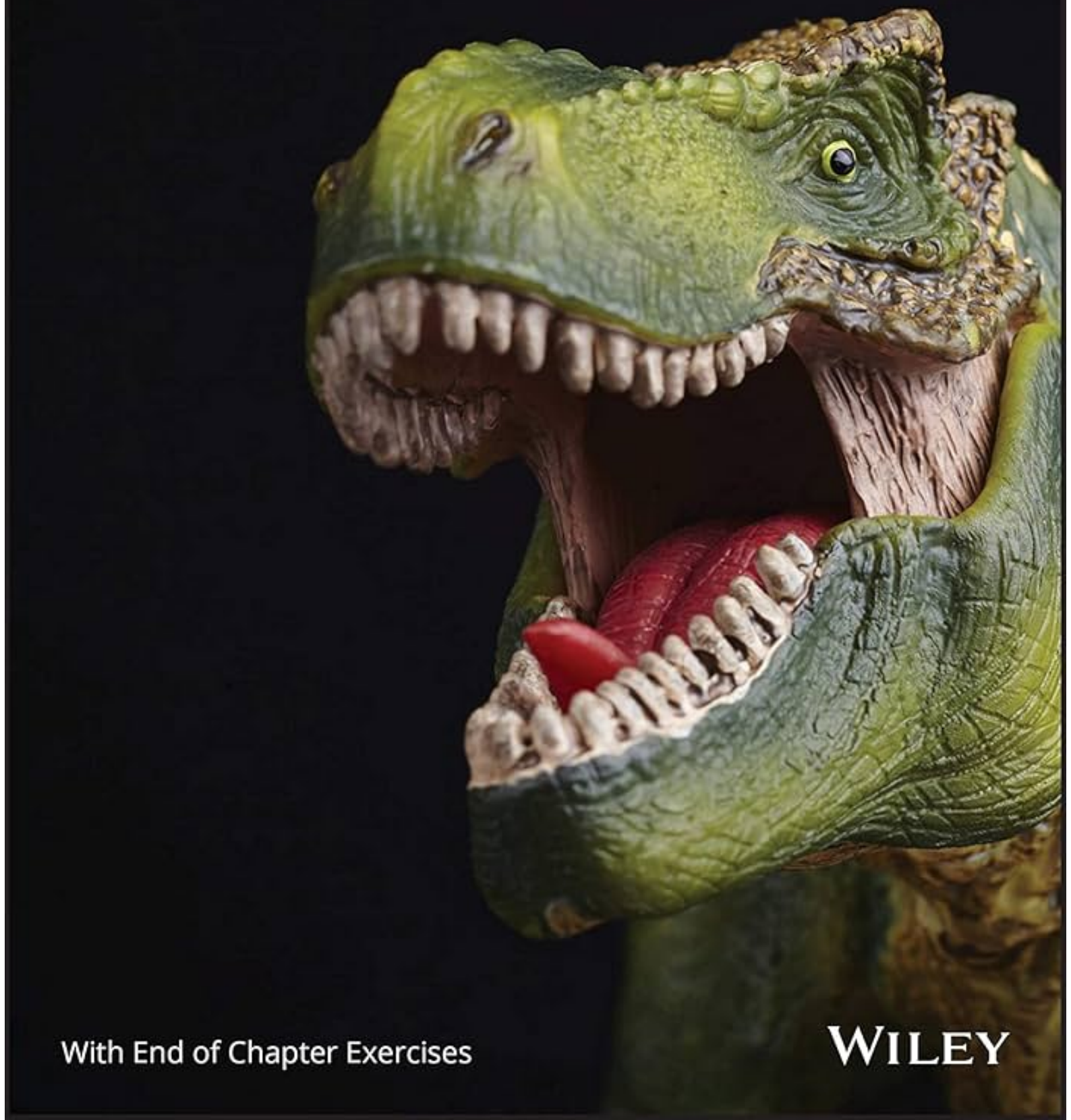
Wiley Loose-Leaf Print Edition

Operating System Concepts

TENTH EDITION

ABRAHAM SILBERSCHATZ • PETER BAER GALVIN • GREG GAGNE



With End of Chapter Exercises

WILEY

by the many-to-many and two-level models discussed in Section 4.3.3. Such coordination allows the number of kernel threads to be dynamically adjusted to help ensure the best performance.

Many systems implementing either the many-to-many or the two-level model place an intermediate data structure between the user and kernel threads. This data structure—typically known as a **lightweight process**, or **LWP**—is shown in Figure 4.20. To the user-thread library, the LWP appears to be a virtual processor on which the application can schedule a user thread to run. Each LWP is attached to a kernel thread, and it is kernel threads that the operating system schedules to run on physical processors. If a kernel thread blocks (such as while waiting for an I/O operation to complete), the LWP blocks as well. Up the chain, the user-level thread attached to the LWP also blocks.

An application may require any number of LWPs to run efficiently. Consider a CPU-bound application running on a single processor. In this scenario, only one thread can run at a time, so one LWP is sufficient. An application that is I/O-intensive may require multiple LWPs to execute, however. Typically, an LWP is required for each concurrent blocking system call. Suppose, for example, that five different file-read requests occur simultaneously. Five LWPs are needed, because all could be waiting for I/O completion in the kernel. If a process has only four LWPs, then the fifth request must wait for one of the LWPs to return from the kernel.

One scheme for communication between the user-thread library and the kernel is known as **scheduler activation**. It works as follows: The kernel provides an application with a set of virtual processors (LWPs), and the application can schedule user threads onto an available virtual processor. Furthermore, the kernel must inform an application about certain events. This procedure is known as an **upcall**. Upcalls are handled by the thread library with an **upcall handler**, and upcall handlers must run on a virtual processor.

One event that triggers an upcall occurs when an application thread is about to block. In this scenario, the kernel makes an upcall to the application informing it that a thread is about to block and identifying the specific thread. The kernel then allocates a new virtual processor to the application. The application runs an upcall handler on this new virtual processor, which saves the

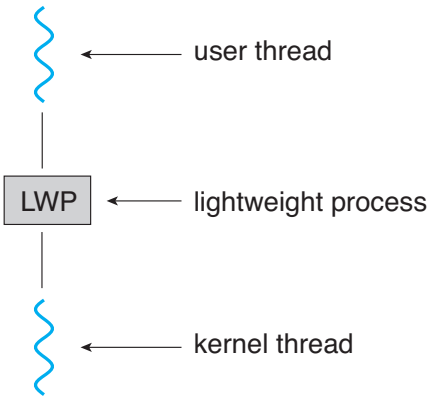


Figure 4.20 Lightweight process (LWP).

state of the blocking thread and relinquishes the virtual processor on which the blocking thread is running. The upcall handler then schedules another thread that is eligible to run on the new virtual processor. When the event that the blocking thread was waiting for occurs, the kernel makes another upcall to the thread library informing it that the previously blocked thread is now eligible to run. The upcall handler for this event also requires a virtual processor, and the kernel may allocate a new virtual processor or preempt one of the user threads and run the upcall handler on its virtual processor. After marking the unblocked thread as eligible to run, the application schedules an eligible thread to run on an available virtual processor.

4.7 Operating-System Examples

At this point, we have examined a number of concepts and issues related to threads. We conclude the chapter by exploring how threads are implemented in Windows and Linux systems.

4.7.1 Windows Threads

A Windows application runs as a separate process, and each process may contain one or more threads. The Windows API for creating threads is covered in Section 4.4.2. Additionally, Windows uses the one-to-one mapping described in Section 4.3.2, where each user-level thread maps to an associated kernel thread.

The general components of a thread include:

- A thread ID uniquely identifying the thread
- A register set representing the status of the processor
- A program counter
- A user stack, employed when the thread is running in user mode, and a kernel stack, employed when the thread is running in kernel mode
- A private storage area used by various run-time libraries and dynamic link libraries (DLLs)

The register set, stacks, and private storage area are known as the **context** of the thread.

The primary data structures of a thread include:

- ETHREAD—executive thread block
- KTHREAD—kernel thread block
- TEB—thread environment block

The key components of the ETHREAD include a pointer to the process to which the thread belongs and the address of the routine in which the thread starts control. The ETHREAD also contains a pointer to the corresponding KTHREAD.

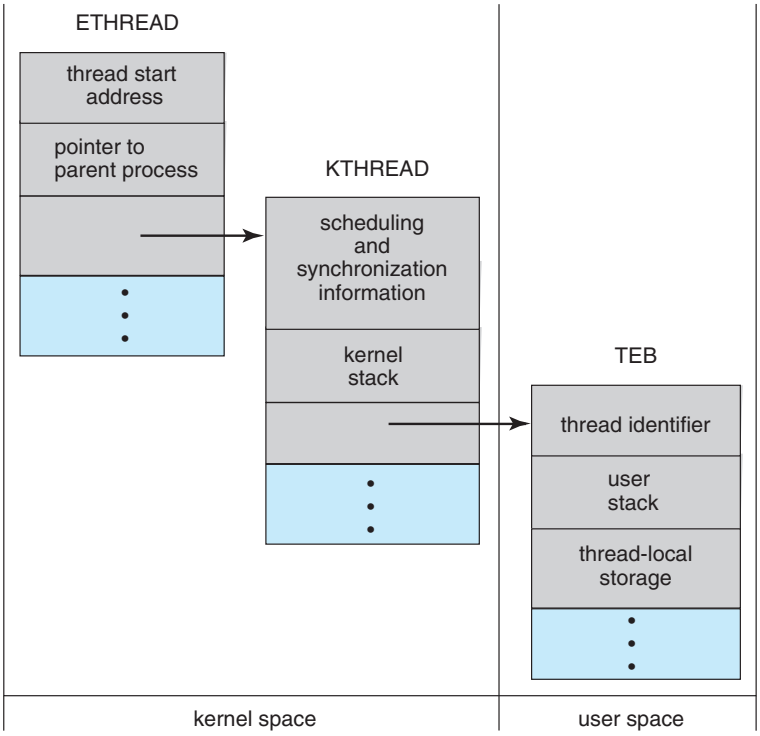


Figure 4.21 Data structures of a Windows thread.

The KTHREAD includes scheduling and synchronization information for the thread. In addition, the KTHREAD includes the kernel stack (used when the thread is running in kernel mode) and a pointer to the TEB.

The ETHREAD and the KTHREAD exist entirely in kernel space; this means that only the kernel can access them. The TEB is a user-space data structure that is accessed when the thread is running in user mode. Among other fields, the TEB contains the thread identifier, a user-mode stack, and an array for thread-local storage. The structure of a Windows thread is illustrated in Figure 4.21.

4.7.2 Linux Threads

Linux provides the `fork()` system call with the traditional functionality of duplicating a process, as described in Chapter 3. Linux also provides the ability to create threads using the `clone()` system call. However, Linux does not distinguish between processes and threads. In fact, Linux uses the term ***task*** —rather than ***process*** or ***thread***— when referring to a flow of control within a program.

When `clone()` is invoked, it is passed a set of flags that determine how much sharing is to take place between the parent and child tasks. Some of these flags are listed in Figure 4.22. For example, suppose that `clone()` is passed the flags `CLONE_FS`, `CLONE_VM`, `CLONE_SIGHAND`, and `CLONE_FILES`. The parent and child tasks will then share the same file-system information (such as the current working directory), the same memory space, the same signal handlers,

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Figure 4.22 Some of the flags passed when `clone()` is invoked.

and the same set of open files. Using `clone()` in this fashion is equivalent to creating a thread as described in this chapter, since the parent task shares most of its resources with its child task. However, if none of these flags is set when `clone()` is invoked, no sharing takes place, resulting in functionality similar to that provided by the `fork()` system call.

The varying level of sharing is possible because of the way a task is represented in the Linux kernel. A unique kernel data structure (specifically, `struct task_struct`) exists for each task in the system. This data structure, instead of storing data for the task, contains pointers to other data structures where these data are stored—for example, data structures that represent the list of open files, signal-handling information, and virtual memory. When `fork()` is invoked, a new task is created, along with a ***copy*** of all the associated data structures of the parent process. A new task is also created when the `clone()` system call is made. However, rather than copying all data structures, the new task ***points*** to the data structures of the parent task, depending on the set of flags passed to `clone()`.

Finally, the flexibility of the `clone()` system call can be extended to the concept of containers, a virtualization topic which was introduced in Chapter 1. Recall from that chapter that a container is a virtualization technique provided by the operating system that allows creating multiple Linux systems (containers) under a single Linux kernel that run in isolation to one another. Just as certain flags passed to `clone()` can distinguish between creating a task that behaves more like a process or a thread based upon the amount of sharing between the parent and child tasks, there are other flags that can be passed to `clone()` that allow a Linux container to be created. Containers will be covered more fully in Chapter 18.

4.8 Summary

- A thread represents a basic unit of CPU utilization, and threads belonging to the same process share many of the process resources, including code and data.
- There are four primary benefits to multithreaded applications: (1) responsiveness, (2) resource sharing, (3) economy, and (4) scalability.
- Concurrency exists when multiple threads are making progress, whereas parallelism exists when multiple threads are making progress simulta-

neously. On a system with a single CPU, only concurrency is possible; parallelism requires a multicore system that provides multiple CPUs.

- There are several challenges in designing multithreaded applications. They include dividing and balancing the work, dividing the data between the different threads, and identifying any data dependencies. Finally, multithreaded programs are especially challenging to test and debug.
- Data parallelism distributes subsets of the same data across different computing cores and performs the same operation on each core. Task parallelism distributes not data but tasks across multiple cores. Each task is running a unique operation.
- User applications create user-level threads, which must ultimately be mapped to kernel threads to execute on a CPU. The many-to-one model maps many user-level threads to one kernel thread. Other approaches include the one-to-one and many-to-many models.
- A thread library provides an API for creating and managing threads. Three common thread libraries include Windows, Pthreads, and Java threading. Windows is for the Windows system only, while Pthreads is available for POSIX-compatible systems such as UNIX, Linux, and macOS. Java threads will run on any system that supports a Java virtual machine.
- Implicit threading involves identifying tasks—not threads—and allowing languages or API frameworks to create and manage threads. There are several approaches to implicit threading, including thread pools, fork-join frameworks, and Grand Central Dispatch. Implicit threading is becoming an increasingly common technique for programmers to use in developing concurrent and parallel applications.
- Threads may be terminated using either asynchronous or deferred cancellation. Asynchronous cancellation stops a thread immediately, even if it is in the middle of performing an update. Deferred cancellation informs a thread that it should terminate but allows the thread to terminate in an orderly fashion. In most circumstances, deferred cancellation is preferred to asynchronous termination.
- Unlike many other operating systems, Linux does not distinguish between processes and threads; instead, it refers to each as a task. The Linux `clone()` system call can be used to create tasks that behave either more like processes or more like threads.

Practice Exercises

4.1 Provide three programming examples in which multithreading provides better performance than a single-threaded solution.

4.2 Using Amdahl's Law, calculate the speedup gain of an application that has a 60 percent parallel component for (a) two processing cores and (b) four processing cores.

4.3 Does the multithreaded web server described in Section 4.1 exhibit task or data parallelism?

4.4 What are two differences between user-level threads and kernel-level threads? Under what circumstances is one type better than the other?

4.5 Describe the actions taken by a kernel to context-switch between kernel-level threads.

4.6 What resources are used when a thread is created? How do they differ from those used when a process is created?

4.7 Assume that an operating system maps user-level threads to the kernel using the many-to-many model and that the mapping is done through LWPs. Furthermore, the system allows developers to create real-time threads for use in real-time systems. Is it necessary to bind a real-time thread to an LWP? Explain.

Further Reading

[Vahalia (1996)] covers threading in several versions of UNIX. [McDougall and Mauro (2007)] describes developments in threading the Solaris kernel. [Russinovich et al. (2017)] discuss threading in the Windows operating system family. [Mauerer (2008)] and [Love (2010)] explain how Linux handles threading, and [Levin (2013)] covers threads in macOS and iOS. [Herlihy and Shavit (2012)] covers parallelism issues on multicore systems. [Aubanel (2017)] covers parallelism of several different algorithms.

Bibliography

[Aubanel (2017)] E. Aubanel, *Elements of Parallel Computing*, CRC Press (2017).

[Herlihy and Shavit (2012)] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*, Revised First Edition, Morgan Kaufmann Publishers Inc. (2012).

[Levin (2013)] J. Levin, *Mac OS X and iOS Internals to the Apple's Core*, Wiley (2013).

[Love (2010)] R. Love, *Linux Kernel Development*, Third Edition, Developer's Library (2010).

[Mauerer (2008)] W. Mauerer, *Professional Linux Kernel Architecture*, John Wiley and Sons (2008).

[McDougall and Mauro (2007)] R. McDougall and J. Mauro, *Solaris Internals*, Second Edition, Prentice Hall (2007).

[Russinovich et al. (2017)] M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals–Part 1*, Seventh Edition, Microsoft Press (2017).

[Vahalia (1996)] U. Vahalia, *Unix Internals: The New Frontiers*, Prentice Hall (1996).

Chapter 4 Exercises

4.8 Provide two programming examples in which multithreading does ***not*** provide better performance than a single-threaded solution.

4.9 Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?

4.10 Which of the following components of program state are shared across threads in a multithreaded process?

a. Register values

b. Heap memory

c. Global variables

d. Stack memory

4.11 Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system? Explain.

4.12 In Chapter 3, we discussed Google's Chrome browser and its practice of opening each new tab in a separate process. Would the same benefits have been achieved if, instead, Chrome had been designed to open each new tab in a separate thread? Explain.

4.13 Is it possible to have concurrency but not parallelism? Explain.

4.14 Using Amdahl's Law, calculate the speedup gain for the following applications:

- 40 percent parallel with (a) eight processing cores and (b) sixteen processing cores
- 67 percent parallel with (a) two processing cores and (b) four processing cores
- 90 percent parallel with (a) four processing cores and (b) eight processing cores

4.15 Determine if the following problems exhibit task or data parallelism:

- Using a separate thread to generate a thumbnail for each photo in a collection
- Transposing a matrix in parallel
- A networked application where one thread reads from the network and another writes to the network
- The fork-join array summation application described in Section 4.5.2
- The Grand Central Dispatch system

4.16 A system with two dual-core processors has four processors available for scheduling. A CPU-intensive application is running on this system. All input is performed at program start-up, when a single file must be

opened. Similarly, all output is performed just before the program terminates, when the program results must be written to a single file. Between start-up and termination, the program is entirely CPU-bound. Your task is to improve the performance of this application by multithreading it. The application runs on a system that uses the one-to-one threading model (each user thread maps to a kernel thread).

- How many threads will you create to perform the input and output? Explain.
- How many threads will you create for the CPU-intensive portion of the application? Explain.

4.17 Consider the following code segment:

```
pid_t pid;

pid = fork();
if (pid == 0) { /* child process */
   fork();
   thread_create( . . .);
}
fork();
```

a. How many unique processes are created?

b. How many unique threads are created?

4.18 As described in Section 4.7.2, Linux does not distinguish between processes and threads. Instead, Linux treats both in the same way, allowing a task to be more akin to a process or a thread depending on the set of flags passed to the `clone()` system call. However, other operating systems, such as Windows, treat processes and threads differently. Typically, such systems use a notation in which the data structure for a process contains pointers to the separate threads belonging to the process. Contrast these two approaches for modeling processes and threads within the kernel.

4.19 The program shown in Figure 4.23 uses the Pthreads API. What would be the output from the program at `LINE C` and `LINE P`?

4.20 Consider a multicore system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads in the program be greater than the number of processing cores in the system. Discuss the performance implications of the following scenarios.

a. The number of kernel threads allocated to the program is less than the number of processing cores.

b. The number of kernel threads allocated to the program is equal to the number of processing cores.

c. The number of kernel threads allocated to the program is greater than the number of processing cores but less than the number of user-level threads.

```
#include <pthread.h>
#include <stdio.h>

int value = 0;
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
pid_t pid;
pthread_t tid;
pthread_attr_t attr;

   pid = fork();

   if (pid == 0) { /* child process */
      pthread_attr_init(&attr);
      pthread_create(&tid,&attr,runner,NULL);
      pthread_join(tid,NULL);
      printf("CHILD: value = %d",value); /* LINE C */
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("PARENT: value = %d",value); /* LINE P */
   }
}

void *runner(void *param) {
   value = 5;
   pthread_exit(0);
}
```

Figure 4.22 C program for Exercise 4.19.

4.21 Pthreads provides an API for managing thread cancellation. The `pthread_setcancelstate()` function is used to set the cancellation state. Its prototype appears as follows:

```
pthread_setcancelstate(int state, int *oldstate)
```

The two possible values for the state are `PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DISABLE`.

Using the code segment shown in Figure 4.24, provide examples of two operations that would be suitable to perform between the calls to disable and enable thread cancellation.

```
int oldstate;

pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);

/* What operations would be performed here? */

pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);
```

Figure 4.23 C program for Exercise 4.21.

Programming Problems

4.22 Write a multithreaded program that calculates various statistical values for a list of numbers. This program will be passed a series of numbers on the command line and will then create three separate worker threads. One thread will determine the average of the numbers, the second will determine the maximum value, and the third will determine the minimum value. For example, suppose your program is passed the integers

90 81 78 95 79 72 85

The program will report

```
The average value is 82
The minimum value is 72
The maximum value is 95
```

The variables representing the average, minimum, and maximum values will be stored globally. The worker threads will set these values, and the parent thread will output the values once the workers have exited. (We could obviously expand this program by creating additional threads that determine other statistical values, such as median and standard deviation.)

4.23 Write a multithreaded program that outputs prime numbers. This program should work as follows: The user will run the program and will enter a number on the command line. The program will then create a separate thread that outputs all the prime numbers less than or equal to the number entered by the user.

4.24 An interesting way of calculating π is to use a technique known as ***Monte Carlo***, which involves randomization. This technique works as follows:

Suppose you have a circle inscribed within a square, as shown in Figure 4.25. (Assume that the radius of this circle is 1.)

- First, generate a series of random points as simple (x, y) coordinates. These points must fall within the Cartesian coordinates that bound the square. Of the total number of random points that are generated, some will occur within the circle.
- Next, estimate π by performing the following calculation:

$$\pi = 4 \times \textit{(number of points in circle)} \ / \ \textit{(total number of points)}$$

Write a multithreaded version of this algorithm that creates a separate thread to generate a number of random points. The thread will count the number of points that occur within the circle and store that result in a global variable. When this thread has exited, the parent thread will calculate and output the estimated value of π. It is worth experimenting with the number of random points generated. As a general rule, the greater the number of points, the closer the approximation to π.

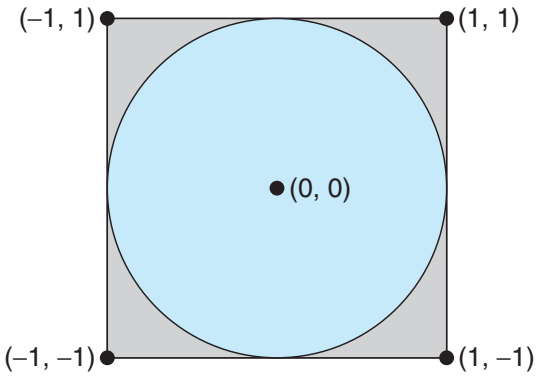


Figure 4.25 Monte Carlo technique for calculating π.

In the source-code download for this text, you will find a sample program that provides a technique for generating random numbers, as well as determining if the random (x, y) point occurs within the circle.

Readers interested in the details of the Monte Carlo method for estimating π should consult the bibliography at the end of this chapter. In Chapter 6, we modify this exercise using relevant material from that chapter.

4.25 Repeat Exercise 4.24, but instead of using a separate thread to generate random points, use OpenMP to parallelize the generation of points. Be careful not to place the calculation of π in the parallel region, since you want to calculate π only once.

4.26 Modify the socket-based date server (Figure 3.27) in Chapter 3 so that the server services each client request in a separate thread.

4.27 The Fibonacci sequence is the series of numbers $0, 1, 1, 2, 3, 5, 8, \ldots$. Formally, it can be expressed as:

$$fib_0 = 0$$
$$fib_1 = 1$$
$$fib_n = fib_{n-1} + fib_{n-2}$$

Write a multithreaded program that generates the Fibonacci sequence. This program should work as follows: On the command line, the user will enter the number of Fibonacci numbers that the program is to generate. The program will then create a separate thread that will generate the Fibonacci numbers, placing the sequence in data that can be shared by the threads (an array is probably the most convenient data structure). When the thread finishes execution, the parent thread will output the sequence generated by the child thread. Because the parent thread cannot begin outputting the Fibonacci sequence until the child thread finishes, the parent thread will have to wait for the child thread to finish. Use the techniques described in Section 4.4 to meet this requirement.

4.28 Modify programming problem Exercise 3.20 from Chapter 3, which asks you to design a pid manager. This modification will consist of writing a

multithreaded program that tests your solution to Exercise 3.20. You will create a number of threads—for example, 100—and each thread will request a pid, sleep for a random period of time, and then release the pid. (Sleeping for a random period of time approximates the typical pid usage in which a pid is assigned to a new process, the process executes and then terminates, and the pid is released on the process's termination.) On UNIX and Linux systems, sleeping is accomplished through the `sleep()` function, which is passed an integer value representing the number of seconds to sleep. This problem will be modified in Chapter 7.

4.29 Exercise 3.25 in Chapter 3 involves designing an echo server using the Java threading API. This server is single-threaded, meaning that the server cannot respond to concurrent echo clients until the current client exits. Modify the solution to Exercise 3.25 so that the echo server services each client in a separate request.

Programming Projects

Project 1—Sudoku Solution Validator

A ***Sudoku*** puzzle uses a 9×9 grid in which each column and row, as well as each of the nine 3×3 subgrids, must contain all of the digits $1 \cdots 9$. Figure 4.26 presents an example of a valid Sudoku puzzle. This project consists of designing a multithreaded application that determines whether the solution to a Sudoku puzzle is valid.

There are several different ways of multithreading this application. One suggested strategy is to create threads that check the following criteria:

- A thread to check that each column contains the digits 1 through 9
- A thread to check that each row contains the digits 1 through 9

6	2	4	5	3	9	1	8	7
5	1	9	7	2	8	6	3	4
8	3	7	6	1	4	2	9	5
1	4	3	8	6	5	7	2	9
9	5	8	2	4	7	3	6	1
7	6	2	3	9	1	4	5	8
3	7	1	9	5	6	8	4	2
4	9	6	1	8	2	5	7	3
2	8	5	4	7	3	9	1	6

Figure 4.26 Solution to a 9×9 Sudoku puzzle.

- Nine threads to check that each of the 3×3 subgrids contains the digits 1 through 9

This would result in a total of eleven separate threads for validating a Sudoku puzzle. However, you are welcome to create even more threads for this project. For example, rather than creating one thread that checks all nine columns, you could create nine separate threads and have each of them check one column.

I. Passing Parameters to Each Thread

The parent thread will create the worker threads, passing each worker the location that it must check in the Sudoku grid. This step will require passing several parameters to each thread. The easiest approach is to create a data structure using a `struct`. For example, a structure to pass the row and column where a thread must begin validating would appear as follows:

```
/* structure for passing data to threads */
typedef struct
{
   int row;
   int column;
} parameters;
```

Both Pthreads and Windows programs will create worker threads using a strategy similar to that shown below:

```
parameters *data = (parameters *) malloc(sizeof(parameters));
data->row = 1;
data->column = 1;
/* Now create the thread passing it data as a parameter */
```

The `data` pointer will be passed to either the `pthread_create()` (Pthreads) function or the `CreateThread()` (Windows) function, which in turn will pass it as a parameter to the function that is to run as a separate thread.

II. Returning Results to the Parent Thread

Each worker thread is assigned the task of determining the validity of a particular region of the Sudoku puzzle. Once a worker has performed this check, it must pass its results back to the parent. One good way to handle this is to create an array of integer values that is visible to each thread. The i^{th} index in this array corresponds to the i^{th} worker thread. If a worker sets its corresponding value to 1, it is indicating that its region of the Sudoku puzzle is valid. A value of 0 indicates otherwise. When all worker threads have completed, the parent thread checks each entry in the result array to determine if the Sudoku puzzle is valid.

Project 2—Multithreaded Sorting Application

Write a multithreaded sorting program that works as follows: A list of integers is divided into two smaller lists of equal size. Two separate threads (which we

will term ***sorting threads***) sort each sublist using a sorting algorithm of your choice. The two sublists are then merged by a third thread—a ***merging thread*** —which merges the two sublists into a single sorted list.

Because global data are shared across all threads, perhaps the easiest way to set up the data is to create a global array. Each sorting thread will work on one half of this array. A second global array of the same size as the unsorted integer array will also be established. The merging thread will then merge the two sublists into this second array. Graphically, this program is structured as in Figure 4.27.

This programming project will require passing parameters to each of the sorting threads. In particular, it will be necessary to identify the starting index from which each thread is to begin sorting. Refer to the instructions in Project 1 for details on passing parameters to a thread.

The parent thread will output the sorted array once all sorting threads have exited.

Project 3—Fork-Join Sorting Application

Implement the preceding project (Multithreaded Sorting Application) using Java's fork-join parallelism API. This project will be developed in two different versions. Each version will implement a different divide-and-conquer sorting algorithm:

1. Quicksort
2. Mergesort

The Quicksort implementation will use the Quicksort algorithm for dividing the list of elements to be sorted into a *left half* and a *right half* based on the

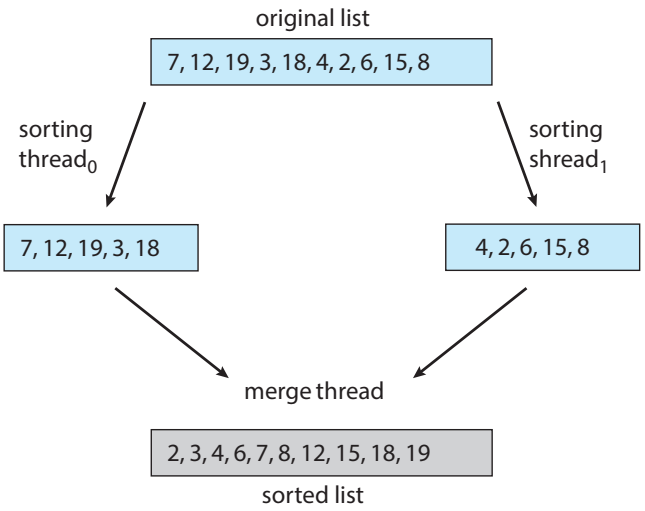


Figure 4.27 Multithreaded sorting.

position of the pivot value. The Mergesort algorithm will divide the list into two evenly sized halves. For both the Quicksort and Mergesort algorithms, when the list to be sorted falls within some threshold value (for example, the list is size 100 or fewer), directly apply a simple algorithm such as the Selection or Insertion sort. Most data structures texts describe these two well-known, divide-and-conquer sorting algorithms.

The class `SumTask` shown in Section 4.5.2.1 extends `RecursiveTask`, which is a result-bearing `ForkJoinTask`. As this assignment will involve sorting the array that is passed to the task, but not returning any values, you will instead create a class that extends `RecursiveAction`, a non result-bearing `ForkJoinTask` (see Figure 4.19).

The objects passed to each sorting algorithm are required to implement Java's `Comparable` interface, and this will need to be reflected in the class definition for each sorting algorithm. The source code download for this text includes Java code that provides the foundations for beginning this project.

CPU Scheduling

CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive. In this chapter, we introduce basic CPU-scheduling concepts and present several CPU-scheduling algorithms, including real-time systems. We also consider the problem of selecting an algorithm for a particular system.

In Chapter 4, we introduced threads to the process model. On modern operating systems it is kernel-level threads—not processes—that are in fact being scheduled by the operating system. However, the terms "process scheduling" and "thread scheduling" are often used interchangeably. In this chapter, we use ***process scheduling*** when discussing general scheduling concepts and ***thread scheduling*** to refer to thread-specific ideas.

Similarly, in Chapter 1 we describe how a ***core*** is the basic computational unit of a CPU, and that a process executes on a CPU's core. However, in many instances in this chapter, when we use the general terminology of scheduling a process to "run on a CPU", we are implying that the process is running on a CPU's core.

CHAPTER OBJECTIVES

- Describe various CPU scheduling algorithms.
- Assess CPU scheduling algorithms based on scheduling criteria.
- Explain the issues related to multiprocessor and multicore scheduling.
- Describe various real-time scheduling algorithms.
- Describe the scheduling algorithms used in the Windows, Linux, and Solaris operating systems.
- Apply modeling and simulations to evaluate CPU scheduling algorithms.
- Design a program that implements several different CPU scheduling algorithms.

5.1 Basic Concepts

In a system with a single CPU core, only one process can run at a time. Others must wait until the CPU's core is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The idea is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU. On a multicore system, this concept of keeping the CPU busy is extended to all processing cores on the system.

Scheduling of this kind is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.

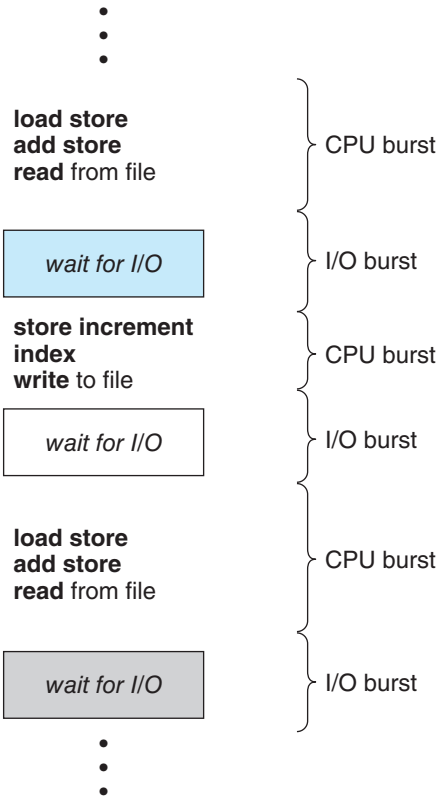


Figure 5.1 Alternating sequence of CPU and I/O bursts.

5.1.1 CPU–I/O Burst Cycle

The success of CPU scheduling depends on an observed property of processes: process execution consists of a **cycle** of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a **CPU burst**. That is followed by an **I/O burst**, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution (Figure 5.1).

The durations of CPU bursts have been measured extensively. Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar to that shown in Figure 5.2. The curve is generally characterized as exponential or hyperexponential, with a large number of short CPU bursts and a small number of long CPU bursts. An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts. This distribution can be important when implementing a CPU-scheduling algorithm.

5.1.2 CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the **CPU scheduler**, which selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue. As we shall see when we consider the various scheduling algorithms, a ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.

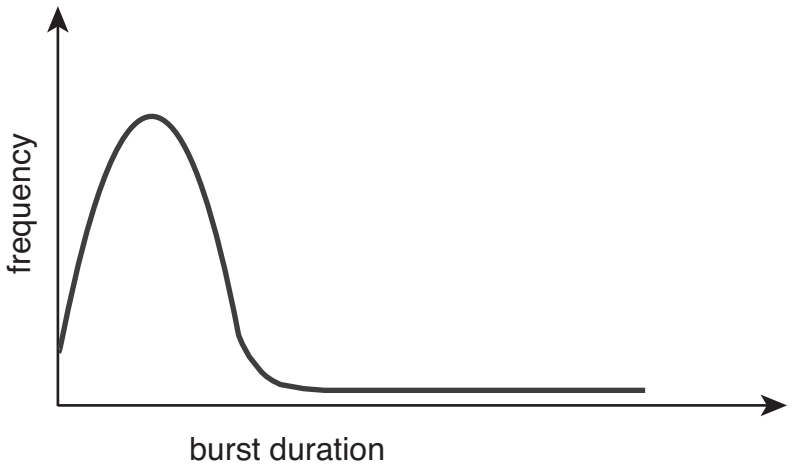


Figure 5.2 Histogram of CPU-burst durations.

5.1.3 Preemptive and Nonpreemptive Scheduling

CPU-scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of `wait()` for the termination of a child process)
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)
4. When a process terminates

For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **nonpreemptive** or **cooperative**. Otherwise, it is **preemptive**. Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state. Virtually all modern operating systems including Windows, macOS, Linux, and UNIX use preemptive scheduling algorithms.

Unfortunately, preemptive scheduling can result in race conditions when data are shared among several processes. Consider the case of two processes that share data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state. This issue will be explored in detail in Chapter 6.

Preemption also affects the design of the operating-system kernel. During the processing of a system call, the kernel may be busy with an activity on behalf of a process. Such activities may involve changing important kernel data (for instance, I/O queues). What happens if the process is preempted in the middle of these changes and the kernel (or the device driver) needs to read or modify the same structure? Chaos ensues. As will be discussed in Section 6.2, operating-system kernels can be designed as either nonpreemptive or preemptive. A nonpreemptive kernel will wait for a system call to complete or for a process to block while waiting for I/O to complete to take place before doing a context switch. This scheme ensures that the kernel structure is simple, since the kernel will not preempt a process while the kernel data structures are in an inconsistent state. Unfortunately, this kernel-execution model is a poor one for supporting real-time computing, where tasks must complete execution within a given time frame. In Section 5.6, we explore scheduling demands of real-time systems. A preemptive kernel requires mechanisms such as mutex locks to prevent race conditions when accessing shared kernel data structures. Most modern operating systems are now fully preemptive when running in kernel mode.

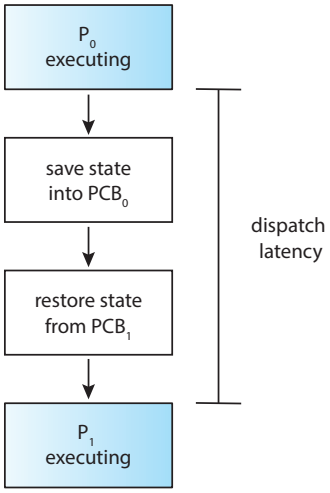


Figure 5.3 The role of the dispatcher.

Because interrupts can, by definition, occur at any time, and because they cannot always be ignored by the kernel, the sections of code affected by interrupts must be guarded from simultaneous use. The operating system needs to accept interrupts at almost all times. Otherwise, input might be lost or output overwritten. So that these sections of code are not accessed concurrently by several processes, they disable interrupts at entry and reenable interrupts at exit. It is important to note that sections of code that disable interrupts do not occur very often and typically contain few instructions.

5.1.4 Dispatcher

Another component involved in the CPU-scheduling function is the **dispatcher**. The dispatcher is the module that gives control of the CPU's core to the process selected by the CPU scheduler. This function involves the following:

- Switching context from one process to another
- Switching to user mode
- Jumping to the proper location in the user program to resume that program

The dispatcher should be as fast as possible, since it is invoked during every context switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency** and is illustrated in Figure 5.3.

An interesting question to consider is, how often do context switches occur? On a system-wide level, the number of context switches can be obtained by using the `vmstat` command that is available on Linux systems. Below is the output (which has been trimmed) from the command

```
vmstat 1 3
```

This command provides 3 lines of output over a 1-second delay:

```
------cpu-----
24
225
339
```

The first line gives the average number of context switches over 1 second since the system booted, and the next two lines give the number of context switches over two 1-second intervals. Since this machine booted, it has averaged 24 context switches per second. And in the past second, 225 context switches were made, with 339 context switches in the second prior to that.

We can also use the `/proc` file system to determine the number of context switches for a given process. For example, the contents of the file `/proc/2166/status` will list various statistics for the process with `pid = 2166`. The command

```
cat /proc/2166/status
```

provides the following trimmed output:

```
voluntary_ctxt_switches          150
nonvoluntary_ctxt_switches       8
```

This output shows the number of context switches over the lifetime of the process. Notice the distinction between ***voluntary*** and ***nonvoluntary*** context switches. A voluntary context switch occurs when a process has given up control of the CPU because it requires a resource that is currently unavailable (such as blocking for I/O.) A nonvoluntary context switch occurs when the CPU has been taken away from a process, such as when its time slice has expired or it has been preempted by a higher-priority process.

5.2 Scheduling Criteria

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

Many criteria have been suggested for comparing CPU-scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

- **CPU utilization**. We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system). (CPU utilization can be obtained by using the `top` command on Linux, macOS, and UNIX systems.)
- **Throughput**. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed

per time unit, called **throughput**. For long processes, this rate may be one process over several seconds; for short transactions, it may be tens of processes per second.

- **Turnaround time**. From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting time**. The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
- **Response time**. In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. However, under some circumstances, we prefer to optimize the minimum or maximum values rather than the average. For example, to guarantee that all users get good service, we may want to minimize the maximum response time.

Investigators have suggested that, for interactive systems (such as a PC desktop or laptop system), it is more important to minimize the variance in the response time than to minimize the average response time. A system with reasonable and predictable response time may be considered more desirable than a system that is faster on the average but is highly variable. However, little work has been done on CPU-scheduling algorithms that minimize variance.

As we discuss various CPU-scheduling algorithms in the following section, we illustrate their operation. An accurate illustration should involve many processes, each a sequence of several hundred CPU bursts and I/O bursts. For simplicity, though, we consider only one CPU burst (in milliseconds) per process in our examples. Our measure of comparison is the average waiting time. More elaborate evaluation mechanisms are discussed in Section 5.8.

5.3 Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU's core. There are many different CPU-scheduling algorithms. In this section, we describe several of them. Although most modern CPU architectures have multiple processing cores, we describe these scheduling algorithms in the context of only one processing core available. That is, a single CPU that has a single processing core, thus the system is

capable of only running one process at a time. In Section 5.5 we discuss CPU scheduling in the context of multiprocessor systems.

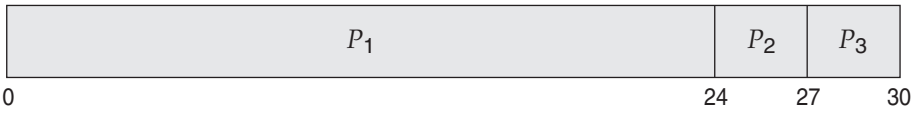
5.3.1 First-Come, First-Served Scheduling

By far the simplest CPU-scheduling algorithm is the **first-come first-serve** (**FCFS**) scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand.

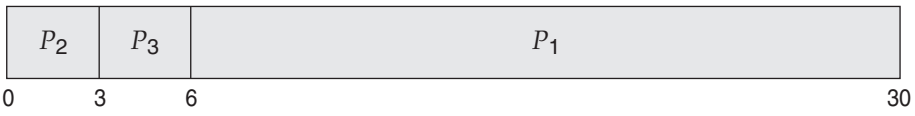
On the negative side, the average waiting time under the FCFS policy is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If the processes arrive in the order P_1, P_2, P_3, and are served in FCFS order, we get the result shown in the following **Gantt chart**, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:



The waiting time is 0 milliseconds for process P_1, 24 milliseconds for process P_2, and 27 milliseconds for process P_3. Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds. If the processes arrive in the order P_2, P_3, P_1, however, the results will be as shown in the following Gantt chart:



The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly.

In addition, consider the performance of FCFS scheduling in a dynamic situation. Assume we have one CPU-bound process and many I/O-bound processes. As the processes flow around the system, the following scenario may result. The CPU-bound process will get and hold the CPU. During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU. While the processes wait in the ready queue, the I/O

devices are idle. Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues. At this point, the CPU sits idle. The CPU-bound process will then move back to the ready queue and be allocated the CPU. Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done. There is a **convoy effect** as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

Note also that the FCFS scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. The FCFS algorithm is thus particularly troublesome for interactive systems, where it is important that each process get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

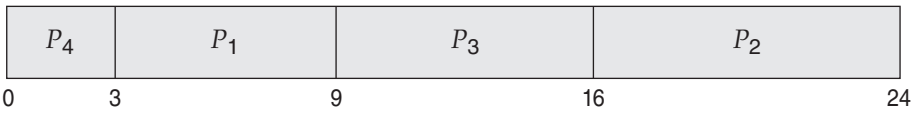
5.3.2 Shortest-Job-First Scheduling

A different approach to CPU scheduling is the **shortest-job-firs** (**SJF**) scheduling algorithm. This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. Note that a more appropriate term for this scheduling method would be the ***shortest-next-CPU-burst*** algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length. We use the term SJF because most people and textbooks use this term to refer to this type of scheduling.

As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:



The waiting time is 3 milliseconds for process P_1, 16 milliseconds for process P_2, 9 milliseconds for process P_3, and 0 milliseconds for process P_4. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes. Moving a short process

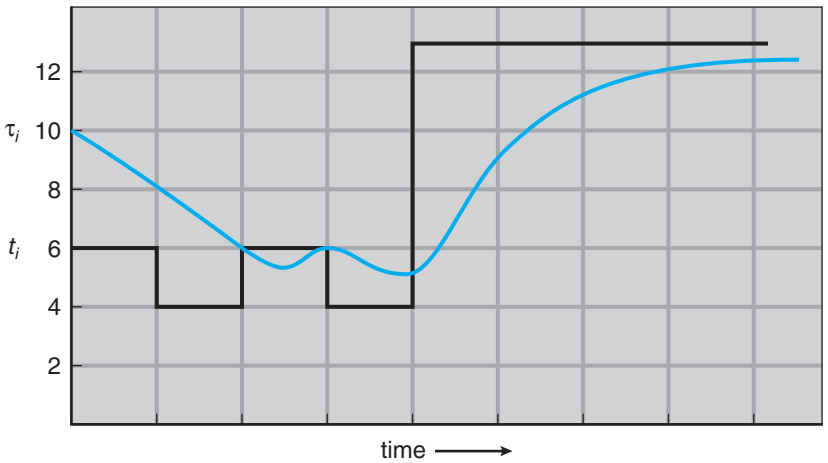
before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

Although the SJF algorithm is optimal, it cannot be implemented at the level of CPU scheduling, as there is no way to know the length of the next CPU burst. One approach to this problem is to try to approximate SJF scheduling. We may not know the length of the next CPU burst, but we may be able to predict its value. We expect that the next CPU burst will be similar in length to the previous ones. By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.

The next CPU burst is generally predicted as an **exponential average** of the measured lengths of previous CPU bursts. We can define the exponential average with the following formula. Let t_n be the length of the nth CPU burst, and let τ_{n+1} be our predicted value for the next CPU burst. Then, for α, $0 \leq \alpha \leq 1$, define

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$$

The value of t_n contains our most recent information, while τ_n stores the past history. The parameter α controls the relative weight of recent and past history in our prediction. If $\alpha = 0$, then $\tau_{n+1} = \tau_n$, and recent history has no effect (current conditions are assumed to be transient). If $\alpha = 1$, then $\tau_{n+1} = t_n$, and only the most recent CPU burst matters (history is assumed to be old and irrelevant). More commonly, $\alpha = 1/2$, so recent history and past history are equally weighted. The initial τ_0 can be defined as a constant or as an overall system average. Figure 5.4 shows an exponential average with $\alpha = 1/2$ and $\tau_0 = 10$.



CPU burst (t_i)	6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Figure 5.4 Prediction of the length of the next CPU burst.

To understand the behavior of the exponential average, we can expand the formula for τ_{n+1} by substituting for τ_n to find

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \cdots + (1 - \alpha)^j \alpha t_{n-j} + \cdots + (1 - \alpha)^{n+1} \tau_0.$$

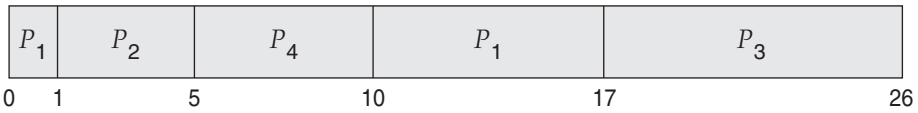
Typically, α is less than 1. As a result, $(1 - \alpha)$ is also less than 1, and each successive term has less weight than its predecessor.

The SJF algorithm can be either preemptive or nonpreemptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called **shortest-remaining-time-firs** scheduling.

As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



Process P_1 is started at time 0, since it is the only process in the queue. Process P_2 arrives at time 1. The remaining time for process P_1 (7 milliseconds) is larger than the time required by process P_2 (4 milliseconds), so process P_1 is preempted, and process P_2 is scheduled. The average waiting time for this example is $[(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)]/4 = 26/4 = 6.5$ milliseconds. Nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

5.3.3 Round-Robin Scheduling

The **round-robin** (RR) scheduling algorithm is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a **time quantum** or **time slice**, is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

To implement RR scheduling, we again treat the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue.

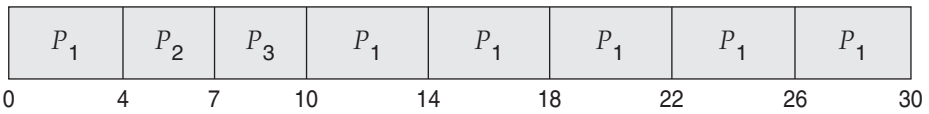
The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If we use a time quantum of 4 milliseconds, then process P_1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P_2. Process P_2 does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process P_3. Once each process has received 1 time quantum, the CPU is returned to process P_1 for an additional time quantum. The resulting RR schedule is as follows:



Let's calculate the average waiting time for this schedule. P_1 waits for 6 milliseconds ($10 - 4$), P_2 waits for 4 milliseconds, and P_3 waits for 7 milliseconds. Thus, the average waiting time is $17/3 = 5.66$ milliseconds.

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is thus preemptive.

If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units. Each process must wait no longer than $(n - 1) \times q$ time units until its next time quantum. For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy. In contrast, if the time quantum is extremely small (say, 1 millisecond), the RR approach can result in a large

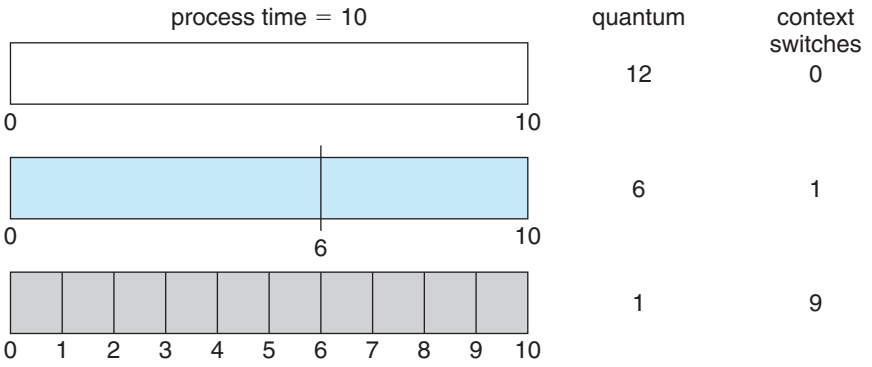


Figure 5.5 How a smaller time quantum increases context switches.

number of context switches. Assume, for example, that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead. If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch. If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly (Figure 5.5).

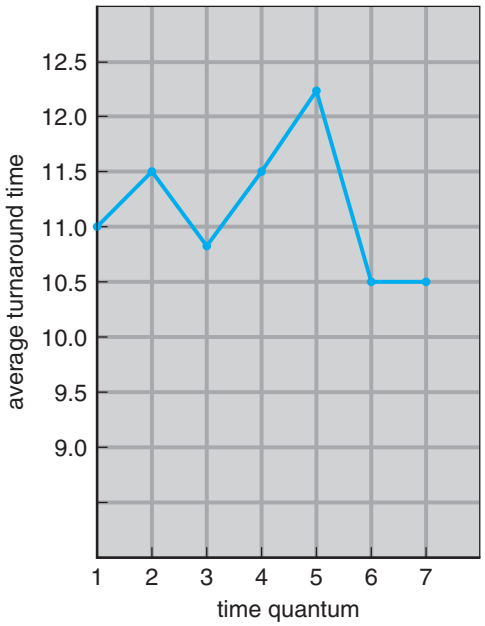
Thus, we want the time quantum to be large with respect to the context-switch time. If the context-switch time is approximately 10 percent of the time quantum, then about 10 percent of the CPU time will be spent in context switching. In practice, most modern systems have time quanta ranging from 10 to 100 milliseconds. The time required for a context switch is typically less than 10 microseconds; thus, the context-switch time is a small fraction of the time quantum.

Turnaround time also depends on the size of the time quantum. As we can see from Figure 5.6, the average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases. In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum. For example, given three processes of 10 time units each and a quantum of 1 time unit, the average turnaround time is 29. If the time quantum is 10, however, the average turnaround time drops to 20. If context-switch time is added in, the average turnaround time increases even more for a smaller time quantum, since more context switches are required.

Although the time quantum should be large compared with the context-switch time, it should not be too large. As we pointed out earlier, if the time quantum is too large, RR scheduling degenerates to an FCFS policy. A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.

5.3.4 Priority Scheduling

The SJF algorithm is a special case of the general **priority-scheduling** algorithm. A priority is associated with each process, and the CPU is allocated to the



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Figure 5.6 How turnaround time varies with the time quantum.

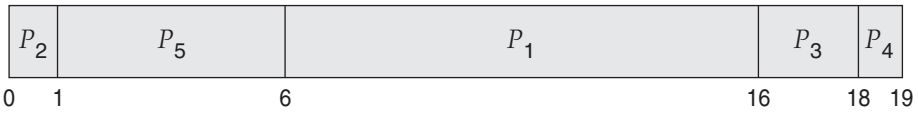
process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

Note that we discuss scheduling in terms of ***high*** priority and ***low*** priority. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. In this text, we assume that low numbers represent high priority.

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order $P_1, P_2, \cdots, P_5$, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Using priority scheduling, we would schedule these processes according to the following Gantt chart:



The average waiting time is 8.2 milliseconds.

Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities. External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.

Priority scheduling can be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithms is **indefinit blocking**, or **starvation**. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low-priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. Generally, one of two things will happen. Either the process will eventually be run (at 2 A.M. Sunday, when the system is finally lightly loaded), or the computer system will eventually crash and lose all unfinished low-priority processes. (Rumor has it that when they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run.)

A solution to the problem of indefinite blockage of low-priority processes is **aging**. Aging involves gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 127 (low) to 0 (high), we could periodically (say, every second) increase the priority of a waiting process by 1. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed. In fact, it would take a little over 2 minutes for a priority-127 process to age to a priority-0 process.

Another option is to combine round-robin and priority scheduling in such a way that the system executes the highest-priority process and runs processes with the same priority using round-robin scheduling. Let's illustrate with an example using the following set of processes, with the burst time in milliseconds:

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

Using priority scheduling with round-robin for processes with equal priority, we would schedule these processes according to the following Gantt chart using a time quantum of 2 milliseconds:

In this example, process P_4 has the highest priority, so it will run to completion. Processes P_2 and P_3 have the next-highest priority, and they will execute in a round-robin fashion. Notice that when process P_2 finishes at time 16, process P_3 is the highest-priority process, so it will run until it completes execution. Now, only processes P_1 and P_5 remain, and as they have equal priority, they will execute in round-robin order until they complete.

5.3.5 Multilevel Queue Scheduling

With both priority and round-robin scheduling, all processes may be placed in a single queue, and the scheduler then selects the process with the highest priority to run. Depending on how the queues are managed, an $O(n)$ search may be necessary to determine the highest-priority process. In practice, it is often easier to have separate queues for each distinct priority, and priority scheduling simply schedules the process in the highest-priority queue. This is illustrated in Figure 5.7. This approach—known as **multilevel queue**—also works well when priority scheduling is combined with round-robin: if there are multiple processes in the highest-priority queue, they are executed in round-robin order. In the most generalized form of this approach, a priority is assigned statically to each process, and a process remains in the same queue for the duration of its runtime.

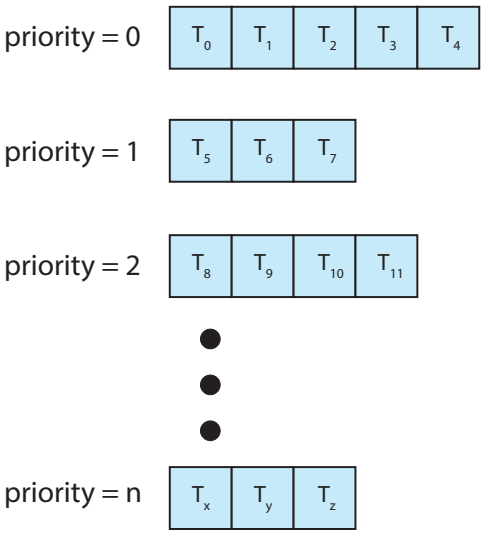


Figure 5.7 Separate queues for each priority.

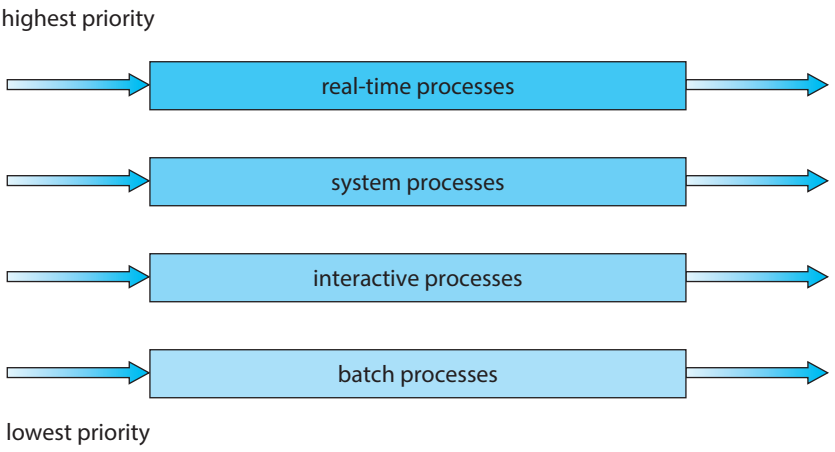


Figure 5.8 Multilevel queue scheduling.

A multilevel queue scheduling algorithm can also be used to partition processes into several separate queues based on the process type (Figure 5.8). For example, a common division is made between **foreground** (interactive) processes and **background** (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs. In addition, foreground processes may have priority (externally defined) over background processes. Separate queues might be used for foreground and background processes, and each queue might have its own scheduling algorithm. The foreground queue might be scheduled by an RR algorithm, for example, while the background queue is scheduled by an FCFS algorithm.

In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the real-time queue may have absolute priority over the interactive queue.

Let's look at an example of a multilevel queue scheduling algorithm with four queues, listed below in order of priority:

1. Real-time processes
2. System processes
3. Interactive processes
4. Batch processes

Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for real-time processes, system processes, and interactive processes were all empty. If an interactive process entered the ready queue while a batch process was running, the batch process would be preempted.

Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For instance, in the foreground–background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling

among its processes, while the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

5.3.6 Multilevel Feedback Queue Scheduling

Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but it is inflexible.

The **multilevel feedback queue** scheduling algorithm, in contrast, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes—which are typically characterized by short CPU bursts—in the higher-priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2 (Figure 5.9). The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.

An entering process is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty. To prevent starvation, a process that waits too long in a lower-priority queue may gradually be moved to a higher-priority queue.

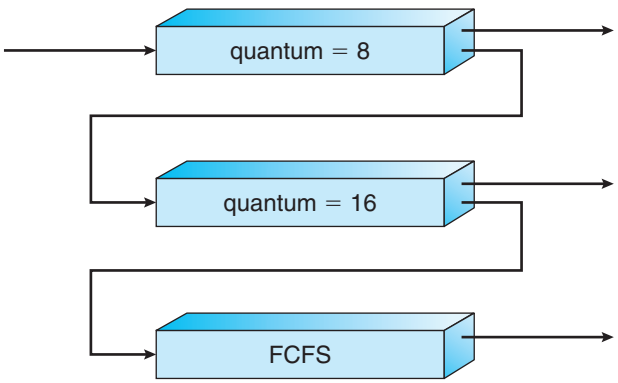


Figure 5.9 Multilevel feedback queues.

This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst. Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

In general, a multilevel feedback queue scheduler is defined by the following parameters:

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher-priority queue
- The method used to determine when to demote a process to a lower-priority queue
- The method used to determine which queue a process will enter when that process needs service

The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm. It can be configured to match a specific system under design. Unfortunately, it is also the most complex algorithm, since defining the best scheduler requires some means by which to select values for all the parameters.

5.4 Thread Scheduling

In Chapter 4, we introduced threads to the process model, distinguishing between ***user-level*** and ***kernel-level*** threads. On most modern operating systems it is kernel-level threads—not processes—that are being scheduled by the operating system. User-level threads are managed by a thread library, and the kernel is unaware of them. To run on a CPU, user-level threads must ultimately be mapped to an associated kernel-level thread, although this mapping may be indirect and may use a lightweight process (LWP). In this section, we explore scheduling issues involving user-level and kernel-level threads and offer specific examples of scheduling for Pthreads.

5.4.1 Contention Scope

One distinction between user-level and kernel-level threads lies in how they are scheduled. On systems implementing the many-to-one (Section 4.3.1) and many-to-many (Section 4.3.3) models, the thread library schedules user-level threads to run on an available LWP. This scheme is known as **process-contention scope** (**PCS**), since competition for the CPU takes place among threads belonging to the same process. (When we say the thread library *schedules* user threads onto available LWPs, we do not mean that the threads are actually *running* on a CPU as that further requires the operating system to schedule the LWP's kernel thread onto a physical CPU core.) To decide which

kernel-level thread to schedule onto a CPU, the kernel uses **system-contention scope** (**SCS**). Competition for the CPU with SCS scheduling takes place among all threads in the system. Systems using the one-to-one model (Section 4.3.2), such as Windows and Linux schedule threads using only SCS.

Typically, PCS is done according to priority—the scheduler selects the runnable thread with the highest priority to run. User-level thread priorities are set by the programmer and are not adjusted by the thread library, although some thread libraries may allow the programmer to change the priority of a thread. It is important to note that PCS will typically preempt the thread currently running in favor of a higher-priority thread; however, there is no guarantee of time slicing (Section 5.3.3) among threads of equal priority.

5.4.2 Pthread Scheduling

We provided a sample POSIX Pthread program in Section 4.4.1, along with an introduction to thread creation with Pthreads. Now, we highlight the POSIX Pthread API that allows specifying PCS or SCS during thread creation. Pthreads identifies the following contention scope values:

- PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling.
- PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling.

On systems implementing the many-to-many model, the PTHREAD_SCOPE_PROCESS policy schedules user-level threads onto available LWPs. The number of LWPs is maintained by the thread library, perhaps using scheduler activations (Section 4.6.5). The PTHREAD_SCOPE_SYSTEM scheduling policy will create and bind an LWP for each user-level thread on many-to-many systems, effectively mapping threads using the one-to-one policy.

The Pthread IPC (Interprocess Communication) provides two functions for setting—and getting—the contention scope policy:

- `pthread_attr_setscope(pthread_attr_t *attr, int scope)`
- `pthread_attr_getscope(pthread_attr_t *attr, int *scope)`

The first parameter for both functions contains a pointer to the attribute set for the thread. The second parameter for the `pthread_attr_setscope()` function is passed either the PTHREAD_SCOPE_SYSTEM or the PTHREAD_SCOPE_PROCESS value, indicating how the contention scope is to be set. In the case of `pthread_attr_getscope()`, this second parameter contains a pointer to an `int` value that is set to the current value of the contention scope. If an error occurs, each of these functions returns a nonzero value.

In Figure 5.10, we illustrate a Pthread scheduling API. The program first determines the existing contention scope and sets it to PTHREAD_SCOPE_SYSTEM. It then creates five separate threads that will run using the SCS scheduling policy. Note that on some systems, only certain contention scope values are allowed. For example, Linux and macOS systems allow only PTHREAD_SCOPE_SYSTEM.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
   int i, scope;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;

   /* get the default attributes */
   pthread_attr_init(&attr);

   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
       printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
       printf("PTHREAD_SCOPE_SYSTEM");
      else
       fprintf(stderr, "Illegal scope value.\n");
    }

   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
       pthread_create(&tid[i],&attr,runner,NULL);

   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
       pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */

   pthread_exit(0);
}
```

Figure 5.10 Pthread scheduling API.

5.5 Multi-Processor Scheduling

Our discussion thus far has focused on the problems of scheduling the CPU in a system with a single processing core. If multiple CPUs are available, **load sharing**, where multiple threads may run in parallel, becomes possible, however scheduling issues become correspondingly more complex. Many possibilities have been tried; and as we saw with CPU scheduling with a single-core CPU, there is no one best solution.

Traditionally, the term **multiprocessor** referred to systems that provided multiple physical processors, where each processor contained one single-core CPU. However, the definition of multiprocessor has evolved significantly, and on modern computing systems, ***multiprocessor*** now applies to the following system architectures:

- Multicore CPUs
- Multithreaded cores
- NUMA systems
- Heterogeneous multiprocessing

Here, we discuss several concerns in multiprocessor scheduling in the context of these different architectures. In the first three examples we concentrate on systems in which the processors are identical—homogeneous—in terms of their functionality. We can then use any available CPU to run any process in the queue. In the last example we explore a system where the processors are not identical in their capabilities.

5.5.1 Approaches to Multiple-Processor Scheduling

One approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor — the master server. The other processors execute only user code. This **asymmetric multiprocessing** is simple because only one core accesses the system data structures, reducing the need for data sharing. The downfall of this approach is the master server becomes a potential bottleneck where overall system performance may be reduced.

The standard approach for supporting multiprocessors is **symmetric multiprocessing** (**SMP**), where each processor is self-scheduling. Scheduling proceeds by having the scheduler for each processor examine the ready queue and select a thread to run. Note that this provides two possible strategies for organizing the threads eligible to be scheduled:

1. All threads may be in a common ready queue.
2. Each processor may have its own private queue of threads.

These two strategies are contrasted in Figure 5.11. If we select the first option, we have a possible race condition on the shared ready queue and therefore must ensure that two separate processors do not choose to schedule the same thread and that threads are not lost from the queue. As discussed in

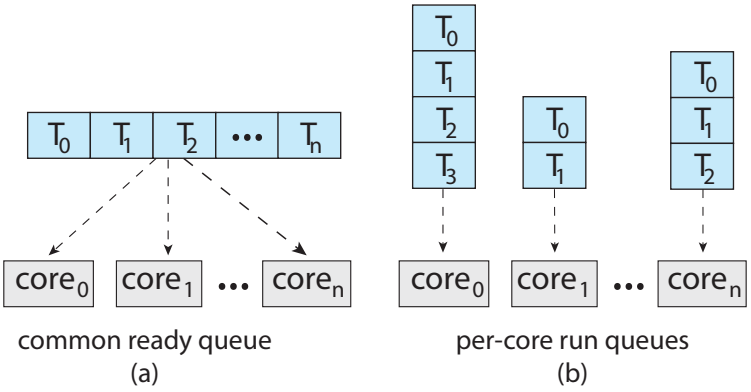


Figure 5.11 Organization of ready queues.

Chapter 6, we could use some form of locking to protect the common ready queue from this race condition. Locking would be highly contended, however, as all accesses to the queue would require lock ownership, and accessing the shared queue would likely be a performance bottleneck. The second option permits each processor to schedule threads from its private run queue and therefore does not suffer from the possible performance problems associated with a shared run queue. Thus, it is the most common approach on systems supporting SMP. Additionally, as described in Section 5.5.4, having private, per-processor run queues in fact may lead to more efficient use of cache memory. There are issues with per-processor run queues—most notably, workloads of varying sizes. However, as we shall see, balancing algorithms can be used to equalize workloads among all processors.

Virtually all modern operating systems support SMP, including Windows, Linux, and macOS as well as mobile systems including Android and iOS. In the remainder of this section, we discuss issues concerning SMP systems when designing CPU scheduling algorithms.

5.5.2 Multicore Processors

Traditionally, SMP systems have allowed several processes to run in parallel by providing multiple physical processors. However, most contemporary computer hardware now places multiple computing cores on the same physical chip, resulting in a **multicore processor**. Each core maintains its architectural state and thus appears to the operating system to be a separate logical CPU. SMP systems that use multicore processors are faster and consume less power than systems in which each CPU has its own physical chip.

Multicore processors may complicate scheduling issues. Let's consider how this can happen. Researchers have discovered that when a processor accesses memory, it spends a significant amount of time waiting for the data to become available. This situation, known as a **memory stall**, occurs primarily because modern processors operate at much faster speeds than memory. However, a memory stall can also occur because of a cache miss (accessing data that are not in cache memory). Figure 5.12 illustrates a memory stall. In this scenario, the processor can spend up to 50 percent of its time waiting for data to become available from memory.

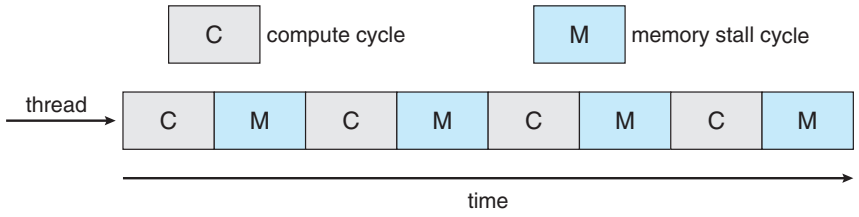


Figure 5.12 Memory stall.

To remedy this situation, many recent hardware designs have implemented multithreaded processing cores in which two (or more) **hardware threads** are assigned to each core. That way, if one hardware thread stalls while waiting for memory, the core can switch to another thread. Figure 5.13 illustrates a dual-threaded processing core on which the execution of thread 0 and the execution of thread 1 are interleaved. From an operating system perspective, each hardware thread maintains its architectural state, such as instruction pointer and register set, and thus appears as a logical CPU that is available to run a software thread. This technique—known as **chip multithreading** (CMT)—is illustrated in Figure 5.14. Here, the processor contains four computing cores, with each core containing two hardware threads. From the perspective of the operating system, there are eight logical CPUs.

Intel processors use the term **hyper-threading** (also known as **simultaneous multithreading** or SMT) to describe assigning multiple hardware threads to a single processing core. Contemporary Intel processors—such as the i7—support two threads per core, while the Oracle Sparc M7 processor supports eight threads per core, with eight cores per processor, thus providing the operating system with 64 logical CPUs.

In general, there are two ways to multithread a processing core: **coarse-grained** and **fine-graine** multithreading. With coarse-grained multithreading, a thread executes on a core until a long-latency event such as a memory stall occurs. Because of the delay caused by the long-latency event, the core must switch to another thread to begin execution. However, the cost of switching between threads is high, since the instruction pipeline must be flushed before the other thread can begin execution on the processor core. Once this new thread begins execution, it begins filling the pipeline with its instructions. Fine-grained (or interleaved) multithreading switches between threads at a much finer level of granularity—typically at the boundary of an instruction

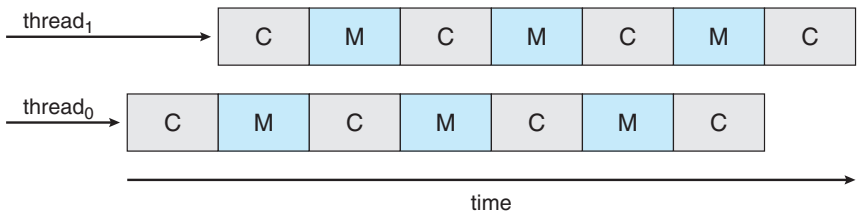


Figure 5.13 Multithreaded multicore system.

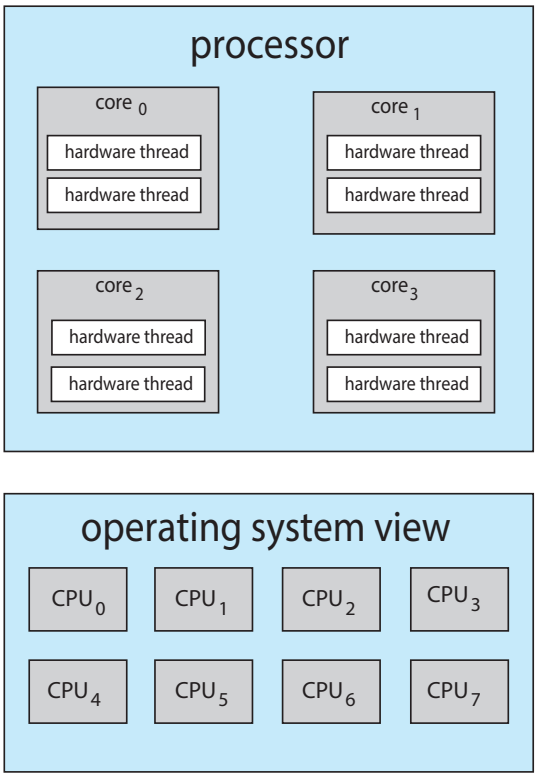


Figure 5.14 Chip multithreading.

cycle. However, the architectural design of fine-grained systems includes logic for thread switching. As a result, the cost of switching between threads is small.

It is important to note that the resources of the physical core (such as caches and pipelines) must be shared among its hardware threads, and therefore a processing core can only execute one hardware thread at a time. Consequently, a multithreaded, multicore processor actually requires two different levels of scheduling, as shown in Figure 5.15, which illustrates a dual-threaded processing core.

On one level are the scheduling decisions that must be made by the operating system as it chooses which software thread to run on each hardware thread (logical CPU). For all practical purposes, such decisions have been the primary focus of this chapter. Therefore, for this level of scheduling, the operating system may choose any scheduling algorithm, including those described in Section 5.3.

A second level of scheduling specifies how each core decides which hardware thread to run. There are several strategies to adopt in this situation. One approach is to use a simple round-robin algorithm to schedule a hardware thread to the processing core. This is the approach adopted by the UltraSPARC T3. Another approach is used by the Intel Itanium, a dual-core processor with two hardware-managed threads per core. Assigned to each hardware thread is a dynamic ***urgency*** value ranging from 0 to 7, with 0 representing the lowest urgency and 7 the highest. The Itanium identifies five different events that may

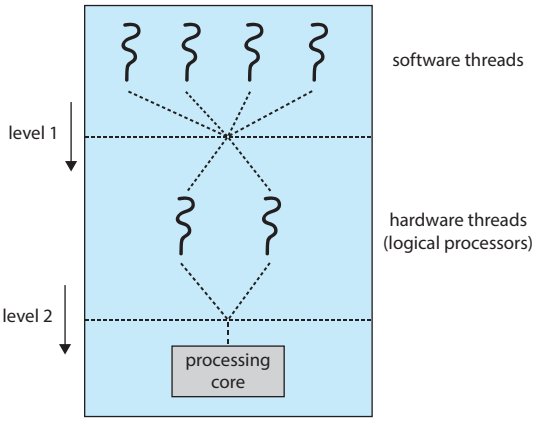


Figure 5.15 Two levels of scheduling.

trigger a thread switch. When one of these events occurs, the thread-switching logic compares the urgency of the two threads and selects the thread with the highest urgency value to execute on the processor core.

Note that the two different levels of scheduling shown in Figure 5.15 are not necessarily mutually exclusive. In fact, if the operating system scheduler (the first level) is made aware of the sharing of processor resources, it can make more effective scheduling decisions. As an example, assume that a CPU has two processing cores, and each core has two hardware threads. If two software threads are running on this system, they can be running either on the same core or on separate cores. If they are both scheduled to run on the same core, they have to share processor resources and thus are likely to proceed more slowly than if they were scheduled on separate cores. If the operating system is aware of the level of processor resource sharing, it can schedule software threads onto logical processors that do not share resources.

5.5.3 Load Balancing

On SMP systems, it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor. Otherwise, one or more processors may sit idle while other processors have high workloads, along with ready queues of threads awaiting the CPU. **Load balancing** attempts to keep the workload evenly distributed across all processors in an SMP system. It is important to note that load balancing is typically necessary only on systems where each processor has its own private ready queue of eligible threads to execute. On systems with a common run queue, load balancing is unnecessary, because once a processor becomes idle, it immediately extracts a runnable thread from the common ready queue.

There are two general approaches to load balancing: push migration and pull migration. With **push migration**, a specific task periodically checks the load on each processor and—if it finds an imbalance—evenly distributes the load by moving (or pushing) threads from overloaded to idle or less-busy processors. **Pull migration** occurs when an idle processor pulls a waiting task from a busy processor. Push and pull migration need not be mutually exclusive and are, in fact, often implemented in parallel on load-balancing systems. For

example, the Linux CFS scheduler (described in Section 5.7.1) and the ULE scheduler available for FreeBSD systems implement both techniques.

The concept of a "balanced load" may have different meanings. One view of a balanced load may require simply that all queues have approximately the same number of threads. Alternatively, balance may require an equal distribution of thread priorities across all queues. In addition, in certain situations, neither of these strategies may be sufficient. Indeed, they may work against the goals of the scheduling algorithm. (We leave further consideration of this as an exercise.)

5.5.4 Processor Affinity

Consider what happens to cache memory when a thread has been running on a specific processor. The data most recently accessed by the thread populate the cache for the processor. As a result, successive memory accesses by the thread are often satisfied in cache memory (known as a "warm cache"). Now consider what happens if the thread migrates to another processor—say, due to load balancing. The contents of cache memory must be invalidated for the first processor, and the cache for the second processor must be repopulated. Because of the high cost of invalidating and repopulating caches, most operating systems with SMP support try to avoid migrating a thread from one processor to another and instead attempt to keep a thread running on the same processor and take advantage of a warm cache. This is known as **processor affinit** —that is, a process has an affinity for the processor on which it is currently running.

The two strategies described in Section 5.5.1 for organizing the queue of threads available for scheduling have implications for processor affinity. If we adopt the approach of a common ready queue, a thread may be selected for execution by any processor. Thus, if a thread is scheduled on a new processor, that processor's cache must be repopulated. With private, per-processor ready queues, a thread is always scheduled on the same processor and can therefore benefit from the contents of a warm cache. Essentially, per-processor ready queues provide processor affinity for free!

Processor affinity takes several forms. When an operating system has a policy of attempting to keep a process running on the same processor—but not guaranteeing that it will do so—we have a situation known as **soft affinit** . Here, the operating system will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors during load balancing. In contrast, some systems provide system calls that support **hard affinit** , thereby allowing a process to specify a subset of processors on which it can run. Many systems provide both soft and hard affinity. For example, Linux implements soft affinity, but it also provides the `sched_setaffinity()` system call, which supports hard affinity by allowing a thread to specify the set of CPUs on which it is eligible to run.

The main-memory architecture of a system can affect processor affinity issues as well. Figure 5.16 illustrates an architecture featuring non-uniform memory access (NUMA) where there are two physical processor chips each with their own CPU and local memory. Although a system interconnect allows all CPUs in a NUMA system to share one physical address space, a CPU has faster access to its local memory than to memory local to another CPU. If the operating system's CPU scheduler and memory-placement algorithms are ***NUMA-aware***

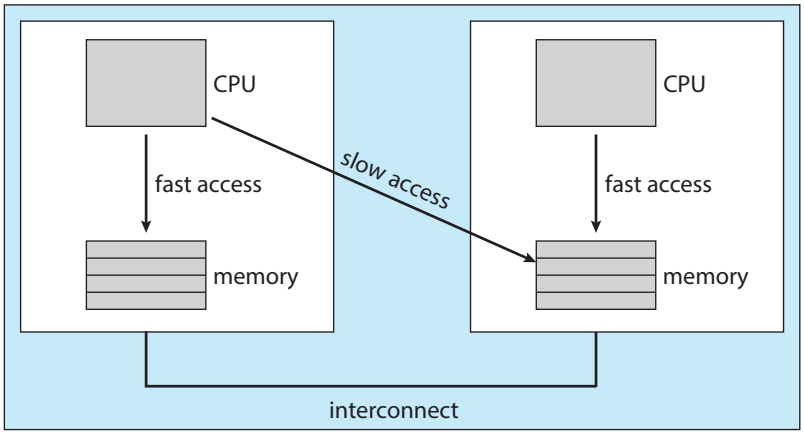


Figure 5.16 NUMA and CPU scheduling.

and work together, then a thread that has been scheduled onto a particular CPU can be allocated memory closest to where the CPU resides, thus providing the thread the fastest possible memory access.

Interestingly, load balancing often counteracts the benefits of processor affinity. That is, the benefit of keeping a thread running on the same processor is that the thread can take advantage of its data being in that processor's cache memory. Balancing loads by moving a thread from one processor to another removes this benefit. Similarly, migrating a thread between processors may incur a penalty on NUMA systems, where a thread may be moved to a processor that requires longer memory access times. In other words, there is a natural tension between load balancing and minimizing memory access times. Thus, scheduling algorithms for modern multicore NUMA systems have become quite complex. In Section 5.7.1, we examine the Linux CFS scheduling algorithm and explore how it balances these competing goals.

5.5.5 Heterogeneous Multiprocessing

In the examples we have discussed so far, all processors are identical in terms of their capabilities, thus allowing any thread to run on any processing core. The only difference being that memory access times may vary based upon load balancing and processor affinity policies, as well as on NUMA systems.

Although mobile systems now include multicore architectures, some systems are now designed using cores that run the same instruction set, yet vary in terms of their clock speed and power management, including the ability to adjust the power consumption of a core to the point of idling the core. Such systems are known as **heterogeneous multiprocessing** (HMP). Note this is not a form of asymmetric multiprocessing as described in Section 5.5.1 as both system and user tasks can run on any core. Rather, the intention behind HMP is to better manage power consumption by assigning tasks to certain cores based upon the specific demands of the task.

For ARM processors that support it, this type of architecture is known as **big.LITTLE** where higher-peformance ***big*** cores are combined with energy efficient ***LITTLE*** cores. ***Big*** cores consume greater energy and therefore should

only be used for short periods of time. Likewise, ***little*** cores use less energy and can therefore be used for longer periods.

There are several advantages to this approach. By combining a number of slower cores with faster ones, a CPU scheduler can assign tasks that do not require high performance, but may need to run for longer periods, (such as background tasks) to little cores, thereby helping to preserve a battery charge. Similarly, interactive applications which require more processing power, but may run for shorter durations, can be assigned to big cores. Additionally, if the mobile device is in a power-saving mode, energy-intensive big cores can be disabled and the system can rely solely on energy-efficient little cores. Windows 10 supports HMP scheduling by allowing a thread to select a scheduling policy that best supports its power management demands.

5.6 Real-Time CPU Scheduling

CPU scheduling for real-time operating systems involves special issues. In general, we can distinguish between soft real-time systems and hard real-time systems. **Soft real-time systems** provide no guarantee as to when a critical real-time process will be scheduled. They guarantee only that the process will be given preference over noncritical processes. **Hard real-time systems** have stricter requirements. A task must be serviced by its deadline; service after the deadline has expired is the same as no service at all. In this section, we explore several issues related to process scheduling in both soft and hard real-time operating systems.

5.6.1 Minimizing Latency

Consider the event-driven nature of a real-time system. The system is typically waiting for an event in real time to occur. Events may arise either in software —as when a timer expires—or in hardware—as when a remote-controlled vehicle detects that it is approaching an obstruction. When an event occurs, the system must respond to and service it as quickly as possible. We refer to **event latency** as the amount of time that elapses from when an event occurs to when it is serviced (Figure 5.17).

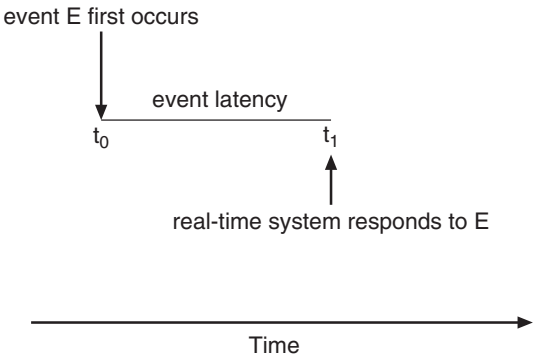


Figure 5.17 Event latency.

Usually, different events have different latency requirements. For example, the latency requirement for an antilock brake system might be 3 to 5 milliseconds. That is, from the time a wheel first detects that it is sliding, the system controlling the antilock brakes has 3 to 5 milliseconds to respond to and control the situation. Any response that takes longer might result in the automobile's veering out of control. In contrast, an embedded system controlling radar in an airliner might tolerate a latency period of several seconds.

Two types of latencies affect the performance of real-time systems:

1. Interrupt latency
2. Dispatch latency

Interrupt latency refers to the period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt. When an interrupt occurs, the operating system must first complete the instruction it is executing and determine the type of interrupt that occurred. It must then save the state of the current process before servicing the interrupt using the specific interrupt service routine (ISR). The total time required to perform these tasks is the interrupt latency (Figure 5.18).

Obviously, it is crucial for real-time operating systems to minimize interrupt latency to ensure that real-time tasks receive immediate attention. Indeed, for hard real-time systems, interrupt latency must not simply be minimized, it must be bounded to meet the strict requirements of these systems.

One important factor contributing to interrupt latency is the amount of time interrupts may be disabled while kernel data structures are being updated. Real-time operating systems require that interrupts be disabled for only very short periods of time.

The amount of time required for the scheduling dispatcher to stop one process and start another is known as **dispatch latency**. Providing real-time

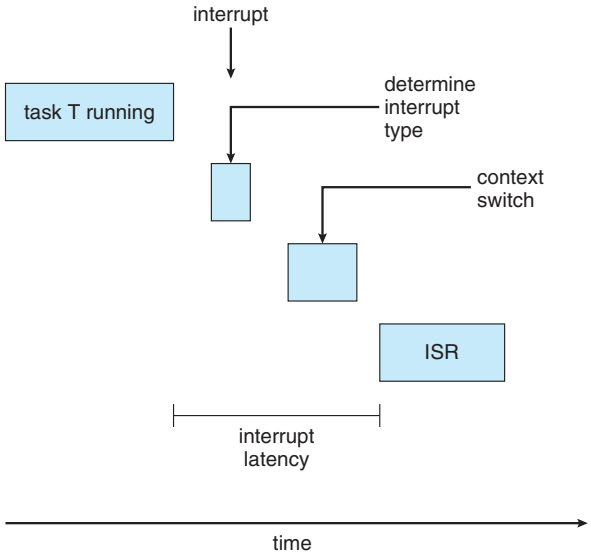


Figure 5.18 Interrupt latency.

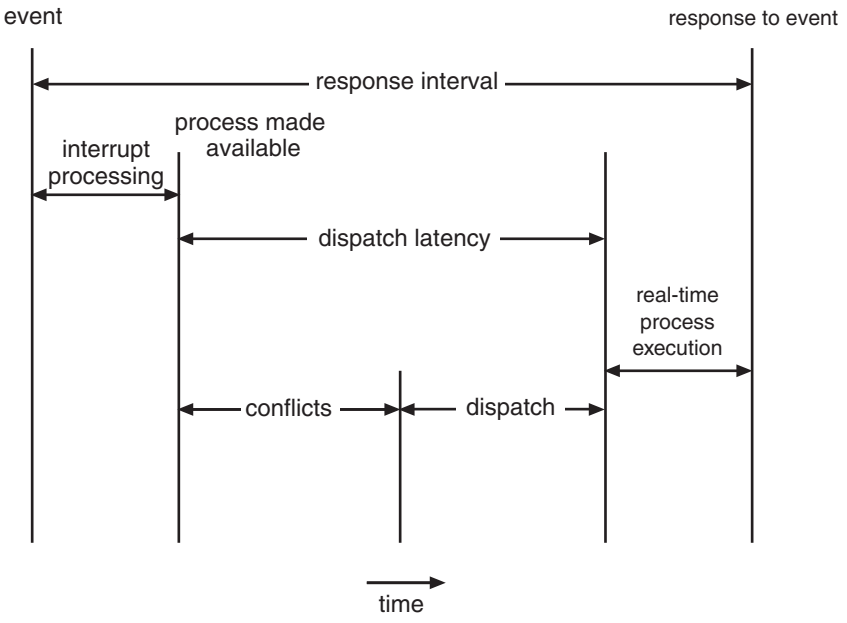


Figure 5.19 Dispatch latency.

tasks with immediate access to the CPU mandates that real-time operating systems minimize this latency as well. The most effective technique for keeping dispatch latency low is to provide preemptive kernels. For hard real-time systems, dispatch latency is typically measured in several microseconds.

In Figure 5.19, we diagram the makeup of dispatch latency. The **conflic phase** of dispatch latency has two components:

1. Preemption of any process running in the kernel
2. Release by low-priority processes of resources needed by a high-priority process

Following the conflict phase, the dispatch phase schedules the high-priority process onto an available CPU.

5.6.2 Priority-Based Scheduling

The most important feature of a real-time operating system is to respond immediately to a real-time process as soon as that process requires the CPU. As a result, the scheduler for a real-time operating system must support a priority-based algorithm with preemption. Recall that priority-based scheduling algorithms assign each process a priority based on its importance; more important tasks are assigned higher priorities than those deemed less important. If the scheduler also supports preemption, a process currently running on the CPU will be preempted if a higher-priority process becomes available to run.

Preemptive, priority-based scheduling algorithms are discussed in detail in Section 5.3.4, and Section 5.7 presents examples of the soft real-time scheduling features of the Linux, Windows, and Solaris operating systems. Each of these systems assigns real-time processes the highest scheduling priority. For

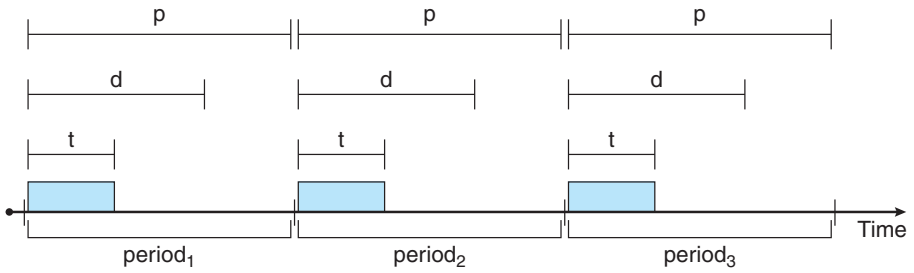


Figure 5.20 Periodic task.

example, Windows has 32 different priority levels. The highest levels—priority values 16 to 31—are reserved for real-time processes. Solaris and Linux have similar prioritization schemes.

Note that providing a preemptive, priority-based scheduler only guarantees soft real-time functionality. Hard real-time systems must further guarantee that real-time tasks will be serviced in accord with their deadline requirements, and making such guarantees requires additional scheduling features. In the remainder of this section, we cover scheduling algorithms appropriate for hard real-time systems.

Before we proceed with the details of the individual schedulers, however, we must define certain characteristics of the processes that are to be scheduled. First, the processes are considered **periodic**. That is, they require the CPU at constant intervals (periods). Once a periodic process has acquired the CPU, it has a fixed processing time t, a deadline d by which it must be serviced by the CPU, and a period p. The relationship of the processing time, the deadline, and the period can be expressed as $0 \leq t \leq d \leq p$. The **rate** of a periodic task is $1/p$. Figure 5.20 illustrates the execution of a periodic process over time. Schedulers can take advantage of these characteristics and assign priorities according to a process's deadline or rate requirements.

What is unusual about this form of scheduling is that a process may have to announce its deadline requirements to the scheduler. Then, using a technique known as an **admission-control** algorithm, the scheduler does one of two things. It either admits the process, guaranteeing that the process will complete on time, or rejects the request as impossible if it cannot guarantee that the task will be serviced by its deadline.

5.6.3 Rate-Monotonic Scheduling

The **rate-monotonic** scheduling algorithm schedules periodic tasks using a static priority policy with preemption. If a lower-priority process is running and a higher-priority process becomes available to run, it will preempt the lower-priority process. Upon entering the system, each periodic task is assigned a priority inversely based on its period. The shorter the period, the higher the priority; the longer the period, the lower the priority. The rationale behind this policy is to assign a higher priority to tasks that require the CPU more often. Furthermore, rate-monotonic scheduling assumes that the process-

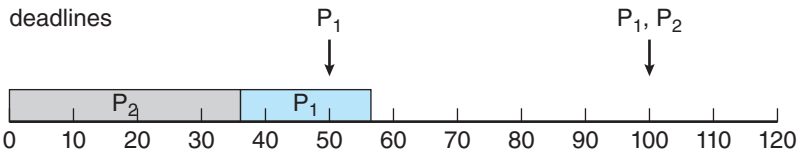


Figure 5.21 Scheduling of tasks when P_2 has a higher priority than P_1.

ing time of a periodic process is the same for each CPU burst. That is, every time a process acquires the CPU, the duration of its CPU burst is the same.

Let's consider an example. We have two processes, P_1 and P_2. The periods for P_1 and P_2 are 50 and 100, respectively—that is, $p_1 = 50$ and $p_2 = 100$. The processing times are $t_1 = 20$ for P_1 and $t_2 = 35$ for P_2. The deadline for each process requires that it complete its CPU burst by the start of its next period.

We must first ask ourselves whether it is possible to schedule these tasks so that each meets its deadlines. If we measure the CPU utilization of a process P_i as the ratio of its burst to its period—t_i/p_i—the CPU utilization of P_1 is $20/50 = 0.40$ and that of P_2 is $35/100 = 0.35$, for a total CPU utilization of 75 percent. Therefore, it seems we can schedule these tasks in such a way that both meet their deadlines and still leave the CPU with available cycles.

Suppose we assign P_2 a higher priority than P_1. The execution of P_1 and P_2 in this situation is shown in Figure 5.21. As we can see, P_2 starts execution first and completes at time 35. At this point, P_1 starts; it completes its CPU burst at time 55. However, the first deadline for P_1 was at time 50, so the scheduler has caused P_1 to miss its deadline.

Now suppose we use rate-monotonic scheduling, in which we assign P_1 a higher priority than P_2 because the period of P_1 is shorter than that of P_2. The execution of these processes in this situation is shown in Figure 5.22. P_1 starts first and completes its CPU burst at time 20, thereby meeting its first deadline. P_2 starts running at this point and runs until time 50. At this time, it is preempted by P_1, although it still has 5 milliseconds remaining in its CPU burst. P_1 completes its CPU burst at time 70, at which point the scheduler resumes P_2. P_2 completes its CPU burst at time 75, also meeting its first deadline. The system is idle until time 100, when P_1 is scheduled again.

Rate-monotonic scheduling is considered optimal in that if a set of processes cannot be scheduled by this algorithm, it cannot be scheduled by any other algorithm that assigns static priorities. Let's next examine a set of processes that cannot be scheduled using the rate-monotonic algorithm.

Assume that process P_1 has a period of $p_1 = 50$ and a CPU burst of $t_1 = 25$. For P_2, the corresponding values are $p_2 = 80$ and $t_2 = 35$. Rate-monotonic

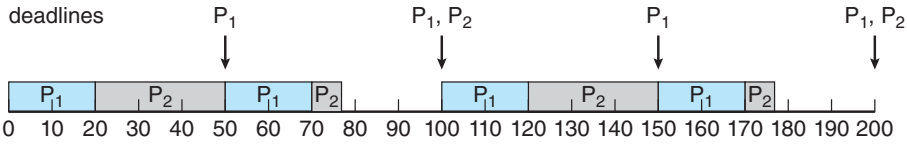


Figure 5.22 Rate-monotonic scheduling.

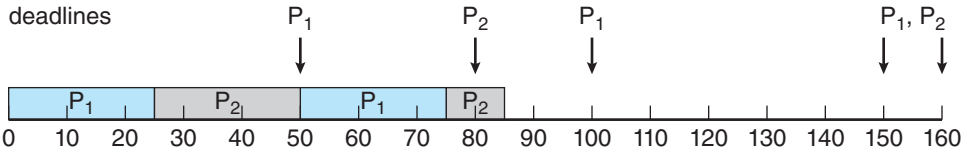


Figure 5.23 Missing deadlines with rate-monotonic scheduling.

scheduling would assign process P_1 a higher priority, as it has the shorter period. The total CPU utilization of the two processes is (25/50) + (35/80) = 0.94, and it therefore seems logical that the two processes could be scheduled and still leave the CPU with 6 percent available time. Figure 5.23 shows the scheduling of processes P_1 and P_2. Initially, P_1 runs until it completes its CPU burst at time 25. Process P_2 then begins running and runs until time 50, when it is preempted by P_1. At this point, P_2 still has 10 milliseconds remaining in its CPU burst. Process P_1 runs until time 75; consequently, P2 finishes its burst at time 85, after the deadline for completion of its CPU burst at time 80.

Despite being optimal, then, rate-monotonic scheduling has a limitation: CPU utilization is bounded, and it is not always possible to maximize CPU resources fully. The worst-case CPU utilization for scheduling N processes is

$$N(2^{1/N} - 1).$$

With one process in the system, CPU utilization is 100 percent, but it falls to approximately 69 percent as the number of processes approaches infinity. With two processes, CPU utilization is bounded at about 83 percent. Combined CPU utilization for the two processes scheduled in Figure 5.21 and Figure 5.22 is 75 percent; therefore, the rate-monotonic scheduling algorithm is guaranteed to schedule them so that they can meet their deadlines. For the two processes scheduled in Figure 5.23, combined CPU utilization is approximately 94 percent; therefore, rate-monotonic scheduling cannot guarantee that they can be scheduled so that they meet their deadlines.

5.6.4 Earliest-Deadline-First Scheduling

Earliest-deadline-firs (EDF) scheduling assigns priorities dynamically according to deadline. The earlier the deadline, the higher the priority; the later the deadline, the lower the priority. Under the EDF policy, when a process becomes runnable, it must announce its deadline requirements to the system. Priorities may have to be adjusted to reflect the deadline of the newly runnable process. Note how this differs from rate-monotonic scheduling, where priorities are fixed.

To illustrate EDF scheduling, we again schedule the processes shown in Figure 5.23, which failed to meet deadline requirements under rate-monotonic scheduling. Recall that P_1 has values of $p_1 = 50$ and $t_1 = 25$ and that P_2 has values of $p_2 = 80$ and $t_2 = 35$. The EDF scheduling of these processes is shown in Figure 5.24. Process P_1 has the earliest deadline, so its initial priority is higher than that of process P_2. Process P_2 begins running at the end of the CPU burst for P_1. However, whereas rate-monotonic scheduling allows P_1 to preempt P_2

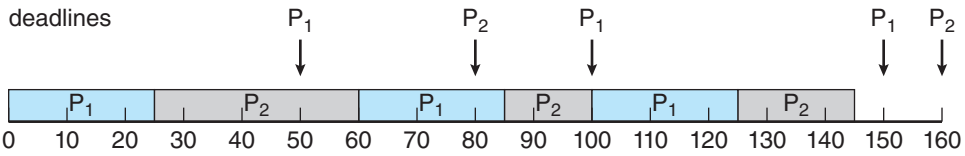


Figure 5.24 Earliest-deadline-first scheduling.

at the beginning of its next period at time 50, EDF scheduling allows process P_2 to continue running. P_2 now has a higher priority than P_1 because its next deadline (at time 80) is earlier than that of P_1 (at time 100). Thus, both P_1 and P_2 meet their first deadlines. Process P_1 again begins running at time 60 and completes its second CPU burst at time 85, also meeting its second deadline at time 100. P_2 begins running at this point, only to be preempted by P_1 at the start of its next period at time 100. P_2 is preempted because P_1 has an earlier deadline (time 150) than P_2 (time 160). At time 125, P_1 completes its CPU burst and P_2 resumes execution, finishing at time 145 and meeting its deadline as well. The system is idle until time 150, when P_1 is scheduled to run once again.

Unlike the rate-monotonic algorithm, EDF scheduling does not require that processes be periodic, nor must a process require a constant amount of CPU time per burst. The only requirement is that a process announce its deadline to the scheduler when it becomes runnable. The appeal of EDF scheduling is that it is theoretically optimal—theoretically, it can schedule processes so that each process can meet its deadline requirements and CPU utilization will be 100 percent. In practice, however, it is impossible to achieve this level of CPU utilization due to the cost of context switching between processes and interrupt handling.

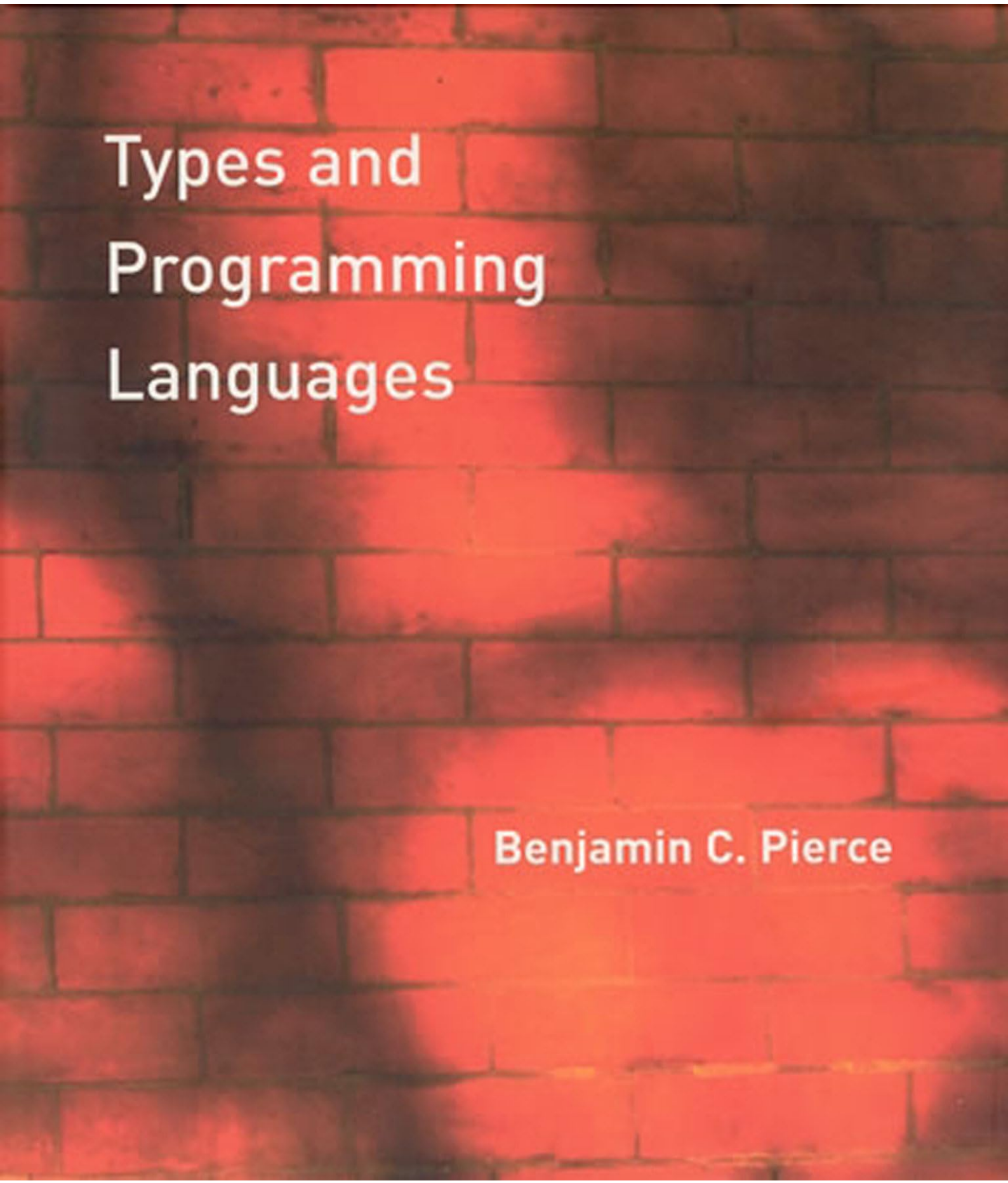
5.6.5 Proportional Share Scheduling

Proportional share schedulers operate by allocating T shares among all applications. An application can receive N shares of time, thus ensuring that the application will have N/T of the total processor time. As an example, assume that a total of $T = 100$ shares is to be divided among three processes, A, B, and C. A is assigned 50 shares, B is assigned 15 shares, and C is assigned 20 shares. This scheme ensures that A will have 50 percent of total processor time, B will have 15 percent, and C will have 20 percent.

Proportional share schedulers must work in conjunction with an admission-control policy to guarantee that an application receives its allocated shares of time. An admission-control policy will admit a client requesting a particular number of shares only if sufficient shares are available. In our current example, we have allocated $50 + 15 + 20 = 85$ shares of the total of 100 shares. If a new process D requested 30 shares, the admission controller would deny D entry into the system.

5.6.6 POSIX Real-Time Scheduling

The POSIX standard also provides extensions for real-time computing—POSIX.1b. Here, we cover some of the POSIX API related to scheduling real-time threads. POSIX defines two scheduling classes for real-time threads:


Types and
Programming
Languages
Benjamin C. Pierce

$$\frac{\Gamma \vdash t_1 : \exists X.T_{12} \qquad \Gamma, X, x{:}T_{12} \vdash t_2 : T_2}{\Gamma \vdash \texttt{let } \{X,x\}=t_1 \texttt{ in } t_2 : T_2} \qquad \text{(T-UNPACK)}$$

19.2 Data Abstraction with Existentials

Abstract Data Types

For a more interesting example, here is a simple package defining an **abstract data type** of (purely functional) counters.

```
counterADT =
   {∃Counter = Nat,
    {new = 0,
     get = λi:Nat. i,
     inc = λi:Nat. succ(i)}}
 as {∃Counter,
     {new: Counter,
      get: Counter→Nat,
      inc: Counter→Counter}};
```

```
▶ counterADT : {∃Counter,
                {new:Counter,get:Counter→Nat,inc:Counter→Counter}}
```

The concrete representation of a counter is just a number. The package provides three operations on counters: a constant `new`, a function `get` for extracting a counter's current value, and a function `inc` for creating a new counter whose stored value is one more than the given counter's. Having created the counter package, we next open it, exposing the operations as the fields of a record `counter`:

```
let {Counter,counter}=counterADT in
counter.get (counter.inc counter.new);
```

```
▶ 1 : Nat
```

If we organize our code so that the body of this `let` contains the whole remainder of the program, then this idiom

```
let {Counter,counter} = <counter package> in
<rest of program>
```

has the effect of declaring a fresh type `Counter` and a variable `counter` of type `{new:Counter,get:Counter→Nat,inc: Counter→Counter}`.

It is instructive to compare the above with a more standard abstract data type declaration, such as might be found in a program in Ada [oD80] or Clu [LAB+81]:

```
ADT counter =
  type Counter
  representation Nat
  operations
    {new = 0
         : Counter,
     get = λi:Nat. i
         : Counter→Nat,
     inc = λi:Nat. succ(i)}
         : Counter→Counter};

counter.get (counter.inc counter.new);
```

The version using existential types is somewhat harder to read, compared to the syntactically sugared second version, but otherwise the two programs are essentially identical.

Note that we can substitute an alternative implementation of the `Counter` ADT—for example, one where the internal representation is a record containing a `Nat` rather than just a single `Nat`

```
counterADT =
   {∃Counter = {x:Nat},
    {new = {x=0},
     get = λi:{x:Nat}. i.x,
     inc = λi:{x:Nat}. {x=succ(i.x)}}}
 as {∃Counter,
     {new: Counter,
      get: Counter→Nat,
      inc: Counter→Counter}};

▶ counterADT : {∃Counter,
                {new:Counter,get:Counter→Nat,inc:Counter→Counter}}
```

in complete confidence that the whole program will remain typesafe, since we are guaranteed that the rest of the program cannot access instances of `Counter` except using `get` and `inc`. This is the essence of data abstraction by information hiding.

In the body of the program, the type name `Counter` can be used just like the base types built into the language. We can define functions that operate on counters:

```
let {Counter,counter}=counterADT
in

let addthree = λc:Counter.
                 counter.inc (counter.inc (counter.inc c))
in

counter.get (addthree counter.new);
```

```
▶ 3 : Nat
```

We can even define new abstract data types whose representation involves counters. For example, the following program defines an ADT of flip-flops, using a counter as the (not very efficient) representation type:

```
let {Counter,counter} =
     {∃Counter = Nat,
      {new = 0,
       get = λi:Nat. i,
       inc = λi:Nat. succ(i)}}
   as {∃Counter,
       {new: Counter,
        get: Counter→Nat,
        inc: Counter→Counter}}
in

let {FlipFlop,flipflop} =
     {∃FlipFlop = Counter,
      {new    = counter.new,
       read   = λc:Counter. iseven (counter.get c),
       toggle = λc:Counter. counter.inc c,
       reset  = λc:Counter. counter.new}}
   as {∃FlipFlop,
       {new:    FlipFlop,
        read:   FlipFlop→Bool,
        toggle: FlipFlop→FlipFlop,
        reset:  FlipFlop→FlipFlop}}
in

flipflop.read (flipflop.toggle (flipflop.toggle flipflop.new));
▶ false : Bool
```

19.2.1 Exercise [Recommended]: Follow the model of the above example to define an abstract data type of *stacks* of numbers, with operations `new`, `push`, `pop`, and `isempty`. Use the `List` type introduced in Exercise **??** as the underlying representation. Write a simple main program that creates a stack, pushes a couple of numbers onto it, pops off the top element, and returns it.

This exercise is best done on-line. Use the checker named "`everything`" and copy the contents of the file `test.f` from the `everything` directory (which contains definitions of the `List` constructor and associated operations) to the top of your own input file. □

Existential Objects

The sequence of "pack then open" that we saw in the last section is the hallmark of ADT-style programming using existential packages. A package defines an abstract type and its associated operations, and each package is opened immediately after it is built, binding a type variable for the abstract type and exposing the ADT's operations abstractly, with this variable in place of the concrete representation type. Existential types can also be used to model other common types of data abstraction. In this section, we show how a simple form of objects can be understood in terms of a different idiom based on existentials.

We will again use simple counters as our running example, as we did both in the previous section on existential ADTs and in our previous encounter with objects, in Chapter 14. Unlike the counters of Chapter 14, however, the counter objects in this section will be purely functional: sending the message `inc` to a counter will not change its internal state in-place, but rather will return a *fresh* counter object with incremented internal state.

A counter object, then, will comprise two basic components: a number (its internal state), and a pair of methods, `get` and `inc`, that can be used to query and update the state. We also need to ensure that the only way that the state of a counter object can be queried or updated is by using one of its two methods. This can be accomplished by wrapping the state and methods in an existential package, abstracting the type of the internal state. For example, a counter object holding the value `5` might be written

```
c = {∃X = Nat,
     {state = 5,
      methods = {get = λx:Nat. x,
                 inc = λx:Nat. succ(x)}}}
 as Counter;
```

where:

```
Counter = {∃X, {state:X, methods: {get:X→Nat, inc:X→X}}};
```

To use a method of a counter object, we will need to open it up and apply the appropriate element of its `methods` to its `state` field. For example, to get the current value of `c` we can write:

```
let {X,body} = c in
          body.methods.get(body.state);
▶ 5 : Nat
```

More generally, we can define a little function that "sends the `get` message" to any counter:

```
sendget = λc:Counter.
            let {X,body} = c in
            body.methods.get(body.state);
```

```
▶ sendget : Counter → Nat
```

Invoking the `inc` method of a counter object is a little more complicated. If we simply do the same as for `get`, the typechecker complains

```
 let {X,body} = c in
          body.methods.inc(body.state);
▶ Error: Scoping error!
```

because the type variable `X` appears free in the type of the body of the `let`. Indeed, what we've written doesn't make intuitive sense either, since the result of the `inc` method is a "bare" internal state, not an object. To satisfy both the typechecker and our informal understanding of what invoking `inc` should do, we must take this fresh internal state and "repackage" it as a counter object, using the same record of methods and the same internal state type as in the original object:

```
c1 = let {X,body} = c in
       {∃X = X,
        {state = body.methods.inc(body.state),
         methods = body.methods}}
 as Counter;
```

More generally, to "send the `inc` message" to an arbitrary counter object, we can write:

```
sendinc = λc:Counter.
            let {X,body} = c in
              {∃X = X,
               {state = body.methods.inc(body.state),
                methods = body.methods}}
              as Counter;
▶ sendinc : Counter → Counter
```

More complex operations on counters can be implemented in terms of these two basic operations:

```
addthree = λc:Counter. sendinc (sendinc (sendinc c));
▶ addthree : Counter → Counter
```

19.2.2 Exercise: Implement `FlipFlop` objects with `Counter` objects as their internal representation type, following the model of the `FlipFlop` ADT in Section 19.2. □

Objects vs. ADTs

What we have seen in Section 19.2 falls significantly short of a full-blown model of object-oriented programming. Many of the features that we saw in Chapter 14, including subtyping, classes, inheritance, and recursion through `self` and `super`, are missing here. We will come back to modeling these features in later chapters, when we have added a few necessary refinements to our modeling language. But even for the simple objects we have developed so far, there are several interesting comparisons to be made with ADTs.

At the coarsest level, the two programming idioms fall at opposite ends of a spectrum: when programming with ADTs, packages are opened immediately after they are built; on the other hand, when packages are used to model objects they are kept closed as long as possible—until the moment when they *must* be opened so that one of the methods can be applied to the internal state.

A consequence of this difference is that the "abstract type" of counters refers to different things in the two styles. In an ADT-style program, the counter values manipulated by client code such as `addthree` are elements of the underlying representation type (e.g., simple numbers). In an object-style program, a counter value is a whole package—not only a number, but also the implementations of the `get` and `inc` methods. This stylistic difference is reflected in the fact that, in the ADT style, the type `Counter` is a bound type variable introduced by the `let` construct, while in the object style `Counter` abbreviates the whole existential type $\{\exists X, \{state:X, methods: \{get:X\to Nat, inc:X\to X\}\}\}$. Thus:

- All the counter values generated from the counter ADT are elements of the same internal representation type; there is a single implementation of the counter operations that works on this internal representation.
- Each counter object, on the other hand, carries its own representation type and its own set of methods that work for this representation type.

19.2.3 Exercise: In what ways do the *classes* found in mainstream object-oriented languages like C++ and Java resemble the simple object types discussed here? In what ways do they resemble ADTs? □

19.3 Encoding Existentials

The encoding of pairs as a polymorphic type in Exercise 18.4.5 suggests a similar encoding for existential types in terms of universal types, using the intuition that an element of an existential type is a pair of a type and a value:

$$\{\exists X, T\} \stackrel{\text{def}}{=} \forall Y.\ (\forall X.\ T\to Y) \to Y.$$

That is, an existential package is thought of as a data value that, given a result type and a "continuation," calls the continuation to yield a final result. The continuation

takes two arguments—a type X and a value of type T—and uses them in computing the final result.

Given this encoding of existential types, the encoding of the packaging and unpackaging constructs is essentially forced. To encode a package

```
{∃X=S,t} as {∃X,T}
```

we must exhibit a value of type ∀Y. (∀X. T→Y) → Y. This type begins with a universal quantifier, the body of which is an arrow. An element of this type should therefore begin with two abstractions:

```
{∃X=S,lt} as {∃X,T}  ≝  λY. λf:(∀X.T→Y). ...
```

To complete the job, we need to return a result of type Y; clearly, the only way to do this is to apply f to some appropriate arguments. First, we supply the type S (this is a natural choice, being the only type we have lying around at the moment):

```
{∃X=S,t} as {∃X,T}  ≝  λY. λf:(∀X.T→Y). f [S] ...
```

Now, the type application f [S] has type {X ↦ S}(T→Y), i.e., ({X ↦ S}T)→Y. We can thus supply t (which, by rule T-PACK, has type {X ↦ S}T) as the next argument:

```
{∃X=S,t} as {∃X,T}  ≝  λY. λf:(∀X.T→Y). f [S] t
```

The type of the whole application f [S] t is now Y, as required.

To encode the unpacking construct

```
let {X,x}=t1 in t2  ≝  ...
```

we proceed as follows. First, the typing rule T-UNPACK tells us that t_1 should have some type {∃X,T_{11}}, that t_2 should have type T_2 (under an extended context binding X and x:T_{11}), and that T_2 is the type we expect for the whole let...in... expression.[2] As in the Church encodings in Section 18.4, the intuition here is that the introduction form ({∃X=S,t}) is encoded as an active value that "performs its own elimination." So the encoding of the elimination form here should simply take the existential package t_1 and apply it to enough arguments to yield a result of the desired type T_2:

```
let {X,x}=t1 in t2  ≝  t1 ...
```

The first argument to t_1 should be the desired result of the whole expression, i.e., T_2:

```
let {X,x}=t1 in t2  ≝  t1 [T2] ...
```

[2] Strictly speaking, the fact that the translation requires these extra bits of type information not present in the syntax of terms means that what we are translating is actually *typing derivations*, not terms.

Now, the application $t_1\ [T_2]$ has type $(\forall X.\ T{\to}T_2) \to T_2$. That is, if we can now supply another argument of type $(\forall X.T{\to}T_2)$, we will be finished. Such an argument can be obtained by *abstracting* the body t_2 on the variables `X` and `x`:

$$\texttt{let } \{X,x\}{=}t_1 \texttt{ in } t_2 \stackrel{\text{def}}{=} t_1\ [T_2]\ (\lambda X.\ \lambda x{:}T_{11}.\ t_2).$$

This finishes the encoding.

19.3.1 Exercise: What must we prove to show that our encoding of existentials is correct? □

19.3.2 Exercise [Recommended]: Take a blank piece of paper and, without looking at the above encoding, regenerate it from scratch. □

19.3.3 Exercise: Can universal types be encoded in terms of existential types? □

19.4 Implementation

```
type ty =
    ...
  | TySome of string * ty

type term =
    ...
  | TmPack of info * string * ty * term * ty
  | TmUnpack of info * string * string * term * term
```

19.5 Historical Notes

The correspondence between ADTs and existential types was first developed by Mitchell and Plotkin [MP88]. (They also noticed the correspondance with objects.)

Chapter 20

Bounded Quantification

Many of the interesting problems in type systems arise from the *combination* of features that, in themselves, may be quite simple. In this chapter, we encounter our first substantial example: a system that mixes subtyping with polymorphism.

The most basic combination of these features is actually quite straightforward. We simply add to the subtyping relation a rule for comparing quantified types:

$$\frac{S <: T}{\forall X.S <: \forall X.T}$$

We consider here a more interesting combination, in which the syntax, typing, and subtyping rules for universal quantifiers are actually refined to take subtyping into account. The resulting notion of **bounded quantification** substantially increases both the expressive power of the system and its metatheoretic complexity.

20.1 Motivation

To see why we might want to combine subtyping and polymorphism in this more intimate manner, consider the identity function on records with a numeric field `a`:

```
f = λx:{a:Nat}. x;
▶ f : {a:Nat} → {a:Nat}
```

If we define a record of this form

```
ra = {a=0};
```

then we can apply `f` to `ra` (in any of the systems that we have seen), yielding a record of the same form.

```
(f ra);
▶ {a=0} : {a:Nat}
```

If we define a larger record `rab` with two fields, `a` and `b`,

```
rab = {a=0, b=true};
```

we can also apply `f` to `rab`, using the rule of subsumption introduced in Chapter 13.

```
(f rab);
▶ {a=0, b=true} : {a:Nat}
```

However, the type of the result has only the field `a`, which means that a term like `(f rab).b` will be judged ill typed. In other words, by passing `rab` through the identity function, we have lost the ability to access its `b` field!

Using the polymorphism of System F, we can write `f` in a different way:

```
fpoly = λX. λx:X. x;
▶ fpoly : ∀X. X → X
```

The application of `fpoly` to `rab` (and an appropriate type argument) yields the desired result:

```
(fpoly [{a:Nat, b:Bool}] rab);
▶ {a=0, b=true} : {a:Nat, b:Bool}
```

But in making the type of `x` into a variable, we have given up some information that `f` might have wanted to use. For example, suppose we intend that `f` return a pair of its original argument and the numeric successor of its `a` field.

```
f2 = λx:{a:Nat}. {orig=x, asucc=succ(x.a)};
▶ f2 : {a:Nat} → {orig:{a:Nat}, asucc:Nat}
```

Again, we can apply `f2` to both `ra` and `rab`, losing the `b` field in the second case.

```
(f2 ra);
▶ {orig={a=0}, asucc=1} : {orig:{a:Nat}, asucc:Nat}

(f2 rab);
▶ {orig={a=0,b=true}, asucc=1} : {orig:{a:Nat}, asucc:Nat}
```

But this time polymorphism offers us no solution. If we replace the type of `x` by a variable `X` as before, we lose the constraint that `x` must have an `a` field, which is required to compute the `asucc` field of the result.

```
f2poly = λX. λx:X. {orig=x, asucc=succ(x.a)};
▶ Error: Expected record type
```

The fact about the operational behavior of `f2` that we want to express in its type is:

> `f2` takes an argument of any record type `R` that includes a numeric `a` field and returns as its result a record containing a field of type `R` and a field of type `Nat`.

We can use the subtype relation to express this concisely as follows:

> `f2` takes an argument of any subtype `R` of the type `{a:Nat}` and returns a record containing a field of type `R` and a field of type `Nat`.

This intuition can be formalized by introducing a **subtyping constraint** on the bound variable `X` of `f2poly`.

```
f2poly = λX<:{a:Nat}. λx:X. {orig=x, asucc=succ(x.a)};
▶ f2poly : ∀X<:{a:Nat}. X → {orig:X, asucc:Nat}
```

This interaction of subtyping and polymorphism, called **bounded quantification**, leads us to a type system commonly called System $F_{<:}$ ("F sub"), which is the topic of this chapter.

20.2 Definitions

We form System $F_{<:}$ by combining the types and terms of System F with the subtype relation from Chapter 13 and refining universal quantifiers with subtyping constraints on their bound variables. When we define the subtyping rule for these bounded quantifiers, there will actually be two choices: a more tractable but less flexible rule called the **kernel** rule and a more expressive **full** subtyping rule, which will turn out to raise some unexpected difficulties when we come to designing typechecking algorithms.

Kernel $F_{<:}$

Since type variables now have associated bounds (just as ordinary variables have associated types), we must keep track of them during both subtyping and typechecking. We change the type bindings in contexts to include an upper bound for each type variable, and add contexts to all the rules in the subtype relation. These bounds will be used during subtyping to justify steps of the form "the type variable `X` is a subtype of the type `T` because we assumed it was."

$$\frac{X<:T \in \Gamma}{\Gamma \vdash X <: T} \qquad \text{(S-TVAR)}$$

20.2.1 Exercise [Quick check]: Exhibit a subtyping derivation showing that

$$B<:\mathtt{Top},\ X<:B,\ Y<:X \vdash B\to Y <:\ X\to B.$$

□

Next, we introduce bounded universal types, extending the syntax and typing rules for ordinary universal types in the obvious way. The only rule where the extension is not completely obvious is the subtyping rule for quantified types, S-ALL. We give here the simpler variant, called the **kernel** subtyping rule for universal quantifiers, in which the bounds of the two quantifiers being compared must be identical. (The term "kernel" comes from Cardelli and Wegner's original paper [CW85], where this variant of $F_{<:}$ was called **Kernel Fun**.)

$$\frac{\texttt{X<:T} \in \Gamma}{\Gamma \vdash \texttt{X} <: \texttt{T}} \qquad \text{(S-TVAR)}$$

For easy reference, here is the complete definition of kernel $F_{<:}$, with differences from previous systems highlighted:

$F^k_{<:}$: Bounded quantification $\rightarrow \forall <: bq$

Syntax

t ::=		(*terms...*)
	x	*variable*
	λx:T.t	*abstraction*
	t t	*application*
	λX<:T.t	*type abstraction*
	t [T]	*type application*
v ::=		(*values...*)
	λx:T.t	*abstraction value*
	λX<:T.t	*type abstraction value*
T ::=		(*types...*)
	X	*type variable*
	Top	*maximum type*
	T→T	*type of functions*
	∀X<:T.T	*universal type*
Γ ::=		(*contexts...*)
	∅	*empty context*
	Γ, x:T	*term variable binding*
	Γ, X<:T	*type variable binding*

Evaluation $(t \longrightarrow t')$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow \{x \mapsto v_2\}t_{12} \qquad \text{(E-BETA)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-APP1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-App2)}$$

$$(\lambda X <: T_{11}.t_{12})\ [T_2] \longrightarrow \{X \mapsto T_2\}t_{12} \quad \text{(E-Beta2)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]} \quad \text{(E-TApp)}$$

Subtyping ($\Gamma \vdash S <: T$)

$$\Gamma \vdash S <: S \quad \text{(S-Refl)}$$

$$\frac{\Gamma \vdash S <: U \qquad \Gamma \vdash U <: T}{\Gamma \vdash S <: T} \quad \text{(S-Trans)}$$

$$\Gamma \vdash S <: \mathtt{Top} \quad \text{(S-Top)}$$

$$\frac{X <: T \in \Gamma}{\Gamma \vdash X <: T} \quad \text{(S-TVar)}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash S_1 \to S_2 <: T_1 \to T_2} \quad \text{(S-Arrow)}$$

$$\frac{\Gamma, X <: U_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X <: U_1.S_2 <: \forall X <: U_1.T_2} \quad \text{(S-All)}$$

Typing ($\Gamma \vdash t : T$)

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-Var)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \to T_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-App)}$$

$$\frac{\Gamma, X <: T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X <: T_1.t_2 : \forall X <: T_1.T_2} \quad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash t_1 : \forall X <: T_{11}.T_{12} \qquad \Gamma \vdash T_2 <: T_{11}}{\Gamma \vdash t_1\ [T_2] : \{X \mapsto T_2\}T_{12}} \quad \text{(T-TApp)}$$

$$\frac{\Gamma \vdash t : S \qquad \Gamma \vdash S <: T}{\Gamma \vdash t : T} \quad \text{(T-Sub)}$$

Full $F_{<:}$

In kernel $F_{<:}$, two quantified types can only be compared if their upper bounds are identical. If we think of quantifiers as a kind of arrow types (since they classify functions from types to terms), then the kernel rule corresponds to a "covariant" version of the subtyping rule for arrows, in which the domain of an arrow type is not allowed to vary in subtypes:

$$\frac{S_2 <: T_2}{U \to S_2 <: U \to T_2}$$

This restriction feels rather unnatural, both for arrows and for quantifiers. Carrying the analogy a little further, we can allow the "left-hand side" of bounded quantifiers to vary (contravariantly) during subtyping:

$F^f_{<:}$: "Full" bounded quantification $\to \forall <: bq\ full$

New subtyping rules ($\Gamma \vdash S <: T$)

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma, X<:T_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X<:S_1.S_2 <: \forall X<:T_1.T_2} \quad \text{(S-ALL)}$$

Intuitively, the "full $F_{<:}$" quantifier subtyping rule can be understood as follows. A type $T = \forall X<:T_1.T_2$ describes a collection of polymorphic values (functions from types to values), each mapping subtypes of T_1 to instances of T_2. If T_1 is a subtype of S_1, then the domain of T is smaller than that of $S = \forall X<:S_1.S_2$, so S is a stronger constraint and describes a smaller collection of polymorphic values. Moreover, if, for each type U that is an acceptable argument to the functions in both collections (i.e., one that satisfies the more stringent requirement $U <: T_1$), the U-instance of S_2 is a subtype of the U-instance of T_2, then S is a "pointwise stronger" constraint and again describes a smaller collection of polymorphic values.

The system with just the kernel subtyping rule for quantified types is called Kernel $F_{<:}$ (or $F^k_{<:}$). The same system with the full quantifier subtyping rule is called Full $F_{<:}$ (or $F^f_{<:}$). The bare name $F_{<:}$ refers ambiguously to both systems.

20.2.2 Exercise [Quick check]: Give a couple of examples of pairs of types that are related by the subtype relation of full $F_{<:}$ but are not subtypes in kernel $F_{<:}$. □

20.2.3 Exercise [Challenging]: Can you find any *useful* examples with this property? □

20.3 Examples

We now present some simple examples of programming in $F_{<:}$. More sophisticated uses of bounded quantification will appear in later chapters.

Encoding Products

In Exercise **??**, we gave the following encoding of pairs in System F. The elements of the type

```
Pair T1 T2 = ∀X. (T1→T2→X) → X;
```

correspond to pairs of T_1 and T_2. The constructor `pair` and the destructors `fst` and `snd` were defined as follows:

```
pair = λX. λY. λx:X. λy:Y.
         (   λR. λp:X→Y→R. p x y
          as Pair X Y);

fst = λX. λY. λp: Pair X Y.
        p [X] (λx:X. λy:Y. x);

snd = λX. λY. λp: Pair X Y.
        p [Y] (λx:X. λy:Y. y);
```

Of course, the same encoding can be used in $F_{<:}$, since $F_{<:}$ contains all the features of System F. What is interesting, though, is that this encoding also has some natural subtyping properties. In fact, the expected subtyping rule for pairs

$$\frac{\Gamma \vdash S_1 <: T_1 \qquad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash \texttt{Pair}\ S_1\ S_2 <: \texttt{Pair}\ T_1\ T_2}$$

follows directly from the encoding.

20.3.1 Exercise [Quick check]: Show this. □

Encoding Records

It is interesting to notice that records and record types—including subtyping—can actually be encoded in the pure calculus. The encoding presented here was discovered by Cardelli [Car92]. We begin by defining **flexible tuples** as follows:

20.3.2 Definition: For each $n \geq 0$ and types T_1 through T_n, let

$$\{T_{i\ i\in 1..n}\} \stackrel{\text{def}}{=} \texttt{Pair}\ T_1\ (\texttt{Pair}\ T_2\ \ldots\ (\texttt{Pair}\ T_n\ \texttt{Top})\ldots).$$

In particular, $\{\} = \texttt{Top}$. Similarly, for terms t_1 through t_n, let

$$\{t_{i\ i\in 1..n}\} \stackrel{\text{def}}{=} \texttt{pair}\ t_1\ (\texttt{pair}\ t_2\ \ldots\ (\texttt{pair}\ t_n\ \texttt{top})\ldots),$$

where we elide the type arguments to `pair`, for the sake of brevity. (Recall that `top` is just some element of `Top`.) The projection `t.n` (again eliding type arguments) is:

$$\texttt{fst}(\underbrace{\texttt{snd}(\texttt{snd}\ldots(\texttt{snd}}_{n-1 \text{ times}}\texttt{t})\ldots) \qquad \square$$

From this abbreviation, we immediately obtain the following rules for subtyping and typing.

$$\frac{\Gamma \vdash^{i\in 1..n} S_i <: T_i}{\Gamma \vdash \{S_i{}^{i\in 1..n+k}\} <: \{T_i{}^{i\in 1..n}\}}$$

$$\frac{\Gamma \vdash^{i\in 1..n} t_i : T_i}{\Gamma \vdash \{t_i{}^{i\in 1..n}\} : \{T_i{}^{i\in 1..n}\}}$$

$$\frac{\Gamma \vdash t : \{T_i{}^{i\in 1..n}\}}{\Gamma \vdash t.i : T_i}$$

Now, let $\mathcal{L}$ be a countable set of labels, with a fixed total ordering given by the bijective function *label-with-index* : $\mathbb{N} \to \mathcal{L}$. We define records as follows:

20.3.3 Definition: Let L be a finite subset of $\mathcal{L}$ and let S_l be a type for each $l \in L$. Let m be the maximal index of any element of L, and

$$\hat{S}_i = \begin{cases} S_l & \text{if } \textit{label-with-index}(i) = l \in L \\ \texttt{Top} & \text{if } \textit{label-with-index}(i) \notin L. \end{cases}$$

The record type $\{l{:}S_l{}^{l\in L}\}$ is defined as the flexible tuple $\{\hat{S}_i{}^{i\in 1..m}\}$. Similarly, if t_l is a term for each $l : L$, then

$$\hat{t}_i = \begin{cases} t_i & \text{if } \textit{label-with-index}(i) = l \in L \\ \texttt{top} & \text{if } \textit{label-with-index}(i) \notin L. \end{cases}$$

The record value $\{l{=}t_l{}^{l\in L}\}$ is $\{\hat{t}_i{}^{i\in 1..m}\}$. The projection $t.l_i$ is just the tuple projection `t.i`. $\square$

This encoding validates the expected rules for typing and subtyping:

$$\Gamma \vdash \{l_i{:}T_i{}^{i\in 1..n+k}\} <: \{l_i{:}T_i{}^{i\in 1..n}\} \qquad \text{(S-RCD-WIDTH)}$$

$$\frac{\text{for each } i \quad \Gamma \vdash S_i <: T_i}{\Gamma \vdash \{l_i{:}S_i{}^{i\in 1..n}\} <: \{l_i{:}T_i{}^{i\in 1..n}\}} \qquad \text{(S-RCD-DEPTH)}$$

Church Encodings with Subtyping

As a last simple illustration of the expressiveness of $F_{<:}$, let's take a look at what happens when we add bounded quantification to the encoding of Church Numerals in System F that we saw in Section 18.4. The original polymorphic type of church numerals was:

```
CNat = ∀X. (X→X) → X → X;
```

The intuitive reading of this type was: "Tell me a result type `T` and give me a function on `T` and an element of `T`, and I'll give you back another element of `T` formed by iterating the function you gave me `n` times over the base value you gave."

We can generalize this by adding two bounded quantifiers and refining the types of the parameters `s` and `z`.

```
SNat = ∀X<:Top. ∀S<:X. ∀Z<:X. (X→S) → Z → X;
```

Intuitively, this type can be read as follows: "Give me a generic result type `T` and two subtypes `S` and `Z`. Then give me a function that maps from the whole set `T` into the subset `S` and an element of the special set `Z`, and I'll return you an element of `T` formed in the same way as before."

To see why this is an interesting generalization, consider this slightly different type:

```
SZero = ∀X<:Top. ∀S<:X. ∀Z<:X. (X→S) → Z → Z;
```

Although `SZero` has almost the same form as `SNat`, it says something much stronger about the behavior of its elements, since it promises that its result will be an element of `Z`, not just of `T`. In fact, there is just one way that an element of `Z` could be returned—namely by yielding just `z` itself. In other words, the value

```
szero = (λX. λS<:X. λZ<:X. λs:X→S. λz:Z. z) as SZero;
▶ szero : SZero
```

is the *only* inhabitant of the type `SZero`. On the other hand, the similar type

```
SPos = ∀X<:Top. ∀S<:X. ∀Z<:X. (X→S) → Z → S;
```

has more inhabitants; for example,

```
sone   = (λX. λS<:X. λZ<:X. λs:X→S. λz:Z. s z) as SPos;
stwo   = (λX. λS<:X. λZ<:X. λs:X→S. λz:Z. s (s z)) as SPos;
sthree = (λX. λS<:X. λZ<:X. λs:X→S. λz:Z. s (s (s z))) as SPos;
```

and so on.

Moreover, notice that `SZero` and `SPos` are both subtypes of `SNat` (Exercise: check this), so we also have `szero` ∶ `SNat`, `sone` ∶ `SNat`, `stwo` ∶ `SNat`, etc.

Finally, we can similarly refine the typings of operations defined on church numerals. For example, the type system is capable of detecting that the successor function always returns a positive number:

```
ssucc = λn:SNat.
          (λX. λS<:X. λZ<:X. λs:X→S. λz:Z.
            s (n [X] [S] [Z] s z))
          as SPos;
```

```
▶ ssucc : SNat → SPos
```

Similarly, by refining the types of its parameters, we can write the function `plus` in such a way that the typechecker gives it the refined type SPos→SZero→SPos.

```
spluspz = λn:SPos. λm:SZero.
             (λX. λS<:X. λZ<:X. λs:X→S. λz:Z.
               n [X] [S] [Z] s (m [X] [S] [Z] s z))
             as SPos;
```

```
▶ spluspz : SPos → SZero → SPos
```

20.3.4 Exercise: Write a similar version of `plus` that has type SPos→SPos→SPos. Write one that has type SNat→SNat→SNat. □

The previous example and exercise raise an interesting point: obviously, we don't want to have several different versions of `plus` lying around and have to decide which to apply based on the expected types of its arguments: we want to have a *single* version of `plus` whose type contains all these possibilities—something like

```
plus :   SZero→SZero→SZero
       ∧ SNat→SPos→SPos
       ∧ SPos→SNat→SPos
       ∧ SNat→SNat→SNat
```

The desire to support this kind of overloading has led to the study of systems with **intersection types**.

20.3.5 Exercise [Recommended]: Generalize the type `CBool` of Church Booleans from Section 18.4 in a similar way by defining a type `SBool` and two subtypes `STrue` and `SFalse`. Write a function `notft` with type SFalse→STrue and a similar one `nottf` with type STrue→SFalse. □

The examples that we have seen in this section are amusing to play with, but they might not convince you that $F_{<:}$ is a system of tremendous practical importance! We will come to some more interesting uses of bounded quantification in Chapter 28, but these will require just a little more machinery, which we will develop in the intervening chapters.

20.4 Safety

We now consider the metatheory of both kernel and full systems of bounded quantification ($F^k_{<:}$ and $F^f_{<:}$). Much of the development is the same for both systems: we carry it out first for the simpler case of $F^k_{<:}$ and then consider $F^f_{<:}$.

The type preservation property can actually be proved quite directly for both systems, with minimal technical preliminaries. This is good, since the soundness of the type system is a critical property, while other properties such as decidability may be less important in some contexts. (The soundness theorem belongs in the language definition, while decision procedures are buried in the compiler.) We develop the proof in detail for $F^k_{<:}$. The argument for $F^f_{<:}$ is very similar.

We begin with a couple of technical facts about the typing and subtyping relations. The proofs go by straightforward induction on derivations.

20.4.1 Lemma [Permutation]:

1. If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and Δ is a permutation of Γ, then $\Delta \vdash \mathtt{t} : \mathtt{T}$.
2. If $\Gamma \vdash \mathtt{S} <: \mathtt{T}$ and Δ is a permutation of Γ, then $\Delta \vdash \mathtt{S} <: \mathtt{T}$. □

20.4.2 Lemma [Weakening]:

1. If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{x} \notin \mathit{dom}(\Gamma)$, then $\Gamma, \mathtt{x}:\mathtt{S} \vdash \mathtt{t} : \mathtt{T}$.
2. If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{X} \notin \mathit{dom}(\Gamma)$, then $\Gamma, \mathtt{X}<:\mathtt{S} \vdash \mathtt{t} : \mathtt{T}$.
3. If $\Gamma \vdash \mathtt{S} <: \mathtt{T}$ and $\mathtt{x} \notin \mathit{dom}(\Gamma)$, then $\Gamma, \mathtt{x}:\mathtt{S} \vdash \mathtt{S} <: \mathtt{T}$.
4. If $\Gamma \vdash \mathtt{S} <: \mathtt{T}$ and $\mathtt{X} \notin \mathit{dom}(\Gamma)$, then $\Gamma, \mathtt{X}<:\mathtt{S} \vdash \mathtt{S} <: \mathtt{T}$. □

As usual, the proof of type preservation relies on several lemmas relating substitution with the typing and subtyping relations.

20.4.3 Definition: We write $\{\mathtt{X} \mapsto \mathtt{S}\}\Gamma$ for the context obtained by substituting $\mathtt{S}$ for $\mathtt{X}$ in the right-hand sides of all of the bindings in Γ. □

20.4.4 Exercise [Quick check]: Show the following properties of subtyping and typing derivations:

1. if $\Gamma, \mathtt{X}<:\mathtt{Q}, \Delta \vdash \mathtt{S} <: \mathtt{T}$ and $\Gamma \vdash \mathtt{P} <: \mathtt{Q}$, then $\Gamma, \mathtt{X}<:\mathtt{P}, \Delta \vdash \mathtt{S} <: \mathtt{T}$;
2. if $\Gamma, \mathtt{X}<:\mathtt{Q}, \Delta \vdash \mathtt{t} : \mathtt{T}$ and $\Gamma \vdash \mathtt{P} <: \mathtt{Q}$, then $\Gamma, \mathtt{X}<:\mathtt{P}, \Delta \vdash \mathtt{t} : \mathtt{T}$;

These properties are often called **narrowing** because they involve restricting the range of the variable $\mathtt{X}$. □

Next, we have the usual lemma relating substitution and the typing relation.

20.4.5 Lemma [Substitution preserves typing]: If $\Gamma, \mathtt{x}:\mathtt{Q}, \Delta \vdash \mathtt{t} : \mathtt{T}$ and $\Gamma \vdash \mathtt{q} : \mathtt{Q}$, then $\Gamma, \Delta \vdash \{\mathtt{x} \mapsto \mathtt{q}\}\mathtt{t} : \mathtt{T}$. □

Proof: Straightforward induction on a derivation of $\Gamma, \mathtt{x}:\mathtt{Q}, \Delta \vdash \mathtt{t} : \mathtt{T}$, using the properties proved above. □

Since we may substitute types for type variables during reduction, we also need a lemma relating type substitution and typing, as we did in System F. Here, though, we must deal with one new twist: the proof of this lemma (specifically, the T-SUB case) depends on a new lemma relating substitution and subtyping:

20.4.6 Lemma [Type substitution preserves subtyping]: If $\Gamma, \mathtt{X}<:\mathtt{Q}, \Delta \vdash \mathtt{S} <: \mathtt{T}$ and $\Gamma \vdash \mathtt{P} <: \mathtt{Q}$, then $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{S} <: \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}$. □

Proof: By induction on a derivation of $\Gamma, \mathtt{X}<:\mathtt{Q}, \Delta \vdash \mathtt{S} <: \mathtt{T}$. The only interesting cases are the last two:

Case S-TVAR: $\mathtt{S} = \mathtt{Y}$ $\mathtt{Y}<:\mathtt{T} \in (\Gamma, \mathtt{X}<:\mathtt{Q}, \Delta)(\mathtt{Y})$

There are two subcases to consider. If $\mathtt{Y} \neq \mathtt{X}$, then the result follows immediately from S-TVAR. On the other hand, if $\mathtt{Y} = \mathtt{X}$, then we have $\mathtt{T} = \mathtt{Q}$ and $\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{S} = \mathtt{Q}$, and the result follows by S-REFL.

Case S-ALL: $\mathtt{S} = \forall \mathtt{Z}<:\mathtt{U}_1.\mathtt{S}_2$ $\mathtt{T} = \forall \mathtt{Z}<:\mathtt{U}_1.\mathtt{T}_2$
$\Gamma, \mathtt{X}<:\mathtt{Q}, \Delta, \mathtt{Z}<:\mathtt{U}_1 \vdash \mathtt{S}_2 <: \mathtt{T}_2$

By the induction hypothesis, $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta, \mathtt{Z}<:\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{U}_1 \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{S}_2 <: \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}_2$. By S-ALL, $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \forall \mathtt{Z}<:\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{U}_1.\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{S}_2 <: \forall \mathtt{Z}<:\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{U}_1.\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}_2$, that is, $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}(\forall \mathtt{Z}<:\mathtt{U}_1.\mathtt{S}_2) <: \{\mathtt{X} \mapsto \mathtt{P}\}(\forall \mathtt{Z}<:\mathtt{U}_1.\mathtt{T}_2)$, as required. □

20.4.7 Lemma [Type substitution preserves typing]: If $\Gamma, \mathtt{X}<:\mathtt{Q}, \Delta \vdash \mathtt{t} : \mathtt{T}$ and $\Gamma \vdash \mathtt{P} <: \mathtt{Q}$, then $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{t} : \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}$. □

Proof: By induction on a derivation of $\Gamma, \mathtt{X}<:\mathtt{Q}, \Delta \vdash \mathtt{t} : \mathtt{T}$. We give just the interesting cases.

Case T-TAPP: $\mathtt{t} = \mathtt{t}_1\ [\mathtt{T}_2]$ $\Gamma, \mathtt{X}<:\mathtt{Q}, \Delta \vdash \mathtt{t}_1 : \forall \mathtt{Z}<:\mathtt{T}_{11}.\mathtt{T}_{12}$
$\mathtt{T} = \{\mathtt{Z} \mapsto \mathtt{T}_2\}\mathtt{T}_{12}$

By the induction hypothesis, $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{t}_1 : \{\mathtt{X} \mapsto \mathtt{P}\}(\forall \mathtt{Z}<:\mathtt{T}_{11}.\mathtt{T}_{12})$, i.e, $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{t}_1 : \forall \mathtt{Z}<:\mathtt{T}_{11}.\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}_{12}$. By T-TAPP, $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{t}_1\ [(\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}_2]) : \{\mathtt{Z} \mapsto \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}_2\}(\{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}_{12})$, i.e., $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}(\mathtt{t}_1\ [\mathtt{T}_2]) : \{\mathtt{X} \mapsto \mathtt{P}\}(\{\mathtt{Z} \mapsto \mathtt{T}_2\}\mathtt{T}_{12})$.

Case T-SUB: $\Gamma, \mathtt{X}<:\mathtt{Q}, \Delta \vdash \mathtt{t} : \mathtt{S}$ $\Gamma, \mathtt{X}<:\mathtt{Q}, \Delta \vdash \mathtt{S} <: \mathtt{T}$

By the induction hypothesis, $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{t} : \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}$. By the preservation of subtyping under substitution (20.4.6), $\Gamma, \{\mathtt{X} \mapsto \mathtt{P}\}\Delta \vdash \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{S} <: \{\mathtt{X} \mapsto \mathtt{P}\}\mathtt{T}$, and the result follows by T-SUB. □

Next, we establish some simple structural facts about the subtype relation.

20.4.8 Lemma [Inversion of the subtyping relation, from right to left]:

1. If $\Gamma \vdash S <: X$, then S is a type variable.
2. If $\Gamma \vdash S <: T_1 \to T_2$, then either S is a type variable or else $S = S_1 \to S_2$ with $\Gamma \vdash T_1 <: S_1$ and $\Gamma \vdash S_2 <: T_2$.
3. If $\Gamma \vdash S <: \forall X<:U_1.T_2$, then either S is a type variable or else $S = \forall X<:U_1.S_2$ with $\Gamma, X<:U_1 \vdash S_2 <: T_2$. □

Proof: Part (1) follows by an easy induction on subtyping derivations. The only interesting case is the rule S-TRANS, which proceeds by two uses of the induction hypothesis, first on the right premise and then on the left. The arguments for the other parts are similar (part (1) is used in the transitivity cases). □

20.4.9 Exercise: Show the following "left to right inversion" properties:

1. If $\Gamma \vdash S_1 \to S_2 <: T$, then either $T = \mathrm{Top}$ or else $T = T_1 \to T_2$ with $\Gamma \vdash T_1 <: S_1$ and $\Gamma \vdash S_2 <: T_2$.
2. If $\Gamma \vdash \forall X<:U.S_2 <: T$, then either $T = \mathrm{Top}$ or else $T = \forall X<:U.T_2$ with $\Gamma, X<:U \vdash S_2 <: T_2$.
3. If $\Gamma \vdash X <: T$, then either $T = \mathrm{Top}$ or $T = X$ or $\Gamma \vdash S <: T$, where $X<:S \in \Gamma$.
4. If $\Gamma \vdash \mathrm{Top} <: T$, then $T = \mathrm{Top}$. □

We use Lemma 20.4.8 for one straightforward structural property of the typing relation that will be needed in the critical cases of the type preservation proof.

20.4.10 Lemma:

1. If $\Gamma \vdash \lambda x:S_1.s_2 : T$ and $\Gamma \vdash T <: U_1 \to U_2$, then $\Gamma \vdash U_1 <: S_1$ and there is some S_2 such that $\Gamma, x:S_1 \vdash s_2 : S_2$ and $\Gamma \vdash S_2 <: U_2$.
2. If $\Gamma \vdash \lambda X<:S_1.s_2 : T$ and $\Gamma \vdash T <: \forall X<:U_1.U_2$, then $U_1 = S_1$ and there is some S_2 such that $\Gamma, X<:S_1 \vdash s_2 : S_2$ and $\Gamma, X<:S_1 \vdash S_2 <: U_2$. □

Proof: Straightforward induction on typing derivations, using Lemma 20.4.8 for the induction case (rule T-SUB). □

With all these facts in-hand, the actual proof of type preservation is straightforward.

20.4.11 Theorem [Preservation]: If $\Gamma \vdash t : T$ and $\Gamma \vdash t \longrightarrow t'$, then $\Gamma \vdash t' : T$. □

Proof: By induction on a derivation of $\Gamma \vdash t : T$. All of the cases are straightforward, using the facts established in the above lemmas.

Case T-VAR: $t = x$

This case cannot actually arise, since we assumed $\Gamma \vdash t \longrightarrow t'$ and there are no evaluation rules for variables.

Case T-ABS: $t = \lambda x{:}T_1.t_2$

Ditto.

Case T-APP: $t = t_1\ t_2 \qquad \Gamma \vdash t_1 : T_{11}{\to}T_{12}$
$T = T_{12} \qquad \Gamma \vdash t_2 : T_{11}$

By the definition of the evaluation relation, there are three subcases to consider:

Subcase: $\Gamma \vdash t_1 \longrightarrow t_1' \qquad t' = t_1'\ t_2$

Then the result follows from the induction hypothesis and T-APP.

Subcase: t_1 is a value $\qquad \Gamma \vdash t_2 \longrightarrow t_2' \qquad t' = t_1\ t_2'$

Similar.

Subcase: $t_1 = \lambda x{:}U_{11}.u_{12} \qquad t' = \{x \mapsto t_2\}u_{12}$

By Lemma 20.4.10, $\Gamma, x{:}U_{11} \vdash u_{12} : U_{12}$ with $\Gamma \vdash T_{11} <: U_{11}$ and $\Gamma \vdash U_{12} <: T_{12}$. By the preservation of typing under substitution (Lemma 20.4.5), $\Gamma \vdash \{x \mapsto t_2\}u_{12} : U_{12}$, from which $\Gamma \vdash \{x \mapsto t_2\}u_{12} : T_{12}$ follows by T-SUB.

Case T-TABS: $t = \lambda X{<:}U.t$

Can't happen.

Case T-TAPP: $t = t_1\ [T_2] \qquad \Gamma \vdash t : \forall X{<:}T_{11}.T_{12}$
$T = \{X \mapsto T_2\}T_{12} \qquad \Gamma \vdash T_2 <: T_{11}$

By the definition of the evaluation relation, there are two subcases to consider:

Subcase: $t_1 \longrightarrow t_1' \qquad t' = t_1'\ [T_2]$

The result follows from the induction hypothesis and T-TAPP.

Subcase: $t_1 = \lambda X{<:}U_{11}.u_{12} \qquad t' = \{X \mapsto T_2\}u_{12}$

By Lemma 20.4.10, $U_{11} = T_{11}$ and $\Gamma, X{<:}U_{11} \vdash u_{12} : U_{12}$ with $\Gamma, X{<:}U_{11} \vdash U_{12} <: T_{12}$. By the preservation of typing under substitution (20.4.5), $\Gamma \vdash \{X \mapsto T_2\}u_{12} : \{X \mapsto T_2\}U_{12}$, from which $\Gamma \vdash \{X \mapsto T_2\}u_{12} : \{X \mapsto T_2\}T_{12}$ follows by Lemma 20.4.6 and T-SUB.

Case T-SUB: $\Gamma \vdash t : S \qquad \Gamma \vdash S <: T$

By the induction hypothesis, $\Gamma \vdash t' : S$; the result follows by T-SUB. □

20.4.12 Exercise: Show how to extend the argument in this section to $F^f_{<:}$. □

20.5 Bounded Existential Types

This final section remains to be written.

Bounded existential quantification $F^k_{<:} + \exists$

New syntactic forms

T ::=	...	(*types...*)
	$\{\exists X <: T, T\}$	*existential type*

New subtyping rules ($\Gamma \vdash S <: T$)

$$\frac{\Gamma, X<:U \vdash S_2 <: T_2}{\Gamma \vdash \{\exists X<:U, S_2\} <: \{\exists X<:U, T_2\}} \quad \text{(S-SOME)}$$

New typing rules ($\Gamma \vdash t : T$)

$$\frac{\Gamma \vdash t_2 : \{X \mapsto U\}T_2 \qquad \Gamma \vdash U <: T_1}{\Gamma \vdash \{\exists X=U, t_2\} \text{ as } \{\exists X <: T_1, T_2\} : \{\exists X <: T_1, T_2\}} \quad \text{(T-PACK)}$$

$$\frac{\Gamma \vdash t_1 : \exists X <: T_{11}. T_{12} \qquad \Gamma, X <: T_{11}, x:T_{12} \vdash t_2 : T_2}{\Gamma \vdash \text{let } \{X,x\}=t_1 \text{ in } t_2 : T_2} \quad \text{(T-UNPACK)}$$

20.5.1 Exercise: Show how the subtyping rule S-SOME can be obtained from the subtyping rules for universals by extending the encoding of existential types in terms of universal types described in Section 19.3. □

20.6 Historical Notes and Further Reading

The idea of bounded quantification was introduced by Cardelli and Wegner [CW85] in the language Fun. (Their "Kernel Fun" calculus corresponds to our $F^k_{<:}$.) Based on informal ideas by Cardelli and formalized using techniques developed by Mitchell [Mit84b], Fun integrated Girard-Reynolds polymorphism [Gir72, Rey74] with Cardelli's first-order calculus of subtyping [**?**, Car84]. The original Fun was simplified and slightly generalized by Bruce and Longo [BL90], and again by Curien and Ghelli [CG92], yielding the calculus we call $F^f_{<:}$.

The most comprehensive single paper on bounded quantification is the survey by Cardelli, Martini, Mitchell, and Scedrov [CMMS94].

Fun and its relatives have been studied extensively by programming language theorists and designers. Cardelli and Wegner's survey paper gives the first programming examples using bounded quantification; more are developed in Cardelli's study of power kinds [Car88]. Curien and Ghelli [CG92, Ghe90] address a number of syntactic properties of $F_{<:}$. Semantic aspects of closely related systems have been studied by Bruce and Longo [BL90], Martini [Mar88], Breazu-Tannen, Coquand,

Gunter, and Scedrov [BCGS91], Cardone [Car89], Cardelli and Longo [CL91], Cardelli, Martini, Mitchell, and Scedrov [**?**], Curien and Ghelli [CG92, CG91], and Bruce and Mitchell [BM92]. $F_{<:}$ has been extended to include record types and richer notions of inheritance by Cardelli and Mitchell [CM91], Bruce [Bru91], Cardelli [Car92], and Canning, Cook, Hill, Olthoff, and Mitchell [CCH+89]. Bounded quantification also plays a key role in Cardelli's programming language Quest [Car91, CL91] and in the Abel language developed at HP Labs [CCHO89, CCH+89, CHO88, CHC90].

The undecidability of full $F_{<:}$ was shown by Pierce [Pie94] and further analyzed by Ghelli [Ghe95].

The effect of bounded quantification on Church encodings of algebraic datatypes (cf. Section 20.3) was considered by Ghelli thesis [Ghe90] and Cardelli, Martini, Mitchell, and Scedrov [CMMS94].

An extension of $F_{<:}$ with intersection types was studied by Pierce [Pie91a, Pie97] and applied to the modeling of object-oriented languages with multiple inheritance by Compagnoni and Pierce [CP96].

Chapter 21

Implementing Bounded Quantification

Next we consider the problem of building a typechecking algorithm for a language with bounded quantifiers. The algorithm that we construct will be parametric in an algorithm for the subtype relation, which we consider in the following section.

21.1 Promotion

In the typechecking algorithm for $\lambda_{<:}$ in Section 13.2, the key idea was that we can calculate a *minimal* type for each term from the minimal types of its subterms. We will use the same basic idea to typecheck $F^{k}_{<:}$, but we need to take into account one slight complication arising from the presence of type variables in the system.

Consider the term

```
f = λX<:Nat→Nat. λy:X. y 5;
▶ f : ∀X<:Nat→Nat. X → Nat
```

This term is clearly well typed, since the type of the variable `y` in the application `(y 5)` is `X`, which can be promoted to `Nat→Nat` by T-SUB. But the *minimal* type of `y` is not an arrow type. In order to find the minimal type of the application, we need to find the minimal arrow type that `y` possesses—i.e., the minimal arrow type that is a supertype of `X`. Not too surprisingly, the correct way to find this type is to **promote** the minimal type of `y` until it is something other than a type variable.

Formally, write $\Gamma \vdash \mathtt{S} \Uparrow \mathtt{T}$ to mean "`T` is the *least nonvariable supertype* of `S`," defined by repeated promotion of variables as follows:

Exposure

Exposure ($\Gamma \vdash_{\blacktriangleright} T \Uparrow T'$)

$$\frac{X{<:}T \in \Gamma \qquad \Gamma \vdash_{\blacktriangleright} T \Uparrow T'}{\Gamma \vdash_{\blacktriangleright} X \Uparrow T'} \qquad \text{(XA-PROMOTE)}$$

$$\frac{T \text{ is not a type variable}}{\Gamma \vdash_{\blacktriangleright} T \Uparrow T} \qquad \text{(XA-OTHER)}$$

It is easy to check that these rules define a total function. Moreover, the result of promotion is always the least supertype that has some shape other than a variable.

21.1.1 Lemma: Suppose $\Gamma \vdash S \Uparrow T$.

1. $\Gamma \vdash S <: T$.
2. If $\Gamma \vdash S <: U$ and U is not a variable, then $\Gamma \vdash T <: U$. □

Proof: Part (1) is easy. Part (2) goes by straightforward induction on a derivation of $\Gamma \vdash S <: U$. □

21.2 Minimal Typing

The algorithm for calculating minimal types is built along the same basic lines as the one for $\lambda_{<:}$, with one additional twist: the minimal type of a term may always be a type variable, and such a type will need to be promoted to its smallest non-variable supertype (its smallest **concrete** supertype, we might say) in order to be used on the left of an application or type application.

Algorithmic typing

Algorithmic typing ($\Gamma \vdash_{\blacktriangleright} t : T$)

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash_{\blacktriangleright} x : T} \qquad \text{(TA-VAR)}$$

$$\frac{\Gamma, x{:}T_1 \vdash_{\blacktriangleright} t_2 : T_2}{\Gamma \vdash_{\blacktriangleright} \lambda x{:}T_1.t_2 : T_1 \to T_2} \qquad \text{(TA-ABS)}$$

$$\frac{\Gamma \vdash_{\blacktriangleright} t_1 : T_1 \qquad \Gamma \vdash_{\blacktriangleright} T_1 \Uparrow (T_{11} \to T_{12}) \qquad \Gamma \vdash_{\blacktriangleright} t_2 : T_2 \qquad \Gamma \vdash_{\blacktriangleright} T_2 <: T_{11}}{\Gamma \vdash_{\blacktriangleright} t_1\ t_2 : T_{12}} \qquad \text{(TA-APP)}$$

$$\frac{\Gamma, X <: T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X <: T_1 . t_2 : \forall X <: T_1 . T_2} \quad \text{(TA-TABS)}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash T_1 \Uparrow \forall X <: T_{11} . T_{12} \qquad \Gamma \vdash T_2 <: T_{11}}{\Gamma \vdash t_1\ [T_2] : \{X \mapsto T_2\} T_{12}} \quad \text{(TA-TAPP)}$$

The proofs of soundness and completeness of this algorithm with respect to the original typing rules are fairly routine.

21.2.1 Theorem [Minimal typing]:

1. If $\Gamma \vdash t : T$, then $\Gamma \vdash t : T$.
2. If $\Gamma \vdash t : T$, then $\Gamma \vdash t : M$ where $\Gamma \vdash M <: T$. □

Proof: Part (1) proceeds by a straightforward induction on algorithmic derivations. Part (2) is more interesting; it goes by induction on a derivation of $\Gamma \vdash t : T$. (The most important cases are those for the rules T-APP and T-TAPP.)

Case T-VAR: $t = x \qquad x : T \in \Gamma$

By TA-VAR, $\Gamma \vdash x : T$. By S-REFL, $\Gamma \vdash T <: T$.

Case T-ABS: $t = \lambda x : T_1 . t_2 \qquad \Gamma, x : T_1 \vdash t_2 : T_2 \qquad T = T_1 \to T_2$

By the induction hypothesis, $\Gamma, x : T_1 \vdash t_2 : M_2$ for some M_2 with $\Gamma, x : T_1 \vdash M_2 <: T_2$, i.e. (since subtyping does not depend on term variable bindings), $\Gamma \vdash M_2 <: T_2$. By TA-ABS, $\Gamma \vdash t : T_1 \to M_2$. Finally, by S-REFL and S-ARROW, we have $\Gamma \vdash T_1 \to M_2 <: T_1 \to T_2$.

Case T-APP: $t = t_1\ t_2 \qquad \Gamma \vdash t_1 : T_{11} \to T_{12}$
$T = T_{12} \qquad \Gamma \vdash t_2 : T_{11}$

By the induction hypothesis, we have $\Gamma \vdash t_1 : M_1$ and $\Gamma \vdash t_2 : M_2$, with $\Gamma \vdash M_1 <: T_{11} \to T_{12}$ and $\Gamma \vdash M_2 <: T_{11}$. Let N_1 be the least nonvariable supertype of M_1—i.e., suppose $\Gamma \vdash M_1 \Uparrow N_1$. By the promotion lemma (21.1.1), $\Gamma \vdash N_1 <: T_{11} \to T_{12}$. But we know that N_1 is not a variable, so the inversion lemma for the subtype relation (20.4.8) tells us that $N_1 = N_{11} \to N_{12}$, with $\Gamma \vdash T_{11} <: N_{11}$ and $\Gamma \vdash N_{12} <: T_{12}$. By transitivity, $\Gamma \vdash M_2 <: N_{11}$, so rule TA-APP applies and gives us $\Gamma \vdash t_1\ t_2 : N_{12}$, which satisfies the requirements.

Case T-TABS: $t = \lambda X <: T_1 . t_2 \qquad \Gamma, X <: T_1 \vdash t_2 : T_2 \qquad T = \forall X <: T_1 . T_2$

By the induction hypothesis, $\Gamma, X <: T_1 \vdash t_2 : M_2$ for some M_2 with $\Gamma, X <: T_1 \vdash M_2 <: T_2$. By TA-TABS, $\Gamma \vdash t : \forall X <: T_1 . M_2$. Finally, by S-REFL and S-ALL, we have $\Gamma \vdash \forall X <: T_1 . M_2 <: \forall X <: T_1 . T_2$.

Case T-TAPP: $t = t_1\ [T_2]$ $\quad \Gamma \vdash t_1 : \forall X{<:}T_{11}.T_{12}$
$T = \{X \mapsto T_2\}T_{12}$ $\quad \Gamma \vdash T_2 <: T_{11}$

By the induction hypothesis, we have $\Gamma \vdash t_1 : M_1$, with $\Gamma \vdash M_1 <: \forall X{<:}T_{11}.T_{12}$. Let N_1 be the least nonvariable supertype of M_1—i.e., suppose $\Gamma \vdash M_1 \Uparrow N_1$. By the promotion lemma (21.1.1), $\Gamma \vdash N_1 <: \forall X{<:}T_{11}.T_{12}$. But we know that N_1 is not a variable, so the inversion lemma for the subtype relation (20.4.8) tells us that $N_1 = \forall X{<:}N_{11}.N_{12}$, with $N_{11} = T_{11}$ and $\Gamma, X{<:}T_{11} \vdash N_{12} <: T_{12}$. Rule TA-TAPP gives us $\Gamma \vdash t_1\ [T_2] : \{X \mapsto T_2\}N_{12}$, and the preservation of subtyping under substitution (20.4.6) yields $\Gamma \vdash \{X \mapsto T_2\}N_{12} <: \{X \mapsto T_2\}T_{12} = T$.

Case T-SUB: $\Gamma \vdash t : S$ $\quad \Gamma \vdash S <: T$

By the induction hypothesis, $\Gamma \vdash t : M$ with $\Gamma \vdash M <: S$. By S-TRANS, $\Gamma \vdash M <: T$, from which T-SUB yields the desired result. □

21.2.2 Corollary [Decidability of typing]: The $F^k_{<:}$ typing relation is decidable (if we are given a decision procedure for the subtyping relation). □

Proof: Given Γ and t, we can check whether there is some T such that $\Gamma \vdash t : T$ by using the algorithmic typing rules to generate a proof of $\Gamma \vdash t : T$. If we succeed, then this T is also a type for T in the original typing relation (by part (1) of 21.2.1). If not, then part (2) of 21.2.1 implies that t has no type in the original typing relation.

Finally, note that the algorithmic typing rules constitute a terminating algorithm, since they are syntax-directed and always reduce the size of t when read from bottom to top. □

21.2.3 Exercise: Show how to add primitive booleans and conditionals to the minimal typing algorithm for $F^k_{<:}$. (Solution on page 261.) □

21.3 Subtyping in $F^k_{<:}$

As we saw in the simply typed lambda-calculus with subtyping, the subtyping rules in their present form do not constitute an algorithm for deciding the subtyping relation. We cannot use them "from bottom to top," for two reasons:

1. There are some overlaps between the conclusions of different rules (specifically, between S-REFL and nearly all the other rules). That is, looking at the form of a derivable subtyping statement $\Gamma \vdash S <: T$, we cannot decide which of the rules must have been used last in deriving it.

2. More seriously, one rule (S-TRANS) contains a metavariable in the premises that does not appear in the conclusion. To apply this rule from botom to top, we'd need to guess what type to replace this metavariable with.

The overlap between S-REFL and the other rules is easily dealt with, using exactly the same technique as we used in Chapter 13: we remove the full reflexivity rule and replace it by a restricted reflexivity rule that applies only to type variables.

$$\Gamma \vdash X <: X$$

Next we must deal with S-TRANS. Unfortunately, unlike the simple subtyping relation studied in Chapter 13, the transitivity rule here interacts in an important way with another rule—namely S-TVAR, which allows assumptions about type variables to be used in deriving subtyping statements. For example, if

$$\Gamma \quad = \quad W<:\mathtt{Top},\ X<:W,\ Y<:X,\ Z<:Y$$

then the statement

$$\Gamma \vdash Z <: W$$

is provable using all the subtyping rules, but cannot be proved if S-TRANS is removed. That is, an instance of S-TRANS whose left-hand subderivation is an instance of the axiom S-TVAR, as in

$$\dfrac{\dfrac{}{\Gamma \vdash Z <: Y}\,(\text{S-TVAR}) \qquad \dfrac{\vdots}{\Gamma \vdash Y <: W}}{\Gamma \vdash Z <: W}\,(\text{S-TRANS})$$

cannot, in general, be eliminated.

Fortunately, it turns out that derivations of this form are the *only* essential uses of transitivity in subtyping. This observation can be made precise by introducing a new subtyping rule

$$\dfrac{X<:U \in \Gamma \qquad \Gamma \vdash U <: T}{\Gamma \vdash X <: T}$$

that captures exactly this pattern of variable lookup followed by transitivity, and showing (as we will do below) that replacing the transitivity and variable rules by this one does not change the set of derivable subtyping statements.

These intuitions are summarized in the following definition. The algorithmic subtype relation of $F^k_{<:}$ is the least relation closed under the following rules:

Algorithmic subtyping

Algorithmic subtyping ($\Gamma \vdash\!\!\blacktriangleright S <: T$)

$$\Gamma \vdash\!\!\blacktriangleright S <: \mathtt{Top} \qquad \text{(SA-TOP)}$$

$$\Gamma \vdash\!\!\blacktriangleright X <: X \qquad \text{(SA-REFL-TVAR)}$$

$$\dfrac{X<:U \in \Gamma \qquad \Gamma \vdash\!\!\blacktriangleright U <: T}{\Gamma \vdash\!\!\blacktriangleright X <: T} \qquad \text{(SA-TRANS-TVAR)}$$

$$\frac{\Gamma \vdash\!\!\blacktriangleright T_1 <: S_1 \qquad \Gamma \vdash\!\!\blacktriangleright S_2 <: T_2}{\Gamma \vdash\!\!\blacktriangleright S_1 \to S_2 <: T_1 \to T_2} \quad \text{(SA-ARROW)}$$

$$\frac{\Gamma, X<:U_1 \vdash\!\!\blacktriangleright S_2 <: T_2}{\Gamma \vdash\!\!\blacktriangleright \forall X<:U_1.S_2 <: \forall X<:U_1.T_2} \quad \text{(SA-ALL)}$$

21.3.1 Lemma [Reflexivity of the algorithmic subtype relation]: The statement $\Gamma \vdash\!\!\blacktriangleright T <: T$ is provable for all Γ and T. □

Proof: Easy induction on T. □

21.3.2 Lemma [Transitivity of the algorithmic subtype relation]: If $\Gamma \vdash\!\!\blacktriangleright S <: Q$ and $\Gamma \vdash\!\!\blacktriangleright Q <: T$, then $\Gamma \vdash\!\!\blacktriangleright S <: T$. □

Proof: By induction on the sum of the sizes of the two derivations. Given two derivations of some total size, we proceed by considering the final rules in each.

First, if the right-hand derivation is an instance of SA-TOP, then we are done, since $\Gamma \vdash S <: \mathrm{Top}$ by SA-TOP. Moreover, if the left-hand derivation is an instance of SA-TOP, then $Q = \mathrm{Top}$ and by looking at the algorithmic rules we see that the right-hand derivation must also be an instance of SA-TOP.

If either derivation is an instance of SA-REFL-TVAR, then we are again done since the other derivation is the desired result.

Next, if the left-hand derivation ends with an instance of SA-TRANS-TVAR, then $S = X$ with $X<:U \in \Gamma$ and we have a subderivation with conclusion $\Gamma \vdash U <: Q$. By the induction hypothesis, $\Gamma \vdash\!\!\blacktriangleright U <: T$, and, by SA-TRANS-TVAR again, $\Gamma \vdash\!\!\blacktriangleright X <: T$, as required.

If the left-hand derivation ends with an instance of SA-ARROW, then we have $S = S_1 \to S_2$ and $Q = Q_1 \to Q_2$, with subderivations $\Gamma \vdash\!\!\blacktriangleright Q_1 <: S_1$ and $\Gamma \vdash\!\!\blacktriangleright S_2 <: Q_2$. But, since we have already considered the case where the right-hand derivation is SA-TOP, the only remaining possibility is that the right-hand derivation also ends with SA-ARROW; we therefore have $T = T_1 \to T_2$, and two more subderivations $\Gamma \vdash\!\!\blacktriangleright T_1 <: Q_1$ and $\Gamma \vdash\!\!\blacktriangleright Q_2 <: T_2$. We now apply the induction hypothesis twice, obtaining $\Gamma \vdash\!\!\blacktriangleright T_1 <: S_1$ and $\Gamma \vdash\!\!\blacktriangleright S_2 <: T_2$. Finally, SA-ARROW yields $\Gamma \vdash\!\!\blacktriangleright S_1 \to S_2 <: T_1 \to T_2$, as required.

The case where the left-hand derivation ends with an instance of SA-ALL is similar. □

21.3.3 Theorem [Soundness and completeness of algorithmic subtyping]:

1. If $\Gamma \vdash S <: T$ then $\Gamma \vdash\!\!\blacktriangleright S <: T$.
2. If $\Gamma \vdash\!\!\blacktriangleright S <: T$ then $\Gamma \vdash S <: T$. □

Proof: Both directions proceed by induction on derivations. Soundness is routine. Completeness is also straightforward, since we have already done the hard work (for the reflexivity and transitivity rules of the original subtype relation) in Lemmas 21.3.1 and 21.3.2. □

Finally, we should check that the subtyping rules define an algorithm that is *total*—i.e., that always terminates no matter what input it is given. We do this by assigning a weight to each subtyping statement, and checking that the algorithmic rules all have conclusions with greater weight than their premises.

21.3.4 Definition: The **weight** of a type $\mathtt{T}$ in a context Γ, written $\mathit{weight}_\Gamma(\mathtt{T})$, is defined as follows:

$$\begin{array}{lcll}
\mathit{weight}_\Gamma(\mathtt{X}) & = & \mathit{weight}_{\Gamma_1}(\mathtt{U}) + 1 & \text{if } \Gamma = \Gamma_1, \mathtt{X<:U}, \Gamma_2 \\
\mathit{weight}_\Gamma(\mathtt{Top}) & = & 1 & \\
\mathit{weight}_\Gamma(\mathtt{T_1 \to T_2}) & = & \mathit{weight}_\Gamma(\mathtt{T_1}) + \mathit{weight}_\Gamma(\mathtt{T_2}) + 1 & \\
\mathit{weight}_\Gamma(\forall \mathtt{X<:T_1}.\mathtt{T}) & = & \mathit{weight}_{\Gamma,\mathtt{X<:T_1}}(\mathtt{T_2}) + 1 &
\end{array}$$

The **weight** of a subtyping statement "$\Gamma \vdash \mathtt{S} <: \mathtt{T}$" is the maximum weight of $\mathtt{S}$ and $\mathtt{T}$ in Γ. □

21.3.5 Theorem: The weight of the conclusion in an instance of any of the algorithmic subtyping rules is strictly greater than the weight of any of the premises. □

Proof: Straightforward inspection of the rules. □

21.4 Subtyping in $F^f_{<:}$

The only difference in the full system $F^f_{<:}$ is that the quantifier subtyping rule S-ALL is replaced by the more expressive variant:

$$\frac{\Gamma \vdash \mathtt{T_1} <: \mathtt{S_1} \qquad \Gamma, \mathtt{X<:T_1} \vdash \mathtt{S_2} <: \mathtt{T_2}}{\Gamma \vdash \forall \mathtt{X<:S_1}.\mathtt{S_2} <: \forall \mathtt{X<:T_1}.\mathtt{T_2}} \qquad \text{(S-ALL)}$$

The algorithmic subtype relation of $F^f_{<:}$ consists of exactly the same set of rules as the algorithm for $F^k_{<:}$, except that SA-ALL is refined to reflect the new version of S-ALL:

$$\frac{\Gamma \vdash \mathtt{T_1} <: \mathtt{S_1} \qquad \Gamma, \mathtt{X<:T_1} \vdash\!\!\blacktriangleright \mathtt{S_2} <: \mathtt{T_2}}{\Gamma \vdash\!\!\blacktriangleright \forall \mathtt{X<:S_1}.\mathtt{S_2} <: \forall \mathtt{X<:T_1}.\mathtt{T_2}} \qquad \text{(SA-ALL)}$$

As with $F^k_{<:}$, the soundness and completeness of this algorithmic relation with respect to the original subtype relation can be shown easily, once we have established that the algorithmic relation is reflexive and transitive. For reflexivity, the argument is exactly the same as before. For transitivity, on the other hand, the issues are more subtle.