# CS24 Assignment 7

The files for this assignment are provided in the archive **`cs24hw7.tgz`** on the course website. The archive contains a top-level directory **`cs24hw7`**, in which all other files are contained.

For the questions you need to answer, and the programs you need to write, put them into this **`cs24hw7`** directory. Then, when you have finished the assignment, you can re-archive this directory into a tarball and submit the resulting file:

> *(From the directory containing* ***`cs24hw7`****; replace* ***`username`*** *with your username.)*
> **`tar -czvf cs24hw7-`*`username`*`.tgz cs24hw7`**

Then, submit the resulting file through the course website.

**Note:** This assignment is a continuation of Assignment 6. This means that it contains answers to some of the questions in Assignment 6. **Do not start working on Assignment 7 until you have completed and turned in Assignment 6.**

# Assignment Overview

In Assignment 6 you completed a simple user-space threading library, which relied on *cooperative multitasking*: each thread was responsible for yielding the CPU so that other threads could also run.

This week you will continue working on the simple threading library, making it significantly more powerful and generalized. This week we will implement *preemptive multitasking* by adding a timer interrupt to schedule the threads asynchronously. This will necessitate the introduction of process synchronization primitives to ensure that concurrent threads will still execute properly.

The write-up for the assignment is rather long and detailed, and the specific problems to complete are interspersed in the documentation, so here is a quick summary of what you must do on this assignment:

1. Explain why cooperative thread execution is not fair.
2. Implement a mutual-exclusion lock (a "mutex," a very important concurrency primitive) by completing the **`__sthread_lock()`** and **`__sthread_unlock()`** functions in assembly.
3. Modify all entry points in the scheduler to use the **`__sthread_lock()`** function, and exit points to use the **`__sthread_unlock()`** function.
4. Implement semaphores (another common concurrency primitive) in the **`semaphore.c`** file.
5. Using semaphores, modify the bounded buffer implementation so that it is thread-safe.

**Important Note:**

For parts 3-5, you must write code that is thread-safe by using synchronization primitives – locks, semaphores, etc. – to protect shared resources (queues, other shared values) from concurrent access by multiple threads. **To obtain full credit on these parts, you must clearly explain (in your code's comments) why you need each synchronization primitive you utilize.**

# The Producer-Consumer Problem

The *producer-consumer problem* is defined as a system where there are one or more *producers* that generate values to be used by one or more *consumers*. The producers and consumers are typically independent processes, which produce (or consume) data at varying rates over time. Clearly the producers must somehow pass data to the consumers; this is implemented by using a shared buffer or queue. The producers store information onto the end of the queue, and the consumers fetch data from the front of the queue.

One potential problem is that producers may generate data faster than consumers can process. If the size of the queue were unbounded, it could easily overflow the available memory; therefore, the queue's capacity is usually capped at some maximum size. This also introduces new challenges, because producers must stop producing data while the queue is full, and consumers must stop consuming data if the queue becomes empty. It is very important to properly coordinate the interactions these processes have with the shared queue.

There are many examples of this problem in a typical operating system. One of the simplest is the keyboard process, which places values from the keyboard into a buffer where they can be used by applications. Another extremely common example is piping the output of one command into the input of another command; this is in fact something you will see in the assignment.

# Generating Fibonacci Numbers

This week we will work with the program **`fibtest.c`**, which implements the producer-consumer problem using a bounded buffer, two producers, and a single consumer. In this program, the producers produce Fibonacci numbers (the slow, recursive way) and place their output in the buffer. The consumer reads values from the buffer, validates them, and prints out the result.

## Bounded Buffer Data Structure

A *bounded buffer* is a queue with a fixed storage size. The buffer supports two operations:

- **`add(buf, elem)`**: Add the **`elem`** to the buffer **`buf`**. If the buffer is full, block the calling thread until the buffer has space.
- **`take(buf, elem)`**: Get a value from the buffer **`buf`**, copying the result into the **`elem`** structure. If the buffer is empty, block the calling thread until the buffer is nonempty.

C code to implement the bounded buffer is provided in the file **`bounded_buffer.c`**. Here is the data type for buffers:

```
typedef struct _bounded_buffer {
    /* The maximum number of elements in the buffer */
    int length;

    /* The index of the first element in the buffer */
    int first;

    /* The number of elements currently stored in the buffer */
    int count;

    /* The values in the buffer */
    BufferElem *buffer;
} BoundedBuffer;
```

The buffer is implemented as a circular array of values stored in the **`buffer`** array. The maximum number of elements that can be stored is in the **`length`** field. The index of the first non-empty element is in the **`first`** field, and the total number of non-empty entries is in the **`count`** field. The buffer is circular: the index of the first entry is **`first`**, the index of the second entry (if it exists) is **`((first + 1) % length)`**, and so forth.

The code to add an element to the queue is in the **`bounded_buffer_add()`** function, defined as follows:

```
void bounded_buffer_add(BoundedBuffer *bufp, const BufferElem *elem) {
    /* Wait until the buffer has space */
    while (bufp->count == bufp->length)
        sthread_yield();

    /* Now the buffer has space */
    bufp->buffer[(bufp->first + bufp->count) % bufp->length] = *elem;
    bufp->count++;
}
```

This code has two parts: code for process synchronization, and code to implement the enqueue operation. As stated before, a producer cannot add a value to the queue if it's already full, so the first part of the function "waits" as long as the queue is full. This is actually done by yielding execution to another thread using **`sthread_yield()`**, in the hopes that another thread will remove an element from the queue so that this thread can continue.

Once the buffer is no longer full, the remainder of the function simply stores the value and increments the count field.

The code for **`bounded_buffer_take()`** performs a similar sequence of operations, although the first part of the function waits for an element to become available, rather than waiting for space to become available.

## Fibonacci Producers and Consumers

In our simple test program, we have two producers generating Fibonacci numbers and one consumer printing out the values. We use our bounded buffer to pass the values from the producers to the consumer. The elements of the bounded buffer are **`BufferElem`** structs, defined in **`bounded_buffer.h`**. Each element contains the **`id`** of the producer, the **`arg`** to the **`fib()`** function, and the computed result stored in the **`val`** field.

The core loop from the producer is defined as follows.

```
elem.id = (int) arg;
while (1) {
    /* Place the next computed Fibonacci result into the buffer */
    elem.arg = i;
    elem.val = fib(i);
    bounded_buffer_add(queue, &elem);
    i = (i + 1) % FIB_MODULUS;
}
```

The consumer validates the result by recomputing the Fibonacci result, and printing out a line summarizing the result:

```
while (1) {
    bounded_buffer_take(queue, &elem);
    i = fib(elem.arg);
    printf("Result from producer %d: Fib %2d = %6d; should be %6d: %s\n",
           elem.id, elem.arg, elem.val, i,
           i == elem.val ? "matched" : "NO MATCH");
}
```

## Fairness

The implementations of the Fibonacci producers and consumer are all in **`fibtest.c`**, which you can build with the provided **`Makefile`**. The initialization code is defined in the **`main()`** function as follows:

```
sthread_create(producer, (void *) 0);
sthread_create(producer, (void *) 1);
sthread_create(consumer, (void *) 0);
```

Go ahead and try running the **`fibtest`** program. If everything is correct, the program should print out the number sequences generated by *both* Fibonacci producers. Unfortunately, the actual result is not what we hope to see; it is not fair.

**Problem 1**: In a file **`problem1.txt`**, describe the specific problem you see in the program's output, and explain why cooperative thread execution is not fair in this program. *(10 points)*

## Adding A Timer Interrupt

To solve this fairness problem, we propose to use a timer that interrupts the program's execution at periodic intervals, selecting another unblocked thread for execution. This code is defined in the file **`timer.c`**, which installs a timer signal handler that will force periodic calls to the **`__sthread_schedule()`** function. (You don't have to understand this code, but it's very cool, so if you are curious then see Appendix 2 at the end of this assignment.) **Timer events are asynchronous; the timer can go off at any point during program execution.** This means that any statement or function call can be interrupted by the timer signal handler.

The timer code is already set up; all you have to do is to turn it on. To enable it, update the **`main()`** function in **`fibtest.c`** to pass 1 as the argument to **`sthread_start()`**.

Let's try it:

```
...edit the fibtest.c file...
% make
% ./fibtest
...
```

Looking at the output, you will see that this definitely works better than the last version. This time the execution appears to be fair; both producers are generating results.

### Vetting the Program

As the next step, let's look for errors in the computations. Since the consumer verifies each producer's results, we should be able to look for lines containing "NO MATCH" to see any issues.

But before you do this, open up **fibtest.c** again, and change **FIB_MODULUS** to 20 instead of 30. This change will make any correctness issues more prominent.

```
...edit the fibtest.c file again...
% make
% ./fibtest | grep "NO MATCH"
Result from producer -1: Fib -1 = -1; should be 1; NO MATCH
Result from producer -1: Fib -1 = -1; should be 1; NO MATCH
Segmentation fault (core dumped)
```

Results may differ from run to run. Also, the exact nature of the crash may differ from run to run.

It's obvious that we have at least two problems: first, we are getting bad values from the buffer, and second, our program actually crashes at times. *Both problems are mutual exclusion problems.*

## Mutual Exclusion in the sthread Scheduler

The first thing we need to fix is the fact that our scheduler is not reentrant: we have to ensure that at most one thread is executing in the scheduler at any time.

To ensure mutual exclusion in the scheduler, we can require that threads establish a lock using the **__sthread_lock()** function before entering the scheduler. We will use an integer variable to represent the lock state: 0 will mean that the lock is available (i.e. "unlocked"), and 1 will mean that the lock is already held by another thread. The **__sthread_lock()** function will attempt to acquire the lock, and will return 1 if the lock was successfully acquired by this thread, or 0 if the lock is already held by another thread.

A very simplistic, literal translation of this description would be:

```
int scheduler_lock = 0;

int __sthread_lock() {
    if (scheduler_lock == 0) {    /* Is the lock available?      */
        scheduler_lock = 1;       /* Yes!  Lock it and return 1. */
        return 1;
    }
    else {                        /* Lock wasn't available.      */
        return 0;                 /* Return 0.                   */
    }
}

void __sthread_unlock() {
    scheduler_lock = 0;
}
```

Now, if you examine this code carefully, you will notice two interesting things:

1) At the end of this function, **scheduler_lock** will always be 1, regardless of whether the caller acquired the lock or not.

2) The return-value of this function is simply the inverse of the old value of **scheduler_lock**. If **scheduler_lock** was 0 when we entered the function, the result is 1; if it was 1 then the result is 0.

This means that we can rewrite **`__sthread_lock()`** without the **`if`**-statement, like this:

```
int __sthread_lock() {
    int old_value = scheduler_lock;  /* These two operations must */
    scheduler_lock = 1;              /* be performed atomically!  */

    return !old_value;
}
```

Unfortunately, we have one problem: the first two operations in **`__sthread_lock()`** must be performed *atomically* (i.e. as an indivisible unit), so that no other thread can interrupt us while in the middle of this operation. So, this code doesn't quite work as it is written, but the processor can help us with that problem.

**Problem 2:** Implement the **`__sthread_lock()`** and **`__sthread_unlock()`** functions in IA32 assembly language, in the file **`glue.s`**. *(20 points)*

You might consider these IA32 instructions for implementing **`__sthread_lock()`**: **`xchg`**, **`bts`**, the **`lock`** prefix, **`set`*`cc`***. Note that the IA32 architecture only provides a way to lock a single instruction, but this is perfectly sufficient for our needs; we only need to perform the first two statements in **`__sthread_lock()`** atomically.

There are multiple ways to implement the locking operation on IA32, so feel free to choose a mechanism that makes sense to you. Some ways are more like the first version of **`__sthread_lock()`** above, and others are more like the second version. Take your pick!

The IA32 Manual Vol. 3A, section 7.1 has much more detailed information about atomic memory accesses, including more information about these instructions. Of course, you are encouraged to look at the instruction references in Vol. 2A and 2B as well (probably first).

Note that the IA32 manual explicitly states that writing to a doubleword (4 bytes) aligned on a 32-bit boundary will always be atomic, even in multicore systems.[1] This means you can implement the unlock operation with a simple **`movl`** instruction, without any additional precautions, and without using the **`lock`** prefix.

**Problem 3**: Modify all entry points in the scheduler to use the **`__sthread_lock()`** function, and exit points to use the **`__sthread_unlock()`** function. Note that **`timer.c`** already calls **`__sthread_lock()`**; you shouldn't have to change any code in **`timer.c`** at all. *(20 points)*

**Note:** As stated initially, you must explain why you need each lock and unlock call in your code. Your explanation can be short, but it must be complete.

**Note:** You do not need to consider the return-value from **`__sthread_lock()`** in any of the calls you add to **`sthread.c`** code. (If the reason why is unclear to you, ask Donnie or a TA for an explanation.)

---

[1] Intel 64/IA32 Manual, Volume 3A, Section 7.1.1, Page 285.
[2] Dijkstra named these operations after Dutch words describing their purposes, hence the odd [to us] letters. *P*(*s*) stands for "*prolaag*," short for "*probeer te verlagen*" which means "try to reduce." *V*(*s*) stands for "*verhogen*," meaning "increase." (CS:APP2e says that *P*(*s*) means "*proberen*," but Dijkstra himself stated that

## Mutual Exclusion in the Bounded Buffer

The second problem in our program is that the bounded-buffer code is not thread-safe. The **`add()`** and **`take()`** functions are *critical sections*. The code within a critical section should only be executed by one thread at a time; in other words, the critical section must be executed atomically.

To address this problem, we will use another synchronization mechanism called a *semaphore*.

### Semaphores

Semaphores were originally proposed by Edsger Dijkstra. Abstractly, a semaphore *s* is simply a nonnegative integer variable initialized to some value – perhaps 0, 1, or some other nonnegative value – and threads can perform two operations on the semaphore, *P*(*s*) and *V*(*s*).[2] The value of the integer itself is not directly accessible or manipulable by threads; they can only use *P*(*s*) and *V*(*s*).

The two operations work as follows:

- *P*(*s*) performs a "test and decrement" operation. If *s* is nonzero then *P*(*s*) decrements *s* and returns immediately. If *s* is zero then we cannot decrement it, so *P*(*s*) must wait until *s* becomes nonzero. Once another thread increments the semaphore, this thread can resume executing, and can decrement *s* and return. Because this operation can block the caller, it is also frequently called "wait."

- *V*(*s*) performs an "increment" operation on the semaphore. If any threads are waiting for the semaphore then one of these threads is awakened and goes on to acquire the semaphore. Because semaphores are frequently used to communicate synchronization status between processes or threads, this operation is also frequently called "signal" or "post."

**It is very important that we maintain the "semaphore invariant" that its value never goes negative.** To ensure that this is the case, the wait operation blocks the calling process until the semaphore's value is positive. Pseudocode for these operations is as follows:

```
typedef struct {
    int i;
} Semaphore;

void wait(Semaphore *semp) {
    if (semp->i == 0)
        ...block the thread...

    semp->i--;
}

void signal(Semaphore *semp) {
    semp->i++;

    if (...a thread is blocked on this semaphore...)
        ...unblock one of them...
}
```

[2] Dijkstra named these operations after Dutch words describing their purposes, hence the odd [to us] letters. *P*(*s*) stands for "*prolaag*," short for "*probeer te verlagen*" which means "try to reduce." *V*(*s*) stands for "*verhogen*," meaning "increase." (CS:APP2e says that *P*(*s*) means "*proberen*," but Dijkstra himself stated that this is not the case.)

These operations are concurrency primitives, but we know that incrementing and decrementing a variable are multi-step operations, not to mention "testing and then conditionally decrementing," which is obviously multi-step! If we do not enforce that these operations are executed atomically, then they will eventually run in an interleaved manner that generates completely broken results. Thus, we must ensure that each of these operations is executed atomically.

## Locks and Fairness

Semaphores and mutexes both have an interesting characteristic: there is no specification of which thread is resumed if multiple threads are waiting on the lock. In general, they only specify that *some* waiting thread is resumed. This means that if multiple threads are waiting on a single lock, there is no guarantee that the lock will be granted to each one in order. Worse, if new threads continually attempt to acquire the lock, it is even possible that some threads may never get the lock.

Therefore, we will further specify that our semaphore must treat blocked threads *fairly*. In other words, a thread cannot remain blocked on the semaphore forever if the signal operation is executed infinitely often. A simple way to do this is to put newly blocked threads onto the end of a queue, and retrieve a blocked thread to resume from the front of the queue. This ensures that all blocked threads will eventually acquire the semaphore.

**Problem 4:** Implement semaphores in the **`semaphore.c`** file. Your implementation may use the **`sthread_block()`** and **`sthread_unblock()`** functions to block and resume threads, and it may use **`__sthread_lock()`** and **`__sthread_unlock()`** to enforce critical sections within the semaphore implementation. *(30 points)*

Your implementation may not "busy-wait" or "spin-wait" for the semaphore to become available; in general this is a *very* inefficient way to use the CPU, and is only useful in special circumstances.

**Note:** As stated initially, you must explain the data structure(s) you use for your implementation, and why you need each block/unblock/lock/unlock call in your code. Your explanation can be short, but it must be complete.

**Problem 5:** Using semaphores, modify the bounded buffer implementation so that it is both thread-safe and efficient. *(20 points)*

Just like the semaphore implementation, your bounded-buffer functions should not busy-wait for data (or space) to become available. For example, the provided implementation of **`bounded_buffer_take()`** is unacceptable because the thread yields instead of blocking if no data is available. Imagine the case where there is only one thread in the system, and it calls **`bounded_buffer_take()`**. The thread will yield, but there is no other thread to switch to, so the yielding thread resumes running again. But then it yields again, and again and again. This is not a passive wait; this is a busy-wait.

To make the bounded buffer more efficient, waiting threads should actually block instead of simply yielding, so that they don't consume any CPU resources until they can proceed.

**Note:** As stated initially, to receive full credit, you must make sure to clearly document the purpose of each semaphore you use, and the need for any synchronization call you make in your implementation (wait/signal, block/unblock, lock/unlock).

**Hint 1**: You will need multiple semaphores in the bounded buffer to implement this functionality.

**Hint 2**: As stated previously, a semaphore may be initialized to any value, not just 1 (which would create a binary semaphore or a mutex – see below). You can use this feature to implement passive waiting on empty/full buffers very easily…

## Appendix 1: More Semaphore Notes

Semaphores are actually a generalization of mutexes, because the semaphore value may be initialized to any nonzero value. For example, if a semaphore is initialized to 3, it means that three threads can acquire the semaphore (i.e. call $P(s)$ without blocking), before subsequent calls to $P(s)$ will block the calling threads. This is a nice feature; it allows us to place upper limits on the number of threads that can access shared resources, as well as using them to impose mutual exclusion. (Of course, you can only enforce mutual exclusion if the semaphore only holds the values of 0 or 1.)

Conceptually, a mutex is simply a binary semaphore; i.e. a semaphore whose value is initialized to 1. Of course, mutexes are usually not implemented this way, because processors often provide special instructions like **`bts`** and **`xchg`** that can be used to implement mutexes much more efficiently. If the only values are 1 or 0, there's not a lot of point in incrementing or decrementing things.

As stated earlier, neither mutexes nor semaphores make any guarantees about fairness. Specific implementations might make guarantees about fairness (as does our semaphore above), but in general you must be very careful about assuming that a library will provide such guarantees.

## Appendix 2: `timer.c` Operation

The code in **`timer.c`** uses UNIX signals to periodically switch between threads, but in a much more advanced way than we ever discussed in class. As you will recall, when a process is interrupted by a hardware or software exception, the kernel must save the process' context before invoking the signal handler, so that when the signal handler returns, the kernel can resume executing the process exactly where it was interrupted, with the same register and flag state as when it was interrupted.

The UNIX signal API actually provides a mechanism for signal handlers to access the context of the interrupted process, via the **`sigaction()`** function. This is a much more advanced version of the **`signal()`** function, that allows one to register a signal handler with the following signature:

```
void handler(int, siginfo_t *info, ucontext_t *uap);
```

The final argument references the machine context of the interrupted code, and generally it should not be monkeyed with if your code is going to be portable across different operating systems. But, in our case it can be used to do something very neat.

On Linux, the **`ucontext_t`** struct contains a member called **`gregs`**, which is an array containing the register state at the time the process was interrupted by the timer-interrupt. The **`ucontext.h`** header file defines these structures, as well as a variety of constants for accessing the elements of the **`gregs`** member, such as **`REG_EIP`** and **`REG_ESP`**.

What is even crazier is that these values are not read-only; we can actually modify these values, and when the signal handler returns, these values will be used to restore the process' context. This

means we can push values onto the process' stack, and we can even modify the **`%eip`** of the interrupted process to control where it resumes executing.

This is exactly what the **`timer_action()`** signal handler does: It modifies the process state to look as if the code had just called **`__sthread_schedule()`**, so that when the signal handler returns, the thread will in fact begin executing **`__sthread_schedule()`**. The signal handler also makes sure to update the stack properly, so that when **`__sthread_schedule()`** returns, it will resume executing the thread exactly where it was preempted.

This is why **`__sthread_schedule()`** must be careful to save *all* registers and the flags; we really don't have any idea what the thread is doing when it's interrupted. It might be performing a comparison, or setting up to call a function, or doing some math. Thus, we must save and restore *all* register state, not just the caller-save registers.