Tinkertoy: Build your own operating systems for IoT devices

by

Bingyao Wang

B.Sc., The University of British Columbia, 2019

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Computer Science)

The University of British Columbia
(Vancouver)

April 2021

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Tinkertoy: Build your own operating systems for IoT devices

submitted by **Bingyao Wang** in partial fulfillment of the requirements for the degree of **Master of Science** in **Computer Science**.

Examining Committee:

Margo Seltzer, Professor, Computer Science, UBC
Supervisor

Alan Wagner, Associate Professor, Computer Science, UBC
Additional Examiner

Abstract

The advent of the Internet of Things (IoT) makes it possible for tiny devices with sensing and communication capabilities to be interconnected and interact with the cyber physical world. However, these tiny devices are typically powered by batteries and have limited memory, so they cannot run commodity operating systems that are designed for general-purpose computers, such as Windows and Linux. Embedded operating systems addressed this issue and established a solid foundation for developers to write applications on these tiny devices.

IoT devices are deployed everywhere, from smart home appliances to self-driving vehicles, and their applications impose ever-increasing and more heterogeneous demands on software architecture. There are many special-purpose and embedded operating systems built to satisfy these wildly different requirements, from early sensor network operating systems, such as TinyOS and Contiki, to more modern robot and real-time control systems, such as FreeRTOS and Zephyr. However, the rapid evolution and heterogeneity of IoT applications call for a different solution. Specifically, this work introduces Tinkertoy, a set of standard operating system components from which developers can assemble a custom system. Not only does the custom system provide precisely the functionality needed by an application, but it does so in up to four time less memory than other IoT operating systems and still has comparable performance to them.

Lay Summary

The Internet, long the domain of large and/or expensive devices, is now so pervasive that it is possible for tiny devices ranging from fitness trackers to doorbells to be interconnected, forming a bridge between the physical and digital worlds. Unfortunately, general-purpose operating systems, such as Windows and Linux, cannot run on these tiny devices that have limited hardware resources. Instead, they frequently run special purpose embedded operating systems. However, the range of capabilities of these small devices is immense, and their wildly different application requirements have led to the birth of many different embedded operating systems. We present an alternative solution to address these heterogeneous requirements. Tinkertoy is a set of components, allowing developers to choose the right set and assemble a custom system for their applications. We show that the assembled system consumes up to four times less memory and has comparable performance to other embedded operating systems.

Preface

All of the work presented henceforth was done at the Systopia laboratory at the University of British Columbia, Vancouver campus. This thesis is original, unpublished work by Bingyao Wang, done under the supervision of Margo Seltzer.

Table of Contents

List of Tables

List of Figures

Acknowledgements

Thanks to my supervisor and committee: Dr. Margo Seltzer and Dr. Alan Wagner. I really appreciate your continued support, valuable suggestions, and helpful feedback.

To Nodir Kodirov:
Thank you for sharing your experience and providing suggestions when I am worried about the future.

To Joel Nider:
It's always a pleasure talking with you. Thank you for your helpful advice.

To Christopher Chen:
Thank you for being my beacon. Your suggestions and experience help me find my path.

To Maryam Raiyat Aliabadi:
Thank you for introducing the cyber physical system to me. Your expertise on this area helps me explore applications running on tiny devices.

To the Systopia Lab and Systopians:
It's my honor to work in such an awesome lab with great people. Thank you all.

Keeping productive and writing a thesis during the pandemic are extremely challenging. If you are reading this in the future and had a hard time as well, I totally understand your feelings.

A big thank you to my parents.
Your love and support across the ocean give me motivation and help me finish all of this.

Also thanks to my 17 years old canine buddy.
Every time I see you with a 200ms delay, my worries are gone.

Dedication

To My Parents

Chapter 1 Introduction

At the dawn of the 21st century, computing technology had reached the point where it was suddenly possible to assemble small, lower-powered devices that combined computing, sensing and computation. Proposed applications that could leverage such devices ranged from volcano monitoring [54] to wildlife tracking [60] to monitoring of the world's infrastructure [57].

1.1 Early Wireless Sensor Networks

Wireless sensor networks (WSNs) are composed of tiny sensor devices with wireless communication capabilities. Applications running on these devices are simple; their sole purpose is to take measurements of the physical world via sensors, such as ambient temperature and light sensors, process the data, and transmit it back to a server [41]. However, due to limited computing power and memory (§2.1), these tiny devices cannot run commodity operating systems, such as Windows® and Linux, so researchers began developing embedded operating systems, such as TinyOS [29] and Contiki [14], to provide a platform on which it was possible to develop sensor applications on these first-generation devices.

1.2 From Wireless Sensor Networks to Internet of Things

Early wireless sensor networks used communication protocols, such as ZigBee [61] and Z-Wave [58], tailored to the devices and the specific communication hardware on the devices. As sensor device capabilities increased, they became able to make use of more general-purpose protocols, such as IPv6 and LoWPAN [44], thus increasing device interoperability [51]. For example, a light sensor device can ask smart lamp controller devices to make a room brighter or darker, based on the brightness level it measures. These more capable devices are interconnected and jointly create the network of physical objects, the Internet of Things (IoT) [32].

1.3 Ever-increasing application requirements

The IoT has attracted a lot of attention, because tiny smart devices play an important role in applications such as home automation [50], smart city management [1] and healthcare [39]. These applications are more sophisticated than previous sensor applications, demanding multitasking, efficient memory management, and real-time operations. For example, when an electrocardiogram device detects an irregular heart rhythm, it is crucial that it alert both the patient and the physician in a timely fashion. Subsequently, the physician must be able to retrieve historical data from the device for diagnostic purposes [12][47]. These second-generation devices frequently run newer, real-time operating systems, such as FreeRTOS [19] and Zephyr [49], to address those demands.

1.4 Key Challenge

To date, the IoT has had tremendous impact in applications ranging from healthcare to self-driving vehicles, but there remains much potential. As the class of applications for IoT devices expands, each generation is likely to impose ever-increasing demands on the software infrastructure, so how should we build system software to deal with wildly different application requirements?

Building a unified general-purpose operating system is one solution but not necessarily the best one, because such a system will always have features that particular applications do not need and may not provide precisely the right behavior that an application expects. For example, there are two common programming models, thread-based and event-driven. Controller devices that wait for commands and trigger actuators are easier to express in an event-driven model (§11.3.2, §11.3.3), while gateway devices that translate messages from one protocol to another concurrently are better implemented using a thread-based model (§11.3.4). However, operating system designers typically make the choice on behalf of developers, thus exposing abstractions that may not be suitable for particular applications.

1.5 Our approach and motivation

We propose that the solution lies in making it easy to develop application-specific IoT operating systems to meet these rapidly increasing and diverging demands. However, building a special-purpose system from scratch is overwhelmingly burdensome, so we introduce Tinkertoy, a collection of standard components, from which one can assemble a custom operating system in only a few lines of code. We are inspired by the success of the Unikernel [10] and the design concept of a library operating system [17]. In the same way that Unikernels let developers select only those libraries needed by an application, Tinkertoy lets developers select a set of components from which to assemble a custom operating system that provides precisely the functionality needed by an application. As such, we answer the following three questions:

1. How much effort does it take to assemble a custom system for applications running on an IoT device?
2. How does the memory footprint of such a system compare to other IoT operating systems?
3. How does the runtime performance of such a system compare to other IoT operating systems?

1.6 Odyssey to Tinkertoy

While Unikernels are motivated largely by application-level resource management and removal of protection boundaries, our goal is to assemble customized operating systems that have a small memory footprint and runtime performance comparable to that of other IoT operating systems. Overall, this work makes the following contributions:

- We present Tinkertoy, a collection of standard components that can be assembled into a custom operating system.

- We introduce the design concept behind Tinkertoy to make each component as flexible as possible.

- We exploit recent C++ language features that allow for efficient implementation of customized components.

- We show that the effort of assembling a custom kernel is insignificant in terms of the number of lines of code required.

- We show through an empirical case-based study that assembled kernels have a smaller memory footprint and better runtime performance than other popular IoT operating systems.

Our journey begins with an introduction to microcontrollers and C++ design concepts in Chapter 2. We then present an architectural overview of Tinkertoy components in Chapter 3 and discuss each component in detail in Chapter 4 through Chapter 9. We introduce our evaluation baseline embedded operating systems in Chapter 10. We then present two use cases for which we assemble custom kernels and compare their memory footprint and runtime performance against the baseline systems in Chapter 11. Finally, we discuss future research directions to improve Tinkertoy further and conclude in Chapter 12.

Chapter 2 Background

Our work focuses on supporting IoT devices with a collection of general-purpose system software components that free application developers from having to worry about details of the underlying platform. This chapter presents a brief background on microcontrollers, which are the core technology of these platforms, so we begin with a brief introduction of the technology underlying these devices. Then we briefly discuss several of the core C++ features that we use to ensure the generality, flexibility, and performance of our components.

2.1 Microcontrollers

Microcontrollers, electronic devices that control the analog world, are the heart of tiny, embedded devices, and they are pervasive. For example, the control panel on a refrigerator monitors the temperature in the freezer and turns the compressor off to reduce energy consumption. The collision detector inside a robotic vacuum adjusts the direction to avoid being stuck to a wall. A microcontroller consists of a processor, memory units, and I/O peripherals. Memory units include volatile random-access memory (RAM) and one or more persistent read-only flash memory (ROM) devices. Unlike general-purpose systems, some microcontroller architectures such as the ARM Cortex-M support Execution-In-Place (XIP), a technique that allows the processor to execute code from flash memory directly, so developers do not need to load the code into RAM, thus saving memory for the runtime. I/O peripherals enable the processor to interact with the outside world, such as capturing environmental data via a sensor, listing statistics on an LED display, or sending messages to other devices.

In addition, microcontroller-based systems are constrained in terms of processing power, storage resources, and energy budgets, and lack of a memory management unit (MMU), making them inadequate to run general-purpose operating systems. The Internet Engineering Task Force (IETF) classifies IoT devices into three categories based on their memory size as shown in **Table 2.1** [42]. Class 0 devices have the most limited resources. For example, the

Arduino Nano board is equipped with an ATmega328 processor running at a clock speed of 16 MHz and having only 2 KB RAM and 32 KB flash memory [2]. Class 1 devices are more common and capable of running richer applications. For example, the STM32F102RB board has a 48 MHz Cortex-M3 processor with 16 KB RAM and 128 KB flash memory [48]. Class 2 devices have even more resources but are still not comparable to high-end devices such as a Raspberry Pi 4. These microcontroller-based devices are frequently powered either by batteries or harvested energy [52], so developers must typically allocate and schedule resources cautiously to execute code efficiently.

Device Classes	RAM Size	Flash Size
Class 0	Less Than 10 KB	Less Than 100 KB
Class 1	About 10 KB	About 100 KB
Class 2	About 50 KB	About 250 KB

Table 2.1: *Classes of constrained devices in terms of their RAM and Flash sizes.*

Tinkertoy targets devices that fall into these three classes. Specifically, these devices all have a single-core ARM Cortex-M processor without an MMU, limited volatile and flash memory, sensors and/or actuators, and communication hardware to interact with other devices. However, Tinkertoy is not restricted to support only this kind of device, because the majority of the building blocks are, in fact, general-purpose. With more building blocks becoming available, we can imagine that developers can assemble kernels for devices, for example, that have multiple symmetric cores (§12.2.2).

2.2 Design Concepts

Tinkertoy's design philosophy includes object orientation and polymorphism, as realized in C++ using the technologies described below.

2.2.1 The golden rule: A level of indirection

Tinkertoy's core concept of designing a modular operating system structure is decomposing and assembling. We separate a kernel into multiple standalone components, such as a scheduler, memory allocator and execution model, and then decompose each of them into primitive building blocks that can be combined to assemble a custom component. Both components and building blocks interact with each other via carefully specified interfaces. An interface defines a set of requirements that must be satisfied by custom components and also introduces a level of indirection that allows us to provide different implementations and a clean separation between architecture-dependent and -independent parts of the system.

2.2.2 Virtual functions are good but at what cost

We implement Tinkertoy in C++, so interfaces can be expressed as pure virtual functions. Virtual functions are convenient, because programmers do not have to define and keep track of function pointers manually, but their cost can be significant on resource-constrained devices. First, the compiler generates a virtual function table for each abstract class and adds a hidden instance variable that points to the table to support dynamic dispatch. The extra memory overhead of an abstract class on a 32-bit system is at least $4(M + N)$, where M is the number of virtual functions and N is the number of instances. For example, if we define 4 abstract classes, each of which declares 8 virtual functions and has 8 instances, the overhead is 256 bytes, 12.5% of the memory on a device with 2 KB RAM. As we show in §11.7.1, the actual overhead is even worse in the case of multiple inheritance. Second, the processor needs to read from the table before jumping to the function, so the execution time is longer than an ordinary

function call. As a result, it may not be suitable to have many virtual function calls in a time-sensitive context.

2.2.3 Introduction to C++20 Concepts

We exploit the latest concept feature in C++20 to reduce virtual functions' negative effects on execution time and memory footprint. A concept defines constraints on types to express requirements on classes and functions and is closely related to C++ templates. The compiler validates a concept at compile time and does not need extra data structures at run time, eliminated both of the overheads that virtual functions introduce. For example, **Figure 2.1** shows how we use a built-in concept to filter out sequences of unsupported types by restricting an iterator. The function consumes a pair of input iterators whose value types must be SInt64. As such, the compiler will report an error if programmers pass a vector of Booleans or a list of strings to this function. We can define virtual functions in the form of concepts. **Figure 2.2** presents an abstract class that declares a virtual function and its equivalent concept representation.

```
template <typename Iterator,
          typename Element = std::iter_value_t<Iterator>>
requires std::input_iterator<Iterator> &&
         std::same_as<Element, SInt64>
bool encode(Iterator begin, Iterator end) {…}
```

Figure 2.1: *Use of concepts to reject arguments of unsupported types.*

The encode function accepts only a sequence of SInt64 values. `std::input_iterator<Iterator>` *states that* `Iterator` *should be an input iterator that provides read-only access to its value, while* `std::same_as<Element, SInt64>` *requires that the element type of the sequence is SInt64.*

```
class Interface
{
    virtual ~Interface() = default;
    virtual bool function(int arg) = 0;
};

template <typename T>
concept InterfaceConcept = requires(T& instance, int arg)
{
    { instance.function(arg) } -> std::same_as<bool>;
};
```

Figure 2.2: *Translate an abstract class to a concept.*

Concepts are powerful but cannot fully replace virtual functions, due to their dependence on concrete type parameters, which makes it difficult to use as an opaque base type. As a result, we express requirements as a combination of C++ interface classes and concepts. As readers will see in upcoming chapters, we use concepts extensively to avoid relying on concrete types in our components, and we occasionally use virtual functions to expose public interfaces. Developers are free to customize or create new building blocks, while our constraints guarantee that they are able to assemble those blocks into a complete operating system.

Chapter 3 Architecture Overview

Tinkertoy is composed of the 10 components, shown in **Table 3.1**, from which developers assemble custom kernels. Currently, Tinkertoy limits such kernels to being single-threaded and non-reentrant, so it does not yet support nested hardware interrupts and multiple kernel stacks.

Tinkertoy Components	Component Providers	Related Chapter
Scheduler	Prebuilt / Building Blocks / Developer	Chapter 4
Memory Allocator	Prebuilt / Building Blocks / Developer	Chapter 5
Context Switcher	Prebuilt / Developer	Chapter 6
Execution State	Prebuilt / Developer	
System Call	Prebuilt / Developer	
Dispatcher	Building Blocks / Developer	Chapter 7
Service Identifier Finder	Prebuilt / Developer	
Service Routine Mapper	Developer	
Kernel Service Routines	Prebuilt / Building Blocks / Developer	Chapter 8
Execution Models	Building Blocks / Developer	Chapter 9
Task Control Block Components	Building Blocks / Developer	
Constraints (C++ Concepts)	Tinkertoy	Ubiquitous
==========	==========	==========
Bootloader	Prebuilt / Future Work	Chapter 12
File System	Future Work (TinkerFS)	Chapter 12
IPC Primitives & Framework	Future Work (TinkerIPC)	Chapter 12
RPC Primitives & Framework	Future Work (TinkerRPC)	Chapter 12

Table 3.1: *Tinkertoy kernel components.*

Developers are free to choose a prebuilt component, assemble a custom component from building blocks, or provide their own component to satisfy their application requirements.

Tinkertoy provides building blocks for two common execution models, the thread-based model and the event-driven model, so developers can customize the contents of the task control block, define kernel service routines, and expose related system calls to assemble an execution model that best suits their applications. In either case, there is a kernel stack and at least one user stack on the system. While it is possible to assemble a system that runs on a single stack, we have not explored combinations of building blocks to do so. Furthermore, the constraints component is composed of C++ concepts, ensuring that other components are independent of the execution model and that developer-specified components satisfy all the requirements needed to assemble a custom kernel.

3.1 Overview of Component Interactions

Before digging into the details of each kernel component and its building blocks, we illustrate how they interact with each other to provide services to user applications. Consider a DNS client system consisting of the following four tasks. The sender task constructs a DNS query request and sends it to a DNS server, while the receiver task waits for the response. The printer task decodes the response message and prints the IP address to a serial port. The waiter task waits for the other three tasks to finish, watching for any unexpected terminations. **Figure 3.1** depicts the applications' waiter task making a system call to wait and then turning the control over to the sender task.

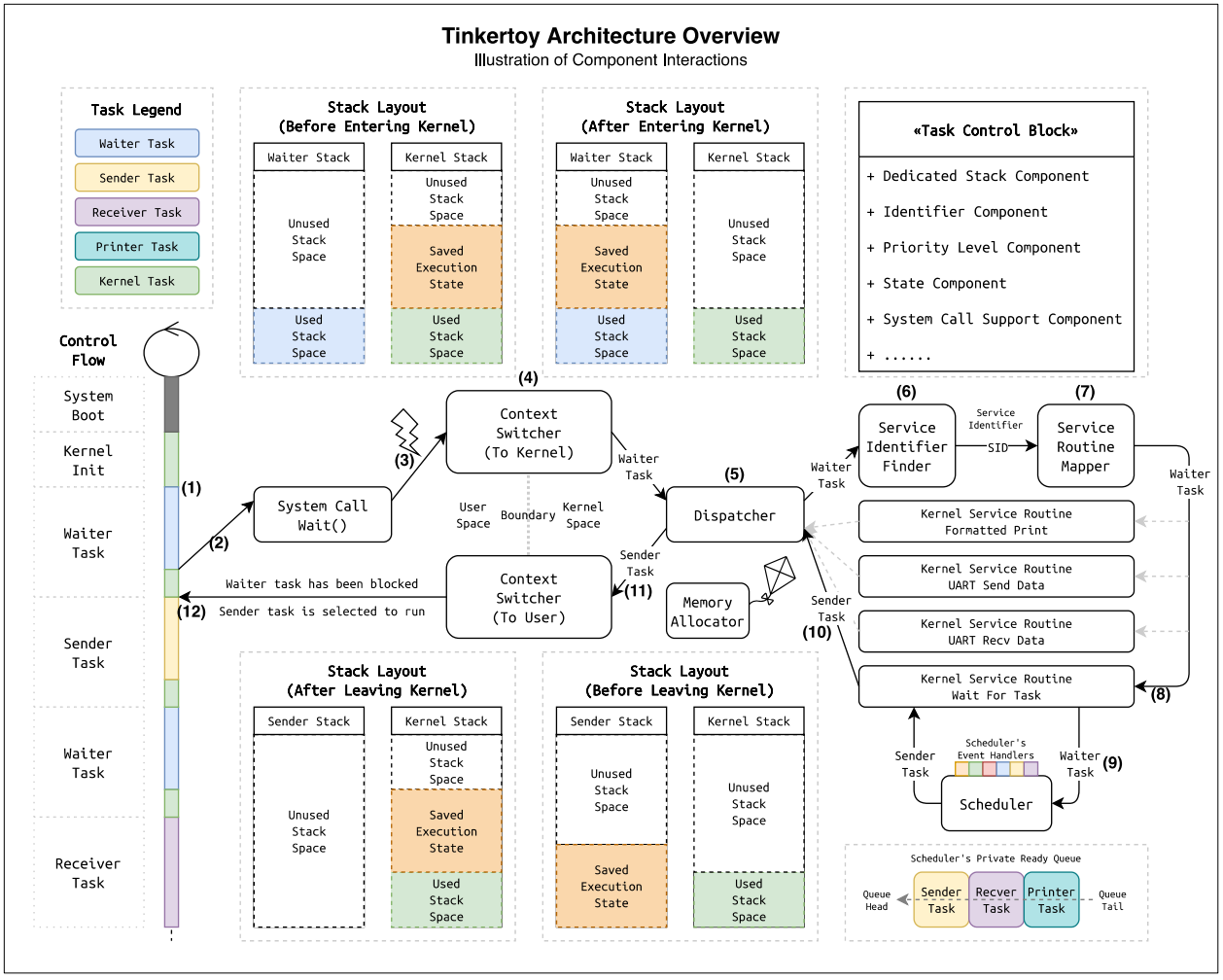


Figure 3.1: *Illustration of interactions between Tinkertoy components to service user requests.*

When the system boots, the kernel initializes itself and lets the waiter task run (1). The waiter task invokes a system call to wait until the sender task finishes (2). The system call generates an exception, causing the processor to switch to privileged mode and jump to a predefined kernel entry point in the context switcher (3). The context switcher preserves machine execution state on the waiter task stack and then restores the kernel state from the kernel stack (4), after which it returns to the dispatcher. The dispatcher relies on two companion components to process the user request (5); it calls the service identifier finder to retrieve a unique service identifier (6) that is subsequently needed by the service routine mapper to select the service routine (7) that implements the system call *wait*(). The kernel service routine receives a reference to the waiter task (8) and finds that the task should be blocked, so it asks

the scheduler to dequeue the sender task (9) which is then returned to the dispatcher (10). The dispatcher knows the next task to run, so it asks the context switcher to exit from the kernel and switch back to the unprivileged mode (11). At the end, the control is handed back to user programs, and the sender task is running (12). Although not used in this example, there are multiple memory allocators that provide dynamic allocated memory for the kernel and user applications as needed.

3.2 Unavailable Components

The existing components are sufficient to assemble custom kernels to run simple user programs as a demonstration of our design concepts (§11.3), but there are several interesting components left to explore, such as the boot sequence, filesystem, and intra- and inter-domain communications (§12.2). We leave exploration of more building blocks, such as a kernel-to-kernel context switcher (to assemble a reentrant and/or multithreaded kernel) for future work.

3.3 Epilogue

We will take a closer look at the architecture diagram in the following chapters and show how we design and implement building blocks for each component. We now move onto the internals of each component, starting with the scheduler.

Chapter 4 Scheduler

A scheduler decides which task should be run at any given moment and for how long it should execute. The scheduling policy affects the quality of service provided by an IoT device. However, it is not possible to design a scheduler that works well for the wide range of IoT applications, because applications have wildly different scheduling requirements. For example, long-running batch tasks, such as a data analysis task that analyzes a collection of environmental samples, are not sensitive to latency. In contrast, real-time tasks must finish before hard deadlines. For example, when the proximity sensor on a self-driving vehicle detects an object, the event handler must deliver the signal to the brake immediately.

Scheduling algorithms are the heart of schedulers and can be classified along different dimensions, such as preemptive versus cooperative, prioritized versus non-prioritized, fair versus biased, and real time versus non-real time. A preemptive scheduler can force a task to relinquish the processor, while a cooperative one waits for a task to finish or yield voluntarily. A prioritized scheduler assigns a priority level to each task and guarantees that the task running on a processor always has the highest priority. The priority level is not limited to predefined numeric values but also can be expressed in the form of deadlines, periods, etc. A fair scheduler ensures that the processor is equally distributed to all tasks, and a hard real time scheduler guarantees that all tasks can meet their deadline.

Tinkertoy provides a collection of fine-grained building blocks from which developers can construct customized schedulers most appropriate for their applications. We discuss how we design these building blocks to encompass the range of the design space described above. We begin with a survey of common scheduling algorithms. We then present how we use C++ concepts to decouple a scheduler from the actual scheduled entities without being restricted to a specific execution model. Subsequently, we illustrate how to build a custom scheduler that meets the demands of applications. We then conclude this chapter with a summary of our scheduling components.

4.1 A survey of common scheduling algorithms

We introduce common scheduling algorithms in a hierarchical manner as presented in **Figure 4.1** and focus on the properties that can be used to classify each of them as summarized in **Table 4.1**. Some of the properties are expressible by others, and we show how they are reflected in our design in §4.2.

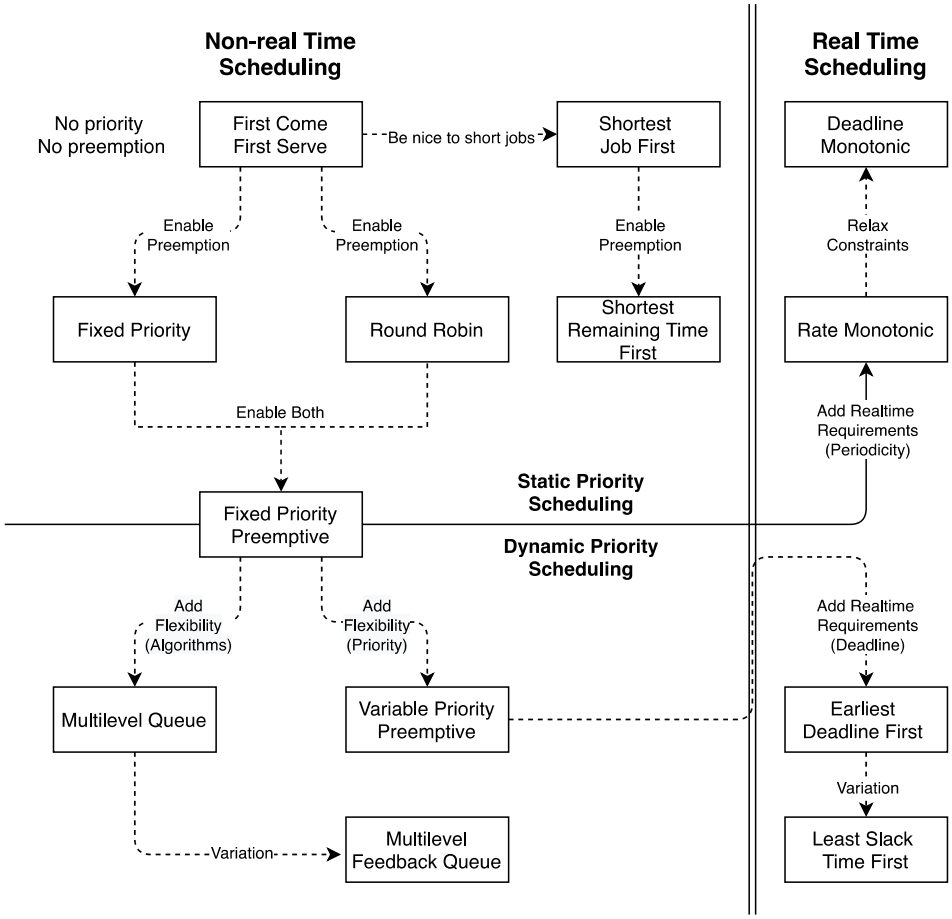


Figure 4.1: *Common scheduling algorithms and their relations.*

First Come First Serve (FCFS) is a simple scheduling algorithm that schedules tasks in order of arrival. The scheduler maintains pending tasks in a first in first out (FIFO) queue. Once a task has acquired the processor, it runs to completion, unless it voluntarily relinquishes the processor. Consequently, a newly arrived task cannot interrupt the running one and must wait

until all other tasks have finished. As such, FCFS is classified as a cooperative scheduling algorithm, and tasks suffer from long delays.

FCFS clearly doesn't work when some tasks are more urgent than others. Such urgent tasks *tend* to have a shorter execution times, so the Shortest Task First (STF) algorithm often addresses urgency and reduces, but does not eliminate, starvation. STF runs the task that has the shortest execution time. Even though it is still non-preemptive, a new task that has a shorter execution time now waits less time to run. However, starvation remains an issue, because longer tasks will never run in the presence of a never-ending sequence of shorter ones. In addition, it is difficult to precisely predict a task's execution time in advance. Statistical methods are used to approximate the execution time based on the binary size and historical runtime data, but this requires additional accounting information and calculation [16]. As a result, STF might introduce extra computational overhead and is seldom deployed in reality [4].

Round Robin (RR) is another attempt to resolve the starvation issue, and it succeeds by enabling preemption. While tasks are still placed in a FIFO queue, the scheduler associates a quantum with each task, determining for how long a task can run before it is preempted. As such, tasks are executed in a cyclic fashion and thus do not suffer from starvation. However, the scheduler is faced with two main challenges. First, it is difficult to choose a perfect quantum for a specific system. If the quantum is too large, all pending tasks have to wait longer and thus the system might not be responsive. In contrast, if the quantum is too small, the system might spend most of its time context switching. Second, the scheduler treats all tasks equally, so there is no mechanism for a task to receive a higher priority. Consequently, a newly arrived critical task cannot run immediately and has to wait until others have used up their quanta.

Priority-based scheduling algorithms address the challenges of round robin [4]. These algorithms extend the round robin scheduler by adding multiple priority levels, using round robin within each level. The result is Prioritized Round Robin (PRR), also known as Fixed Priority Preemptive scheduler. While tasks are still executed in the same manner as before,

the scheduler associates a fixed priority level with each task. As a result, higher priority tasks take precedence over lower ones, and lower priority tasks can run only after higher ones have finished. Furthermore, a PRR scheduler can be extended to support heterogeneous scheduling algorithms for each priority level, producing a Multilevel Queue (MLQ) scheduler. For example, if there are three priority levels, named interactive, default and background, in a system, we can schedule background tasks on a first-come, first-serve basis whereas others in a round robin fashion.

Multilevel Feedback Queue [9] is another priority-based scheduling algorithm that incorporates the concept of dynamic priority scheduling. Initially, all tasks start at the highest priority level with a small quantum. Once a task has used up its quantum, it is demoted to the next priority level with a larger quantum. Eventually, the task is placed on the lowest priority queue and assigned a maximum quantum, so it will run to completion. However, the scheduler could also choose not to demote a task due to its behavior. For example, it could retain the current priority level if a task blocks before the quantum expires, so interactive tasks that frequently wait for user input are able to stay at a high priority. Similarly, it could promote a task that deals with I/O requests to improve its responsiveness.

Real-time tasks are time-sensitive, so the scheduler must ensure that tasks finish before their deadlines. Earliest Deadline First (EDF) exploits the idea of dynamic priority and treats a task's absolute deadline as its priority level. The nearer the deadline, the higher the priority. As a result, when a new task arrives in the system, the scheduler implicitly adjusts the priority level of each active task. Similarly, its variant, Least Slack Time First (LST), determines a task's priority level by its slack time, defined as the difference between the relative deadline and the remaining execution time of a task. In other words, the slack time measures the amount of time left before the deadline, so a task that has the least slack time will have the highest priority.

Not all real-time scheduling algorithms rely on dynamic priority, and some of them are still able to schedule tasks without missing deadlines under some constraints. For example, the Rate Monotonic (RM) scheduler statically determines the priority level of a task by its period.

The period measures how often a task needs to run. A task will be assigned the highest priority level if it has the shortest period. However, the scheduler assumes that all tasks must be periodic and independent of each other, and the deadline must be equal to the period. While the conditions might be severe, the algorithm guarantees that the scheduling is optimal under these assumptions [31]. Its variant, Deadline Monotonic (DM), relaxes the deadline-period equality constraint and gives precedence to the task that has the nearest deadline, but is more complicated to implement in practice [5].

Scheduling Algorithms	**Classifications**			
	Preemptive	**Prioritized**	**Fair**	**Real Time**
First Come First Serve	No	No	Yes	No
Shortest Task First	No	No	Yes	No
Round Robin	Yes	No	Yes	No
Prioritized Round Robin	Yes	Yes	No	No
Multilevel Feedback Queue	Yes	Yes	No	No
Earliest Deadline First	Yes	Yes	No	Yes
Least Slack Time First	Yes	Yes	No	Yes
Rate Monotonic	Yes	Yes	No	Yes
Deadline Monotonic	Yes	Yes	No	Yes

Table 4.1: *Classify common scheduling algorithms along four dimensions.*

4.2 Execution Model Independent Design

Our goal in Tinkertoy is to provide the building blocks so that a developer can assemble a scheduler precisely matched to an application's needs. We first introduce the main classes on which our design is based and then explain each component in detail.

4.2.1 Overview

A scheduler maintains one or more queues to keep track of ready tasks (1) and decides which task to run in response to external events (2), such as a new task arriving in the system (3). **Figure 4.2** depicts the interactions between the scheduler and other kernel components.

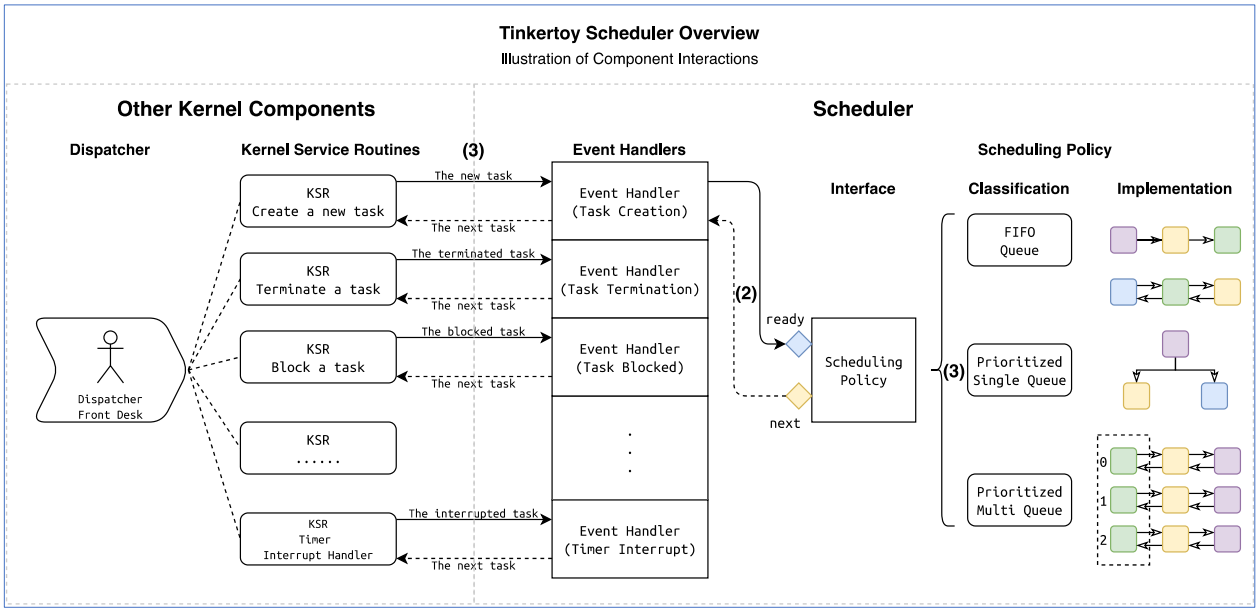


Figure 4.2: *Illustration of interactions between scheduler components.*

However, a scheduler does not implement any policy to prevent priority inversion and deadlock; instead, it is the responsibility of the kernel service routines (details in **Chapter 8**) to detect such situations and take precautions before asking the scheduler to reschedule a task. For example, **Figure 4.3** illustrates how the scheduler is involved in preventing priority inversion. When a high priority task, T_1, should be blocked waiting for a resource currently held by a low priority task, T_2, the kernel service routine that implements the system call should change T_2's priority to high (A) and notify the scheduler of the change (B), so the scheduler can reorder tasks in the ready queue (C). Finally, the service routine blocks T_1 by asking the scheduler to dequeue the next task T_3 (D).

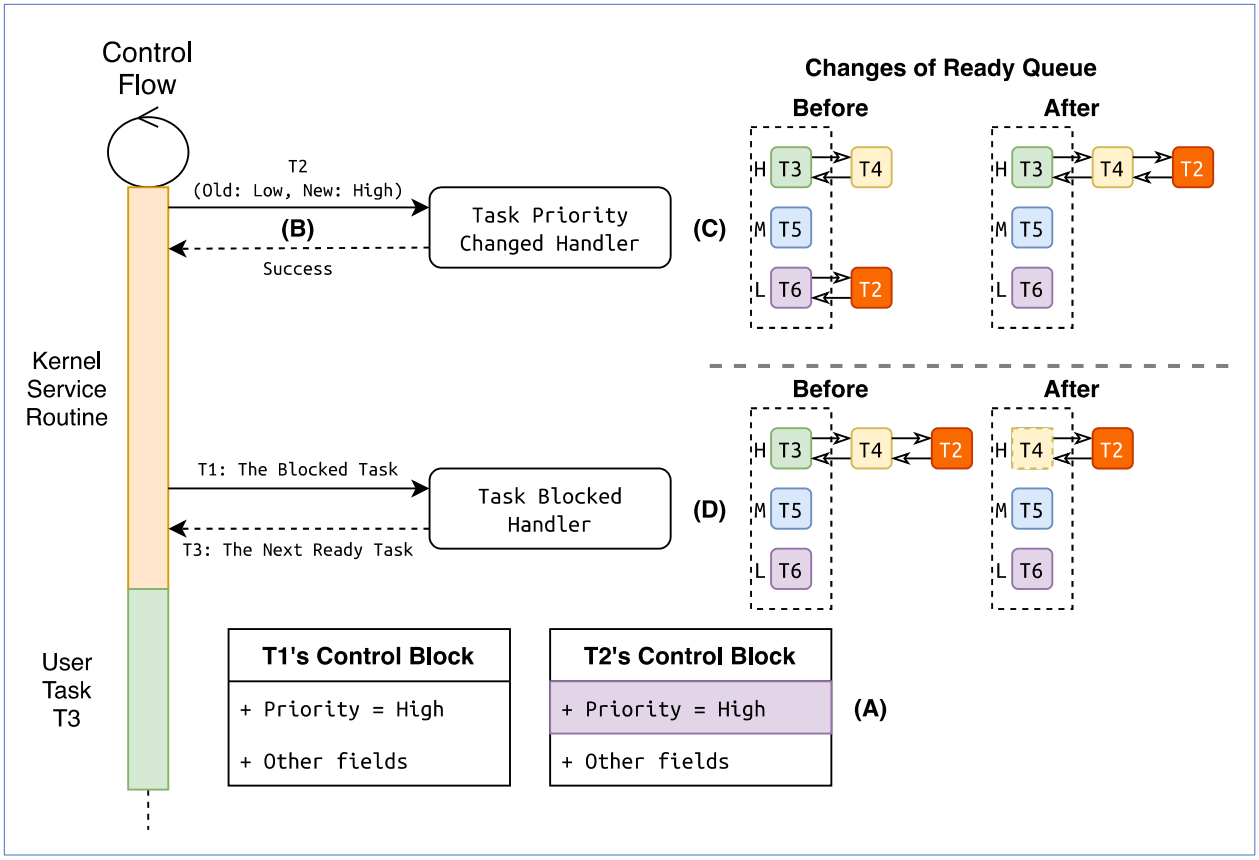


Figure 4.3: *Illustration of interactions between the scheduler and the kernel service routine.*

On some systems, a scheduler schedules processes; on others it schedules threads. A flexible scheduler should be agnostic with respect to the particular object it is scheduling yet should be constrained to handle only those objects that can reasonably be scheduled. We express constraints on the type of the item being scheduled using C++ concepts. Developers must provide a concrete type that satisfies all the requirements specified by the concept. To accommodate these requirements, Tinkertoy's scheduler is composed of three components: Policy, Event Handlers and Task Control Block Constraints. We focus on the first two components in the following sections and define specific constraints as they become necessary. **Table 4.2** summarizes the different types for each component.

Scheduling Policies	Event Handlers
First In First Out Queue	Timer Interrupt
Prioritized Single Queue	Task Creation
Prioritized Multi Queue	Task Termination
Task Control Block Constraints	Task Yielded
Schedulable	Task Blocked
Implicitly Prioritizable	Task Unblocked
Prioritizable By Priority	Task Killed
Prioritizable By Mutable Priority	Task Priority Changed
Prioritizable By Auto Mutable Priority	Task Self Priority Changed
Quantizable	Task Quantum Used Up

Table 4.2: *Scheduler components.*

4.2.2 Policy Component

The policy component is responsible for managing the queue and reflects whether a scheduler is prioritized or not. It exposes two primitive functions, *ready* that enqueues a ready task and *next* that dequeues the next ready task. The queue accepts tasks only if they are schedulable, so we define an empty base class *Schedulable Marker*, from which all schedulable tasks must inherit.

Tinkertoy provides three types of queues, FIFO Queue, Prioritized Single Queue and Prioritized Multi Queue. A scheduler that adopts a priority queue must assign priorities, and we define the following constraints to ensure that such a scheduler can prioritize tasks properly. *Implicitly Prioritizable* requires that a concrete task type overloads the C++ comparison operators. In this case, a task can keep its priority level private to the scheduler. On the contrary, *Prioritizable By Priority* requires that a task explicitly reveals its priority level via a getter function. The actual type and the meaning of a priority level is determined by developers. **Figure 4.4** presents the concept definition.

```cpp
template <typename T>
concept ImplicitlyPrioritizable = requires(const T& a, const T& b)
{
    { a <  b } -> std::convertible_to<bool>;
    { a >  b } -> std::convertible_to<bool>;
    { a <= b } -> std::convertible_to<bool>;
    { a >= b } -> std::convertible_to<bool>;
};

template <typename Task>
concept PrioritizableByPriority = requires(const std::remove_reference_t<Task>& task)
{
    /// The task must explicitly define its priority level type
    typename Task::Priority;

    /// The priority level must also be comparable
    /// @note The larger the priority level, the higher the task priority.
    requires ImplicitlyPrioritizable<typename Task::Priority>;

    /// The task must be able to provide its immutable priority level
    { task.getPriority() } -> std::same_as<const typename Task::Priority&>;
};
```

Figure 4.4: *Concept definitions of Implicitly Prioritizable and Prioritizable By Priority.*

The Prioritized Multi Queue component allows developers to specify a potentially different scheduling policy for each priority level. A multilevel queue scheduler can then use the priority level of a task to decide onto which queue to place it. If the queue has not been initialized yet, the scheduler uses a *mapper* provided by developers to retrieve an instance. The mapper is a C++ functor that overloads the operator (), consuming a priority level and producing the corresponding queue. For example, one can build a Prioritized Round Robin scheduler by providing a mapper that returns a FIFO queue for every priority level. The scheduler manages the memory of these internal queues, so we also allow developers to specify a deleter in the mapper, if queues are dynamically allocated. When the scheduler is deleted, it invokes the deleter function on each queue. **Figure 4.5** presents a sample mapper implementation.

```
template <typename Task>
requires PrioritizableByPriority<Task>
struct DynamicFIFO
{
    /// Maps the given priority level to a specific scheduling policy
    SchedulingPolicy<Task>* operator()(const typename Task::Priority& priority)
    {
        return new SchedulingPolicies::FIFO::LinkedListImp<Task>();
    }

    /// A custom deleter that releases the policy properly
    struct Deleter
    {
        void operator()(SchedulingPolicy<Task>* policy)
        {
            delete policy;
        }
    };
};
```

Figure 4.5: *Implementation of a mapper that dynamically creates a FIFO queue for each priority level.*

Tinkertoy also provides code extension support for developers to customize enqueue and dequeue behaviors conveniently. Traditionally, developers add more functionality in two ways, subclassing or using delegation, but neither of them make it easier to reuse code. Our design encapsulates extra functionality as individual reusable injectors, so developers can build a new policy component from an existing one given a list of injectors as shown in **Figure 4.6**. An injector is a C++ functor that takes a reference to a task. Depending upon when the new policy component invokes an injector, the task can be either the one to be enqueued or the one that has just been dequeued. We demonstrate an injection that allocates a quantum to a task before enqueuing the task in §4.3.2. In general, the policy component makes it possible to materialize the queue for all common scheduling algorithms.

```
/// Defines the interface of a scheduling policy that supports code extension to `ready()` via injectors
template <typename Policy, typename... Injector>
struct SchedulingPolicyWithEnqueueInjectors: public Policy
{
public:
    /// Enqueue a ready schedulable task
    void ready(typename Policy::SchedulableTask* task) override
    {
        /// C++ fold expression to invoke a list of injectors in order
        ((Injector{}(task)), ...);

        /// Invoke the original ready() function to enqueue the task
        Policy::ready(task);
    }
};

/// Defines the interface of a scheduling policy that supports code extension to `next()` via injectors
template <typename Policy, typename... Injector>
struct SchedulingPolicyWithDequeueInjectors: public Policy
{
public:
    /// Dequeue the next ready schedulable task
    typename Policy::SchedulableTask* next() override
    {
        /// Invoke the original next() function to dequeue the task
        typename Policy::SchedulableTask* task = Policy::next();

        /// C++ fold expression to invoke a list of injectors in order
        ((Injector{}(task)), ...);

        return task;
    }
};

/// Examples
using FIFO = SchedulingPolicies::FIFO::LinkedListImp<SimpleTask>;
using SSPQ = SchedulingPolicies::PrioritizedSingleQueue::LinkedListImp<SimpleTask>;
using MyPolicy1 = SchedulingPolicyWithEnqueueInjectors<FIFO, MyInjector1, MyInjector2>;
using MyPolicy2 = SchedulingPolicyWithDequeueInjectors<SSPQ, MyInjector1, MyInjector3, MyInjector2>;
```

Figure 4.6: *Composition of a new policy component out of an existing one and a list of injectors.*

We build a new policy $MyPolicy1$*, on top of FIFO. Its ready method invokes* $MyInjector1$ *and* $MyInjector2$ *before placing the task on the queue. Similarly, we build* $MyPolicy2$ *on top of a Prioritized Single Queue. It invokes* $MyInjector1$*,* $MyInjector3$*, and* $MyInjector2$ *after removing the task from the queue.*

4.2.3 Event Handler Component

The event handler component is responsible for reacting to all scheduling events that can occur on a system and reflects scheduler characteristics such as whether a scheduler is preemptive or not. For example, when a new task is created, a cooperative scheduler might put the new task into the queue and keep the current task running, while a preemptive one might check the priority level of both tasks and run the task that has a higher priority. These two different reactions are reflected in the implementation of the task creation event handler.

Tinkertoy provides ten types of event handlers, allowing developers to specify which types of events a scheduler can respond to and how it should respond. Each of them defines the interface through which other kernel modules interact with the scheduler and uses primitive functions provided by the policy component to manipulate the ready queue. **Table 4.2** lists all the event handlers. Conceptually, the scheduler prohibits direct accesses to its ready queue and expects kernel routines that service system calls to invoke its handlers. For example, if the kernel allows tasks to change their priority at run time, the system call handler invokes the Priority Changed handler to inform the scheduler that the priority level of a task has been changed; it is then up to the scheduler to make that priority change and reschedule if necessary. Our design has no limit on the number of handler types, so developers are free to declare and implement new types of handler to suit their needs.

Each type of handler has several default implementations, such as one that is designed for a preemptive scheduler, one that takes the idle task into consideration, and one that keeps track of the time left in a quantum assigned to a task. Normally, event handlers always return the next ready task each time they are called, but some of them support group operations, which makes it easier for developers to deal with multiple tasks. For example, developers may use the task unblocked handler to unblock tasks one after another and dequeue the next task at the last invocation. Such usage can be convenient if the kernel supports system calls, such as *sysWaitForTask*, which causes multiple tasks to be blocked waiting for an event. In general, developers can extend default implementations or provide their own. Subsequently, they choose or create event handlers that are necessary for their system to assemble a custom scheduler as needed. In the next section, we demonstrate how to combine components to produce a scheduler.

4.3 Building Custom Schedulers

Tinkertoy provides a convenient *Scheduler* class for developers to use to assemble a scheduler from a scheduling policy component and a collection of event handlers. We illustrate this

process to assemble a simple FIFO scheduler and a more complex Multilevel Feedback Queue scheduler.

4.3.1 FIFO Scheduler

4.3.1.1 System Requirements

Let's say that we want to build a scheduler for a kernel that allows a process to create another process, relinquish the processor voluntarily and wait for that process to finish. We assume the existence of a user process that never terminates (e.g., Unix's init process). The kernel expects all processes to run in arrival time order. The system does not have a hardware timer.

4.3.1.2 Assembling Schedulers

The above requirements suggest that we need five event handlers: Task Creation Handler, Task Termination Handler, Task Yielded Handler, Task Blocked Handler and Task Unblocked Handler. Since the system has a never terminating user process, there is always a runnable process, so we do not need an idle task. We assemble the FIFO scheduler as shown in **Figure 4.7**.

```
template<typename Task>
class FIFO : public Scheduler<
        SchedulingPolicies::FIFO::LinkedListImp<Task>,
        SchedulingEventHandlers::TaskCreation::Cooperative::KeepRunningCurrent<Task>,
        SchedulingEventHandlers::TaskTermination::Common::RunNext<Task>,
        SchedulingEventHandlers::TaskBlocked::Common::RunNext<Task>,
        SchedulingEventHandlers::TaskUnblocked::Cooperative::KeepRunningCurrent<Task>,
        SchedulingEventHandlers::TaskYielded::Common::RunNext<Task>> { … }
```

Figure 4.7: *Composition of a FIFO scheduler using existing Tinkertoy components.*

We use a FIFO queue to ensure that tasks run in arrival time order. We choose the cooperative version of Task Creation Handler and Task Unblocked Handler to ensure that the running task cannot be preempted. The class constructor that initializes the queue is not shown in the figure.

4.3.2 Multilevel Feedback Queue Scheduler

4.3.2.1 System Requirements

Next, let's examine how to construct a scheduler for a kernel that allows a process to create another process and wait for that process to finish, given different system capabilities. This time, the system has a hardware timer that generates interrupts at fixed intervals, and the kernel supports three priority levels, Low, Medium and High, and their quanta are 40, 20 and 10 ticks respectively. Each process starts running at the highest priority level and will be demoted to the next lower level if it does not relinquish the processor before using up its quantum. If a task is preempted by a higher priority task, the system resets its remaining ticks to the initial value specified by its priority level as a compensation. We assume that every process is guaranteed to finish, so the system can become idle with no user processes to run.

4.3.2.2 Assembling Schedulers

The above requirements suggest that we need five event handlers: Task Creation Handler, Task Termination Handler, Task Blocked Handler, Task Unblocked Handler and Timer Interrupt Handler. Since the system can run out of user processes to run, we also need the idle task component. We also need the Prioritized Multi Queue policy component with a mapper that returns a FIFO queue for each priority level and an injector that allocates ticks to a process.

Timer Interrupt Handler and Companion Constraints

The scheduler needs to check the remaining time of a task in the timer interrupt handler, so Tinkertoy defines the constraint *Quantizable* as shown in **Figure 4.8** to ensure that a task type supports quantum-related operations. A task implements a tick function for the scheduler to adjust its remaining time at each timer tick and a query function to check whether it has used up its time allotment. Once the time left in the quantum reaches zero, the scheduler must demote the task to the next level and reallocate a quantum to it, so, the task should also provide an allocate function to receive the new number of ticks.

```cpp
template <typename Task>
concept Quantizable = requires(Task& task, typename Task::Tick ticks)
{
    /// The task must explicitly define its tick type
    typename Task::Tick;

    /// The tick type must be convertible to an unsigned integer
    std::unsigned_integral<typename Task::Tick>;

    /// The task should adjust its remaining ticks properly on a timer tick
    { task.tick() } -> std::same_as<void>;

    /// The task should report whether it has used up its time allotment
    { task.hasUsedUpTimeAllotment() } -> std::same_as<bool>;

    /// Other entity should be able to allocate a certain number of ticks to the task
    { task.allocateTicks(ticks) } -> std::same_as<void>;
};
```

Figure 4.8: *Concept definition of Quantizable.*

Unlike in the FIFO example, the priority level of a task is no longer a constant, so we need new constraints to model such behavior. *Prioritizable By Mutable Priority* inherits from *Prioritizable By Priority* and requires a setter function to modify the task priority. Subsequently, *Prioritizable By Auto Mutable Priority* adds two more functions, promote and demote, to adjust the priority level conveniently, because levels may not be contiguous numeric values. **Figure 4.9** presents the concept definition.

```cpp
template <typename Task>
concept PrioritizableByMutablePriority = requires(Task& task, const typename Task::Priority& priority)
{
    /// The task must be prioritizable by priority
    requires PrioritizableByPriority<Task>;

    /// The task must be able to accept a new priority level
    { task.setPriority(priority) } -> std::same_as<void>;
};

template <typename Task>
concept PrioritizableByAutoMutablePriority = requires(std::remove_reference_t<Task>& task)
{
    /// The task must be prioritizable by its mutable priority
    requires PrioritizableByMutablePriority<Task>;

    /// The task can be promoted to the next priority level
    /// @note The priority level should remain unchanged if it is the highest one.
    { task.promote() } -> std::same_as<void>;

    /// The task can be demoted to the next priority level
    /// @note The priority level should remain unchanged if it is the lowest one.
    { task.demote() } -> std::same_as<void>;
};
```

Figure 4.9: *Concept definition of Prioritizable By Mutable Priority and Prioritizable By Auto Mutable Priority.*

The scheduler also needs to allocate a quantum for every task placed into the ready queue, so we use a code injector that performs tick allocation to extend the existing Prioritized Multi Queue component as shown in **Figure 4.10**. The injector simply invokes *allocateTicks* to assign a certain number of ticks to a task based on the task's current priority level.

```
template <typename Task, typename QuantumSpecifier>
requires Quantizable<Task> &&
         PrioritizableByPriority<Task>
struct PriorityBasedTaskQuantumAllocator
{
    void operator()(Task* task)
    {
        // Guaranteed by the constraint `Prioritizable By Priority`
        typename Task::Priority priority = task->getPriority();

        // Get the number of ticks based on the task priority
        typename Task::Ticks ticks = QuantumSpecifier{}(priority);

        // Guaranteed by the constraint `Quantizable`
        task->allocateTicks(ticks);
    }
};

struct MyQuantumSpecifier
{
    UInt32 operator()(UInt32 priority)
    {
        switch (priority)
        {
            case kLow:
                return 40;

            case kMedium:
                return 20;

            case kHigh:
                return 10;
        }
    }
};

// Pseudocode to build a new policy component upon Prioritized Multi Queue
using MLFQPolicy = SchedulingPolicyWithEnqueueInjectors< /* See The Last Figure */,
    PriorityBasedTaskQuantumAllocator<Task, MyQuantumSpecifier>>;
```

Figure 4.10: *Build a new policy component for the multilevel feedback queue scheduler.*

We can use above primitives to implement the timer interrupt handler in only three steps as shown in **Figure 4.11**. First, we notify the task that it has used one tick by calling its tick function. Then, we invoke its query function to determine whether the task has used up its time allotment. If so, we delegate the rest of work to the Quantum Used Up Handler, otherwise we keep the task running. The Quantum Used Up Handler invokes the demote function to lower the task's priority level. It then asks the policy component to put the task back on the ready queue and return the next task.

```
Task* onTimerInterrupt(Task* current) override
{
    // The current running task has used one tick
    current->tick();

    // Guard: Demote the task if it has used up its time allotment
    if (current->hasUsedUpTimeAllotment())
    {
        return this->onTaskQuantumUsedUp(current);
    }

    // Keep running the current task
    return current;
}

Task* onTaskQuantumUsedUp(Task* current) override
{
    // Decrement the priority level of the current running task
    current->demote();

    // Retrieve the current scheduling policy
    SchedulingPolicy<Task>& policy = this->getPolicy();

    // Enqueue the current task: Ticks allocation happens here
    policy.ready(current);

    // Select the next one to run
    return policy.next();
}
```

Figure 4.11: *Implementation of the timer interrupt handler.*

Finally, we can assemble a Multilevel Feedback Queue scheduler that satisfies all the requirements as follows.

```
template<typename Task, typename QuantumSpecifier, size_t MaxPriorityLevel>
class MultilevelFeedbackQueue : public Scheduler<
        SchedulingPolicyWithEnqueueInjectors<
                SchedulingPolicies::PrioritizedMultiQueue::ArrayMapImp<Task, DynamicFIFO<Task>, MaxPriorityLevel>,
                SchedulingPolicies::CodeInjector::PriorityBasedTaskQuantumAllocator<Task, QuantumSpecifier>>,
        SchedulingEventHandlers::TaskCreation::Preemptive::RunHigherPriorityWithIdleTaskSupport<Task>,
        SchedulingEventHandlers::TaskTermination::Common::RunNextWithIdleTaskSupport<Task>,
        SchedulingEventHandlers::TaskBlocked::Common::RunNextWithIdleTaskSupport<Task>,
        SchedulingEventHandlers::TaskUnblocked::Preemptive::RunNextWithIdleTaskSupport<Task>,
        SchedulingEventHandlers::TimerInterrupt::Preemptive::KeepRunningCurrentWithAutoDemotionAndIdleTaskSupport<Task>>
{ … }
```

Figure 4.12: *Composition of a Multilevel Feedback Queue scheduler using existing components.*

The Priority Based Task Quantum Allocator relies on a Quantum Specifier to allocate certain number of ticks based on the task priority level. The Keep Running Current With Auto Demotion And Idle Task Support combines the Timer Interrupt Handler with the Quantum Used Up Handler discussed above.

4.4 Summary

In this chapter, we described how we decompose a scheduler into three primitive components, Policy, Event Handlers and Constraints, and use them to assemble sample schedulers that

satisfy system requirements. The constraint component ensures that schedulers can work with any schedulable types properly, while developers are responsible for providing the concrete type that satisfies those requirements. The policy component manages the ready queue and is the heart of a scheduler. Our event handlers make it possible for developers to customize how a scheduler reacts to the outside world at a fine-grained level. Our building blocks are reusable and extensible, so developers are able to choose the exact ones and assemble a custom scheduler quickly.

Chapter 5 Memory Allocator

Physical memory is a scarce resource on tiny devices that do not have a memory management unit (§2.1). Operating systems provide two types of memory allocation, static and dynamic. Static memory allocation is simple and has no computation overhead, thus is widely used in IoT systems [37]. Operating systems allocate global variables when loading the process into memory. Static allocation is adequate for applications that can make an accurate prediction on the resource needed in advance. For example, the Constrained Application Protocol (CoAP) [43] is a specialized HTTP-like application-layer protocol for memory-constrained devices. A CoAP server that allocates a session object for each incoming connection can allocate them in advance, if it knows the maximum number of concurrent connections. However, when it cannot predict the length of each incoming request message, static allocation becomes limited, because application developers typically want to pre-allocate such buffers to avoid running out of memory.

Dynamic allocation addresses the concern of over-provisioning but increases the complexity of both user applications and the system. Applications must be prepared to handle allocation failures, while the system has to address fragmentation issues, otherwise it runs out of memory more quickly. The system must also provide a deterministic allocator for real-time applications that require predictable runtime [33]. As such, existing IoT operating systems provide limited support for dynamic allocation and discourage developers from using it [25].

Tinkertoy provides four prebuilt memory allocators, a free list allocator, a fixed-size resource allocator, a fast pool allocator, and a binary buddy allocator, and building blocks from which developers can assemble other custom allocators. We begin with an overview of common allocation strategies and then discuss how we decompose an allocator into individual components. Subsequently, we assemble an efficient binary buddy allocator for memory-constrained devices. We discuss how to reuse existing building blocks and design new ones to

add more functionality to allocators and then summarize our approach to conclude this chapter.

5.1 Overview of common allocation strategies

An allocator partitions a chunk of memory into multiple regions and finds a free one to satisfy an allocation request. Each allocation strategy has its unique set of design decisions and tradeoffs. Some of them are optimized for speed, while others are optimized for space. In general, there are two partitioning schemes: fixed-size partitions and variable-sized partitions.

5.1.1 Fixed-Size Partitions: The Original

As the name suggests, this allocator divides free memory into fixed-size blocks. It cannot satisfy a request larger than its block size; it always returns a single block or fails. As a result, the block size has a direct impact on its performance. The allocator suffers from internal fragmentation if its block size is too large relative to the requested size, in which case the system wastes memory. At the same time, it becomes impractical if the block size is too small, as it cannot fulfill most allocation requests.

The McKusick-Karels allocator used in 4.3BSD maintains multiple pools of memory blocks, each of which keeps track of free blocks of a particular power-of-two size [34]. Internal fragmentation is reduced to some extent but is still high due to the power-of-two constraint, so fixed-sized allocators are seldom used in general-purpose operating systems. However, it inspired special-purpose allocators that allocate objects of specific types.

5.1.2 Fixed-Size Partitions: The Variants

Fixed-sized allocators are commonly used as pool allocators, because most programs exhibit a strong pattern of memory allocations. For example, when a new task is created, the kernel allocates a task control block to store relevant control data. The zone-based allocator [46] and

the slab allocator [26] fully exploit such allocation patterns, and they share the same fundamental idea. The allocator reserves a chunk of memory to store objects of a particular type and then manages the memory as a pool of fixed size blocks. When a program requests a new object, the allocator finds a free slot in the pool. If all slots are in use, it allocates another pool. However, the allocator is not restricted to manage a single type of object. For example, the zone allocator in the XNU kernel on macOS maintains a large collection of common object types, such as task control block, IPC port, Mach message [46]. While a fixed-sized allocator is convenient to allocate objects of known types, it does not work well with allocations of arbitrary size.

5.1.3 Variable-Size Partitions: The Original

Variable-sized allocators do not have internal fragmentation, because the size of each partition is determined by the amount of memory requested by a program. However, external fragmentation becomes an issue as programs start to release memory. Free memory areas may not be contiguous, so a new allocation can fail even though the total amount of free memory is sufficient to satisfy the request. Several policies have been proposed to reduce external fragmentation [56]. For example, the allocator can coalesce adjacent free blocks as much as possible or relocate allocated memory blocks to build a single compact region if necessary. Despite this drawback, a variable-sized allocator can provide fine-grain allocation and is adaptable to general scenarios.

5.1.4 Variable-Size Partitions: The Variants

The binary buddy allocator [40] is a compromise solution to reducing both internal and external fragmentation. The allocator assigns an order i to each block, indicating that the block size is proportional to 2^i. The smallest block has order 0 and determines the minimum size allocation, while the largest block has order N and is the single free block managed by the allocator.

When a program requests an allocation, the allocator computes the smallest order K that will satisfy the request. If no block of that order is available, it recursively looks for a higher order free block. Eventually, it will either fail (in which case the system is essentially out of memory) or it will find some available block of order T. It then splits the order T block in half producing two blocks of order $T - 1$. It uses one block to satisfy the allocation (potentially recursively splitting the block until producing a block of order K); it places the other in the list of free blocks of order $T - 1$.

These split blocks are called buddies, and a block can only be merged with its buddy. There also exist several variants, such as Fibonacci buddy [28] and weighted buddy [45] allocators, but the concept of buddy is the same. In general, the buddy allocator is fast and works well if the requested amounts of memory are carefully chosen, otherwise internal fragmentation remains a severe issue.

5.1.5 Epilogue

We discussed several common strategies for dynamic memory allocation. No allocation scheme works well in every scenario, so we give developers a range of choices so they can choose the one best suited to their applications.

5.2 Building Block Design

Dynamic memory allocation involves two major operations, *allocate* to allocate a specific amount of memory and $free$ to reclaim a previously allocated chunk of memory, so an allocator must keep track of which portions of memory are in use or free. We decompose a memory allocator into four components: Memory Block, Static Aligner, Primitive Steps, and Account Book. **Figure 5.1** illustrates the interactions between these components.

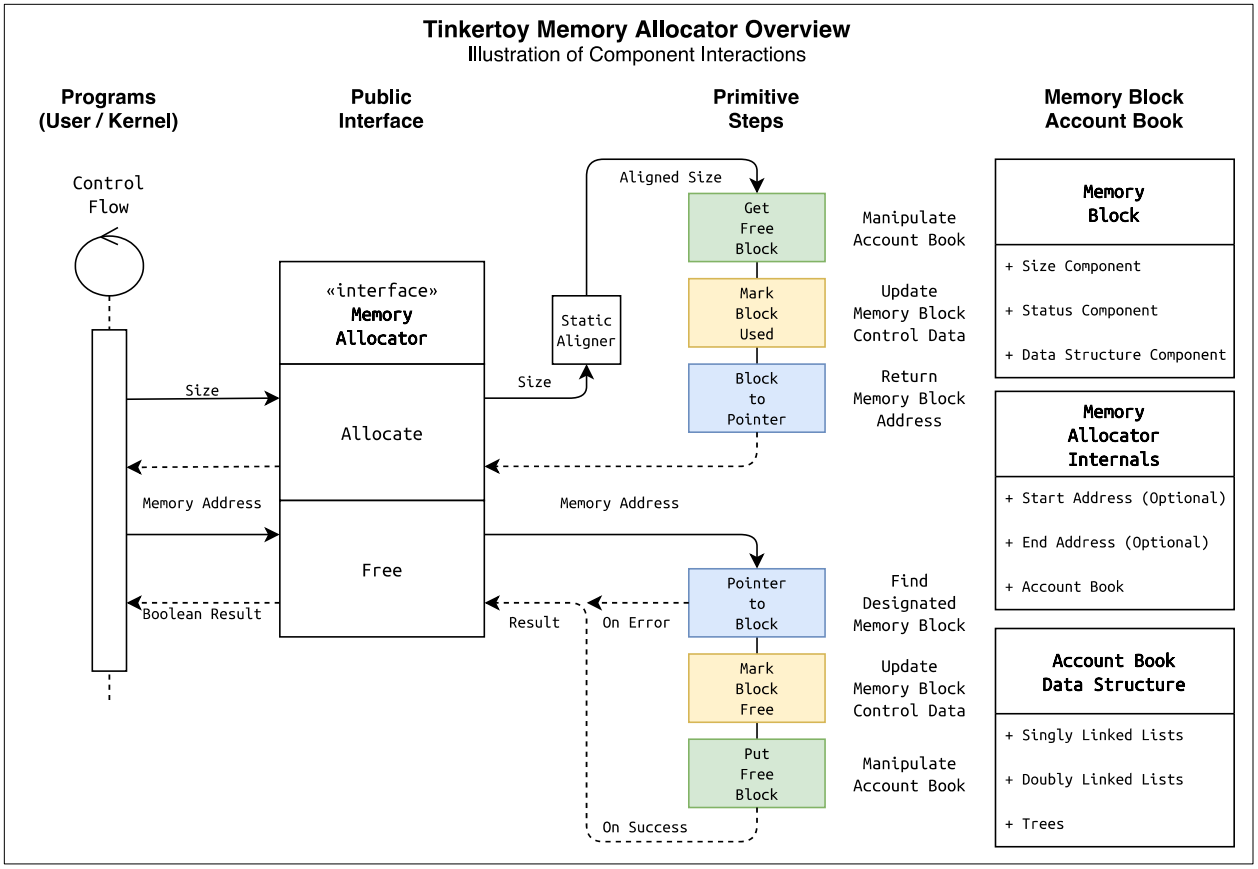


Figure 5.1: *Illustration of interactions between memory allocator components.*

5.2.1 Memory Block

A memory block is an abstraction for a region of allocated memory that might contain control data (metadata) necessary for the allocator to track space that is available/unavailable for allocation. For example, a free list allocator needs to know the size of each memory region, while a fixed-size allocator needs to know only whether a region is allocated. As a result, the content of a memory block is allocator specific.

Tinkertoy provides a collection of standard memory block components to store the size, record the allocation status, maintain pointers to previous or next blocks, etc., allowing developers to assemble a custom memory block simply by selecting existing components. Alternately, they can implement a custom allocator using some or none of the existing components. For example,

if a tiny device has less than 64 KB memory, one might prefer to use two 2-byte integers instead of 4-byte ones to store the block size and the address of the next free block.

5.2.2 Static Aligner

Variable-sized memory allocators might rely on a static aligner to ensure that all allocations are properly aligned. For example, the ARM Cortex v7m architecture requires stacks to be aligned to an 8-byte boundary [3], so a stack allocator must return an address divisible by 8. An aligner is a C++ functor that calculates the amount of memory needed to satisfy both the allocation request and the alignment requirement; it is invoked by the allocate function. Tinkertoy provides three types of aligners: null, constant and power-of-two. **Figure 5.2** presents a sample implementation.

```
template <size_t Alignment>
struct ConstantAligner
{
    constexpr size_t operator()(size_t size)
    {
        size_t result = size / Alignment;

        result += (size % Alignment) ? 1 : 0;

        result *= Alignment;

        return result;
    }
};
```

Figure 5.2: *Implementation of a static aligner that aligns allocations to a constant boundary.*

5.2.3 Primitive Steps

Allocate is composed of three primitive steps, Get Free Block, Mark Block Used, and Block to Pointer. Get Free Block tries to find a free memory block large enough to satisfy the request. Subsequently, Mark Block Used might modify the control data to mark the block in use. At the end, Block to Pointer returns the start address of the block to the program. As such, we can provide a default implementation for *allocate* with just these primitives as shown in **Figure 5.3**.

Similarly, *free* is also composed of three primitive steps, Pointer to Block, Mark Block Free, and Put Free Block, each of which does the reverse of the corresponding step in *allocate*. Specifically, Pointer to Block locates the memory block referenced by the given pointer. Mark Block Free then marks the designated memory block free. Finally, Put Free Block reclaims the memory back for later allocations. Therefore, we can implement *free* as shown in **Figure 5.3**.

```
void* allocate(size_t size)
{
    // Guard: Return null if size is 0
    if (size == 0)
    {
        return nullptr;
    }

    // Guard: Find a block that satisfies the request
    auto block = this->getFreeBlock(Aligner{}(size));

    if (block == nullptr)
    {
        return nullptr;
    }

    // Mark the block used and return its address
    this->markBlockUsed(block);

    auto address = this->block2Pointer(block);

    return address;
}
```

```
bool free(void* pointer)
{
    // Guard: It's OK to free a NULL pointer
    if (pointer == nullptr)
    {
        return true;
    }

    // Guard: Locate the designated block
    auto block = this->pointer2Block(pointer);

    if (block == nullptr)
    {
        return false;
    }

    // Mark the block free and put it back
    this->markBlockFree(block);

    this->putFreeBlock(block);

    return true;
}
```

Figure 5.3: *Default implementation of the allocate and the free function.*

5.2.4 Account Book

Primitive steps track allocations with specific data structures, each of which implements part of an allocation algorithm. Tinkertoy provides two types of account book, overlay and standalone. The former stores control data of a memory block in the block itself, while the latter allocates additional memory for control data. **Figure 5.4** illustrates the difference. We also provide common data structures for each of them, such as a singly linked list, a doubly linked list, a binary tree. For example, as shown in §5.3, we can assemble a binary buddy allocator that uses a binary tree to keep track of buddy blocks.

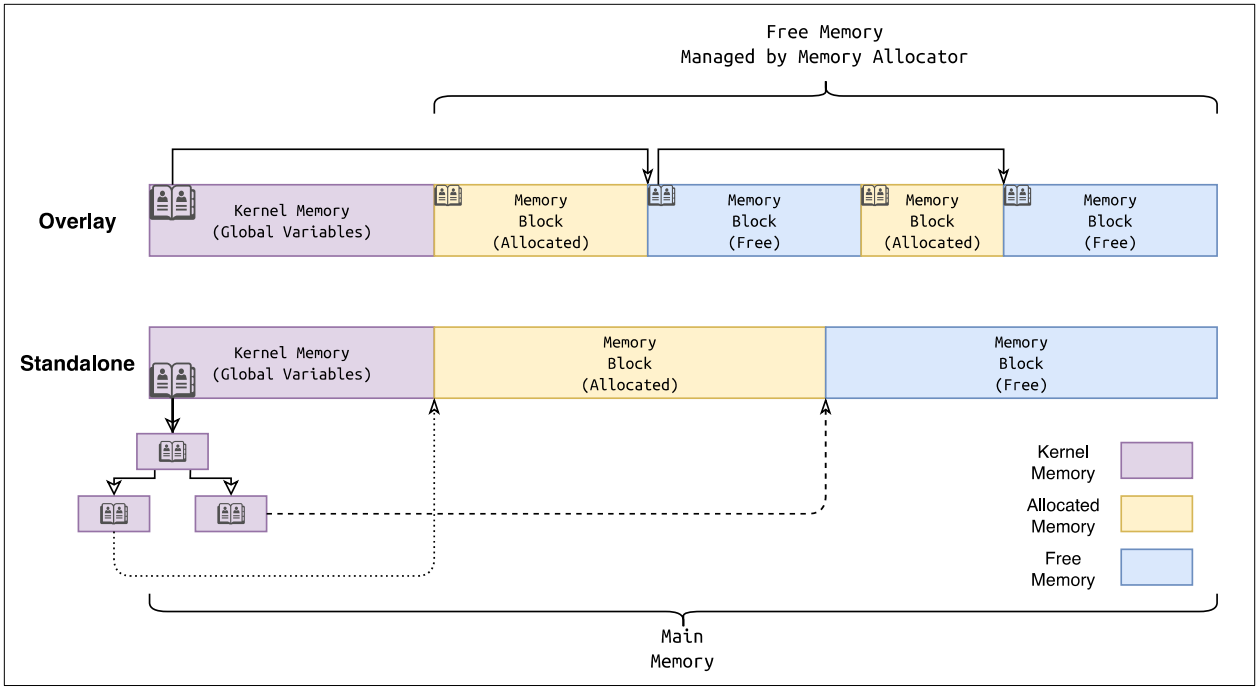


Figure 5.4: *Illustrates the difference between two types of account book.*

5.2.5 Summary and Limitations

Memory allocator components are tightly coupled with each other, so there is not as much flexibility left to developers as they have in assembling schedulers. However, one can still select different fitting policies in Get Free Block, such as first-fit or best-fit, customize control data, and choose data structures that track allocations. More importantly, as discussed in §5.4, we can reuse these components and design new ones to add more functionality to an allocator, such as reallocation, defragmentation and memory protection.

5.3 Assembling a Binary Buddy Allocator

We now demonstrate how we implement each component to build a buddy allocator for memory-constrained devices. Developers can specify the maximum order N and the basic allocation size S. We use only a standalone binary tree represented as a bit array to track the status of memory blocks of every possible size between S and $2^N S$, consuming only 2^{N-2} bytes, so we do not store any control data in memory blocks. A block can be in one of three states:

free if it is available, *split* if it is split into two buddy blocks, or *allocated* if it is in use. Note that a block of order 0 is the smallest unit of allocation thus cannot be split so must be either free or allocated. We encode a free block by setting its bit to 1 and both of its children's bits to 0. We encode an allocated block by setting its bit to 0 and both of its children's bits to 1. We encode a split block by setting its bit to 0 and at least one of its children's bits to 0.

Consider an allocator with a maximum order of 2 and a basic allocation size of 16 bytes. The tree consumes a single byte, and initially, the allocator has a free order 2 block as illustrated in **Figure 5.5.1**. When a program requests 12 bytes, the allocator needs to find an order 0 block to satisfy the request. Thus, it iterates over all order 0 blocks, but none of them are free, because a free block would have its bit set to 1. Subsequently, it iterates over all order 1 blocks, but the result is the same. Finally, the allocator finds that the order 2 block is free, so it splits the block into two order 1 blocks; it changes the block's bit (the first bit in the array) to 0 and its children's bits (the second and the third bits) to 1, indicating that the order 2 block is now split and both order 1 blocks are free as illustrated in **Figure 5.5.2**. The allocator then splits the first order 1 block into two order 0 blocks by changing the second, the fourth, and the fifth bits as illustrated in **Figure 5.5.3**. Finally, it marks one of the order 0 blocks allocated and returns that block to the program as illustrated in **Figure 5.5.4**.

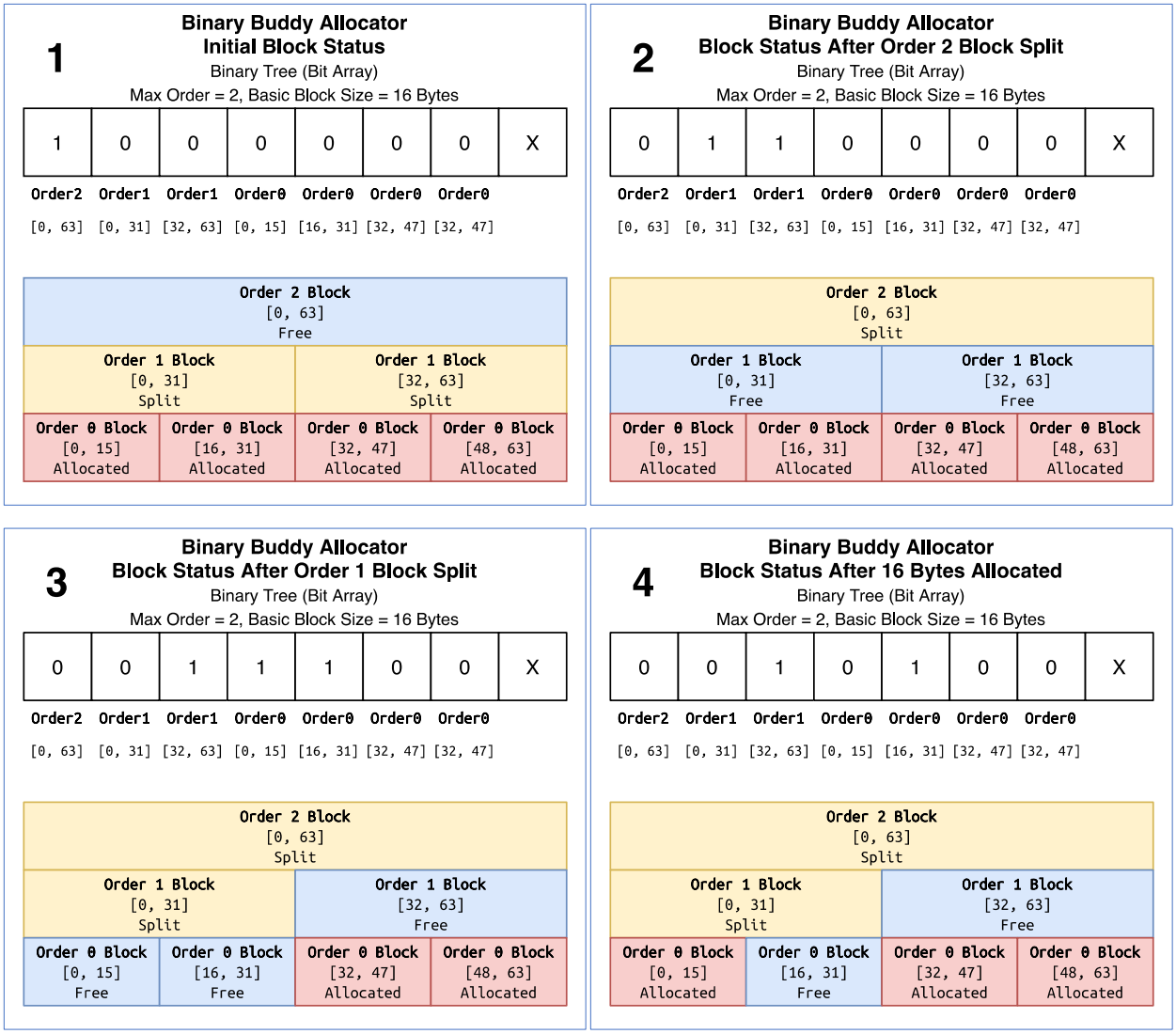


Figure 5.5: *Status of each memory block* throughout allocating 12 bytes.

5.3.1 Get Free Block

We implement Get Free Block in terms of three micro-operations as shown in **Figure 5.6**. Given the allocation size, *Convert Size To Order* calculates the smallest order K of a block large enough to satisfy the request. Subsequently, *Get Free Block Of Order* is a function that searches for a free block of order K and returns its index in the bit array. At the end, *Block Index To Address* translates the index to the memory address. We omit discussion about the first and the last operation, because they essentially exploit properties of a binary tree represented as an array [55].

```
Function GetFreeBlock(Input: Size)
Output: Block Address
    1.  Order = ConvertSizeToOrder(Size)
    2.  Index = GetFreeBlockOfOrder(Order)
    3.  Address = BlockIndexToAddress(Index)
    4.  Returns Address.
```

Figure 5.6: *Implements Get Free Block in three micro-operations.*

To find a free block of order K, the allocator finds a set bit in the range $[2^D - 1, 2^{D+1} - 2]$, where D is the tree depth derived from the order K. If such a block exists, the allocator must check whether the parent block is allocated or not, and if so, it shrinks the range and continues the search. If such a block does not exist, the allocator recursively finds and splits a free block of order $K + 1$ into two order K blocks. If it cannot find a free higher order block after reaching the root block that has the maximum order, the allocator cannot satisfy the request. **Figure 5.**7 presents the algorithm.

```
Function GetFreeBlockOfOrder(Input: K)
Output: Block Index (Bit index in the bit array)
    -   If K > the maximum order, returns -1.
    -   Initially, Range = [2^D − 1, 2^(D+1) − 2] where D = OrderToDepth(K).
    -   While the range is valid:
            ○  Index = BitVector.FindFirstSetBitInRange(Range)
            ○  If Index is invalid:
                    ▪  Index = GetFreeBlockOfOrder(K + 1)
                    ▪  If no higher order block is free, returns -1.
                    ▪  If a higher order block is free, splits it and returns its left child
                       index.
            ○  If Index is valid but its parent block is allocated:
                    ▪  Continue the search with Range.LowerBound = Index + 2 or Index + 1,
                       depending on whether the current “free” block is the left child or
                       not.
            ○  Otherwise, the free block is found and returns the index.
```

Figure 5.7: *Search algorithm to find a free block of order K.*

5.3.2 Mark Block Used

Once it has identified the index of a free block B of order K, the allocator sets B's bit to 0 and both B's children's bits to 1, to indicate that the block is allocated.

5.3.3 Block-to-Pointer

Since the allocator stores nothing in a memory block, it just returns the memory address of the block to the program.

5.3.4 Pointer-to-Block

To find the block referenced by the given pointer, the allocator performs a binary search on the tree, starting from the root block B. At each point in the search, if the bit in the tree represents a block whose start address is equal to the given pointer, we know that the designated block is either B or a left-most descendant of B (i.e., its leftmost child, grandchild, etc.). In the example shown in **Figure 5.5**, the target block can be either the first order 2 block, the first order 1 block, or the first order 0 block. However, if B is allocated, we know that the pointer refers to B. If B is split, we run the same procedure on B's left child to find out which lower order block is allocated, thus referenced by the given pointer. If B is free, then the pointer is invalid. On the other hand, if the pointer is not identical to B's start address, we deduce that the designated block is part of either B's left or right child, so we again run the same procedure on each side until we locate an allocated block. **Figure 5.8** presents the algorithm.

```
Function Pointer2Block(Input: Pointer)
Output: Block Index
    -   Start Address = Start address of the block that has the maximum order
    -   Block Index = 0 // Index of the current block
    -   Start the search from the largest block:
    -   For Order in MaxOrder…0:
            ○  If Pointer == Start Address: // The block itself or one of its left children
                    ▪  If the current block is allocated, returns its index.
                    ▪  If the current block is split, Index = IndexOfLeftChild(Index) and
                       continue.
                    ▪  If the current block is free, the pointer in invalid and returns -1.
            ○  If Pointer < Start Address + BlockSizeOfOrder(Order - 1):
                    ▪  The pointer is part of the left child of the current block.
                    ▪  Index = IndexOfLeftChild(Index)
                    ▪  Continue to search the left child.
            ○  Otherwise:
                    ▪  The pointer is part of the right child of the current block.
                    ▪  Start Address += BlockSizeOfOrder(Order - 1).
                    ▪  Index = IndexOfRightChild(Index)
                    ▪  Continue to search the left child.
```

Figure 5.8: *Search algorithm to find the block referenced by the given pointer.*

5.3.5 Mark Block Free

Once it has the index of the block B referenced by the pointer, the allocator sets B's bit to 1 and both B's children's bits to 0, indicating that it is now free.

5.3.6 Put Free Block

Finally, the allocator checks whether B's buddy block is free. If so, it merges them recursively to reduce external fragmentation. If the allocation pattern is predictable, one can choose to defer merging buddy blocks to avoid unnecessary splitting. Our default implementation merges buddy blocks as soon as possible.

5.3.7 Memory Efficiency

We analyze the memory efficiency of our buddy allocator by comparing it to the one used in Linux to track physical page allocation. As our allocator is designed for memory-constrained devices, we might expect that greater emphasis has been placed on memory efficiency, and indeed, this is what we shall see.

It is common to use an array of free lists to implement a binary buddy allocator. Each list is associated with a bit map, keeping track of the status of each pair of buddy blocks of order K. To allocate a block of order K, the allocator searches in the list of free order K blocks. If no such block exists, the allocator finds and splits a higher order block recursively, after which it places the unused free block on the free list.

Linux uses this approach to allocate physical pages, maintaining an array of struct $free\_area$ as shown in **Figure 5.9**, thus reserving $16N$ bytes for N orders. In contrast, our buddy allocator uses a bit vector to track allocations, consuming only 2^{N-3} bytes. **Figure 5.10** shows the amount of memory, in bytes, reserved by Linux's allocator compared to the amount needed by our allocator.

These results demonstrate both the memory efficiency of Tinkertoy's allocator for the common case where we have relatively few orders in our allocator as well as the advantages of Tinkertoy's design: A developer who finds themselves needing more than ten orders can adopt the Linux strategy by replacing the binary tree with a doubly linked list and extending primitive steps of a free list allocator to update the status of each pair of buddy blocks.

```
typedef struct free_area
{
    struct list_head free_list;
    unsigned long* map;
} free_area_t;

free_area_t free_areas[MAX_ORDER];
```

Figure 5.9: *Linux's free area struct definition.*

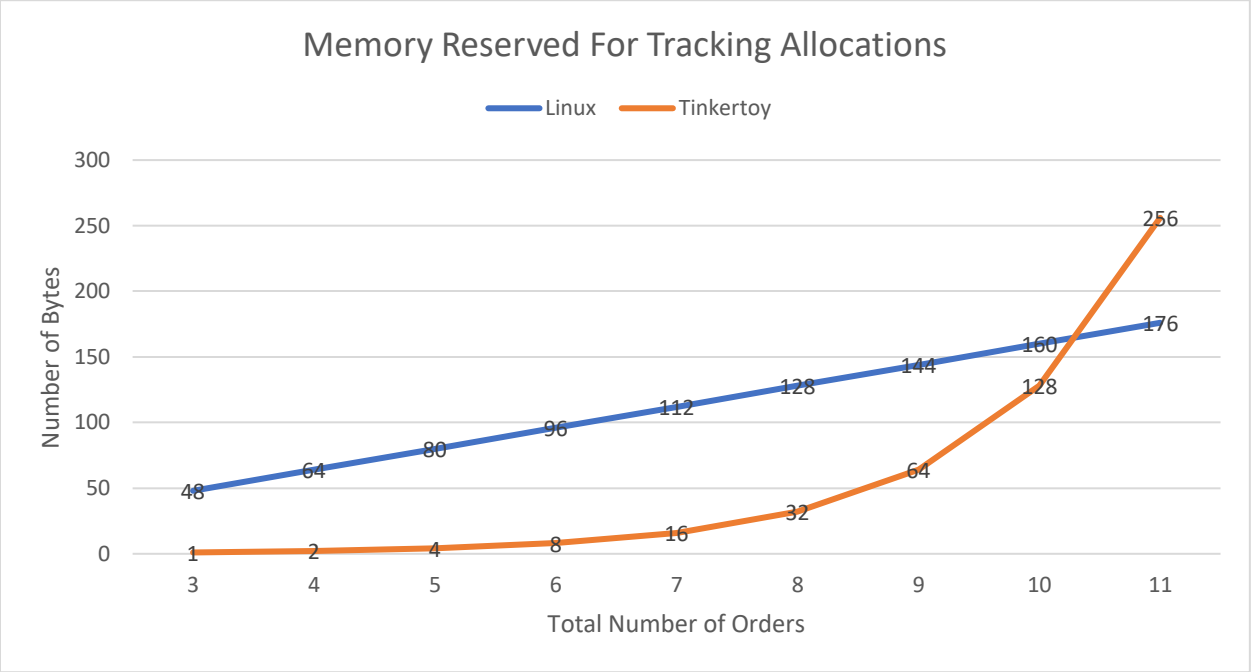


Figure 5.10: *A comparison of the amount of memory reserved for tracking allocations.*

5.4 Extensibility

Tinkertoy's memory allocator interface includes only allocate and free interfaces but can be extended to support other features, such as reallocations, defragmentation, and memory protections.

5.4.1 Allocator Context

An allocator may need additional context for its operation. For example, our binary buddy allocator needs to know the index of a block when it marks the block free or in use. While it is possible to apply the binary search to find the index, it will be more efficient if the allocator can fetch the result from the previous step. Recall that the allocator has already found the index of a free block of order K in *Get Free Block*, so it can save the index for later use. Our current approach is to exploit free memory by writing the index to the first 4 bytes of the block, which implicitly adds a restriction on the minimum size of a block. Although it is rare to set

the minimum block size to less than 4 bytes in the buddy allocator, this approach might not be practical for other allocators that require more space to store control data. Note that these kinds of data are temporary and needed only when the allocator is processing a request, so it is undesirable to reserve “permanent” space inside the allocator beforehand.

Alternately, we can add a context parameter to pass temporary control data between parts of the workflow. The type of the allocation and the release context are determined by the allocator. *Allocate* initializes an instance on the stack and passes it to each primitive step. For example, we can define a structure to store the index of a block. We save the index into the context object in *Get Free Block* and retrieve it in *Mark Block Used*. **Figure 5.11** presents the new implementation of both steps.

```
MarkBlockUsed(Block, Context):
    -   Step 1: Index = Context.Index
    -   Step 2: ClearFreeBit(Index)
    -   Step 3: SetFreeBit(Index.LeftChild)
    -   Step 4: SetFreeBit(Index.RightChild)
    -   Returns.
```

```
GetFreeBlock(Size, Context):
    -   Step 1: Order = ConvertSizeToOrder(Size)
    -   Step 2: Index = GetFreeBlockOfOrder(Order)
    -   Step 3: Address = BlockIndexToAddress(Index)
    -   Step 4: Context.Index = Index
    -   Returns Block Address.
```

Figure 5.11: *Add a new parameter Context to primitive functions.*

5.4.2 Reallocation Support

Reallocation is a convenient feature when one implements a mutable container, such as an array list in Java. The allocator tries to expand the old memory region to satisfy the new size requirement. If there is sufficient free memory after the current allocation, the allocator should mark the extended area in use and update its account book accordingly. Otherwise, it allocates another chunk of memory large enough to satisfy the new request and copies data from the original allocation to the new one. As such, we need two new primitives to get the size of a block and determine if it’s possible to allocate the additional memory contiguously; we can reuse existing components to implement the reallocation function as shown in **Figure 5.12**.

```
-   Function Reallocate(Input: Pointer, Size)
-   Output: Memory Address
        a.  Block = Pointer2Block(Pointer)
        b.  BlockSize = GetBlockSize(Block)
        c.  NeighborBlock = findNeighborBlock(Block, BlockSize)
        d.  If NeighborBlock is NULL:
                i.  NewPointer = Allocate(size); Returns NULL on failure.
               ii.  CopyMemory(from: Pointer, to: NewPointer, of: BlockSize)
              iii.  Free(Pointer) and Returns NewPointer.
        e.  Otherwise:
                i.  MarkBlockUsed(NeighborBlock)
               ii.  MarkBlockUsed(Block) to update necessary control data
              iii.  Returns Pointer.
```

Figure 5.12: *Implement*ation of the *reallocate* interface.

5.4.3 Defragmentation and Memory Compaction

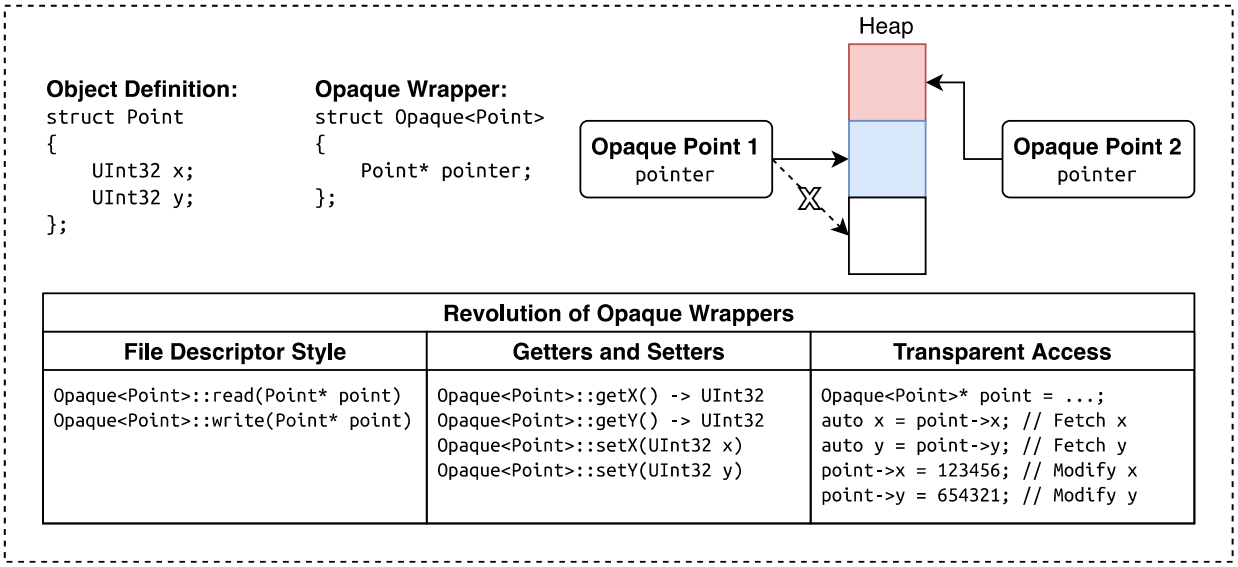
External fragmentation becomes a critical issue as programs release memory. The issue can be reduced in some cases by designing a custom allocator but cannot be prevented in general.

Indirect pointers and opaque handles are common techniques to solve the problem but come with performance penalties. Either the library or the language runtime must provide a double indirection mechanism to access the memory. For example, objects in Java are all reference types, but the Java Virtual Machine (JVM) hides all low-level details, preventing developers' direct access to memory. When accessing an object in Java, the JVM uses the internal pointer to locate the data in the heap. Once the heap usage reaches a certain threshold, the JVM starts to compact the heap by relocating object data and updating internal pointers, and programs are suspended until the compaction has finished [21][24].

Contiki OS provides a similar approach to the JVM but cannot fully hide the low-level details [7]. The system provides a macro to access the underlying pointer to the object, but there is no mechanism to forbid a program from storing it for later use since Contiki is written in C. As a

result, a careless developer can use a stale pointer and unintentionally alter the contents of a memory area that now belongs to another entity.

Tinkertoy can avoid the above issue by providing an opaque wrapper around a relocatable object. Such an object is treated like a file descriptor on Unix-like systems, and developers must provide a buffer to read from or write to it. This approach is simple but does not work well with large objects due to a high overhead of copying memory. However, we could further refine it by providing getters and setters for each public field of a wrapped object, eliminating unnecessary memory copies. We might be able to overload C++'s member access operators to simplify these operations further. **Figure 5.13** illustrates these approaches.



Revolution of Opaque Wrappers		
File Descriptor Style	**Getters and Setters**	**Transparent Access**

`Opaque<Point>::read(Point* point)`
`Opaque<Point>::write(Point* point)`
	`Opaque<Point>::getX() -> UInt32`
	`Opaque<Point>::getY() -> UInt32`
	`Opaque<Point>::setX(UInt32 x)`
	`Opaque<Point>::setY(UInt32 y)`
		`Opaque<Point>* point = ...;`
		`auto x = point->x; // Fetch x`
		`auto y = point->y; // Fetch y`
		`point->x = 123456; // Modify x`
		`point->y = 654321; // Modify y`

Figure 5.13: *An opaque wrapper around a Point type.*

The first instance $point1$ *is relocated to the* 2^{nd} *slot of the heap, and the internal pointer now points to the new location. The table presents pseudocode for each approach to work with an opaque type.*

5.4.4 Security and Protection

5.4.4.1 Enhanced Integrity Checks

Tinkertoy's allocators assume that programs are trusted and behave correctly when releasing memory, so it provides limited protection against invalid pointers. Problems arise when the memory area referenced by a pointer has already been deallocated or is out of the range managed by the allocator. Moreover, a faulty program may unintentionally corrupt the header of an allocated memory block, resulting in unexpected behaviors in the allocator.

We can add more primitives to provide enhanced integrity checks. For example, we can define the predicate *Is Within Range*, so that the allocator can ensure that a pointer falls within a valid range before retrieving the associated memory block. We then define *Is Free Block* to check whether a block is free or not before the allocator puts it back on the free list. Furthermore, we can have a primitive *Is Valid Block* to ensure that programs do not tamper with the control data of a block. At the same time, additional integrity checks introduce increased complexity and memory overhead. An allocator needs to store additional information to provide these guarantees, which might not be practical on low-end devices.

5.4.4.2 Memory Protection

Due to the lack of the Memory Management Unit (MMU), most IoT devices do not support virtual memory, thus missing the standard memory isolation that is available on commodity operating systems. However, hardware manufacturers have introduced Memory Protection Units (MPU), trimmed down versions of an MMU that provide basic memory protection so that a user process cannot access memory that does not belong to it.

An MPU allows the kernel to divide main memory into multiple regions and assign a protection level to each. Each region is described as a <base address, length> pair, which is similar to the segment approach on Intel x86. If a process attempts to access memory illegally, the MPU generates a fault and notifies the processor, so the kernel can take necessary steps to deal with

the faulty process. We have not explored MPUs provided by various platforms yet, so Tinkertoy does not have a hardware abstraction layer for MPUs at this moment; we leave it for future work.

5.5 Summary

We described how we decompose a memory allocator into four components, Memory Block, Static Aligner, Primitives, and Account Book. We customize these components and present a memory-efficient implementation of a binary buddy allocator. While there is less flexibility in the memory allocator than in other components, we showed how we can use existing components and design new ones to add more features to an allocator, such as context-based allocations, reallocation, defragmentation and memory protection.

Chapter 6 Context Switcher

The context switcher is responsible for preserving and restoring machine execution state, allowing multiple tasks to share the processor, and providing the illusion of multitasking. We divide the context switcher into two halves, the *to-kernel* half that defines kernel entry points and switches from a user task to the kernel and the *from-kernel* half that switches from the kernel back to a user task.

We model execution state as an architecture-dependent object, specifying the layout of saved registers on the stack and the calling conventions (§6.2). **Figure 6.1** illustrates how the context switcher interacts with exceptions and the dispatcher. When a user task raises an exception or a device generates a hardware interrupt (1), the processor jumps to the to-kernel half of the context switcher (2). After saving the execution state, the context switcher hands control to the dispatcher (3), which will then process the request (4). Once the request is processed, the dispatcher calls the from-kernel half with the task that should run next (5).

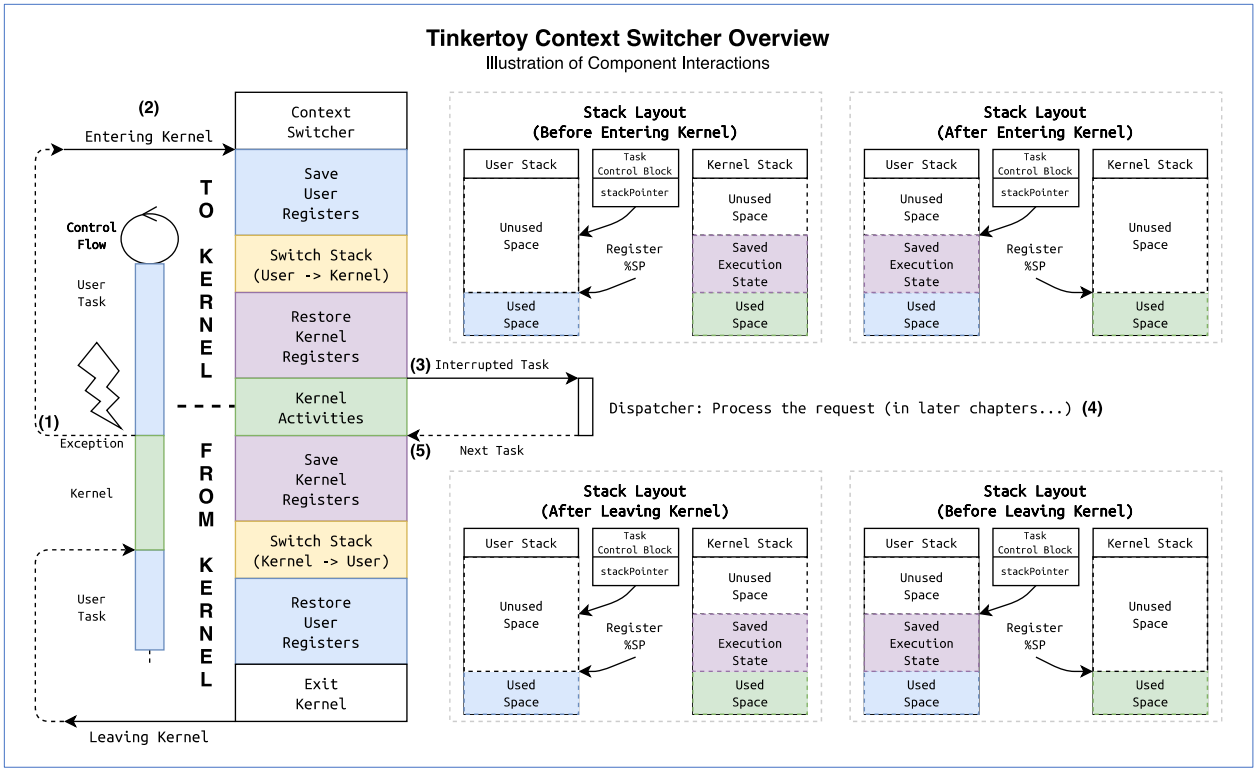


Figure 6.1: *Illustrations of how the context switcher responds to processor exceptions and interrupts and interacts with the dispatcher.*

6.1 Execution Model Independent Design

A common place to store a task's execution state is its stack. The context switcher allocates stack space large enough to hold the execution state and copies the machine's registers into it. It then stores the address of the saved state into the task's control block, enabling restoration of the state later. Tinkertoy's context switcher is independent of whether a kernel is event-based or thread-based, so it relies on two constraints on the task control block, *Task Provides Stack Pointer Read Access* and *Task Provides Stack Pointer Write Access* as shown in **Figure 6.2**, to have full access to a task's instance variable *stack pointer*. As a result, we can implement a generic context switcher as shown in **Figure 6.3**.

```
template <typename Task>
concept TaskProvidesStackPointerReadAccess = requires(Task& task)
{
    /// Task control block provides the getter of the current stack pointer (i.e. the top of the stack)
    { task.getStackPointer() } -> std::same_as<UInt8*>;
};

template <typename Task>
concept TaskProvidesStackPointerWriteAccess = requires(Task& task, UInt8* newStackPointer)
{
    /// Task control block provides the setter of the current stack pointer (i.e. set the new top of the stack)
    { task.setStackPointer(newStackPointer) } -> std::same_as<void>;
};
```

Figure 6.2: *Constraints on the task control block to provide read and/or write access to the stack pointer. A concrete task control block must provide the getter and/or setter to its stack pointer.*

```
template <typename Task>
requires TaskConstraints::TaskProvidesStackPointerReadAccess<Task> &&
         TaskConstraints::TaskProvidesStackPointerWriteAccess<Task>
struct TaskContextSwitcher
{
    static volatile UInt8* gKernStackPointer;
    static volatile UInt8* gUserStackPointer;

    static void switchTask(Task* prev, Task* next)
    {
        // Read the next task's stack pointer
        gUserStackPointer = next->getStackPointer();

        //
        // Assembly Code Block
        // --> From Kernel Half
        // 1. Save kernel registers
        // 2. Save the kernel stack pointer (-> gKernStackPointer)
        // 3. Load the user stack pointer and switch the stack
        // 4. Restore user registers
        // 5. Special instruction to exit the kernel
        // ----------------------------------------
        // <-- To Kernel Half
        // 1. Save user registers
        // 2. Save the user stack pointer (-> gUserStackPointer)
        // 3. Load the kernel stack pointer and switch the stack
        // 4. Restore kernel registers
        //

        // Save the interrupted task's stack pointer
        next->setStackPointer(gUserStackPointer);
    }
};
```

Figure 6.3: *Implement a generic context switcher.*

6.2 System Call and Execution State

Tinkertoy allows developers to define their own system calls, so we provide a broker function, *syscall*, for each architecture to minimize effort. The broker is a variadic function that takes a numeric system call identifier followed by an arbitrary number of arguments and returns a

signed 32-bit integer representing the kernel return value; all system calls are essentially wrappers of this function.

Internally, the broker complies with the calling convention on the target architecture, passing the system call identifier in the first argument register and a *va_list* pointer in the second register, after which it raises an exception to enter the kernel. If the calling convention specifies that arguments are passed on the stack, the broker chooses two caller-saved registers instead, and the compiler guarantees that their original values are preserved and restored.

Since the context switcher saves the execution state on the task's stack, kernel service routines (details in §8.1) that implement system calls must be able to retrieve those arguments from the stack while remaining architecture independent. We achieve this by defining a constraint on the execution state to ensure that it specifies the calling convention of system calls by providing the following functions for kernel service routines in **Figure 6.4**.

```
template <typename Context>
concept ExecutionContextSpecifiesSystemCallConvention = requires(Context& context, Int32 krv)
{
    /// The execution context must provide read access to the register that stores the system call identifier
    { context.getSyscallIdentifier() } -> std::same_as<UInt32>;

    /// The execution context must provide read access to the register that stores the system call argument list
    { context.getSyscallArgumentList() } -> std::same_as<va_list*>;

    /// The execution context must provide write access to the register that stores the kernel return value
    { context.setSyscallKernelReturnValue(krv) } -> std::same_as<void>;
};
```

Figure 6.4: *Definition of the constraint on the execution context to specify the calling convention of system calls.*

As a result, the implementation of the broker function is paired with that of the execution state. For example, on ARM processors, Tinkertoy loads the system call identifier into $\%r0$, so it implements the member function $getSyscallIdentifier$ by returning the register value of $\%r0$ in the saved execution state. Developers do not need to provide their own implementations of both components unless they port Tinkertoy to another architecture.

6.3 Limitations

Tinkertoy provides context switchers for both x86 and ARM Cortex-M, with the assumption that the kernel is non-reentrant and single-threaded. We have not yet developed an elegant way to define assembly code building blocks. However, as the body of the context switcher demonstrates in **Figure 6.1**, it is possible to assemble a kernel-to-kernel context switcher for a single- or multi-threaded reentrant kernel. We might be able to leverage code generation or synthesis [11] techniques to assist developers in building context switchers for such kernels.

Chapter 7 Dispatcher

The dispatcher serves as the front desk of the kernel, bridging the gap between user tasks and devices that request services and kernel service routines (details in §8.1) that provide services. It does not implement any kernel policy and therefore is independent of the execution model. We divide the dispatcher into two halves, the *specific* half that invokes kernel service routines specified by a task or a piece of hardware and the *common* half that invokes routines common to every task.

After receiving a service request from the context switcher, the dispatcher jumps to the *specific* half, invoking the appropriate kernel service routine to process the request. When the service routine returns, the dispatcher knows which task should run next. However, there are special service routines, such as checking pending signals, common to every service being requested, so the dispatcher enters the *common* half to invoke them as well before passing the next task to the context switcher. As a result, developers can specify a list of such routines while assembling a custom dispatcher, which is essentially an infinite loop as shown in **Figure 7.1**.

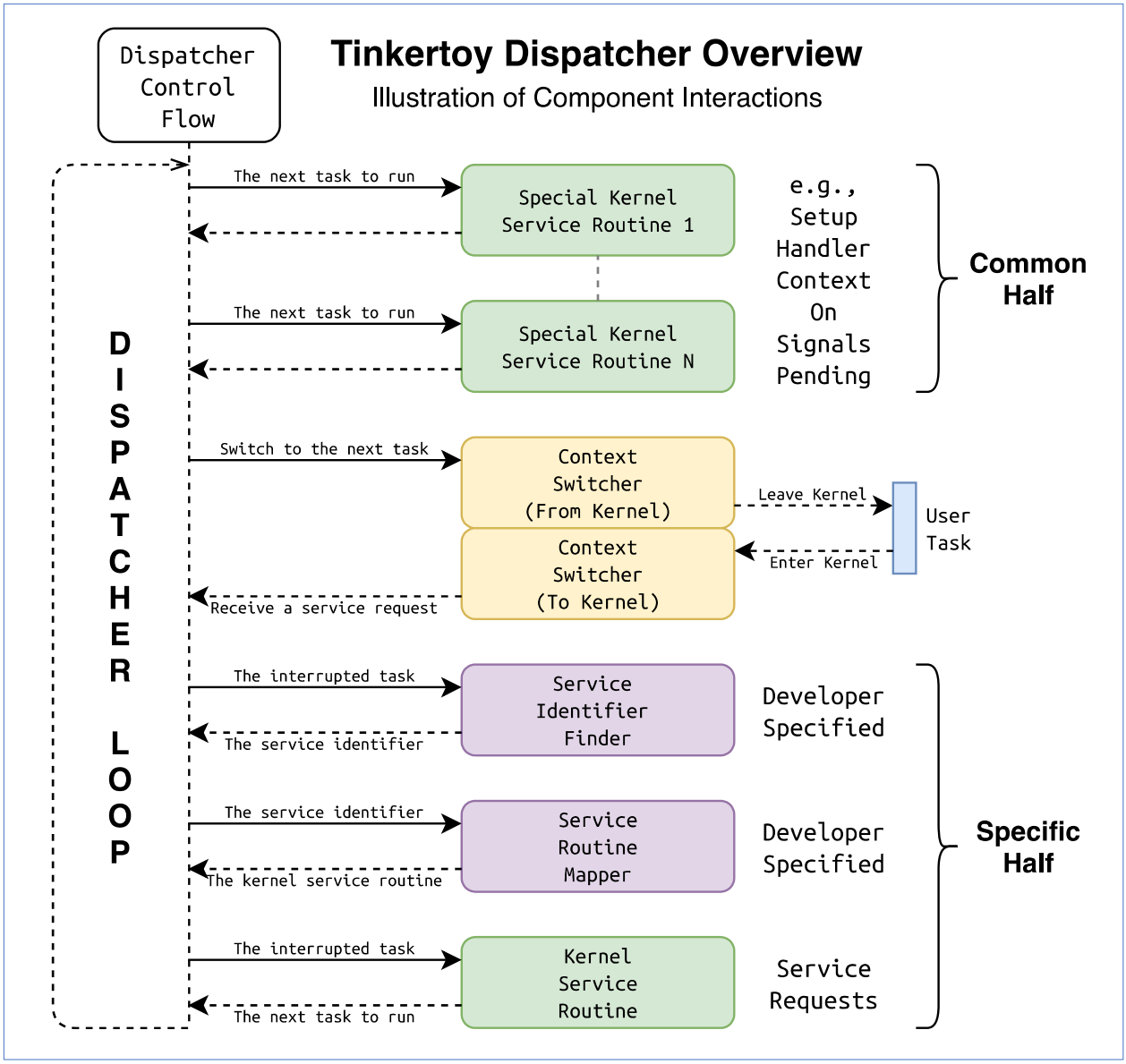


Figure 7.1: *Illustrations of how the dispatcher interacts with the context switcher, companion components and kernel service routines.*

7.1 Companion Components

The dispatcher relies on two developer-specified components to select a kernel service routine. Each of them is encapsulated as a C++ functor, and developers must specify their implementations at compile time. The Service Identifier Finder takes a reference to the interrupted task and returns the service identifier. For example, ARM Cortex-M processors

have a special register called the Interrupt Control and State Register (ICSR) that records the current IRQ number, which can be used as the service identifier. The Service Routine Mapper consumes a service identifier and returns a pointer to the service routine. Since IRQ numbers are fixed on ARM Cortex-M systems, developers can use either a table or a switch statement to implement the mapper.

7.2 Assembling Dispatchers

Tinkertoy provides a builder class for developers to use to assemble a custom dispatcher. Consider an ARM Cortex-M3 system that provides two system calls, *sysSendData* and *sysRecvData*, to send or receive data via a UART port. The UART controller is configured to generate an interrupt when data is available for reading. The system uses the 32-bit unsigned IRQ number as the service identifier and supports signal delivery from the kernel. As such, we implement companion components and then assemble the dispatcher as shown in **Figure** 7.2.

```
struct MyServiceIdentifierFinder
{
    UInt32 operator()(ThreadControlBlock* task)
    {
        // The low 8 bits in the ICSR register stores the current IRQ handler number
        return *reinterpret_cast<volatile UInt32*>(0xE000ED04) & 0xFF;
    }
};

struct MyServiceRoutineMapper
{
    using Routine = ThreadControlBlock* (*)(ThreadControlBlock*);

    Routine operator()(UInt32 identifier)
    {
        switch (identifier)
        {
            case 11:
                return kSyscallServiceRoutine;

            case 22:
                return kUARTRxInterruptHandler;

            default:
                return kUnknownIdentifierHandler;
        }
    }
};

using MyDispatcher = Dispatcher<ThreadControlBlock,
                                UInt32,
                                MyServiceIdentifierFinder,
                                MyServiceRoutineMapper,
                                ContextSwitcher_ARM,
                                SetupSignalHandlerContextServiceRoutine>;
```

***Figure** 7.2: Assemble a custom dispatcher for the system.*

Our custom service identifier finder reads the IRQ number from the ICSR register. If the number is 11, our service routine mapper tells the dispatcher to redirect the request to the kernel service routine that processes system calls. Similarly, if the number is 22, the request will be redirected to the UART RX interrupt handler. Since the kernel can send signals to a thread, we specify the special service routine *Setup Signal Handler Context Service Routine* that builds the execution state for the thread that runs next if a signal is pending.

Chapter 8 Kernel Service Routines

A kernel service routine acts as a servant in the kernel. It is invoked by the dispatcher and either implements a system call or responds to a hardware interrupt. Depending upon the type of request being serviced, a kernel service routine might ask the scheduler to reorder tasks and/or dequeue the next available task. For example, when a task wants to read bytes from a UART port synchronously, the kernel service routine that implements *sysRecvData* checks whether the kernel buffer has enough data to satisfy the request. If so, it copies the data and returns the current task to the dispatcher, so the task will resume. Otherwise, it places the task on the waiting queue of the driver and retrieves the next task from the Task Blocked Event Handler (§4.2.3) of the scheduler.

8.1 Properties

Kernel service routines rely heavily on constraints to provide flexibility. Recall that constraints are a set of C++ concepts that specify requirements on Tinkertoy components, so kernel service routines can provide services only if their requirements are all satisfied. As such, they have the three properties described in the next three subsections.

8.1.1 Non-blocking

Recall that Tinkertoy kernels are currently single threaded, so all kernel service routines must be one-shot and run to completion. However, developers do not need to write routines in a continuation passing style, such as having a callback parameter in the function signature, because a service routine that must be blocked waiting for some resources or conditions in a multithreaded kernel can be naturally expressed as two (or more) non-blocking service routines in a single threaded kernel.

Let us consider the *sysRecvData* example again. The kernel blocks the receiver task if it does not have enough data and runs the next available task. Later, the UART controller generates a hardware interrupt to notify the kernel that data is available for reading, so the UART receive interrupt handler is invoked to service the interrupt, moving data from the hardware buffer to the kernel buffer. When the kernel buffer has enough data to satisfy the receive request, the interrupt handler dequeues the receiver task from the driver's waiting queue and asks the scheduler to unblock the task via the Task Unblocked Event Handler (§4.2.3). Depending upon the actual scheduling policy, the scheduler may preempt the task being interrupted and return the receiver task. Subsequently, the receiver task returns from the system call and proceeds with the data.

8.1.2 Task Control Block-Independent

Recall that Tinkertoy allows developers to assemble a custom task control block, so kernel service routines should be independent of any specific task control block type to ensure maximum flexibility and reusability. However, a service routine may rely on certain task control block components (details in §9.2.2) to provide services. For example, the routine that implements *sysGetTaskID* must be able to retrieve the identifier of the task that issues the request, so it uses the task control block constraint *Task Has Unique Identifier* to guarantee read access to the identifier and *Task Can Invoke System Call* to guarantee write access to the kernel return value. The routine is agnostic about how the identifier is stored in the task control block and which register will hold the kernel return value. However, developers will not be able to compile their kernel, if their custom task control block type does not satisfy all the requirements defined by those two constraints. **Figure 8.1** presents a sample implementation of the service routine.

```
template <typename Task>
requires TaskConstraints::TaskHasUniqueIdentifier<Task> &&
         TaskConstraints::TaskCanInvokeSystemCall<Task>
struct GetTaskIdentifier
{
    Task* operator()(Task* task)
    {
        UInt32 id = task->getUniqueIdentifier();

        task->setSyscallKernelReturnValue(id);

        return task;
    }
};
```

Figure 8.1: *Implementation of the kernel service routine that services the system call sysGetTaskID but is independent of the actual task control block type. The task constraints in the requires clause guarantee that those two member functions exist.*

8.1.3 Component-Independent

We can use a strategy similar to that described in the previous section to make kernel service routines independent of other kernel components, such as the scheduler. For example, the service routine that implements *sysTaskYield* must have access to the scheduler, which provides the Task Yielded Event Handler, while the one implementing *sysRecvData* needs the Task Blocked Event Handler. As a result, we translate these requirements into scheduler constraints and implement these routines as shown in **Figure 8.2**. Furthermore, since developers will provide the scheduler when assembling the kernel, they must implement a helper function *GetScheduler* to give kernel service routines access to the scheduler. Tinkertoy provides a macro *OSDeclareScheduler* to statically allocate a scheduler and implement the helper function simultaneously.

```
template <typename Task, typename Scheduler>
requires SchedulerProvidesTaskYieldingHandler<Scheduler, Task>
struct TaskYield
{
    Task* operator()(Task* task)
    {
        Scheduler& scheduler = GetTaskScheduler<Scheduler>();

        return scheduler.onTaskYielded(task);
    }
};
```

```
template <typename Task, typename Scheduler>
requires SchedulerProvidesTaskBlockedHandler<Scheduler, Task>
struct UARTRecvData
{
    Task* operator()(Task* task)
    {
        Scheduler& scheduler = GetTaskScheduler<Scheduler>();

        return scheduler.onTaskBlocked(task);
    }
};
```

Figure 8.2: *Implements kernel service routines that are independent of the task control block type and the scheduler type.*

Chapter 9 Execution Models

Thread-based and event-driven models are two common execution models, but the debate of which one is better has continued for decades. Advocates of the event-driven model claim that threads are hard to work with because developers must coordinate all accesses to shared data properly, and, failing to do so results in corrupted data or deadlocks [38]. Advocates for threads claim that event-driven programming tends to obfuscate control flow and is difficult to adapt to programs that are not inherently event-driven [53]. Ultimately, these two models are duals of each other, as demonstrated by Lauer and Needham; it is possible to convert a program constructed in one model to the other [23].

In reality, some systems are simply easier to express in one or the other model. To allow applications to use the model best matched to their needs, Tinkertoy provides building blocks for both execution models. While it is possible to build a hybrid execution model, we have not explored this area yet.

9.1 Task Control Block Design

9.1.1 Introduction

Regardless of which execution model developers choose for their applications, the kernel maintains a task control block for each abstract unit of execution, such as a process, a thread, an event handler, or a coroutine. A task control block should contain only the information needed by the kernel to provide services. For example, the kernel needs the priority level of each task to provide priority-based scheduling but does not need the task identifier if there is no system call that references a task by its identifier. To provide maximum flexibility to Tinkertoy developers, we provide building blocks from which one can assemble, initialize and finalize a custom task control block independent of the execution model.

<table>
<tr><th colspan="3">Task Control Block</th></tr>
<tr><th>Components</th><th>Initializer Components</th><th>Satisfied Constraints</th></tr>
<tr><td>Shared Stack Support</td><td>Assign Shared Stack</td><td>Task Has Stack</td></tr>
<tr><td rowspan="2">Dedicated
Non-Recyclable Stack
Support</td><td>Allocate Dedicated
Non-Recyclable Stack</td><td rowspan="2">Task Has Dedicated Stack
Inheriting From
Task Has Stack</td></tr>
<tr><td>Assign Dedicated
Non-Recyclable Stack</td></tr>
<tr><td rowspan="2">Dedicated
Recyclable Stack
Support</td><td>Allocate Dedicated
Recyclable Stack</td><td rowspan="2">Task Has Dedicated
Recyclable Stack
Inheriting From
Task Has Dedicated Stack</td></tr>
<tr><td>Assign Dedicated
Recyclable Stack</td></tr>
<tr><td>System Call Support</td><td>Setup Execution Context</td><td>Task Can Invoke
System Calls</td></tr>
<tr><td>Numeric Identifier Support
(w/ Member Declaration)</td><td rowspan="2">Assign Unique Identifier</td><td rowspan="2">Task Has Unique
Identifier</td></tr>
<tr><td>Numeric Identifier Support
(w/out Member Declaration)</td></tr>
<tr><td>Priority Level Support
(w/ Member Declaration)</td><td rowspan="2">Assign Priority</td><td rowspan="2">Prioritizable
(Scheduler §4.2.2)</td></tr>
<tr><td>Priority Level Support
(w/out Member Declaration)</td></tr>
<tr><td>State Support
(w/ Member Declaration)</td><td rowspan="2">Assign Task State</td><td rowspan="2">Task Has Explicit State</td></tr>
<tr><td>State Support
(w/out Member Declaration)</td></tr>
</table>

Table 9.1: *Components from which it is possible to assemble and initialize a task control block (TCB).*

TCBs that contain a particular component automatically satisfy the associated constraint. For example, a TCB that includes a Dedicated Recyclable Stack Support component will satisfy the "Task Has Dedicated Recyclable Stack" constraint as well as the "Task Has Stack" constraint.

Note that finalizer components are not shown in the table, but every initializer component that allocates resources dynamically (e.g., Allocate Recyclable Stack) has a corresponding finalizer component to release the resource.

9.1.2 Task Control Block Components

9.1.2.1 Components "Meet" Constraints

Tinkertoy provides 10 task control block components, each of which comprises a part of the final block by defining zero or more instance variables and providing functions to manipulate those instance variables. Recall that kernel components are independent of the execution model by means of specifying constraints on the task control block, so developers can use task control block components to satisfy those constraints. For example, the context switcher specifies the *Task Provides Stack Pointer Read* and *Write Access* constraints on tasks being switched (§6.1), which can be satisfied by the Dedicated Non-recyclable Stack component as shown in **Figure 9.1**. **Table 9.1** lists all task control block components and the constraints they satisfy.

```
template <typename Task>
struct DedicatedNonRecyclableStackSupport
{
private:
    UInt8* stackPointer;

public:
    UInt8* getStackPointer()
    {
        return this->stackPointer;
    }

    void setStackPointer(UInt8* newStackPointer)
    {
        this->stackPointer = newStackPointer;
    }
};
```

```
template <typename Task>
struct DedicatedRecyclableStackSupport
     : DedicatedNonRecyclableStackSupport<Task>
{
private:
    UInt8* stack;

public:
    UInt8* getPrivateStack()
    {
        return this->stack;
    }

    void setPrivateStack(UInt8* newStack)
    {
        this->stack = newStack;
    }
};
```

Figure 9.1: *Definition of the stack support component.*

Dedicated recyclable stack support component inherits from the dedicated non-recyclable stack support component and records the start address of the stack. Task control blocks inheriting from either of them get stack support automatically.

9.1.2.2 Stack Components

Tinkertoy is designed to support a variety of different application architecture as efficiently as possible. For example, monitoring systems (§11.3.2, §11.3.3) typically require a single task that loops infinitely collecting and transmitting data, while server systems, such as a CoAP-to-HTTP

proxy (§11.3.4), use an endless supply of short-lived threads. These different application architectures impose different requirements on the kernel. Specifically, the kernel need never reclaim stack space if task lifetimes are essentially forever. Such tasks are best implemented by the non-recyclable stack component that keeps track of the current stack pointer only. In contrast, applications that use short-lived tasks should be implemented by the recyclable stack component as shown in **Figure 9.1**, so the kernel can reclaim the stack space for new tasks.

However, developers do not have to limit themselves to these stack components, and they can design their own. For example, it is possible to eliminate the instance variable *stack* in kernels that allocate a contiguous block of memory to hold both the task control block and the stack. In this case, the task control block can be placed at either the start or the end of the allocated memory region, and the start address of the stack can be calculated by pointer arithmetic.

9.1.2.3 System Call Support Component

Recall that kernel service routines that implement system calls read arguments and set the kernel return value (§8.1.2) by manipulating the saved execution state (§6.2). To read a task's saved state, a service routine must be able to access its stack pointer. Since the execution state component provides an extra layer of indirection to hide architecture-specific details, we provide the system call support component for kernel service routines to access arguments and the return value conveniently. We use static polymorphism to explicitly make the system call support component dependent on one of the above stack components. **Figure 9.2** presents the definition.

9.1.2.4 Other Components

We also provide standard components to declare a task identifier, assign a priority level, adjust the task state, etc. Developers are not required to use these standard components; they can always implement their own to provide more efficient application-specific memory management. For example, if a system supports at most four dynamic priority levels and 16 tasks, one might prefer to use 4-bit task identifiers, 2-bit priority levels, and a 2-bit state

representation, consuming only a single byte instead of requiring three 4-byte integers. **Figure 9.3** presents an example of assembling such a task control block.

```
template <typename Task, typename Context>
requires ExecutionContextSpecifiesSystemCallConvention<Context>
struct SystemCallSupport
{
private:
    Context* getExecutionContext()
    {
        return reinterpret_cast<Context*>(static_cast<Task*>(this)->getStackPointer());
    }

public:
    template <typename Arg>
    Arg getSyscallArgument()
    {
        va_list* ptr = this->getExecutionContext()->getSyscallArgumentList();

        return va_arg(*ptr, Arg);
    }

    void setSyscallKernelReturnValue(int retVal)
    {
        this->getExecutionContext()->setSyscallKernelReturnValue(retVal);
    }
};
```

Figure 9.2: *Definition of the architecture-independent system call support component.*

Developers must choose a stack component to enable this component, otherwise the static polymorphism statement in `getExecutionContext` *will generate a compiler error.*

```
struct Block: SchedulableMarker, Listable<Block>,
              TaskControlBlockComponents::UniqueNumericIdentifierSupportWithoutDeclaration<Block>,
              TaskControlBlockComponents::PriorityLevelSupportWithoutDeclaration<Block>,
              TaskControlBlockComponents::StateSupportWithoutDeclaration<Block>,
              TaskControlBlockComponents::DedicatedNonRecyclableStackSupport<Block>,
              TaskControlBlockComponents::SystemCallSupport<Block, Context>
{
    UInt8 identifier: 4;

    UInt8 priority: 2;

    UInt8 state: 2;
};
```

Figure 9.3: *Assemble a task control block that has a dedicated non-recyclable stack, a unique task identifier, a priority level and a state.*

9.1.3 Task Control Block Initializers and Finalizers

When the kernel creates a new task, it must allocate and initialize a task control block. Similarly, when a task finishes running or is killed by other tasks, the kernel must finalize and release its control block. Both operations are implemented as kernel service routines that rely

on a task controller to allocate and release control blocks, leaving developers responsible for the allocation algorithm; they may, of course, use existing building blocks from the Memory Allocator (§5.2) to assemble a custom control block allocator.

Since developers assemble task control blocks from a collection of components, they should be able to assemble the corresponding initializers and finalizers as well. Tinkertoy provides one or more initializer components for each task control block component and convenient builder classes to assemble a custom initializer and materialize a kernel service routine. The same concept also applies to the finalizer, and as a result, developers are able to initialize or finalize a task control block in one line of code. **Figure 9.4** presents an example of initializing the task control block defined in **Figure 9.3** above.

```
using namespace KernelServiceRoutines::CreateTask::KPI;

using Initializer = TaskInitializerBuilderWithArgs<
            Block,
            AssignUniqueIdentifier<Block>,
            AssignPriority<Block>,
            AllocateDedicatedStack<Block>,
            SetupExecutionContext<Block, ExecutionContextBuilder_ARM>>;

Initializer{}(block, 1, TaskPriority::kHigh, kDefaultTaskStackSize, task_main);
```

Figure 9.4: *Assemble an initializer to initialize a task control block.*

The 1st argument block is the target task control block to be initialized. The 2nd argument, with value 1*, is the task identifier and is passed to the initializer component Assign Unique Identifier. The 3rd argument,* $kHigh$*, is the task priority level and is passed to the Assign Priority component. The 4th argument,* $kDefaultTaskStackSize$*, is used by the Allocate Dedicated Stack component to allocate memory for the task stack. The 5th argument,* $task\_main$*, defines the entry point of the new task and is used by the Setup Execution Context component to initialize the program counter.*

9.2 Thread-based Execution Model

The thread-based execution model allows developers to model their applications as multiple threads, each of which has a dedicated stack and can be blocked waiting for resources or

conditions. Once developers have defined the task control block type and built the initializer and finalizer, they can expose relevant system calls, such as creating a new thread, killing a thread, or yielding the current thread, to user programs in two steps. First, they declare the system call prototypes, assign a unique service identifier to each, and pass the identifier and arguments to the *syscall* function (§6.2). Second, they add an entry to the service routine mapper (§7.1), so that the dispatcher can route the request to the corresponding kernel service routine. **Figure 9.5** shows an example of declaring a system call *sysCreateThread* and routing the request to the task control block initializer built in **Figure 9.4**. In §11.3.4, we will demonstrate assembly of a thread-based kernel to build a CoAP-to-HTTP proxy.

```
int sysCreateThread(int identifier, int priority, size_t stackSize, void(*entryPoint)())
{
    return syscall(SyscallIdentifiers::CreateThread, identifier, priority, stackSize, entryPoint);
}

using SyscallCreateThreadRoutine = CreateThread::ServiceRoutineBuilder<
        ThreadControlBlock,
        ThreadScheduler,
        ThreadController,
        CreateThread::KPI::AssignUniqueIdentifier<Block>,
        CreateThread::KPI::AssignPriority<Block>,
        CreateThread::KPI::AllocateDedicatedStack<Block>,
        CreateThread::KPI::SetupExecutionContext<Block, ExecutionContextBuilder_ARM>>;

OSDefineAndRouteKernelRoutine(kSyscallCreateThread, ThreadControlBlock, SyscallCreateThreadRoutine);

struct KernelServiceRoutineMapper
{
    using Routine = Block* (*)(Block*);

    Routine operator()(int identifier)
    {
        switch (identifier)
        {
            case SyscallIdentifiers::CreateThread:
                return kSyscallCreateThread;

            /* More entries */
        }
    }
};
```

Figure 9.5: *Add a new system call to create a thread at runtime.*

The Service Routine Builder materializes a kernel service routine for the new system call. Internally, it reads system call arguments and forwards them to initializer components. A new entry is added to the mapper to redirect the request to the routine. The kernel service routine is encapsulated as a functor that overloads the operator (), so the macro is needed to convert the functor to a function pointer.

Tinkertoy does not yet provide IPC primitives, such as mutex, semaphores and message queues, so there are limitations in the current thread-based model (e.g., threads cannot communicate).

However, our builder classes are generic enough to work with any number and type of components, so we believe that it is possible to implement such functionality.

9.3 Event-driven Execution Model

The event-driven execution model allows developers to define custom events and model their applications as individual event handlers that run on a single shared stack. Tinkertoy provides task control components specific to this model, such as the event handler component that stores the handler entry point, and standard kernel service routines as well as system calls to (un)register events and their handlers and deliver events. We will assemble event-driven kernels for monitor applications in later chapter (§11.3.2, 11.3.3) and leave IPC primitives for future work.

9.4 Summary

We have now introduced all Tinkertoy's kernel components and demonstrated how they interact to provide services to user applications. We introduced constraints on the task control block to ensure that kernel components are independent of the execution model but still have access to related control data to fulfill the user task requests. We showed how we design task control block components from which developers can assemble custom task control blocks, defining instance variables and public functions that are precisely those needed by services available on their system. Moreover, we provide builder classes for developers to materialize kernel service routines that manipulate their custom task control blocks. We have not yet explored IPC primitives for both execution models, but we believe that Tinkertoy can provide more building blocks for such functionality in future.

Chapter 10 Related Work

Tinkertoy's building blocks make it possible to assemble a custom kernel that is inherently modular. Before evaluating the performance of the kernels we've assembled, we discuss how this work draws on work in library operating systems, Exokernel [17], and The Flux OSKit [18]. We then introduce four popular IoT operating systems and highlight their different approaches to modularity.

10.1 Library Operating Systems

General-purpose operating systems must multiplex hardware resources among user applications, so kernel designers select a policy for sharing each resource on behalf of users and expose a collection of abstractions to them. When applications impose wildly different demands, arbitrating among them becomes a key challenge. Library operating systems address the concern by allowing developers to implement application-specific operating system abstractions in user-space.

10.1.1 Exokernel

Exokernels focus on presenting the right abstractions to directly expose hardware, allowing applications to manage resources efficiently at user level while still protecting them from each other. However, the kernel still defines a set of core functionality and policies from which a library system can use or extend to service a particular application. Tinkertoy allows developers to create or choose abstractions as well by implementing kernel service routines and exposing related system calls, but it is a set of components that are independent of the kernel architecture and execution model, so developers are free to replace, add, or remove any of them. Application-level resource management is not our goal, because we believe that developers can use hardware resources efficiently by carefully assembling a custom operating system from building blocks.

10.1.2 The Flux OSKit

The Flux OSKit provides a set of standard kernel components, such as a boot sequence and memory manager and allows developers to reuse components from different existing operating systems to construct a new system. Its encapsulation technique makes it easier to import code, such as device drivers, network stacks, and file systems, from other systems and its object-oriented design allows one to override individual functions, thus reducing dependencies between components and subsequently making it possible to mix and match different components to produce a new system. The Flux OSKit provides two modes of system construction: separable components and individual functions. Tinkertoy shares the design concept of modularity and separability with the Flux OSKit but does not import components from other operating systems, primarily because such components are frequently too large and expensive for IoT systems, but also because even the overhead of wrapping these components in standard interfaces might be too burdensome for our target environment. Instead, we carefully design each component from scratch to provide a third approach to system construction: composable code, so developers can customize kernel data structures (e.g., task control block) and related functions (e.g., initializers, finalizers, and service routines) by constructing entirely new systems by assembling them out of a collection of modular, flexible, and interacting components.

10.2 Other IoT Operating Systems

10.2.1 TinyOS

TinyOS [29] is a component-based operating system written in NesC (a dialect of C) [20] and provides an event-driven execution model. Software services and hardware resources are encapsulated as components that define interfaces to respond to events of interest and post events to request services from other components. Developers can write their applications in terms of a set of components, thus excluding unused services from the system. Tasks share a

single stack and cannot preempt each other, but in later version, TinyOS provided a cooperative thread-based execution model [27], allowing tasks to relinquish processors voluntarily. TinyOS does not support user space, so applications are linked directly with the operating system. It does not support dynamic memory allocation, so developers must reserve memory at compile time. TinyOS achieves modularity at the component level, so developers cannot customize a component easily without tweaking the code, but the way developers assemble application is similar to Tinkertoy.

10.2.2 Contiki & Contiki-NG

Contiki [14] was originally targeted at wireless sensor networks, and its successor Contiki-NG [8] provides a rich collection of low-power communication protocols for IoT devices. Contiki-NG is written in C and provides an event-driven execution model. Applications are encapsulated as individual processes that are built upon the stack-less protothread library [15]. Processes always run cooperatively and can only be preempted by hardware interrupts. While the original Contiki provided limited support for preemptive multithreading, Contiki-NG removed this feature, making the system strictly cooperative. As a result, the scheduler selects a process to run when there is a pending event and no other processes are running, and processes will run to completion.

Contiki-NG supports both static and dynamic memory allocations. The system provides two types of dynamic memory allocators, Memory Block Allocator ($MEMB$) that allocates typed memory from a static pool and Heap Allocator ($HeapMem$) that allocates untyped memory from a free list. However, there is no distinction between kernel space and user space in Contiki-NG, so system calls and memory protection are not available. Finally, Contiki-NG is modular in a coarse-grained manner. Developers can edit a Makefile to include modules, such as CoAP and IPv6 Multicast, but cannot customize existing the kernel implementation without modifying the source code.

10.2.3 FreeRTOS

FreeRTOS [19] is a real-time thread-based operating system and is written in C. Its kernel is designed to be small and simple and consists of only three source files: list, queue and task. All other kernel functionality, such as semaphores, software timers and event groups, are built on top of the above three constructs. The kernel itself neither provides networking support nor device drivers, but these features are implemented as third-party modules. FreeRTOS can be configured to enable the user space, but it does not provide any system calls, so developers are responsible for implementing support for crossing the supervisor/user boundary and routing user requests to kernel APIs.

Applications are encapsulated as tasks that have a private stack and can be scheduled either preemptively or cooperatively. While developers can write applications as coroutines that share a single stack, all related kernel APIs are deprecated. FreeRTOS supports dynamic memory allocations and provides five different allocators, from a simple one that never releases memory to a more fully featured one that coalesces adjacent free blocks to reduce fragmentation. Furthermore, FreeRTOS heavily relies on C macros to make the kernel modular. Developers provide a header file where they define macros to enable or disable kernel APIs. In general, FreeRTOS provides a straightforward way to customize the system at the API level but not at the code level as Tinkertoy does.

10.2.4 Zephyr

Zephyr [49] is a thread-based operating system and is written in C. Its kernel shares lots of design concepts with the Linux kernel. For example, Zephyr provides multiple scheduling algorithms, such as cooperative, preemptive, time slicing and earliest deadline first, to suit application needs. It supports meta IRQ scheduling to defer executing a function, which behaves similarly to Linux's tasklet and softirq. It uses device trees to describe hardware and provides a consistent driver model to support heterogeneous devices. It provides the standard virtual filesystem interface (VFS) to support multiple filesystems, such as FAT and LittleFS.

Zephyr is modular and provides a Kconfig interface, as Linux does, to customize the system, so developers can easily specify the device drivers they need, the scheduling algorithm, the data structure that implements the queue, etc. In general, Zephyr allows one to customize the system in a finer-grained manner than FreeRTOS, and similar to Tinkertoy, some of the customizations are close to code level.

10.3 Epilogue

We began this chapter by introducing prior work that made it possible to build application-specific operating systems. While Tinkertoy shares that goal with these systems, the scale and design target for the prior work is considerably different from that of Tinkertoy.

We then introduced four popular IoT operating systems whose design targets are more closely aligned with those of Tinkertoy. All of them are modular: Contiki, FreeRTOS, and Zephyr provide a prebuilt kernel, from which developers remove unneeded functionality, while TinyOS provides components, from which developers assemble their applications, but it has limited or no support for functionality, such as preemptive multithreading, dynamic memory allocation, and user space, to meet new demands.

Tinkertoy is not a kernel but a set of modules, from which developers assemble a custom kernel. Developers can customize modules at code level easily and design new ones specific to their applications. We now move onto assembling custom kernels for different applications and demonstrate how Tinkertoy-based kernels behave compared to other IoT operating systems.

Chapter 11 Evaluation

Our previous chapters show that it is possible to design and implement a set of modular components that allows developers to customize kernel functionality by easily constructing a kernel via assembling flexible components. We now evaluate Tinkertoy to answer the three questions we posed in §1.5.

1. How much effort does it take to assemble a custom system for applications running on an IoT device?
2. How does the memory footprint of such a system compare to other IoT operating systems?
3. How does the runtime performance of such a system compare to other IoT operating systems?

We start by describing use cases for which we assemble custom kernels to demonstrate Tinkertoy's configurability and simplicity. We then analyze the memory footprint and the runtime performance of the assembled kernels and compare the results to other competing systems. At the end, we discuss Tinkertoy's potential drawbacks revealed from our experiments and conclude the chapter.

11.1 Use Cases

We introduce the following two scenarios that we will emulate under QEMU and discuss how we emulate unsupported hardware in detail in §11.2.1.

11.1.1 Automatic Watering System

Our first use case is an automatic watering system that takes care of potted plants while people are away from their home. The system monitors the soil moisture level and waters the plant

when the soil's moisture content drops below a threshold value. It is composed of two devices, a monitor and an actuator, each of which requires a custom kernel. We connect sensors to the GPIO pins on the device by a wire.

The monitor device periodically retrieves the moisture level from a moisture sensor in a plant pot. If the level falls below a predefined threshold, it signals the actuator device to start dripping water. Once the level reaches a predefined upper threshold, it notifies the actuator device to stop.

On the other side, the actuator device controls the gate of a water bottle. It opens the gate on receiving a dry alert message from the monitor device and closes the gate on a wet alert message. In addition, a gravity sensor is placed in the water bottle, so the actuator device watches the water level and notifies the user when the bottle runs out of water, by sending a message to a server (introduced in §11.1.2).

Both devices work with multiple potted plants. Users can connect additional moisture sensors to the monitor device and install electronic gates on the water tube to control which plant the actuator device should water. For simplicity, in our deployment, we assume that there is only one potted plant and one water gate. **Figure 11.1** illustrates the setup.

11.1.2 CoAP/HTTP Gateway

Our second use case is a gateway device that bridges the communication between IoT devices and general-purpose computers seamlessly. IoT devices have limited memory, so they exchange messages via specialized protocols, such as CoAP [43] and MQTT [36]. As a result, a local area network is logically divided into two computing zones, the general-purpose zone where devices use standard protocols and the IoT zone where devices use specialized protocols. The gateway device bridges between two zones by acting as a transparent proxy that translates CoAP messages to HTTP messages and vice versa.

In our scenario, the monitor device periodically needs to report the moisture level to a Linux machine that runs an HTTP server, so users can learn how the soil's moisture content in each plant changes throughout the day. The gateway device translates CoAP PUT messages that carry the moisture level into HTTP PUT messages. Similarly, when the bottle runs out of water, the actuator device sends out a CoAP POST message, which is then translated into an HTTP POST message. **Figure 11.1** shows how the gateway device joins the home network.

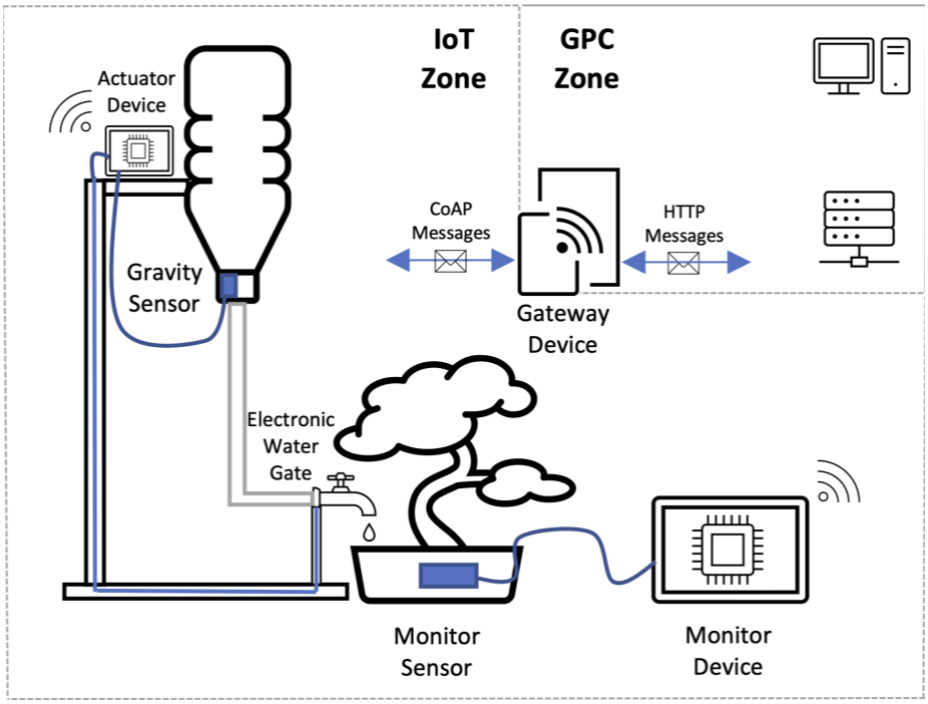


Figure 11.1: *Deployment of the automatic watering system in a home network.*

11.2 Experimental Setup

11.2.1 Hardware Emulation

We have three tiny devices in total, the monitor device, the actuator device and the gateway device. Each of them runs its own custom kernel on a Stellaris LM3S811 board emulated by QEMU (ARM v5.2.0). The host machine that runs QEMU has a 4 GHz Intel Core i7 processor, while we emulate each device as a 50 MHz ARM Cortex-M3 processor, 64 KB Flash and 8 KB SRAM.

Tinkertoy does not yet support networking, so these devices use UART ports to communicate with each other. QEMU allows us to redirect a serial port to a socket, so we run a native controller process on the host machine, acting like Wireshark to capture and display all messages within the network.

However, QEMU does not emulate sensors, so when applications request sensor readings, our kernel reports an artificial value that can be modified by the controller process remotely via special messages. In summary, there are five processes running on the host machine and exchanging messages via TCP sockets. **Figure 11.2** depicts the emulated environment. **Table 11.1** summarizes processes and their roles.

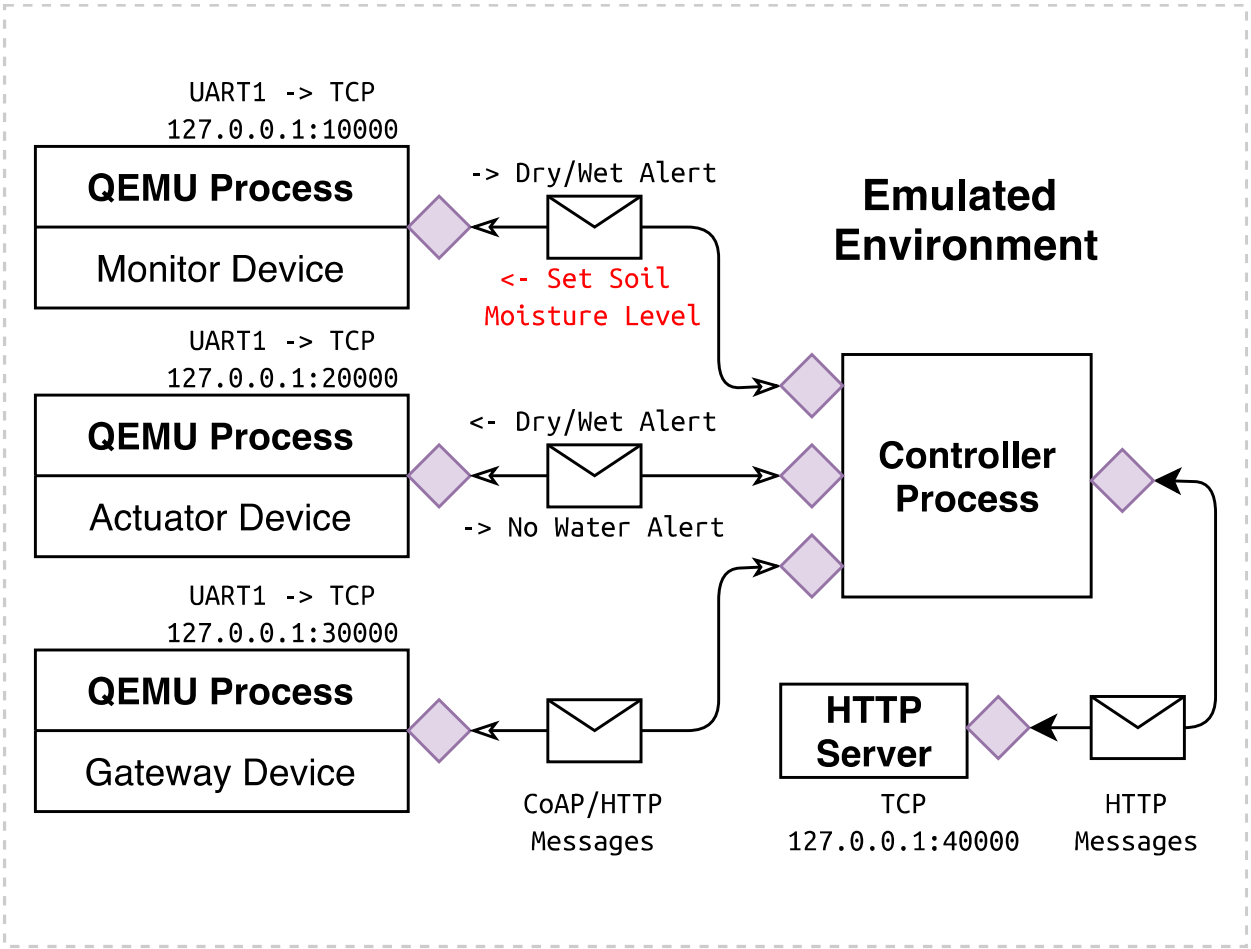


Figure 11.2: *Deployment of the automatic watering system in an emulated environment.*

Process Name	Process Role
Monitor	Emulate the monitor device (QEMU)
Actuator	Emulate the actuator device (QEMU)
Gateway	Emulate the gateway device (QEMU)
Controller	Capture and visualize all messages within emulated network
HTTP Server	Receive moisture levels from the monitor device Receive no water alerts from the actuator device

Table 11.1: *Emulated processes and their roles.*

11.2.2 Compiler Configurations

We use GCC 10.2.1 (GNU ARM Embedded Toolchain 10-2020-q4-major) to compile kernels with code optimization level 2 (-O2), code size optimization (-Os) and link phase garbage collection (--gc-sections) enabled. C++ exceptions, runtime support (RTTI) and C standard library are disabled. As a result, we guarantee that all unreferenced functions and variables are excluded from the final kernel executable.

11.3 Assembling Kernels

11.3.1 Overview

Tinkertoy is not a prebuilt kernel; it is a set of modules that a developer assembles into a kernel. We evaluate the effort in terms of the number of source lines of code (SLOC) needed to produce a kernel for each device. Blank lines and comments are not included, while curly braces are always placed in a new line and counted. Note that application code is not counted, even though the kernel and the application are compiled into one single binary image.

11.3.2 Moisture Kernel

11.3.2.1 System Requirements

This application, which simply acts as an interface between the moisture sensor and the water bottle, can be modeled as a state machine, so an event-driven kernel is adequate. We define the following three events, *Periodic Timer Event*, *Dry Soil Event* and *Wet Soil Event*, and their corresponding handlers, *Sensor Reader*, *Dry Soil Handler* and *Wet Soil Handler*. The system should deliver a *Periodic Timer* event on a regular basis, so that a *Sensor Reader* task fetches and examines the moisture level. The reader task has two versions, *Read Sensor Dry* and *Read Sensor Wet*. The former one checks whether the moisture level falls below the lower bound and if so, delivers a dry soil event, while the latter one checks against the upper bound and delivers a wet soil event. The dry soil handler sends a *Dry Soil Alert* message to the actuator device and switches the periodic timer event handler to *Read Sensor Wet*, while the wet soil handler does the opposite. Furthermore, soil event handlers are more critical than the reader task, so the kernel should provide preemption support to run the handler immediately when an event is delivered.

11.3.2.2 System Deployment

The above requirements suggest that developers need a preemptive scheduler that responds to task creation and termination events, because all tasks are one-shot and therefore guaranteed to complete. The *Simple Event Driven Execution* model is required to support custom events and handlers. Developers also need the UART driver, SysTick Timer and Fake Sensor driver support. As such, they are responsible for the following items.

1. Define the event control block that provides system call management.
2. Define the event controller that manages all registered events and their handlers.
3. Assign a unique identifier to each system call.
4. Define the kernel service routine mapper for the dispatcher.
5. Provide a custom timer interrupt handler to deliver *Periodic Timer* events.

6. Provide a startup routine to declare and initialize all chosen components.

Since all event handlers share the same stack, an event control block needs to record only the handler address as shown in **Figure 11.3**. *Schedulable Marker* (§4.2.1) allows events to be scheduled by a scheduler, while *Listable* provides doubly linked list support needed by the FIFO scheduling policy component (§4.2.2). The *Shared Stack Support* (§9.1.2.2) component provides getter and setter functions that access the stack pointer of an event handler, while the *System Call Support* (§9.1.2.3) component allows other kernel service routines to retrieve system call arguments and set the kernel return value. As a result, an event control block is 12 bytes long.

```
struct EventControlBlock: SchedulableMarker, Listable<EventControlBlock>,
                          TaskControlBlockComponents::SharedStackSupport<EventControlBlock>,
                          TaskControlBlockComponents::SystemCallSupport<EventControlBlock>
{
    EventHandler handler;
};
```

Figure 11.3: *Definition of the event control block for the moisture kernel.*

Event Handler is a function pointer type. Listable defines two pointers, prev and next, so the size of an event control block is 4 + 8 = 12 bytes.

The event controller maintains a table of four event control blocks. The first entry is reserved for the idle event handler, while the rest of them are used by the application. The controller also provides two functions to register the handler for an event and retrieve the handler associated with an event, as specified by the *Simple Event Driven* execution model. **Figure 11.4** presents the controller definition.

```
struct EventController
{
    EventControlBlock table[4];

    void registerEvent(Event event, EventHandler handler)
    {
        this->table[event].handler = handler;
    }

    EventControlBlock* getRegisteredEvent(Event event)
    {
        return this->table + event;
    }
};
```

Figure 11.4: *Definition of the event controller for the moisture kernel.*

When a user task invokes a system call, the dispatcher relies on the mapper to redirect the request to the corresponding kernel service routine (§8.1). The moisture kernel provides five system calls as summarized in **Table 11.2**, so a simple switch statement is sufficient to implement the mapper.

System Call Definition	System Call Provider
sysSetEventHandler(int event, void (*handler)())	Simple Event Driven Execution Model
sysSendEvent(Event event)	Simple Event Driven Execution Model
sysEventHandlerReturn(void* oldStack)	Simple Event Driven Execution Model
sysReadSensor(int sensor)	Fake Sensor Driver
sysSendData(const void* bytes, size_t count)	UART Driver

Table 11.2: *A list of system calls provided by the moisture kernel.*

The timer interrupt handler keeps track of the amount of time elapsed in ticks. Once the number of ticks reaches the threshold, for example 5000 ticks for five seconds, it delivers the *Periodic Timer* event by means of notifying the scheduler that a new task is created. **Figure 11.5** presents the implementation.

```
static EventControlBlock* kSysTickInterruptHandler(EventControlBlock* current)
{
    /// Every 5 seconds
    static UInt32 timeout = 5000;

    timeout -= 1;

    if (timeout == 0)
    {
        timeout = 5000;

        auto handler = GetTaskController<EventController>().getRegisteredEvent(kPeriodicTimerEvent);

        current = GetTaskScheduler<EventScheduler>().onTaskCreated(current, handler));
    }

    return current;
}
```

Figure 11.5: *Timer interrupt handler that delivers an event every 5 seconds.*

Tinkertoy provides a simple bootloader that sets up the kernel stack, initializes all global variables and runs the kernel startup routine in ARM Handler mode. As a result, the scheduler and the event controller are initialized by the bootloader automatically, so we need only set up the interrupt vector table, initialize the SysTick timer and UART ports, register event handlers and start the dispatcher in the startup routine. **Figure 11.6** presents the startup routine, and **in total we need 86 lines of code to assemble the moisture kernel**. **Table 11.3** shows the breakdown by component.

```
OSDeclareTaskScheduler(EventScheduler, scheduler);
OSDeclareTaskController(EventController, controller);

extern "C" void kmain(void)
{
    // Configure the kernel interrupt table
    initInterruptTable();

    // Configure the timer
    initTimers();

    // Configure UART1 and RX interrupts
    initUART1();

    // Setup events and handlers
    initEvents();

    // Dispatcher
    // We assume that the idle handler was running before we first enter the dispatcher
    auto idleHandler = controller.getRegisteredEvent(kIdleEvent);
    EventDispatcher dispatcher(idleHandler, idleHandler);
    dispatcher.dispatch();
}
```

Figure 11.6: *Moisture kernel startup routine that initializes all selected kernel components.*

The `OSDeclareTaskScheduler` macro (§8.1.3) declares a global variable of the given scheduler type and generates the glue function, `GetScheduler`, for other kernel service routines to access the scheduler. The task controller macro essentially does the same thing.

Moisture Kernel Components	Deployment Methods	Source Lines of Code
Scheduler	Assemble from building blocks	15
Task Control Block	Write from scratch	4
Task Controller	Write from scratch	13
Kernel Service Routine Mapper	Write from scratch	21
Dispatcher	Assemble from building blocks	2
Timer Interrupt Handler	Write from scratch	12
Startup Routine	Write from scratch	19
Total		86

Table 11.3: *Moisture kernel per-component line counts.*

11.3.3 Actuator Kernel

11.3.3.1 System Requirements

The actuator device behaves similarly to the monitor device, and there are also three events on the system, *Start Watering Event*, *Stop Watering Event* and *No Water Event*. The UART RX interrupt handler receives messages from the monitor device and delivers the first two types of events. Message handlers then open or close the water gate to start or stop watering the plant. If the water bottle is empty, the handler delivers a *No Water* event, so that the *No Water* handler will send an alert message to the gateway device. However, developers must choose a cooperative scheduler to ensure that message handlers access the water gate in order. In other words, the *Stop Watering* handler cannot preempt the *Start Watering* handler that is running, in case the moisture level changes before the actuator device opens the gate.

11.3.3.2 Differences in Deployment

As such, the actuator kernel shares about 80% of its code of the moisture kernel. Developers need to implement the UART RX interrupt handler instead of the SysTick timer interrupt

handler. The actuator kernel provides two more system calls to control the water gate, but the one that changes the event handler at runtime is no longer needed. As a result, the actuator kernel requires 111 lines of code, 70 of which are shared with the moisture kernel. We remove 17 lines of code (related to the timer interrupt) and add 42 lines of code (related to the UART interrupt handler) to finish assembling the kernel. **Table 11.4** summarizes the per-component line counts.

Actuator Kernel Components	Deployment Methods	Source Lines of Code
Scheduler	Shared with the moisture kernel	15
Task Control Block	Shared with the moisture kernel	4
Task Controller	Shared with the moisture kernel	13
Kernel Service Routine Mapper	Shared base code	9
	Shared entries = 4	8
	Added entries = 3	+ 6
	Removed entries = 2	- 4
	Total	23
Dispatcher	Shared with the moisture kernel	2
Timer Interrupt Handler	Removed	- 12
UART RX Interrupt Handler	Write from scratch	+ 36
Startup Routine	Shared	14
	Removed (Init Timer)	- 1
	Modified (Init Events)	- 4 + 4
	Total	18
Shared (Including modified lines)		69
Added		42
Removed		17
Total		111

Table 11.4: *Actuator kernel per-component line counts.*

11.3.4 Gateway Kernel

11.3.4.1 System Requirements

The gateway device should be able to translate multiple CoAP/HTTP messages concurrently, so a thread-based execution model is necessary. The translator thread should finish its work without being interrupted, so that it can be ready to service the next message immediately. To reduce the unnecessary complexity of detecting the size of an incoming message due to a lack of network stack, we assume that each CoAP PUT request sent by the monitor device is always 32 bytes long.

11.3.4.2 System Deployment

The above requirements suggest that we need a cooperative scheduler that can handle Task Creation, Task Blocked and Task Unblocked events. Task Termination support is not needed, because a translator thread works in an endless loop and remains blocked until it receives a message. A memory allocator is convenient to allocate private stacks to each thread. Additionally, we need the UART driver module to send and receive messages. As a result, we must provide the following items to assemble the gateway kernel.

1. Define the thread control block that provides system call management.
2. Define the thread controller that allocates and releases a free thread control block.
3. Assign a unique identifier to each system call.
4. Define the kernel service routine mapper for the dispatcher.
5. Provide a custom UART RX interrupt handler to service the *sysRecvData* system call.
6. Provide a startup routine to declare and initialize all chosen components.

Since a translator thread never terminates, its dedicated stack will never be released, so the thread control block needs to keep track of its current stack pointer only. It also maintains a data structure to record the status of the active request and provides four functions to assist the UART RX interrupt handler in servicing the request. **Figure 11.7** presents the definition.

The *Dedicated Nonrecyclable Stack Support* (§9.1.2.2) component defines an instance variable, *stackPointer*, to store the current stack pointer and provide getter and setter functions to access the thread stack. *IORequest* maintains a reference to the user buffer and stores the number of bytes requested by the thread and being processed by the UART driver. The rest of the components are the same as the event control block. As such, a thread control block is 24 bytes long.

```
struct ThreadControlBlock: SchedulableMarker, Listable<ThreadControlBlock>,
                           TaskControlBlockComponents::DedicatedNonrecyclableStackSupport<ThreadControlBlock>,
                           TaskControlBlockComponents::SystemCallSupport<ThreadControlBlock>
{
    IORequest request;

    void beginRequest()
    {
        this->request.buffer = this->getSyscallArgument<void*>();
        this->request.count = this->getSyscallArgument<size_t>();
        this->request.processed = 0;
    }

    void serviceRequest(UInt8 byte)
    {
        this->request.buffer[this->request.processed] = byte;
        this->request.processed += 1;
    }

    void endRequest() const
    {
        this->setKernelReturnValue(this->request.count);
    }

    bool isRequestFulfilled() const
    {
        return this->request.processed == this->request.count;
    }
};
```

Figure 11.7: *Definition of the thread control block for the gateway kernel.*

For demonstration purposes, we assume that there are at most three translator threads that can run concurrently, so like the event controller, the thread controller maintains a table of four thread control blocks. Since threads can neither be created nor destroyed at runtime, we choose the simplest and the fastest allocation algorithm and leave the release function empty. **Figure 11.8** presents the definition.

```
struct ThreadController
{
    ThreadControlBlock table[4];

    ThreadControlBlock* allocate()
    {
        static int index = 0;
        return this->table[index++];
    }

    void release(ThreadControlBlock* block) {}

    ThreadControlBlock* getThreadAtIndex(size_t index)
    {
        return this->table + index;
    }
};
```

Figure 11.8: *Definition of the thread controller.*

We have only two system calls in the gateway kernel, *sysSendData* to send HTTP request messages to the server and *sysRecvData* to receive CoAP request messages from other devices. Both of them are provided by the UART driver module, so we need only assign an identifier to each of them and implement the mapper with a switch statement.

The UART RX interrupt handler maintains a queue of waiting threads that request data from the port. When the hardware triggers an interrupt, the handler starts to move bytes to the buffer provided by the thread until either the hardware has no more data or the request is satisfied. In the latter case, the handler dequeues the thread and notifies the scheduler that the thread has been unblocked. **Figure 11.9** shows the implementation.

```
static LinkedList<ThreadControlBlock> kTranslators;

static ThreadControlBlock* kUART1ReceiveInterruptHandler(ThreadControlBlock* current)
{
    // Fetch the waiting thread
    auto waitingThread = kTranslators.peekHead();

    if (waitingThread == nullptr)
    {
        return current;
    }

    // Receive data
    while (!waitingThread->isRequestFulfilled())
    {
        if (usart_is_recv_ready(USART1_BASE))
        {
            waitingThread->serviceRequest(usart_recv(USART1_BASE) & 0xFF);
        }
        else
        {
            // Request is partially serviced, so the waiting thread remains blocked
            usart_clear_rx_interrupt(USART1_BASE);

            return current;
        }
    }

    // The request is fully serviced, so the waiting thread can be unblocked
    waitingThread->endRequest();

    usart_clear_rx_interrupt(USART1_BASE);

    return GetTaskScheduler<ThreadScheduler>().onTaskUnblocked(current, kTranslators.removeFirst());
}
```

Figure 11.9: *UART RX interrupt handler that services the request of receiving data.*

Finally, we need to initialize the memory manager, the interrupt vector table, UART ports and all the threads before calling the dispatcher in the startup routine. We need 10 lines of code to assemble a custom thread initializer and then initialize an idle thread and three translator threads. **Figure 11.10** presents the thread initialization routine invoked by the kernel main function. As a result, **we need about 127 lines of code to assemble the gateway kernel**. **Table 11.5** summarizes the deployment result.

```
static void initThreads()
{
    static constexpr size_t kDefaultStackSize = 1024;

    using namespace KernelServiceRoutines::CreateThread;

    using ThreadInitializer = KPI::TaskInitializerBuilderWithArgs<
            ThreadControlBlock,
            KPI::AllocateDedicatedStack<ThreadControlBlock>,
            KPI::SetupExecutionContext<ThreadControlBlock, ThreadExecutionContextBuilder_ARM>>;

    // Create the idle thread
    ThreadInitializer{}(controller.allocate(), kDefaultStackSize, idleThread);

    // Create three translator threads
    for (int index = 0; index < 3; index += 1)
    {
        auto thread = controller.allocate();

        ThreadInitializer{}(thread, kDefaultStackSize, translator);

        scheduler.getPolicy().ready(thread);
    }
}
```

Figure 11.10: *Thread initialization routine that creates the idle thread and three translator threads for the gateway kernel.*

Gateway Kernel Components	Deployment Methods	Source Lines of Code
Scheduler	Assemble from building blocks	13
Task Control Block	Write from scratch	29
Task Controller	Write from scratch	15
Kernel Service Routine Mapper	Write from scratch	17
Dispatcher	Assemble from building blocks	1
Memory Allocator	Assemble from building blocks	1
UART RX Interrupt Handler	Write from scratch	29
Startup Routine (Init Threads)	Assemble from building blocks	13
Startup Routine (Init Components)	Shared with other kernels	9
Total		127

Table 11.5: *Gateway kernel per-component line counts.*

11.3.5 Summary

Table 11.6 summarizes the amount of code needed to assemble each kernel. As we can see from these numbers, they are small and assembling them is straightforward: we pick standard

components and “concatenate” them together to build the substrate of our custom task control block; we specify template parameters to materialize kernel services; we provide glue code to fill the gap between kernel components, such as the dispatcher and our custom collection of kernel service routines. As expected, hardware interrupt handlers are always the most challenging part at such an early stage, because we haven’t explored this area yet to provide common building blocks; we leave that for further work.

Kernels	Source Lines of code
Moisture	86
Actuator	111
Gateway	127

Table 11.6: *Summary of the number of lines of code needed to assemble each kernel.*

11.4 Comparison Operating Systems

11.4.1 Criteria of Selecting Points of Comparison

We select our comparison operating systems based on three criteria to ensure the fairest comparison possible. First, the target system must be modular, so we can disable features that are not yet available in Tinkertoy. Second, the target system must be under active development, so it does not present performance or implementation challenges in the kernel. Third, the target system must have an official port that supports the Stellaris LM3S811 board, so we can focus on adopting different kernel APIs when porting user applications to these systems, leaving platform-specific code intact. Based on these criteria, we selected FreeRTOS v202012.00 (§10.2.3) and Zephyr v2.5.99 (§10.2.4), which are all thread-based operating systems.

Since both moisture and actuator kernels are event-driven, we understand that it may be unfair to exclude Contiki-ng (§10.2.2), an event-driven system, from the list. However, Contiki-ng does not support any boards that can be emulated by QEMU, so we would have had to rewrite the entire startup and hardware initialization code to port the kernel to our board.

Such modifications are non-trivial and might have introduced bugs and/or performance penalties that would have threatened the validity of our results.

11.4.2 Porting Applications to Target Systems

11.4.2.1 FreeRTOS Port

Although FreeRTOS is a thread-based operating system, it provides a lightweight mechanism, called *Event Group*, to write event-driven programs. Event handlers in the moisture and the actuator kernels are converted to threads that wait for and process events in an endless loop, while translator threads in the gateway kernel are ported as is. We take example applications in FreeRTOS's official repository as references and modify our system calls to adopt their APIs. However, since there is no strict distinction between the kernel space and the user space in FreeRTOS, our system calls behave like ordinary library calls and thus require no context switches.

11.4.2.2 Zephyr Port

Zephyr is also a thread-based operating system, so we follow the same procedure as we did with FreeRTOS to port our applications, except that *Event Group* is replaced by Zephyr's *Binary Semaphore*. Zephyr can be configured to enable a user space, so the kernel checks system call arguments and performs a context switch; otherwise, system calls are the same as library calls. We choose to disable argument checking, because Tinkertoy does not have any protection against malicious arguments yet, and we did not want to impose an unfair disadvantage on Zephyr.

11.5 Comparison to Other Operating Systems

11.5.1 Flash Footprint

A kernel executable contains a code segment, a read-only data segment and a read/write data segment. Since the processor supports execution in place, the code segment remains in the Flash memory along with the read-only data segment. The read/write data segment contains both initialized and uninitialized global variables and is loaded to main memory by the bootloader.

We use *readelf* to dump the size of each segment and calculate the total number of bytes needed to store code and read-only data segments. **Table 11.**7 summarizes the result.

Devices	Kernels	Flash Footprint (Bytes)	Flash Footprint (Normalized to Tinkertoy)
Monitor	Tinkertoy	4924	100.00
	FreeRTOS	4808	97.64
	Zephyr	11116	225.75
Actuator	Tinkertoy	4601	100.00
	FreeRTOS	4845	105.30
	Zephyr	8568	186.22
Gateway	Tinkertoy	6996	100.00
	FreeRTOS	5976	85.42
	Zephyr	10948	156.49

Table 11.7: *Flash memory footprint of each kernel.*

Tinkertoy has a comparable flash footprint to FreeRTOS but is approximately 50% to 125% smaller than Zephyr. This result is expected as FreeRTOS's kernel is written in C, but our kernels are written in C++, so the executable contains additional functions and data structures generated by the compiler. For example, the scheduler's constructors and destructors comprise 15% of the gateway kernel, while its virtual function tables comprise another 19%.

However, as discussed in §11.7.1, we might be able to reduce the flash footprint by up to 50% if we revise how we assemble schedulers.

Meanwhile, Zephyr reserves about 1 KB to store kernel configuration parameters, such as the number of IRQs, SRAM size, device priorities, etc., and 344 bytes for the software ISR table. Even though we allocate threads and semaphores statically, Zephyr's kernel still requires memory management routines to allocate and release private data structure. However, the result is still reasonable, as the reference minimal Flash footprint with multithreading enabled is about 7 KB to 8 KB [35].

11.5.2 Basic Memory Footprint

The main memory contains global variables and stack areas. **Table 11.8** presents the basic memory footprint that measures the amount of memory reserved for global variables, excluding statically allocated kernel and user stack areas.

Devices	Kernels	Basic Memory Footprint (Bytes)	Basic Memory Footprint (Normalized to Tinkertoy)
Monitor	Tinkertoy	132	100.00
	FreeRTOS	672	509.09
	Zephyr	936	709.09
Actuator	Tinkertoy	135	100.00
	FreeRTOS	888	657.78
	Zephyr	920	681.48
Gateway	Tinkertoy	292	100.00
	FreeRTOS	792	271.23
	Zephyr	1032	353.42

Table 11.8: *Basic memory footprint of each kernel.*

Tinkertoy significantly outperforms both comparison systems, because our building blocks allow developers to include only entries that are needed in kernel data structures. For example, a task control block in Tinkertoy's moisture kernel is 12 bytes long. In comparison, a TCB is 64

bytes long in FreeRTOS and 112 bytes long in Zephyr. Since there are four event handlers registered on the system, it is unsurprising that the other systems require more memory.

11.5.3 Active Stack Usage

We exploit QEMU's GDB interface to trace the execution of each system. The trace log captures the machine state when the processor executes an instruction. In particular, we are interested in the program counter and the stack pointer. We then analyze the kernel executable file to dump the start address and the length of each stack area. As a result, we can visualize how the stack pointer changes and analyze the stack footprint. **Figure 11.11** presents the stack trace of the gateway kernel, **Table 11.9** presents the analysis result of the stack footprint, and **Table 11.10** summarizes the memory footprint.

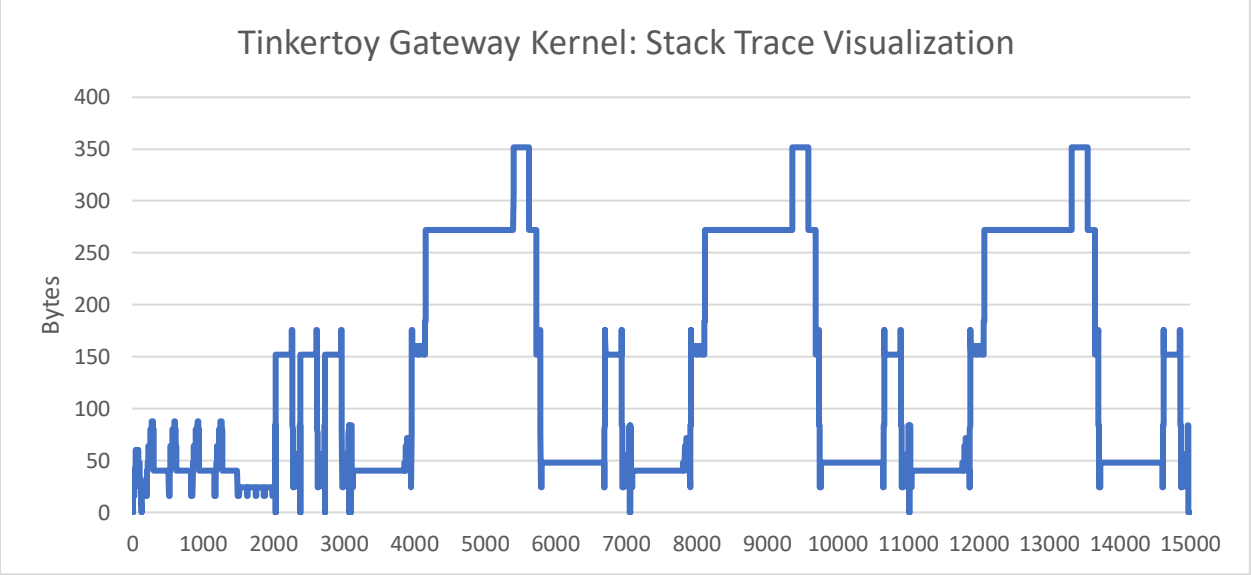


Figure 11.11: *Visualization of stack usage in the Tinkertoy gateway kernel.*

The x-axis represents the instruction number, and the y-axis represents the number of bytes allocated on the stack. The lower four peaks (200 - 1500) correspond to the initialization of the idle thread and three translator threads. The middle three peaks (2000 - 3000) correspond to translator threads invoking `sysRecvData` *to wait for messages. The highest three peaks (4000 - 6000, 8000 - 10000, 12000 - 14000) correspond to translator threads receiving a CoAP message and starting the translation.*

Devices	Kernels	Active Stack Usage (Kernel)	Active Stack Usage (User)	Active Stack Usage (Total)	Active Stack Usage (Normalized to Tinkertoy)
Monitor	Tinkertoy	84	208	292	100.00
	FreeRTOS	112	360	472	161.64
	Zephyr	176	592	768	263.01
Actuator	Tinkertoy	88	192	280	100.00
	FreeRTOS	116	412	528	188.57
	Zephyr	96	584	680	242.86
Gateway	Tinkertoy	88	1056	1144	100.00
	FreeRTOS	112	1160	1272	111.19
	Zephyr	96	1456	1552	135.66

Table 11.9: *Active Stack Footprint of each kernel.*

Devices	Kernels	Memory Footprint (Total)	Memory Footprint (Normalized to Tinkertoy)
Monitor	Tinkertoy	424	100.00
	FreeRTOS	1144	269.81
	Zephyr	1704	401.89
Actuator	Tinkertoy	415	100.00
	FreeRTOS	1416	341.20
	Zephyr	1600	385.54
Gateway	Tinkertoy	1436	100.00
	FreeRTOS	2064	143.73
	Zephyr	2584	179.94

Table 11.10: *Total memory footprint of each kernel.*

Both comparison systems have a larger kernel stack footprint than Tinkertoy does, but the differences are insignificant in all but the moisture kernel on Zephyr. Zephyr provides a special mechanism to use software timers. The kernel runs periodic timer tasks on a shared system work queue instead of an ordinary thread. As a result, sensor reading is performed in the kernel and therefore increases the kernel stack footprint.

On the other hand, our moisture and actuator systems have a significantly lower user stack footprint than others, because both of them are event-driven and all the event handlers share a single user stack. The difference is reduced to about 10% in the thread-based gateway kernel, because translators on each system have the same implementation except for system calls.

11.5.4 Performance

Tinkertoy-based kernels provide precisely the functionality needed by applications, so they should have not only a low memory footprint but also more efficient runtime than other systems. We evaluate the performance by means of the amount of time it takes the gateway kernel to receive a CoAP request message and send out the translated HTTP message.

We use the controller process to send a CoAP message to the gateway device and measure the delay until the translated message is received. We start the system, send 100 32-byte messages, and compute the minimum, maximum, median, mean, and standard deviation of the sample. We then repeat the process, but this time we send 1000 messages to estimate long-term performance. **Table 11.11**, **Figure 11.12**, and **Figure 11.13** summarize the result.

As discussed in §11.3.4, the gateway kernel does not require the hardware timer, and both Tinkertoy and Zephyr allows developers to disable the timer. However, FreeRTOS does not provide such an option, because its scheduler relies on the timer. Instead, FreeRTOS allows us to adjust the timer frequency, so we change the value to 1 Hz, which is the minimum possible value, thus reducing the number of timer interrupts during our experiment. To ensure the fairest comparison possible, we also modify the FreeRTOS kernel to disable the timer forcedly, and we measure the performance of both modified FreeRTOS gateway kernels.

Kernels	Sample Size	Round Trip Time (In Milliseconds)				
		Minimum	Maximum	Median	Mean	Standard Deviation
Tinkertoy	100	0.63	1.45	0.78	0.79	0.09
	1000	0.63	1.38	0.82	0.82	0.08
FreeRTOS (1Hz Timer)	100	0.60	1.56	0.82	0.81	0.12
	1000	0.59	3.22	0.89	0.88	0.14
FreeRTOS (No Timer)	100	0.64	1.38	0.70	0.72	0.09
	1000	0.60	14.1	0.82	0.85	0.45
Zephyr	100	0.68	1.27	0.87	0.88	0.09
	1000	0.66	1.81	0.88	0.89	0.09

Table 11.11: *Statistics of the round trip time in milliseconds on each gateway system.*

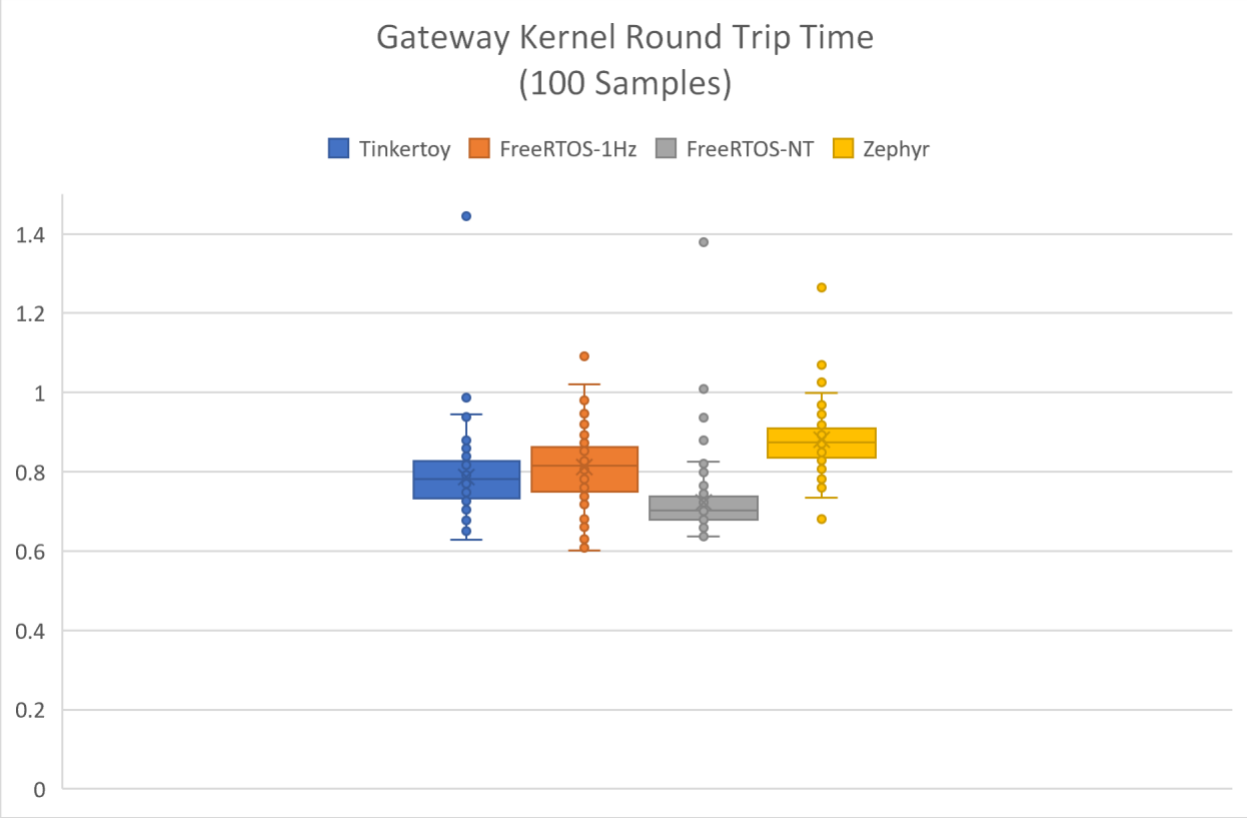


Figure 11.12: *Boxplot of 100 sample round trip times in milliseconds on each gateway system.*

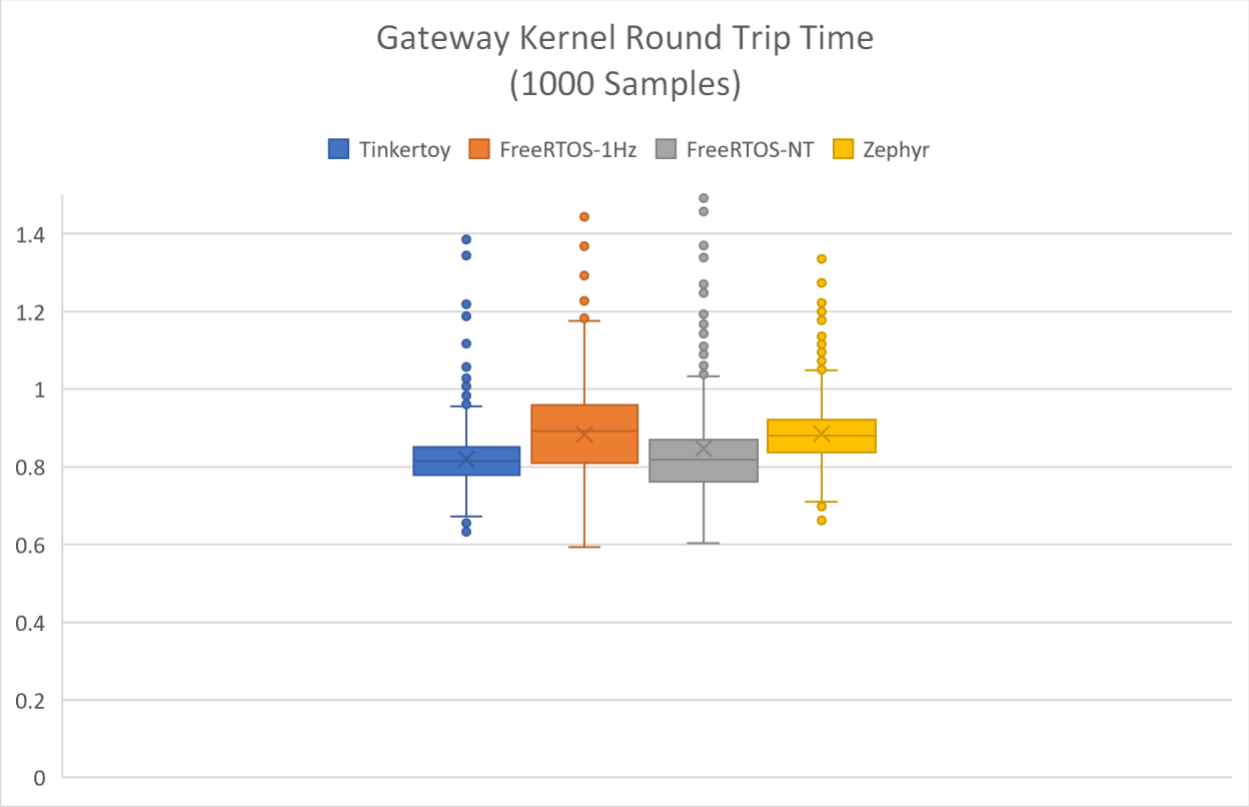


Figure 11.13: *Boxplot of 1000 sample round trip times in milliseconds on each gateway system.*

Our gateway system runs 11% faster than Zephyr but 9% slower than FreeRTOS. We analyze the assembly code and the execution trace and find that the performance overhead is related to how the kernel reacts to hardware interrupts and unblocks the translator thread. Since there is no distinction between the kernel space and the user space on FreeRTOS, the processor jumps to the UART receive interrupt handler directly when a message arrives. Recall that FreeRTOS's semaphore is built on top of the list construct, so the kernel then removes the translator thread from the waiting list in the semaphore and puts it on the scheduler's ready queue by three ordinary C function calls.

In contrast, Tinkertoy allows developers to select which functionality the kernel should provide, so the processor must traverse through the context switcher and the dispatcher (service identifier finder and the service routine mapper) before entering the interrupt handler, thus having a longer code path than FreeRTOS. Inside the UART receive interrupt

handler, Tinkertoy essentially does the same thing as FreeRTOS to unblock the translator thread but by four virtual function calls as shown in **Figure 11.14**.

```
Task* onTaskUnblocked(Task* current, Task* task) override
{
    // 1. Virtual Function Call: Retrieve the current scheduling policy
    SchedulingPolicy<Task>& policy = this->getPolicy();

    // Guard: [Special] Check whether the caller performs an intermediate call
    // This event handler supports group operation. Not used in this example.
    if (current == nullptr)
    {
        // Intermediate call
        policy.ready(task);

        return nullptr;
    }

    // Guard: [Special] Check whether the caller only wants to fetch the next task
    // This event handler supports group operation. Not used in this example.
    if (task == nullptr)
    {
        // The current running task keeps running
        return current;
    }

    // Default: Ready queue might be modified before this method is called
    // 2. Virtual Function Call: Enqueue the unblocked task
    policy.ready(task);

    // Guard: Check whether the current task is the idle task
    // 3. Virtual Function Call: Get the idle task
    if (current == this->getIdleTask())
    {
        // 4. Virtual Function Call: Get the next ready task from the queue
        return policy.next();
    }
    else
    {
        // The current running task keeps running
        return current;
    }
}
```

Figure 11.14: *Implementation of the cooperative task unblocked event handler.*

Since our scheduler class inherits from multiple component classes, to place the translator thread on the ready queue, the scheduler retrieves its policy component by a virtual function call and then locates the policy's virtual function table to calculate the address of the *next* function, involving at least additional 12 memory accesses thus slowing down the process. **Figure 11.15** presents the pseudocode of the Task Unblocked Event Handler derived from the assembly code. The result again reminds us to revise how we assemble the scheduler in future work.

Meanwhile, Zephyr takes the multithreaded kernel into consideration; the kernel relies on spinlocks to protect semaphores and the scheduler, thus introducing unnecessary barrier instructions for every list operation on a single-threaded system. However, Zephyr does not provide a separate implementation for such a system, so we learn that Tinkertoy must provide building blocks for both types of system in the future.

```
int onTaskUnblocked(void * arg0, void * arg1)
{
    sp = sp - 0x18;
    r4 = arg0;
    r6 = arg1;
    // Locate the policy component (2 memory reads)
    r0 = *(*arg0 + 0xfffffffffffffff4);
    r7 = r2;
    // Virtual function call: this->getPolicy() (2 memory reads)
    r0 = (*(*(r4 + r0) + 0x8))(r0 + r4, arg1);
    r5 = r0;
    if (r6 != 0x0) goto loc_7f8;

loc_7ea:
    (*(*r0 + 0xc))();
    goto loc_7f2;

loc_7f2:
    r0 = r6;
    return r0;

loc_7f8:
    if (r7 == 0x0) goto loc_7f2;

loc_7fc:
    // Virtual function call: policy->ready() (2 memory reads)
    (*(*r0 + 0xc))();
    // Locate the idle task support component (2 memory reads)
    r0 = *(*r4 + 0xfffffffffffffff0);
    // Virtual function call: this->getIdleTask() (2 memory reads)
    if (r6 != (*(*(r4 + r0) + 0x8))(r0 + r4)) goto loc_7f2;

loc_816:
    // Virtual function call: policy->next() (2 memory reads)
    r0 = (*(*r5 + 0x8))(r5);
    return r0;
}
```

Figure 11.15: *Pseudocode of the task unblocked event handler derived from the assembly code.*

Since kernels run on the emulator instead of real hardware, we also analyze the execution trace to calculate the total number of instructions executed to receive, translate and send out a message. **Table 11.12** presents the result.

Instructions	Tinkertoy	FreeRTOS (1Hz Timer)	FreeRTOS (No Timer)	Zephyr
Total Number of Instructions	3825	3217	3056	4236
Normalized to Tinkertoy	100	84.10	79.89	110.74

Table 11.12: *The total number of instructions executed on each gateway system.*

11.6 Interoperability

Tinkertoy offers both great flexibility and comparable performance but should not limit itself to work only with other Tinkertoy-based systems. We mixed devices running different kernels to ensure interoperability. For example, we assemble an automatic watering system from a monitor device that runs FreeRTOS, an actuator device that runs Tinkertoy and a gateway device that runs Zephyr. We examined the data received by the HTTP server and found that all combinations passed our tests.

11.7 Drawbacks Discussion

Tinkertoy's modularity and flexibility do not come for free, we observed two categories of overheads that Tinkertoy introduced.

11.7.1 Increased Code Size

First, Tinkertoy is written in C++, so compilers have to generate additional code and data structures to support language features, such as object-oriented design, multiple inheritance, and virtual functions, but the size of this auto-generated code is significant on resource-constrained tiny devices. For example, the moisture kernel has a code size of 4808 bytes, but constructors and destructors consume 1048 bytes, and the virtual function tables consume 1288 bytes. This code and tables comprise 48.5% of the code segment, because the scheduler inherits from multiple interface classes. Even though the FreeRTOS and Zephyr kernels are

written in C, their code sizes are still larger than Tinkertoy's. We see potential to reduce Tinkertoy's code size still further if we revise how we assemble schedulers to avoid unnecessary virtual functions (§11.5.4).

11.7.2 Reduced Portability

Second, Tinkertoy does not provide a unified kernel, meaning that there is no consistent set of APIs. As a result, developers might find it challenging to port their applications from one Tinkertoy-based system to another due to an incompatible execution model, different scheduling assumptions, missing system calls, etc. Since Tinkertoy is still in its infancy, we have not yet addressed this issue, but we could imagine introducing standard API packages, so that a developer could choose an existing API package to reduce the burden of porting applications.

11.8 Summary

We evaluated Tinkertoy from three perspectives: the effort needed to assemble a custom kernel, memory footprint and runtime performance. We find that about 75% of the deployment code is straightforward and easy to write, as developers use our macros and builder classes to concatenate a collection of components and materialize kernel services. The rest of the code deals with hardware interrupts, but Tinkertoy does not yet provide standard building blocks in this area, so developers have to use existing kernel and driver APIs to service the hardware manually at this moment.

The assembled kernels provide exactly the functionality needed by particular applications, so Tinkertoy-based kernels have 2.5x to 4x smaller memory footprints than other popular IoT operating systems and comparable performance to them. However, there is still a large room for future improvement, such as C++ devirtualization to further reduce the memory footprint, standard API packages to improve applications' portability, source-to-source compilers to translate thread-based programs to event-driven ones and vice versa.

Chapter 12 Limitations and Future Work

We showed that operating systems assembled from Tinkertoy building blocks expose minimal performance overhead and have a smaller memory footprint than other systems. However, our journey has just begun, and we summarize the current limitations and identify four future research directions to improve Tinkertoy.

12.1 The Cost of Flexibility and Configurability

Tinkertoy relies on C++ virtual functions and concepts extensively to provide maximum flexibility and configurability. In summary, we use the following three strategies to modularize an operating system and implement each customized component efficiently.

12.1.1 Pure Virtual Functions

C++ interface classes that define one or more pure virtual functions allows us to specify a set of operations that are common to all heterogeneous concrete types. For example, our scheduling policy component (§4.2.2) defines two interface operations, *ready* and *next*, so we can assemble a Multilevel Queue scheduler that implements different scheduling algorithms for each priority level. We also exploit multiple inheritance to assemble an interface from a collection of individual ones. For example, we can select a set of event handlers for a scheduler to interact with kernel service routines (§4.3). With virtual functions, we can provide different implementations for a component, allowing developers to choose the one that suits their needs. However, as we observe from the experiment (§11.5.1, §11.5.4, §11.7.1), multiple inheritance and virtual functions have negative effects on the code size and runtime performance, so we should use them only if necessary.

12.1.2 Pure C++20 Concepts

C++20 concepts allow us to define constraints on a type, so we can decompose any kind of control blocks (e.g., memory block in §5.2.1 and task control block in §9.1.2) into pieces and define a constraint that describes each. A kernel component that relies on a particular part of a control block then uses the corresponding constraint to filter out unsupported types, thus independent of any concrete control block type. For example, our kernel service routines (§8.1.2) that are independent of the execution model cannot satisfy requests of a task that does not satisfy their constraints. As a result, we are able to compose a task control block that contains only the information needed by the kernel to provide services.

Compared to virtual functions, C++20 concepts incur no runtime overhead in terms of both memory footprint and performance, because the compiler verifies constraints at compile time and does not generate additional data structures. As a result, calling a function specified by a constraint is a direct jump in assembly, thus faster than calling a virtual function that requires access to the virtual function table. While we can provide different implementations for a component without any performance penalty, C++20 concepts must be used together with a concrete type, making it impossible to define an opaque type, so they are not suitable for implementing dynamic dispatch.

12.1.3 A Hybrid Approach

Our trials and tribulations in designing modular building blocks show that C++20 concepts cannot replace virtual functions completely and vice versa, so we propose that the solution lies in using both of them, and finding the “balance point” is the key challenge. Specifically, to achieve maximum flexibility while incurring minimum or no runtime overhead, which strategy should we use to design and implement which types of kernel components? What properties do make a component inherently suitable for a particular strategy? As we explore more components (§12.2), we believe that we will have a clear answer to those questions in the future.

12.2 Other Operating System Components

Tinkertoy is still in its infancy and does not yet have support for reentrant kernel, multithreaded kernel, networking, inter-domain communication, filesystem, and etc. We discuss the assumptions that each missing component imposes on the kernel architecture and illustrate how they fit into the system in the next five subsections.

12.2.1 Reentrant Kernel Support

Tinkertoy limits all assembled kernels to being non-reentrant, so the system essentially services hardware interrupts on a first come first served basis. It is simple to implement such a system, because kernel service routines can assume that they have exclusive access to all kernel data structures thus do not need any mechanism to prevent interleaved access.

However, non-reentrant kernels become inadequate if some hardware interrupts are more critical than others and must be serviced immediately to avoid any catastrophic effects. For example, when the proximity sensor on a self-driving vehicle detects an obstacle, its interrupt service routine must run immediately to deliver a signal to the brake. As a result, the kernel ends up with servicing multiple requests concurrently, so Tinkertoy must provide building blocks to coordinate access to shared data structures, such as the ready queue (§4.2.2).

A common approach is to disable hardware interrupts while the kernel is modifying shared resources, so we can provide a companion block for each architecture, allowing developers to construct a reentrant-safe building block on top of an unsafe one. **Figure 12.1** illustrates how we protect access to the ready queue. The system essentially disables hardware interrupts (1) before calling the *ready* function (2) and reenables interrupts (3) after the function returns. Since our building blocks are tiny, we might be able to minimize the no-interrupt window as much as possible. We also need a reentrant-safe context switcher to enter (4) and leave (5) the

kernel recursively, and we believe that we can assemble one once we have assembly building blocks in the future.

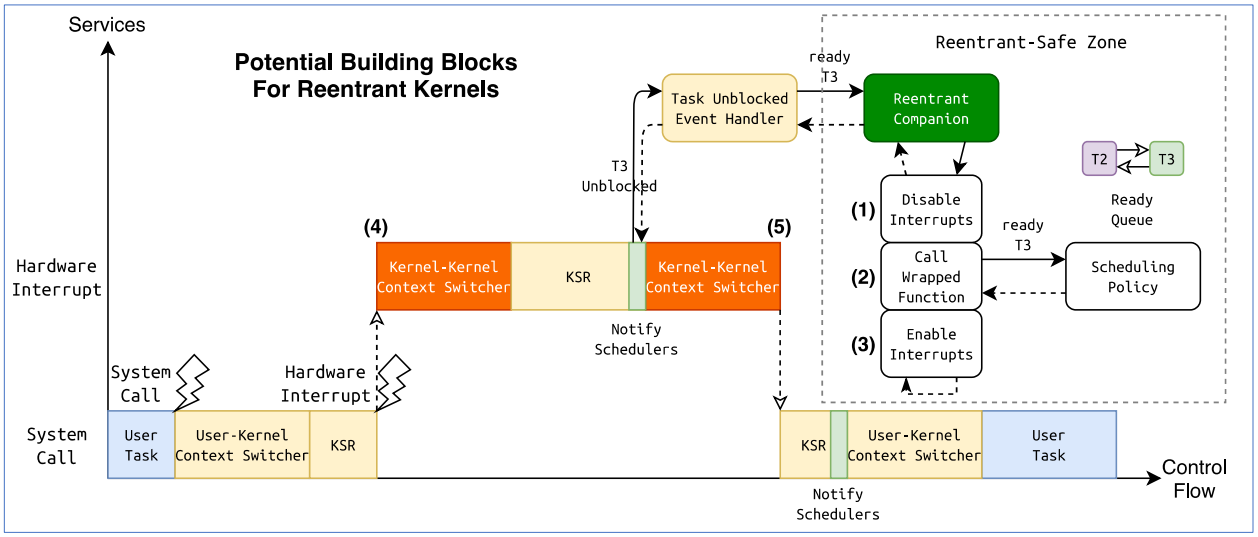


Figure 12.1: *Illustration of building blocks for reentrant kernels.*

12.2.2 Multithreaded Kernel Support

Tinkertoy limits all assembled kernels to being single-threaded, so the system contains only one kernel stack but is sufficient enough for a tiny device that has a single-core processor and limited volatile memory. In contrast, multithreaded kernels can exploit the full potential of multi-core processors, enabling servicing requests in parallel but also imposing new demands on the kernel architecture. For example, we need lock primitives to protect shared resources, a kernel-to-kernel context switcher to switch between two tasks in the kernel, take tasks' affinity to a processor into consideration in schedulers, and etc. We propose that these new demands are also expressible as individual building blocks.

However, some of the new demands make particular kernel service routines no longer have deterministic runtime, thus limiting the collection of kernel functionality suitable for real-time operations. For example, a kernel service routine that allocates dynamic memory can be blocked waiting for a lock that protects the account book (§5.2.4). A common strategy to design

a multithreaded kernel is to assign a kernel thread to each user thread. In other words, a task now has a user stack for running user code and a kernel stack for running kernel service routines, so it can block itself in the kernel by asking the scheduler to dequeue the next task and calling the kernel-to-kernel context switcher. The next task resumes running, finishes up what it has done before blocked, and exits the kernel. As a result, kernel service routines become blocking, but whether it is possible to design companion blocks to convert two (or more) existing non-blocking kernel service routines (§8.1.1) back to a blocking one remains to be explored.

12.2.3 Networking Support

Networking support is crucial for a tiny device to communicate with others and can be broadly decomposed into three parts: task control block components to store socket-related control data, kernel service routines that implement socket system calls, and device drivers that interact with network interface cards. **Figure 12.2** illustrates how these new components fit into the system.

Recall that our DNS system presented in §3.1 has a sender task that constructs a query message and sends it to a DNS server. The sender task invokes a system call to create a UDP socket (1), so the kernel service routine initializes a private data structure in the task control block (2) and returns an opaque handle to it (3). The sender task then makes another system call to send a message via the socket (4), so the kernel service routine initializes a network packet (5), traverses through the network stack (6), and passes the final packet to the driver (7) that instructs the communication device to transmit the data. At the end, the kernel service routine returns to the dispatcher, and the sender task resumes (8). As we can observe from the figure, we do not need change our design at all; instead, we merely add new building blocks to existing components (task control block components, kernel service routines, system calls, and device drivers) to meet the new demands. However, we might be able to extract common building blocks from different network stacks, so that developers can customize their behaviors easily.

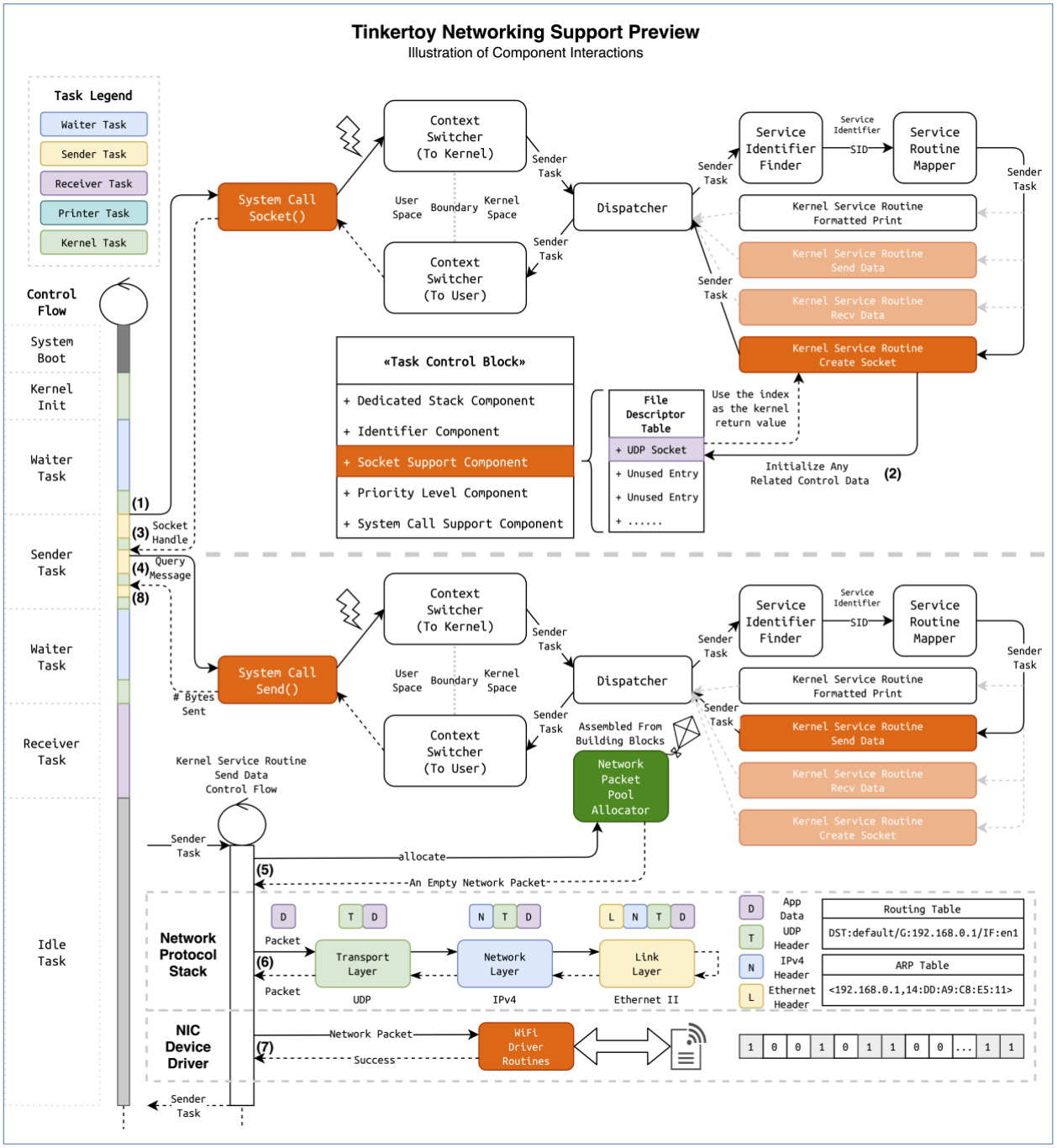


Figure 12.2: *Illustration of interactions between the networking subsystem and other kernel components.*

12.2.4 Advanced Inter-domain Communications

Remote procedure call (RPC) is one of fundamental abstractions in distributed computations. It abstracts away low-level details of a message and allows developers to invoke a function on other machines transparently. TinkerRPC is one of our companion ongoing projects and shares the same design concepts as Tinkertoy. It separates a classic RPC framework into five standalone components and decomposes each of them into building blocks, from which developers can assemble a custom framework that works with their message formats, communication channels, service discovery protocols and execution models.

12.2.5 Filesystem Support

TinkerFS is another companion project that allows developers to take full advantage of tiny devices with secondary storage media. It analyzes existing operations defined in the virtual filesystem layer (VFS) and further decomposes them into primitive micro-operations, from which developers can assemble a custom file system for their applications. A filesystem uses an *inode* to describe a file or a directory, storing metadata, such as the file size, permission, and timestamps. As such, an *inode* is similar to a task control block, so we might be able to use the same technique to decompose an *inode* into components and assemble one for a custom filesystem that contains only the functionality needed by an application. Similar to the networking module, we expect that filesystems will be expressed as task control block components, system calls and kernel service routines as well.

12.2.6 Epilogue

We discussed five modules that are not yet available in Tinkertoy. When these modules are complete, we will be able to explore a richer set of applications for which Tinkertoy will be suitable.

12.3 Operating System Synthesis

Writing operating systems from scratch is challenging, but Tinkertoy has lowered the burden by allowing developers to select a collection of standard building blocks and assemble a custom system that suits their needs. We might be able to reduce the effort further by leveraging program synthesis, a technique of searching for an implementation that satisfies a specification. However, it is inherently difficult to synthesize operating system code due to the complexity of low-level specification and the exponential growth of the search space [22]. In our cases, the specification is the application requirements, while the search space is that of all possible configurations of all combinations of building blocks.

Nonetheless, Tinkertoy's building blocks design may ease the challenge of synthesizing a kernel from three angles. First, each building block is associated with one or more type constraints by design, so we can shrink the search space by excluding all invalid combinations due to mismatched types. Second, each building block has a single responsibility and is tiny in terms of the source lines of code (SLOC), so the synthesizer can spend less effort finding a correct implementation for a block than synthesizing a huge function. As a result, we might be able to solve a large number of tiny problems in parallel and merge the solutions together. Third, each building block is either architecture-dependent or architecture-independent but not both, so we can apply different strategies to synthesize each type of block. For example, we could use an interactive approach to synthesize a block that contains assembly code with a trace-based approach [6] to deal with blocks that manipulate state.

In general, Tinkertoy allows us to divide the synthesis problem into two halves. Given a specification of application requirements, the upper half finds a collection of building blocks that satisfies the requirement, while the lower half finds an implementation for each building block. Moreover, if the implementation itself is expressible as a collection of building blocks defined at a lower layer, we can solve the problem recursively until we are left with primitives.

12.4 Operating System Security

As people increasingly connect tiny devices to the physical world, security has been a major concern on embedded systems. We discuss how we could provide building blocks to secure Tinkertoy-based systems in three aspects in the next subsections.

12.4.1 Memory Protection

Due to the lack of hardware-based isolation mechanisms, both kernel and user applications run in the same address space, so a faulty program can allow attackers to tamper with kernel memory and/or execute arbitrary code. The memory protection unit (MPU) on ARM Cortex-M processors can be configured to restrict user applications' access to certain memory regions, so we can add two building blocks to the context switcher: one that configures and enables the MPU before the system exits from the kernel, and the other one that disables the MPU after the system enters the kernel. Unfortunately, an MPU requires that memory regions must be aligned or have a size of a power of two, resulting in wasted space. Recent language-level isolation [30] and protection [13] techniques and software-based memory management [59] incurs low runtime overhead, so they may be applicable to secure Tinkertoy-based systems running on memory-constrained devices.

12.4.2 Authorizations and Credentials

Tinkertoy does not yet provide any mechanism to restrict a user task's access to particular kernel services. For example, the kernel should not allow a user task to read from a file that does not belong to it, but the kernel service routine that implements *read* does not implement any checks. We could introduce a task control block component that stores the identity of a task and an *inode* component that stores a list of identities (ACLs) who are allowed to read the file. We then design a kernel service routine component that retrieves the access control lists from the *inode* and checks whether the task identity is in the list. If so, the kernel proceeds to the original routine that contains no checks, otherwise it rejects the system call by setting a

proper kernel return value. As such, we can build an authorized service routine on top of the original one and the new component.

Similarly, we can restrict a task's access to particular system calls as well: A task must have a credential to invoke a system call, and a secured kernel service routine must verify the credential before delegating the request to the "unsecured" one. In summary, we propose that it is possible to design building blocks that implement security policies and combine them with existing unsecured ones to assemble a secured system.

Chapter 13 Conclusion

Tinkertoy allows developers to assemble operating systems customized for particular applications in less than 150 lines of code. It does so by providing a set of configurable and composable components. We use constraints to ensure that each component is as flexible as possible while also ensuring that all the requirements are satisfied by developer-specified components. The assembled systems not only provide precisely the functionality that an application needs, but also have four times less memory footprint than existing IoT operating systems and comparable performance to them.

While there are many avenues for future work, we demonstrated that it is possible to assemble realistic operating systems in this manner. Tinkertoy is an embodiment of this approach and demonstrates how to achieve flexibility and configurability to satisfy the requirements of heterogeneous applications.

References

[1] Arasteh, H., Hosseinnezhad, V., Loia, V., Tommasetti, A., Troisi, O., Shafie-khah, M., & Siano, P. (2016). Iot-based smart cities: A survey. *2016 IEEE 16th International Conference on Environment and Electrical Engineering (EEEIC)*, 1–6. https://doi.org/10.1109/eeeic.2016.7555867

[2] *Arduino Board Nano*. (n.d.). Arduino. Retrieved April 8, 2021, from https://www.arduino.cc/en/pmwiki.php?n=Main/ArduinoBoardNano

[3] *ARMv7-M Architecture Reference Manual*. (n.d.). ARM Developer. Retrieved April 8, 2021, from https://developer.arm.com/documentation/ddi0403/latest/

[4] Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C. (2018). *Operating Systems: Three Easy Pieces* (1.00 ed.). CreateSpace Independent Publishing Platform.

[5] Audsley, N., Burns, A., Richardson, M., & Wellings, A. (1992). Deadline Monotonic Scheduling Theory. *IFAC Proceedings Volumes*, *25*(11), 55–60. https://doi.org/10.1016/s1474-6670(17)50124-x

[6] COMET: Tractable Reactive Program Synthesis. (2021). Christopher Chen. Master Thesis. University of British Columbia.

[7] *Contiki 2.6: Managed memory allocator*. (n.d.). Contiki. Retrieved April 8, 2021, from

[8] *Contiki-NG, the OS for Next Generation IoT Devices*. (n.d.). Contiki-NG. Retrieved April 8, 2021, from https://www.contiki-ng.org/

[9] Corbató, F. J., Merwin-Daggett, M., & Daley, R. C. (1962). An experimental time-sharing system. *Proceedings of the May 1–3, 1962, Spring Joint Computer Conference on - AIEE-IRE '62 (Spring)*, 335–344. https://doi.org/10.1145/1460833.1460871

[10] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. 2016. EbbRT: a framework for building per-application library operating systems. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation* (*OSDI'16*). USENIX Association, USA, 671–688.

[11] David A. Holland. (2020). Toward Automatic Operating System Ports via Code Generation and Synthesis. Ph.D. Dissertation. Harvard University.

[12] Divakaran, S., Manukonda, L., Sravya, N., Morais, M. M., & Janani, P. (2017). IoT clinic-Internet based patient monitoring and diagnosis system. *2017 IEEE International Conference on Power, Control, Signals and Instrumentation Engineering (ICPCSI)*, 2858–2862. https://doi.org/10.1109/icpcsi.2017.8392243

[13] Duan, J., Yang, Y., Zhou, J., & Criswell, J. (2020). Refactoring the FreeBSD Kernel with Checked C. *2020 IEEE Secure Development (SecDev)*, 15–22. https://doi.org/10.1109/secdev45635.2020.00018

[14] Dunkels, A., Gronvall, B., & Voigt, T. (2004). Contiki - a lightweight and flexible operating system for tiny networked sensors. 29th Annual IEEE International Conference on Local Computer Networks, 455–462. https://doi.org/10.1109/lcn.2004.38

[15] Dunkels, A., Schmidt, O., Voigt, T., & Ali, M. (2006). Protothreads. *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems - SenSys '06*, 29–42. https://doi.org/10.1145/1182807.1182811

[16] Elmougy, S., Sarhan, S., & Joundy, M. (2017). A novel hybrid of Shortest job first and round Robin with dynamic variable quantum time task scheduling technique. *Journal of Cloud Computing*, *6*(1), 1–12. https://doi.org/10.1186/s13677-017-0085-0

[17] Engler, D. R., Kaashoek, M. F., & O'Toole, J. (1995). Exokernel. *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles - SOSP '95*, 251–266. https://doi.org/10.1145/224056.224076

[18] Ford, B., Back, G., Benson, G., Lepreau, J., Lin, A., & Shivers, O. (1997). The Flux OSKit. *ACM SIGOPS Operating Systems Review*, *31*(5), 38–51. https://doi.org/10.1145/269005.266642

[19] *FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions*. (n.d.). FreeRTOS. Retrieved April 8, 2021, from https://www.freertos.org

[20] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., & Culler, D. (2003). The nesC language. *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation - PLDI '03*, 1–11. https://doi.org/10.1145/781131.781133

[21] *HotSpot Virtual Machine Garbage Collection Tuning Guide*. (n.d.). Oracle Help Center. Retrieved April 8, 2021, from https://docs.oracle.com/en/java/javase/15/gctuning/index.html

[22] Hu, J., Lu, E., Holland, D. A., Kawaguchi, M., Chong, S., & Seltzer, M. I. (2019). Trials and Tribulations in Synthesizing Operating Systems. *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, 67–73. https://doi.org/10.1145/3365137.3365401

[23] Hugh C. Lauer and Roger M. Needham. 1979. On the duality of operating system structures. SIGOPS Operating System. Rev. 13, 2 (April 1979), 3–19.

[24] *Java Garbage Collection Basics*. (n.d.). Oracle. Retrieved April 8, 2021, from https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

[25] Javed, F., Afzal, M. K., Sharif, M., & Kim, B. S. (2018). Internet of Things (IoT) Operating Systems Support, Networking Technologies, Applications, and Challenges: A Comparative Review. *IEEE Communications Surveys & Tutorials*, *20*(3), 2062–2100. https://doi.org/10.1109/comst.2018.2817685

[26] Jeff Bonwick. 1994. The slab allocator: an object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1* (*USTC'94*). USENIX Association, USA, 6.

[27] Klues, K., Liang, C. J. M., Paek, J., Musăloiu-E, R., Levis, P., Terzis, A., & Govindan, R. (2009). TOSThreads. *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems - SenSys '09*, 127–140. https://doi.org/10.1145/1644038.1644052

[28] Knuth, D. E. & Addison-Wesley. (1997). *The Art of Computer Programming: Fundamental algorithms*. Addison-Wesley.

[29] Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., & Culler, D. (2005). TinyOS: An Operating System for Sensor Networks. Ambient Intelligence, 115–148. https://doi.org/10.1007/3-540-27139-2_7

[30] Levy, A., Campbell, B., Ghena, B., Giffin, D. B., Pannuto, P., Dutta, P., & Levis, P. (2017). Multiprogramming a 64kB Computer Safely and Efficiently. *Proceedings of the 26th Symposium on Operating Systems Principles*, 234–251. https://doi.org/10.1145/3132747.3132786

[31] Liu, C. L., & Layland, J. W. (1973). Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, *20*(1), 46–61. https://doi.org/10.1145/321738.321743

[32] Mainetti, L., Patrono, L., & Vilei, A. (2011). Evolution of wireless sensor networks towards the internet of things: A survey. SoftCOM 2011, 19th International Conference on Software, Telecommunications and Computer Networks, 1-6.

[33] Masmano, M., Ripoll, I., Balbastre, P., & Crespo, A. (2008). A constant-time dynamic storage allocator for real-time systems. *Real-Time Systems*, *40*(2), 149–179. https://doi.org/10.1007/s11241-008-9052-7

[34] McKusick, Marshall Kirk, and Michael J. Karels. Design of a General Purpose Memory Allocator for the 4.3BSD UNIX Kernel. (1988). *In: Proceedings of the USENIX Summer Technical Conference*, 295–304.

[35] *Minimal footprint*. (n.d.). Zephyr Project Documentation. Retrieved April 8, 2021, from https://docs.zephyrproject.org/latest/samples/basic/minimal/README.html

[36] *MQTT Specification Version 5.0*. (n.d.). MQTT. Retrieved April 11, 2021, from https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

[37] Musaddiq, A., Zikria, Y. B., Hahm, O., Yu, H., Bashir, A. K., & Kim, S. W. (2018). A Survey on Resource Management in IoT Operating Systems. *IEEE Access*, *6*, 8459–8482. https://doi.org/10.1109/access.2018.2808324

[38] Ousterhout, John. Why threads are a bad idea (for most purposes). Presentation given at the 1996 Usenix Annual Technical Conference. Vol. 5. 1996.

[39] Perera, M. S., Halgamuge, M. N., Samarakody, R., & Mohammad, A. (2021). Internet of Things in Healthcare: A Survey of Telemedicine Systems Used for Elderly People. *IoT in Healthcare and Ambient Assisted Living*, 69–88. https://doi.org/10.1007/978-981-15-9897-5_4

[40] Peterson, J. L., & Norman, T. A. (1977). Buddy systems. *Communications of the ACM*, *20*(6), 421–431. https://doi.org/10.1145/359605.359626

[41] Puccinelli, D., & Haenggi, M. (2005). Wireless sensor networks: applications and challenges of ubiquitous sensing. *IEEE Circuits and Systems Magazine*, *5*(3), 19–31. https://doi.org/10.1109/mcas.2005.1507522

[42] *RFC 7228 - Terminology for Constrained-Node Networks*. (n.d.). Internet Engineering Task Force (IETF). Retrieved April 8, 2021, from https://tools.ietf.org/html/rfc7228

[43] *RFC 7252 - The Constrained Application Protocol (CoAP)*. (n.d.). Internet Engineering Task Force (IETF). Retrieved April 11, 2021, from https://tools.ietf.org/html/rfc7252

[44] Shelby, Z., & Bormann, C. (2011). *6LoWPAN: The wireless embedded Internet* (Vol. 43). John Wiley & Sons.

[45] Shen, K. K., & Peterson, J. L. (1974). A weighted buddy method for dynamic storage allocation. *Communications of the ACM*, *17*(10), 558–562. https://doi.org/10.1145/355620.361164

[46] Singh, A. (2016). *Mac OS X Internals: A Systems Approach* (Reprint ed.). Addison-Wesley Professional.

[47] Sonnis, O., Sunka, A., Singh, R., & Agarkar, T. (2017). IoT based telemedicine system. *2017 IEEE International Conference on Power, Control, Signals and Instrumentation Engineering (ICPCSI)*, 2840–2842. https://doi.org/10.1109/icpcsi.2017.8392239

[48] *STM32F102RB*. (n.d.). STM32F102RB Product Overview. Retrieved April 8, 2021, from https://www.st.com/en/microcontrollers-microprocessors/stm32f102rb.html

[49] *The Zephyr Project*. (n.d.). The Zephyr Project. Retrieved April 8, 2021, from https://www.zephyrproject.org

[50] Toschi, G. M., Campos, L. B., & Cugnasca, C. E. (2017). Home automation networks: A survey. *Computer Standards & Interfaces*, *50*, 42-54. https://doi.org/10.1016/j.csi.2016.08.008

[51] Vasseur, J. P., & Dunkels, A. (2010). *Interconnecting smart objects with ip: The next internet*. Morgan Kaufmann.

[52] Venkat, S., Clyburn, M., & Campbell, B. (2020). Energy Harvesting Systems Need an Operating System Too. *Proceedings of the 8th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, 15–21. https://doi.org/10.1145/3417308.3430274

[53] Von Behren, J. Robert, Jeremy Condit, and Eric A. Brewer. Why Events Are a Bad Idea (for High-Concurrency Servers). HotOS. 2003.

[54] Werner-Allen, G., Johnson, J., Ruiz, M., Lees, J., & Welsh, M. (2005). Monitoring volcanic eruptions with a wireless sensor network. *Proceedings of the Second European Workshop on Wireless Sensor Networks, 2005.*, 108–120. https://doi.org/10.1109/ewsn.2005.1462003

[55] Wikipedia contributors. (n.d.). *Binary tree*. Wikipedia. Retrieved April 11, 2021, from https://en.wikipedia.org/wiki/Binary_tree#Arrays

[56] Wilson, P. R., Johnstone, M. S., Neely, M., & Boles, D. (1995). Dynamic storage allocation: A survey and critical review. *Memory Management*, 1–116. https://doi.org/10.1007/3-540-60368-9_19

[57] Xu, N. (2002). A survey of sensor network applications. *IEEE communications magazine*, *40*(8), 102-114.

[58] Z-Wave: "Z-Wave Protocol Overview", v. 4, May 2007.

[59] Zagieboylo, D., Suh, G. E., & Myers, A. C. (2020). The cost of software-based memory management without virtual memory. https://arxiv.org/abs/2009.06789

[60] Zhang, P., Sadler, C. M., Lyon, S. A., & Martonosi, M. (2004). Hardware design experiences in ZebraNet. *Proceedings of the 2nd international conference on Embedded networked sensor systems* (*SenSys '04*). Association for Computing Machinery, New York, NY, USA, 227–238. https://doi.org/10.1145/1031495.1031522

[61] ZigBee Alliance: "ZigBee Home Automation Public Application Profile", revision 25, v. 1.0, Oct. 2007.