Chapter 14. Critical Sections and Semaphores

Programs that manage shared resources must execute portions of code called critical sections in a mutually exclusive manner. This chapter discusses how critical sections arise and how to protect their execution by means of semaphores. After presenting an overview of the semaphore abstraction, the chapter describes POSIX named and unnamed semaphores. The closing section outlines a license manager project based on semaphores.

Objectives

- Learn about semaphores and their properties
- Experiment with synchronization
- Explore critical section behavior
- Use POSIX named and unnamed semaphores
- Understand semaphore management

14.1 Dealing with Critical Sections

Imagine a computer system in which all users share a single printer and can simultaneously print. How would the output appear? If lines of users' jobs were interspersed, the system would be unusable. Shared devices, such as printers, are called *exclusive resources* because they must be accessed by one process at a time. Processes must execute the code that accesses these shared resources in a *mutually exclusive* manner.

A *critical section* is a code segment that must be executed in a mutually exclusive manner, that is, only one thread of execution can be active in its boundaries. For example, code that modifies a shared variable is considered to be part of a critical section, if other threads of execution might possibly access the shared variable during the modification. The *critical section problem* refers to the problem of executing critical section code in a safe, fair and symmetric manner.

Program 14.1 contains a modification of Program 3.1 on page 67 to generate a process chain. It prints its message one character at a time. The program takes an extra command-line argument giving a delay after each character is output to make it more likely that the process quantum will expire in the output loop. The call to `wait` ensures that the original process does not terminate until all children have completed and prevents the shell prompt from appearing in the middle of the output of one of the children.

Exercise 14.1

Explain why the marked section of code in Program 14.1 is a critical section.

Answer:

After falling out of the forking loop, each process outputs an informative message to standard error one character at a time. Since standard error is shared by all processes in the chain, that part of the code is a critical section and should be executed in a mutually exclusive manner. Unfortunately, the critical section of Program 14.1 is not protected, so output from different processes can interleave in a random manner, different for each run.

Exercise 14.2

Run Program 14.1 with different values of the delay parameter. What happens?

Answer:

When the delay parameter is near 0, each process usually outputs its entire line without losing the CPU. Longer delays make it more likely that a process will lose the CPU before completing the entire message. For large enough values of the delay, each process outputs only one character before losing the CPU. Depending on the speed of the machine, you might need to use values of the delay in excess of 1 million for this last case.

Exercise 14.3

Program 3.1 on page 67 uses a single `fprintf` to standard error to produce the output. Does this have a critical section?

Answer:

Yes. Although the output is in a single C language statement, the compiled code is a sequence of assembly language instructions and the process can lose the CPU anywhere in this sequence. Although this might be less likely to happen in Program 3.1 than in Program 14.1, it is still possible.

Program 14.1 `chaincritical.c`

A program to generate a chain of processes that write to standard error.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include "restart.h"
#define BUFSIZE 1024

int main(int argc, char *argv[]) {
    char buffer[BUFSIZE];
    char *c;
    pid_t childpid = 0;
    int delay;
    volatile int dummy = 0;
    int i, n;

    if (argc != 3){   /* check for valid number of command-line arguments */
        fprintf (stderr, "Usage: %s processes delay\n", argv[0]);
        return 1;
    }
    n = atoi(argv[1]);
    delay = atoi(argv[2]);
    for (i = 1; i < n; i++)
        if (childpid = fork())
            break;
    snprintf(buffer, BUFSIZE,
             "i:%d  process ID:%ld  parent ID:%ld  child ID:%ld\n",
             i, (long)getpid(), (long)getppid(), (long)childpid);

    c = buffer;
   /********************** start of critical section **********************/
    while (*c != '\0') {
        fputc(*c, stderr);
        c++;
        for (i = 0; i < delay; i++)
            dummy++;
    }
   /********************** end of critical section ************************/
    if (r_wait(NULL) == -1)
```

```
        return 1;
    return 0;
}
```

Each process in Program 14.1 executes the statements in sequential order, but the statements (and hence the output) from the different processes can be arbitrarily interleaved. An analogy to this arbitrary interleaving comes from a deck of cards. Cut a deck of cards. Think of each section of the cut as representing one process. The individual cards in each section represent the statements in the order that the corresponding process executes them. Now shuffle the two sections by interleaving. There are many possibilities for a final ordering, depending on the shuffling mechanics. Similarly, there are many possible interleavings of the statements of two processes because the exact timing of processes relative to each other depends on outside factors (e.g., how many other processes are competing for the CPU or how much time each process spent in previous blocked states waiting for I/O). The challenge for programmers is to develop programs that work for all realizable interleavings of program statements.

Code with synchronized critical sections can be organized into distinct parts. The *entry section* contains code to request permission to modify a shared variable or other resource. You can think of the entry section as the gatekeeper—allowing only one thread of execution to pass through at a time. The *critical section* usually contains code to access a shared resource or to execute code that is nonreentrant. The explicit release of access provided in the *exit section* is necessary so that the gatekeeper knows it can allow the next thread of execution to enter the critical section. After releasing access, a thread may have other code to execute, which we separate into the *remainder section* to indicate that it should not influence decisions by the gatekeeper.

A good solution to the critical section problem requires fairness as well as *exclusive access*. Threads of execution that are trying to enter a critical section should not be *postponed indefinitely*. Threads should also make *progress*. If no thread is currently in the critical section, a waiting thread should be allowed to enter.

Critical sections commonly arise when two processes access a shared resource, such as the example of Program 14.1. Be aware that critical sections can arise in other ways. Code in a signal handler executes asynchronously with the rest of the program, so it can be thought of as logically executing in a separate thread of execution. Variables that are modified in the signal handler and used in the rest of the program must be treated as part of a critical section. In Program 8.6 on page 271, the signal handler and the `results` function compete for access to `buf` and `buflen`. The entry section or gatekeeper is the code in `results` to block `SIGUSR1`; the exit section is the code to unblock `SIGUSR1` and to restore the original signal mask.

Program 2.3 on page 39 illustrates a related problem that can arise with recursive calls to nonreentrant functions such as `strtok`. Although this example is not strictly a critical section problem by the definition given above, it has the same characteristics because the single thread of execution changes its execution environment when a function call pushes a new activation record on the stack.

14.2 Semaphores

In 1965, E. W. Dijkstra [30] proposed the semaphore abstraction for high-level management of mutual exclusion and synchronization. A semaphore is an integer variable with two atomic operations, `wait` and `signal`. Other names for `wait` are `down`, `P` and `lock`. Other names for `signal` are `up`, `V`, `unlock` and `post`.

If `S` is greater than zero, `wait` tests and decrements `S` in an atomic operation. If `S` is equal to zero, the `wait` tests `S` and blocks the caller in an atomic operation.

If threads are blocked on the semaphore, then `S` is equal to zero and `signal` unblocks one of these waiting threads. If no threads are blocked on the semaphore, `signal` increments `S`. In POSIX:SEM terminology, the `wait` and `signal` operations are called *semaphore lock* and *semaphore unlock*, respectively. We can think of a semaphore as an integer value and a list of processes waiting for a `signal` operation.

Example 14.4

The following pseudocode shows a blocking implementation of semaphores.

```
void wait(semaphore_t *sp) {
   if (sp->value > 0)
      sp->value--;
   else {
      <Add this process to sp->list>
      <block>
   }
}

void signal(semaphore_t *sp) {
   if (sp->list != NULL)
      <remove a process from sp->list and put in ready state>
   else
      sp->value++;
}
```

The `wait` and `signal` operations must be atomic. An *atomic operation* is an operation that, once started, completes in a logically indivisible way (i.e., without any other related instructions interleaved). In this context, being atomic means that if a process calls `wait`, no other process can change the semaphore until the semaphore is decremented or the calling process is blocked. The signal operation is atomic in a similar way. Semaphore implementations use atomic operations of the underlying operating system to ensure correct execution.

Example 14.5

The following pseudocode protects a critical section if the semaphore variable `S` is initially 1.

```
wait(&S);                                  /* entry section or gatekeeper */
<critical section>
signal(&S);                                                /* exit section */
<remainder section>
```

Processes using semaphores must cooperate to protect a critical section. The code of Example 14.5 works, provided that all processes call `wait(&S)` before entering their critical sections and that they call `signal(&S)` when they leave. If any process fails to call `wait(&S)` because of a mistake or oversight, the processes may not execute the code of the critical section in a mutually exclusive manner. If a process fails to call `signal(&S)` when it finishes its critical section, other cooperative processes are blocked from entering their critical sections.

Exercise 14.6

What happens if `S` is initially 0 in the previous example? What happens if `S` is initially 8? Under what circumstances might initialization to 8 prove useful?

Answer:

If `S` is initially 0, every `wait(&S)` blocks and a deadlock results unless some other process calls `signal` for this semaphore. If `S` is initially 8, at most eight processes execute concurrently in their critical sections. The initialization to 8 might be used when there are eight identical copies of the resource that can be accessed concurrently.

Example 14.7

Suppose process 1 must execute statement `a` before process 2 executes statement `b`. The semaphore `sync` enforces the ordering in the following pseudocode, provided that `sync` is initially 0.

```
Process 1 executes:          Process 2 executes:
  a;                           wait(&sync);
  signal(&sync);               b;
```

Because `sync` is initially 0, process 2 blocks on its `wait` until process 1 calls `signal`.

Exercise 14.8

What happens in the following pseudocode if the semaphores `S` and `Q` are both initially 1? What about other possible initializations?

```
Process 1 executes:          Process 2 executes:
   for( ; ; ) {                 for( ; ; ) {
      wait(&S);                    wait(&Q);
      a;                           b;
      signal(&Q);                  signal(&S);
   }                            }
```

Answer:

Either process might execute its `wait` statement first. The semaphores ensure that a given process is no more than one iteration ahead of the other. If one semaphore is initially 1 and the other 0, the processes proceed in strict alternation. If both semaphores are initially 0, a deadlock occurs.

Exercise 14.9

What happens when `S` is initially 8 and `Q` is initially 0 in Exercise 14.8? Hint: Think of `S` as representing buffer slots and `Q` as representing items in a buffer.

Answer:

Process 1 is always between zero and eight iterations ahead of process 2. If the value of `S` represents empty slots and the value of `Q` represents items in the slots, process 1 acquires slots and produces items, and process 2 acquires items and produces empty slots. This generalization synchronizes access to a buffer with room for no more than eight items.

Exercise 14.10

What happens in the following pseudocode if semaphores `S` and `Q` are both initialized to 1?

```
Process 1 executes:           Process 2 executes:
   for( ; ; ) {                  for( ; ; ) {
      wait(&Q);                     wait(&S);
      wait(&S);                     wait(&Q);
      a;                            b;
      signal(&S);                 signal(&Q);
      signal(&Q);                 signal(&S);
   }                               }
```

Answer:

The result depends on the order in which the processes get the CPU. It should work most of the time, but if process 1 loses the CPU after executing `wait(&Q)` and process 2 gets in, both processes block on their second `wait` call and a deadlock occurs.

A semaphore synchronizes processes by requiring that the value of the semaphore variable be nonnegative. More general forms of synchronization allow synchronization on arbitrary conditions and have mechanisms for combining synchronization conditions. *OR synchronization* refers to waiting until any condition in a specified set is satisfied. The use of `select` or `poll` to monitor multiple file descriptors for input is a form of OR synchronization. *NOT synchronization* refers to waiting until some condition in a set is not true. NOT synchronization can be used to enforce priority ordering [76]. *AND synchronization* refers to waiting until all the conditions in a specified set of conditions are satisfied. AND synchronization can be used for simultaneous control of multiple resources such as that needed for Exercise 14.10. POSIX:XSI semaphore sets described in Chapter 15 are capable of providing AND synchronization.

14.3 POSIX:SEM Unnamed Semaphores

A POSIX:SEM semaphore is a variable of type `sem_t` with associated atomic operations for initializing, incrementing and decrementing its value. The POSIX:SEM Semaphore Extension defines two types of semaphores, named and unnamed. An implementation supports POSIX:SEM semaphores if it defines `_POSIX_SEMAPHORES` in `unistd.h`. The difference between unnamed and named semaphores is analogous to the difference between ordinary pipes and named pipes (FIFOs). This section discusses unnamed semaphores. Named semaphores are discussed in Section 14.5.

Example 14.11

The following code segment declares a semaphore variable called `sem`.

```
#include <semaphore.h>
sem_t sem;
```

The POSIX:SEM Extension does not specify the underlying type of `sem_t`. One possibility is that `sem_t` acts like a file descriptor and is an offset into a local table. The table values point to entries in a system table. A particular implementation may not use the file descriptor table model but instead may store information about the semaphore with the `sem_t` variable. The semaphore functions take a pointer to the semaphore variable as a parameter, so system implementers are free to use either model. You may not make a copy of a `sem_t` variable and use it in semaphore operations.

POSIX:SEM semaphores must be initialized before they are used. The `sem_init` function initializes the unnamed semaphore referenced by `sem` to `value`. The `value` parameter cannot be negative. Our examples use unnamed semaphores with `pshared` equal to 0, meaning that the semaphore can be used only by threads of the process that initializes the semaphore. If `pshared` is nonzero, any process that can access `sem` can use the semaphore. Be aware that simply forking a child after creating the semaphore does not provide access for the child. The child receives a copy of the semaphore, not the actual semaphore.

SYNOPSIS

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned value);
```

POSIX:SEM

If successful, `sem_init` initializes `sem`. Interestingly, POSIX does not specify the return value on success, but the rationale mentions that `sem_init` may be required to return 0 in a future specification. If unsuccessful, `sem_init` returns –1 and sets `errno`. The following table lists the mandatory errors for `sem_init`.

errno	cause
EINVAL	value is greater than SEM_VALUE_MAX
ENOSPC	initialization resource was exhausted, or number of semaphores exceeds SEM_NSEMS_MAX
EPERM	caller does not have the appropriate privileges

Example 14.12

The following code segment initializes an unnamed semaphore to be used by threads of the process.

```
sem_t semA;

if (sem_init(&semA, 0, 1) == -1)
   perror("Failed to initialize semaphore semA");
```

The sem_destroy function destroys a previously initialized unnamed semaphore referenced by the sem parameter.

SYNOPSIS

```
    #include <semaphore.h>

    int sem_destroy(sem_t *sem);
```

POSIX:SEM

If successful, sem_destroy returns 0. If unsuccessful, sem_destroy returns –1 and sets errno. The sem_destroy function sets errno to EINVAL if *sem is not a valid semaphore.

Example 14.13

The following code destroys semA.

```
sem_t semA;

if (sem_destroy(&semA) == -1)
   perror("Failed to destroy semA");
```

Exercise 14.14

What happens if Example 14.13; executes after semA has already been destroyed? What happens if another thread or process is blocked on semA when the sem_destroy function is called?

Answer:

The POSIX standard states that the result of destroying a semaphore that has already been destroyed is undefined. The result of destroying a semaphore on which other threads are blocked is also undefined.

PREVIOUS NEXT

14.4 POSIX:SEM Semaphore Operations

The semaphore operations described in this section apply both to POSIX:SEM unnamed semaphores and to POSIX:SEM named semaphores described in Section 14.5.

The `sem_post` function implements classic semaphore signaling. If no threads are blocked on `sem`, then `sem_post` increments the semaphore value. If at least one thread is blocked on `sem`, then the semaphore value is zero. In this case, `sem_post` causes one of the threads blocked on `sem` to return from its `sem_wait` function, and the semaphore value remains at zero. The `sem_post` function is signal-safe and can be called from a signal handler.

SYNOPSIS

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

POSIX:SEM

If successful, `sem_post` returns 0. If unsuccessful, `sem_post` returns –1 and sets `errno`. The `sem_post` operation sets `errno` to `EINVAL` if `*sem` does not correspond to a valid semaphore.

The `sem_wait` function implements the classic semaphore wait operation. If the semaphore value is 0, the calling thread blocks until it is unblocked by a corresponding call to `sem_post` or until it is interrupted by a signal. The `sem_trywait` function is similar to `sem_wait` except that instead of blocking when attempting to decrement a zero-valued semaphore, it returns –1 and sets `errno` to `EAGAIN`.

SYNOPSIS

```
#include <semaphore.h>

int sem_trywait(sem_t *sem);
int sem_wait(sem_t *sem);
```

POSIX:SEM

If successful, these functions return 0. If unsuccessful, these functions return –1 and set `errno`. These functions set `errno` to `EINVAL` if `*sem` does not correspond to a valid semaphore. The `sem_trywait` sets `errno` to `EAGAIN` if it would block on an ordinary `sem_wait`.

The `sem_wait` and `sem_trywait` functions *may* set `errno` to `EINTR` if they are interrupted by a signal. Any program that catches signals must take care when using semaphore operations, since the standard allows `sem_wait` and `sem_trywait` to return when a signal is caught and the signal handler returns. Program 14.2 restarts the `sem_wait` if it is interrupted by a signal.

Program 14.2 shows how to implement a shared variable that is protected by semaphores. The

`initshared` function initializes the value of the shared variable. It would normally be called only once. The `getshared` function returns the current value of the variable, and the `incshared` function atomically increments the variable. If successful, these functions return 0. If unsuccessful, these functions return –1 and set `errno`. The shared variable (`shared`) is static, so it can be accessed only through the functions of `semshared.c`. Although `shared` is a simple integer in Program 14.2, functions of the same form can be used to implement any type of shared variable or structure.

Program 14.2 `semshared.c`

A shared variable protected by semaphores.

```
#include <errno.h>
#include <semaphore.h>

static int shared = 0;
static sem_t sharedsem;

int initshared(int val) {
    if (sem_init(&sharedsem, 0, 1) == -1)
        return -1;
    shared = val;
    return 0;
}

int getshared(int *sval) {
    while (sem_wait(&sharedsem) == -1)
        if (errno != EINTR)
            return -1;
    *sval = shared;
    return sem_post(&sharedsem);
}

int incshared() {
    while (sem_wait(&sharedsem) == -1)
        if (errno != EINTR)
            return -1;
    shared++;
    return sem_post(&sharedsem);
}
```

Exercise 14.15

Suppose a variable were to be incremented in the `main` program and also in a signal handler. Explain how Program 14.2 could be used to protect this variable.

Answer:

It could not be used without some additional work. If the signal were caught while a call to one of the functions in Program 14.2 had the semaphore locked, a call to one of these in the signal handler would cause a deadlock. The application should block the signals in the `main` program

before calling `getshared` and `incshared`.

Programs 14.3 and 14.4 return to the original critical section problem of Program 14.1. The new version uses threads to illustrate the need to protect the critical section. The function in Program 14.3 is meant to be used as a thread. It outputs a message, one character at a time. To make it more likely to be interrupted in the middle of the message, the thread sleeps for 10 ms after each character is output. Program 14.4 creates a number of `threadout` threads and waits for them to terminate.

Program 14.3 `threadcritical.c`

A thread with an unprotected critical section.

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#define BUFSIZE 1024
#define TEN_MILLION 10000000L

/* ARGSUSED */
void *threadout(void *args) {
    char buffer[BUFSIZE];
    char *c;
    struct timespec sleeptime;

    sleeptime.tv_sec = 0;
    sleeptime.tv_nsec = TEN_MILLION;
    snprintf(buffer, BUFSIZE, "This is a thread from process %ld\n",
             (long)getpid());
    c = buffer;
   /*****************start of critical section ******************/
    while (*c != '\0') {
        fputc(*c, stderr);
        c++;
        nanosleep(&sleeptime, NULL);
    }
   /******************end of critical section ******************/
    return NULL;
}
```

Exercise 14.16

What would happen if Program 14.4 were run with four threads?

Answer:

Most likely each thread would print the first character of its message, and then each would print the second character of its message, etc. All four messages would appear on one line followed by four newline characters.

Exercise 14.17

Why did we use `nanosleep` instead of a busy-waiting loop as in Program 14.1?

Answer:

Some thread-scheduling algorithms allow a busy-waiting thread to exclude other threads of the same process from executing.

Exercise 14.18

Why didn't we have the thread in Program 14.3 print its thread ID?

Answer:

The thread ID is of type `pthread_t`. Although many systems implement this as an integral type that can be cast to an `int` and printed, the standard does not require that `pthread_t` be of integral type. It may be a structure.

Program 14.4 `maincritical.c`

A `main` program that creates a number of threads.

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

void *threadout(void *args);

int main(int argc, char *argv[]) {
    int error;
    int i;
    int n;
    pthread_t *tids;

    if (argc != 2){   /* check for valid number of command-line arguments */
        fprintf (stderr, "Usage: %s numthreads\n", argv[0]);
        return 1;
    }
    n = atoi(argv[1]);
    tids = (pthread_t *)calloc(n, sizeof(pthread_t));
    if (tids == NULL) {
        perror("Failed to allocate memory for thread IDs");
        return 1;
    }
    for (i = 0; i < n; i++)
        if (error = pthread_create(tids+i, NULL, threadout, NULL)) {
            fprintf(stderr, "Failed to create thread:%s\n", strerror(error));
            return 1;
        }
    for (i = 0; i < n; i++)
```

```
        if (error = pthread_join(tids[i], NULL)) {
            fprintf(stderr, "Failed to join thread:%s\n", strerror(error));
            return 1;
        }
    return 0;
}
```

Program 14.5 is a version of Program 14.3 that protects its critical section by using a semaphore passed as its parameter. Although the `main` program does not use signals, this program restarts `sem_wait` if interrupted by a signal to demonstrate how to use semaphores with signals. Program 14.6 shows the corresponding `main` program. The `main` program initializes the semaphore to 1 before any of the threads are created.

Program 14.5 `threadcriticalsem.c`

A thread with a critical section protected by a semaphore passed as its parameter.

```
#include <errno.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>
#define TEN_MILLION 10000000L
#define BUFSIZE 1024

void *threadout(void *args) {
    char buffer[BUFSIZE];
    char *c;
    sem_t *semlockp;
    struct timespec sleeptime;

    semlockp = (sem_t *)args;
    sleeptime.tv_sec = 0;
    sleeptime.tv_nsec = TEN_MILLION;
    snprintf(buffer, BUFSIZE, "This is a thread from process %ld\n",
             (long)getpid());
    c = buffer;
   /****************** entry section *****************************/
    while (sem_wait(semlockp) == -1)        /* Entry section */
        if(errno != EINTR) {
            fprintf(stderr, "Thread failed to lock semaphore\n");
            return NULL;
        }
   /****************** start of critical section *****************/
    while (*c != '\0') {
        fputc(*c, stderr);
        c++;
        nanosleep(&sleeptime, NULL);
    }
   /****************** exit section ******************************/
    if (sem_post(semlockp) == -1)         /* Exit section */
        fprintf(stderr, "Thread failed to unlock semaphore\n");
   /****************** remainder section *************************/
    return NULL;
```

```
}
```

Exercise 14.19

What happens if you replace the following line of Program 14.6

```
semlock = sem_init(*semlock, 0, 1)
```

with the following?

```
semlock = sem_init(*semlock, 0, 0)
```

Answer:

The original `sem_init` sets the initial value of `semlock` to 1, which allows the first process to successfully acquire the semaphore lock when it executes `sem_wait`. The replacement sets the initial value of `semlock` to 0, causing a deadlock. All of the processes block indefinitely on `sem_wait`.

Program 14.6 **`maincriticalsem.c`**

A `main` program that creates a semaphore and passes it to a number of threads.

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void *threadout(void *args);

int main(int argc, char *argv[]) {
    int error;
    int i;
    int n;
    sem_t semlock;
    pthread_t *tids;

    if (argc != 2){   /* check for valid number of command-line arguments */
        fprintf (stderr, "Usage: %s numthreads\n", argv[0]);
        return 1;
    }
    n = atoi(argv[1]);
    tids = (pthread_t *)calloc(n, sizeof(pthread_t));
    if (tids == NULL) {
        perror("Failed to allocate memory for thread IDs");
        return 1;
    }
    if (sem_init(&semlock, 0, 1) == -1) {
        perror("Failed to initialize semaphore");
```

```
        return 1;
    }
    for (i = 0; i < n; i++)
        if (error = pthread_create(tids + i, NULL, threadout, &semlock)) {
            fprintf(stderr, "Failed to create thread:%s\n", strerror(error));
            return 1;
        }
    for (i = 0; i < n; i++)
        if (error = pthread_join(tids[i], NULL)) {
            fprintf(stderr, "Failed to join thread:%s\n", strerror(error));
            return 1;
        }
    return 0;
}
```

Exercise 14.19 illustrates the importance of properly initializing the semaphore value. The `sem_getvalue` function allows a user to examine the value of either a named or unnamed semaphore. This function sets the integer referenced by `sval` to the value of the semaphore without affecting the state of the semaphore. Interpretation of `sval` is a little tricky: It holds the value that the semaphore had at some unspecified time during the call, but not necessarily the value at the time of return. If the semaphore is locked, `sem_getvalue` either sets `sval` to zero or to a negative value indicating the number of threads waiting for the semaphore at some unspecified time during the call.

SYNOPSIS

```
#include <semaphore.h>

int sem_getvalue(sem_t *restrict sem, int *restrict sval);
```

POSIX:SEM

If successful, `sem_getvalue` returns 0. If unsuccessful, `sem_getvalue` returns –1 and sets `errno`. The `sem_getvalue` function sets `errno` to `EINVAL` if `*sem` does not correspond to a valid semaphore.

14.5 POSIX:SEM Named Semaphores

POSIX:SEM named semaphores can synchronize processes that do not share memory. Named semaphores have a name, a user ID, a group ID and permissions just as files do. A semaphore name is a character string that conforms to the construction rules for a pathname. POSIX does not require that the name appear in the filesystem, nor does POSIX specify the consequences of having two processes refer to the same name unless the name begins with the slash character. If the name begins with a slash (`/`), then two processes (or threads) that open the semaphore with that name refer to the same semaphore. Consequently, always use names beginning with a `/` for POSIX:SEM named semaphores. Some operating systems impose other restrictions on semaphore names.

14.5.1 Creating and opening named semaphores

The `sem_open` function establishes the connection between a named semaphore and a `sem_t` value. The `name` parameter is a string that identifies the semaphore by name. This name may or may not correspond to an actual object in the file system. The `oflag` parameter determines whether the semaphore is created or just accessed by the function. If the `O_CREAT` bit of `oflag` is set, the `sem_open` requires two more parameters: a `mode` parameter of type `mode_t` giving the permissions and a `value` parameter of type `unsigned` giving the initial value of the semaphore. If both the `O_CREAT` and `O_EXCL` bits of `oflag` are set, the `sem_open` returns an error if the semaphore already exists. If the semaphore already exists and `O_CREAT` is set but `O_EXCL` is not set, the semaphore ignores `O_CREAT` and the additional parameters. POSIX:SEM does not provide a way to directly set the value of a named semaphore once it already exists.

SYNOPSIS

```
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag, ...);
```

POSIX:SEM

If successful, the `sem_open` function returns the address of the semaphore. If unsuccessful, `sem_open` returns `SEM_FAILED` and sets `errno`. The following table lists the mandatory errors for `sem_open`.

errno	cause
EACCES	permissions incorrect
EEXIST	`O_CREAT` and `O_EXCL` are set and semaphore exists
EINTR	`sem_open` was interrupted by a signal

EINVAL	name can't be opened as a semaphore, or tried to create semaphore with value greater than SEM_VALUE_MAX
EMFILE	too many file descriptors or semaphores in use by process
ENAMETOOLONG	name is longer than PATH_MAX, or it has a component that exceeds NAME_MAX
ENFILE	too many semaphores open on the system
ENOENT	O_CREAT is not set and the semaphore doesn't exist
ENOSPC	not enough space to create the semaphore

Program 14.7 shows a getnamed function that creates a named semaphore if it doesn't already exist. The getnamed function can be called as an initialization function by multiple processes. The function first tries to create a new named semaphore. If the semaphore already exists, the function then tries to open it without the O_CREAT and O_EXCL bits of the oflag parameter set. If successful, getnamed returns 0. If unsuccessful, getnamed returns –1 and sets errno.

Program 14.7 getnamed.c

A function to access a named semaphore, creating it if it doesn't already exist.

```
#include <errno.h>
#include <fcntl.h>
#include <semaphore.h>
#include <sys/stat.h>
#define PERMS (mode_t)(S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
#define FLAGS (O_CREAT | O_EXCL)

int getnamed(char *name, sem_t **sem, int val) {
    while (((*sem = sem_open(name, FLAGS, PERMS, val)) == SEM_FAILED) &&
            (errno == EINTR)) ;
    if (*sem != SEM_FAILED)
        return 0;
    if (errno != EEXIST)
        return -1;
    while (((*sem = sem_open(name, 0)) == SEM_FAILED) && (errno == EINTR)) ;
    if (*sem != SEM_FAILED)
        return 0;
    return -1;
}
```

The first parameter of getnamed is the name of the semaphore and the last parameter is the value to use for initialization if the semaphore does not already exist. The second parameter is a pointer to a pointer to a semaphore. This double indirection is necessary because getnamed needs to change a pointer. Note that if the semaphore already exists, getnamed does not initialize the semaphore.

Program 14.8 shows a modification of Program 14.1 that uses named semaphores to protect

the critical section. Program 14.8 takes three command-line arguments: the number of processes, the delay and the name of the semaphore to use. Each process calls the `getnamed` function of Program 14.7 to gain access to the semaphore. At most, one of these will create the semaphore. The others will gain access to it.

Exercise 14.20

What happens if two copies of `chainnamed`, using the same named semaphore run simultaneously on the same machine?

Answer:

With the named semaphores, each line will be printed without interleaving.

Exercise 14.21

What happens if you enter Ctrl-C while `chainnamed` is running and then try to run it again with the same named semaphore?

Answer:

Most likely the signal generated by Ctrl-C will be delivered while the semaphore has value 0. The next time the program is run, all processes will block and no output will result.

14.5.2 Closing and unlinking named semaphores

Like named pipes or FIFOs (Section 6.3), POSIX:SEM named semaphores have permanence beyond the execution of a single program. Individual programs can close named semaphores with the `sem_close` function, but doing so does not cause the semaphore to be removed from the system. The `sem_close` takes a single parameter, `sem`, specifying the semaphore to be closed.

SYNOPSIS

```
#include <semaphore.h>

int sem_close(sem_t *sem);
```

POSIX:SEM

If successful, `sem_close` returns 0. If unsuccessful, `sem_close` returns –1 and sets `errno`. The `sem_close` function sets `errno` to `EINVAL` if `*sem` is not a valid semaphore.

Program 14.8 `chainnamed.c`

A process chain with a critical section protected by a POSIX:SEM named semaphore.

```
#include <errno.h>
```

```
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include "restart.h"
#define BUFSIZE 1024
int getnamed(char *name, sem_t **sem, int val);

int main  (int argc, char *argv[]) {
    char buffer[BUFSIZE];
    char *c;
    pid_t childpid = 0;
    int delay;
    volatile int dummy = 0;
    int i, n;
    sem_t *semlockp;

    if (argc != 4){       /* check for valid number of command-line arguments */
        fprintf (stderr, "Usage: %s processes delay semaphorename\n", argv[0]);
        return 1;
    }
    n = atoi(argv[1]);
    delay = atoi(argv[2]);
    for (i = 1; i < n; i++)
        if (childpid = fork())
            break;
    snprintf(buffer, BUFSIZE,
             "i:%d  process ID:%ld  parent ID:%ld  child ID:%ld\n",
             i, (long)getpid(), (long)getppid(), (long)childpid);
    c = buffer;
    if (getnamed(argv[3], &semlockp, 1) == -1) {
        perror("Failed to create named semaphore");
        return 1;
    }
    while (sem_wait(semlockp) == -1)                          /* entry section */
        if (errno != EINTR) {
            perror("Failed to lock semlock");
            return 1;
        }
    while (*c != '\0') {                                    /* critical section */
        fputc(*c, stderr);
        c++;
        for (i = 0; i < delay; i++)
            dummy++;
    }
    if (sem_post(semlockp) == -1) {                           /* exit section */
        perror("Failed to unlock semlock");
        return 1;
    }
    if (r_wait(NULL) == -1)                                 /* remainder section */
        return 1;
    return 0;
}
```

The `sem_unlink` function, which is analogous to the `unlink` function for files or FIFOs, performs

the removal of the named semaphore from the system after all processes have closed the named semaphore. A close operation occurs when the process explicitly calls `sem_close`, `_exit`, `exit`, `exec` or executes a return from `main`. The `sem_unlink` function has a single parameter, a pointer to the semaphore that is to be unlinked.

SYNOPSIS

```
#include <semaphore.h>

int sem_unlink(const char *name);
```

POSIX:SEM

If successful, `sem_unlink` returns 0. If unsuccessful, `sem_unlink` returns –1 and sets `errno`. The following table lists the mandatory errors for `sem_unlink`.

errno	cause
`EACCES`	permissions incorrect
`ENAMETOOLONG`	`name` is longer than `PATH_MAX`, or it has a component that exceeds `NAME_MAX`
`ENOENT`	the semaphore doesn't exist

Calls to `sem_open` with the same name refer to a new semaphore after a `sem_unlink`, even if other processes still have the old semaphore open. The `sem_unlink` function always returns immediately, even if other processes have the semaphore open.

Exercise 14.22

What happens if you call `sem_close` for an unnamed semaphore that was initialized by `sem_init` rather than `sem_open`?

Answer:

The POSIX standard states that the result of doing this is not defined.

Program 14.9 shows a function that closes and unlinks a named semaphore. The `destroynamed` calls the `sem_unlink` function, even if the `sem_close` function fails. If successful, `destroynamed` returns 0. If unsuccessful, `destroynamed` returns –1 and sets `errno`.

Remember that POSIX:SEM named semaphores are persistent. If you create one of these semaphores, it stays in the system and retains its value until destroyed, even after the process that created it and all processes that have access to it have terminated. POSIX:SEM does not provide a method for determining which named semaphores exist. They may or may not show up when you display the contents of a directory. They may or may not be destroyed when the system reboots.

Program 14.9 **destroynamed.c**

A function that closes and unlinks a named semaphore.

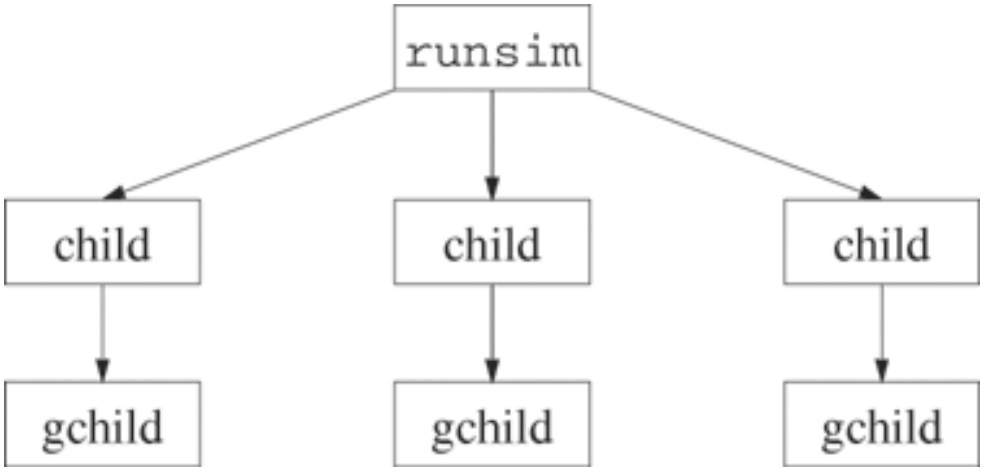
```
#include <errno.h>
#include <semaphore.h>

int destroynamed(char *name, sem_t *sem) {
    int error = 0;

    if (sem_close(sem) == -1)
        error = errno;
    if ((sem_unlink(name) != -1) && !error)
        return 0;
    if (error)        /* set errno to first error that occurred */
        errno = error;
    return -1;
}
```

14.6 Exercise: License Manager

The exercises in this section are along the lines of the `runsim` program developed in the exercises of Section 3.9. In those exercises, `runsim` reads a command from standard input and forks a child that calls `execvp` to execute the command. That `runsim` program takes a single command-line argument specifying the number of child processes allowed to execute simultaneously. It also keeps a count of the children and uses `wait` to block when it reaches the limit.

In these exercises, `runsim` again reads a command from standard input and forks a child. The child in turn forks a grandchild that calls `execvp`. The child waits for the grandchild to complete and then exits. Figure 14.1 shows the structure of `runsim` when three such pairs are executing. This program uses semaphores to control the number of simultaneous executions.

Figure 14.1. The structure of `runsim` when the grandchildren, not the children, call `execvp`.



14.6.1 License object

Implement a `license` object based on a named semaphore generated from the pathname `/tmp.license.uid`, where `uid` is the process user ID. The `license` should have the following public functions.

```
int getlicense(void);
```

blocks until a license is available.

```
int returnlicense(void);
```

increments the number of available licenses.

```
int initlicense(void);
```

performs any needed initialization of the `license` object.

```
int addtolicense(int n);
```

adds a certain number of licenses to the number available.

```
int removelicenses(int n);
```

decrements the number of licenses by the specified number.

14.6.2 The `runsim` main program

Write a `runsim` program that runs up to `n` processes at a time. Start the `runsim` program by typing the following command.

```
runsim n
```

Implement `runsim` as follows.

1. Check for the correct number of command-line arguments and output a usage message if incorrect.
2. Perform the following in a loop until end-of-file on standard input.
    a. Read a command from standard input of up to `MAX_CANON` characters.
    b. Request a license from the `license` object.
    c. Fork a child that calls `docommand` and then exits. Pass the input string to `docommand`.
    d. Check to see if any of the children have finished (`waitpid` with the `WNOHANG` option).

The `docommand` function has the following prototype.

```
void docommand(char *cline);
```

Implement `docommand` as follows.

1. Fork a child (a grandchild of the original). This grandchild calls `makeargv` on `cline` and calls `execvp` on the resulting argument array.

2. Wait for this child and then return the license to the `license` object.

3. Exit.

Test the program as in Section 3.9. Improve the error messages to make them more readable. Write a test program that takes two command-line arguments: the sleep time and the repeat factor. The test program simply repeats a loop for the specified number of times. In the loop, the test program sleeps and then outputs a message with its process ID to standard error. After completing the specified number of iterations, the program exits. Use `runsim` to run multiple copies of the test program.

Try executing several copies of `runsim` concurrently. Since they all use the same semaphore, the number of grandchildren processes should still be bounded by `n`.

14.6.3 Extensions to the license manager

Modify the license object so that it supports multiple types of licenses, each type identified by a numerical key. Test the program under conditions similar to those described in the previous section.

14.7 Additional Reading

Most books on operating systems [107, 122] discuss the classical semaphore abstraction. The book *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers* by Schimmel [103] presents an advanced look at how these issues apply to design of multiprocessor kernels.