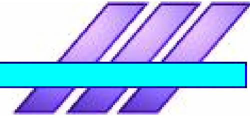
Chapter 9
Synchronizing Primitives & IPC



Design and Implementation of SysPak OS

Abstract

This chapter explains two things: Synchronizing Primitives which are used for imposing locking and execution order and ports which are employed for IPC (Inter Process Communication)

9.1 Semaphore: Overview

A semaphore is a token that's used to synchronize multiple threads, usually for one of these reasons:

- **Locking** The most common use of a semaphore is to create a mutually exclusive lock: It ensures that only one thread can execute a "protected" section of code at a time.
- **Execution Order** Semaphores can also be used to impose an order in which a series of dependent operations are performed by two or more threads..

The semaphore concept is simple: To enter into a semaphore-protected "critical section", a thread must first "acquire" the semaphore, through the **sem_acquire()** function. When it passes out of the critical section, the thread "releases" the semaphore through **sem_release()**. The advantage of the semaphore system is that if a thread can't acquire a semaphore (because the semaphore is yet to be released by the previous acquirer), the thread blocks in the **sem_acquire()** call. While it's blocked, the thread doesn't waste any cycles.

The full story about semaphores is a wee bit more complicated than this quick description, but if all you want to do is create a mutually exclusive lock or impose an execution order, you'll probably learn all you need to know by visiting the examples in the sections referred to above.

9.2 The Full Story

A semaphore acts as a key that a thread must acquire in order to continue execution. Any thread that can identify a particular semaphore can attempt to acquire it by passing its **sem_id** identifier--a system-wide number that's assigned when the semaphore is created--to the **sem_acquire()** function. The function blocks until the semaphore is actually acquired.

- An alternate function, **sem_acquire_etc()** lets you specify the amount of time you're willing to wait for the semaphore to be acquired. Unless otherwise noted, characteristics ascribed to **sem_acquire()** apply to **sem_acquire_etc()** as well.)

When a thread acquires a semaphore, that semaphore (typically) becomes unavailable for acquisition by other threads (less typically, more than one thread is allowed to acquire the semaphore at a time. The semaphore remains unavailable until it's passed in a call to the **sem_release()** function.

The code that a semaphore "protects" lies between the calls to **sem_acquire()** and **sem_release()**. The disposition of these functions in your code usually follows this pattern:

```
if (sem_acquire(my_semaphore) == NO_ERROR) {
   /* Protected code goes here. */
   sem_release(my_semaphore);
}
```

Keep in mind that...

- The calls to the acquire and release functions needn't be so locally balanced (although this is by far the most common use). A semaphore can be acquired within one function and released in another. Acquisition and release of the same semaphore can even be performed by two different threads.
- Checking the value returned by **sem_acquire()** is *extremely* important. If an acquire-blocked thread is unblocked by a signal, the thread shouldn't procede to the

critical section.

9.2.1 The Thread Queue

Every semaphore has its own *thread queue*: This is a list that identifies the threads that are waiting to acquire the semaphore. A thread that attempts to acquire an unavailable semaphore is placed at the tail of the semaphore's thread queue where it sits blocked in the **sem_acquire()** call. Each call to **sem_release()** umblocks the thread at the head of that semaphore's queue, thus allowing the thread to return from its call to **sem_acquire()**.
Semaphores don't discriminate between acquisitive threads--they don't prioritize or otherwise reorder the threads in their queues--the oldest waiting thread is always the next to acquire the semaphore.

9.2.2 The Thread Count

To assess availability, a semaphore looks at its *thread count*. This is a counting variable that's initialized when the semaphore is created. Ostensibly, a thread count's initial value (which is passed as the first argument to **sem_create()**) is the number of threads that can acquire the semaphore at a time. (As we'll see later, this isn't the entire story, but it's good enough for now.) For example, a semaphore that's used as a mutually exclusive lock takes an initial thread count of 1--in other words, only one thread can acquire the semaphore at a time.

- An initial thread count of 1 is by far the most common use; a thread count of 0 is also useful. Other counts are much less common.

Calls to **sem_acquire()** and **sem_release()** alter the semaphore's thread count: **sem_acquire()** decrements the count, and **sem_release()** increments it. When you call **sem_acquire()**, the function looks at the thread count (before decrementing it) to determine if the semaphore is available:

- If the count is greater than zero, the semaphore is available for acquisition, so the function returns immediately.
- If the count is zero or less, the semaphore is unavailable, and the thread is placed in the semaphore's thread queue.

The initial thread count isn't an inviolable limit on the number of threads that can acquire a given semaphore--it's simply the initial value for the sempahore's thread count variable. For example, if you create a semaphore with an initial thread count of 1 and then immediately call **sem_release()** five times, the semaphore's thread count will increase to 6. Furthermore, although you can't initialize the thread count to less-than-zero, an initial value of zero itself is common--it's an integral part of using semaphores to impose an execution order (as demonstrated later).
Summarizing the description above, there are three significant thread count value ranges:

- A positive thread count (*n*) means that there are no threads in the semaphore's queue, and the next *n* **sem_acquire()** calls will return without blocking.
- If the count is 0, there are no queued threads, but the next **sem_acquire()** call will block.
- A negative count (-*n*) means there are *n* threads in the semaphore's thread queue, and the next call to **sem_acquire()** will block.

Although it's possible to retrieve the value of a semaphore's thread count (by looking at a field in the semaphore's **sem_info** structure), you should only do so for amusement--while you're debugging, for example.

9.2.3 Deleting a Semaphore

Every semaphore is owned by a team (the team of the thread that called **sem_create()**). When the last thread in a team dies, it takes the team's semaphores with it.

Prior to the death of a team, you can explicitly delete a semaphore through the **sem_delete()** call. Note, however, that **sem_delete()** must be called from a thread that's a member of the team that owns the semaphore--you can't delete another team's semaphores.

You're allowed to delete a semaphore even if it still has threads in its queue. However, you usually want to avoid this, so deleting a semaphore may require some thought: When you delete a semaphore (or when it dies naturally), all its queued threads are immediately allowed to continue--they all return from **sem_acquire()** at once. You can distinguish between a "normal" acquisition and a "semaphore deleted" acquisition by the value that's returned by **sem_acquire()** (the specific return values are listed in the function descriptions, below).

9.2.4 Broadcasting Semaphores

The **sem_id** number that identifies a semaphore is a system-wide token--the **sem_id** values that you create in your application will identify your semaphores in all other applications as well. It's possible, therefore, to broadcast the **sem_id** numbers of the semaphores that you create and so allow other applications to acquire and release them--but it's not a very good idea.

- A semaphore is best controlled if it's created, acquired, released, and deleted within the same team.

If you want to provide a protected service or resource to other applications, you should accept messages from other applications and then spawn threads that acquire and release the appropriate semaphores.

9.3 Ports: Overview

A port is a system-wide message repository into which a thread can copy a buffer of data, and from which some other thread can then retrieve the buffer. This repository is implemented as a first-in/first-out message queue: A port stores its messages in the order in which they're received, and it relinquishes them in the order in which they're stored. Each port has its own message queue.

9.3.1 Creating a Port

A port is represented by a unique, system-wide **port_id** number (a positive 32-bit integer). The **port_create** function creates a new port and assigns it a **port_id** number. Although ports are accessible to all threads, the **port_id** numbers aren't disseminated by the operating system; if you create a port and want some other thread to be able to write to or read from it, you have to broadcast the **port_id** number to that thread. Typically, ports are used within a single team. The easiest way to broadcast a **port_id** number to the threads in a team is to declare it as a global variable.

A port is owned by the team in which it was created. When a team dies (when all its threads are killed, by whatever hand), the ports that belong to the team are deleted. A team can bestow ownership of its ports to some other team through the **port_set_owner** function.

If you want explicitly get rid of a port, you can call port delete.You can delete any port, not just those that are owned by the team of the calling thread.

9.3.2 The Message Queue: Reading and Writing Port Messages

The length of a port's message queue--the number of messages that it can hold at a time--is set when the port is created. The **MAX_PORT_COUNT** constant provides a reasonable queue length.

The functions **port_write()** and **port_read** manipulate a port's message queue: **port_write** places a message at the tail of the port's message queue; **port_read()** removes the message at the head of the queue and returns it the caller.**port_write()** blocks if the queue is full; it returns when room is made in the queue by an invocation of **port_read()**. Similarly, if the queue is empty, **port_read()** blocks until **port_write()** is called. When a thread is waiting in

a **port_write()** or **port_read()** call, its state is **THREAD_SEM_WAIT** (it's waiting to acquire a system-defined, port-specific semaphore).

You can provide a timeout for your port-writing and port-reading operations by using the "full-blown" functions **port_write_etc()** and **port_read_etc()**. By supplying a timeout, you can ensure that your port operations won't block forever.

Although each port has its own message queue, all ports share a global "queue slot" pool--there are only so many message queue slots that can be used by all ports taken cumulatively. If too many port queues are allowed to fill up, the slot pool will drain, which will cause **port_write()** calls on less-than-full ports to block. To avoid this situation, you should make sure that your **port_write()** and **port_read()** calls are reasonably balanced.

The **port_write()** and **port_read()** functions are the only way to traverse a port's message queue. There's no notion of "peeking" at the queue's unread messages, or of erasing messages that are in the queue.

9.3.3 Port Messages

A port message--the data that's sent through a port--consists of a "message code" and a "message buffer." Either of these elements can be used however you like, but they're intended to fit these purposes:

- The message code (a single four-byte value) should be a mask, flag, or other predictable value that gives a general representation of the flavor or import of the message. For this to work, the sender and receiver of the message must agree on the meanings of the values that the code can take.
- The data in the message buffer can elaborate upon the code, identify the sender of the message, or otherwise supply additional information. The length of the buffer isn't restricted. To get the length of the message buffer that's at the head of a port's queue, you call the **port_buffer_size()** function.

The message that you pass to **port_write()** is copied into the port. After **port_write()** returns, you may free the message data without affecting the copy that the port holds.

When you read a port, you have to supply a buffer into which the port mechanism can copy the message. If the buffer that you supply isn't large enough to accommodate the message, the unread portion will be lost--the next call to **port_read()** won't finish reading the message.

You typically allocate the buffer that you pass to **port_read()** by first calling **port_buffer_size()**, as shown below:

```
char *buf = NULL;
ssize_t size;
int32 code;

/* We'll assume that my_port is valid.
 * port_buffer_size() will block until a message shows up.
 */
if ((size = port_buffer_size(my_port)) < B_NO_ERROR)
   /* Handle the error */

if (size > 0)
   buf = (char *)malloc(size);

if (buf) {
   /* Now we can read the buffer. */
   if (port_read(my_port, &code, (void *)buf, size) < B_OK)
   /* Handle the error */
```

Obviously, there's a race condition (in the example) between **port_buffer_size()** and the subsequent **port_read()** call--some other thread could read the port in the interim. If you're

going to use **port_buffer_size()** as shown in the example, you shouldn't have more than one thread reading the port at a time.

As stated in the example, **port_buffer_size()** blocks until a message shows up. If you don't want to (potentially) block forever, you should use the **port_buffer_size_etc()** version of the function. As with the other `...etc()` functions, **port_buffer_size_etc()** provides a timeout option.