Systems Programming and Operating Systems	Induprakas Kodukula
CS414, Summer 1996	Class#3, 7/10/96

1 Class #3

We first examine the need to study concurrency in a conventional operating systems. For this purpose, we consider the case of I/O in an operating system that happens "concurrently" with the execution of a user program. Figure 1 shows a typical instance of how I/O works in a modern computer. Because I/O is asynchronous with respect to the processor, the I/O contoller and the CPU communicate via a bounded buffer *buf* in memory which has a certain amount of preallocated storage. For an input device, the I/O channel is responsible for transferring data to the bounded buffer and the CPU is responsible for reading it. All data unread by the CPU is represented as a circular queue in the bounded buffer. The circular queue is completely specified by two parameters *first* and *num*. *First* indicates the index of the position in *buf* that the CPU should read from next. It is where the start of the circular queue is. *Num* indicates the number of elements in the circular queue. In other words, the circular queue is stored from $buf[first]$ to $buf[first \oplus num - 1]$. To read data from *buf*, the reading process typically does the following:(Figure 2) The process of writing data to *buf* is accomplished by a similar piece of code (Figure 3) Each of these algorithms in isolation have the desired effect of reading from the right location or writing to the right location. However, when these algorithms are interleaved, the results can be quite interesting. We assume the following *memory atomicity model*

> Reading/Writing a single word of shared memory is an atomic operation

Let's focus on the operations involving num for a moment. In the reader, num is decremented and it is incremented in the writer. Decrementing of num happens in the following sequence:

```
load num, R
sub R, 1
store R, num
```

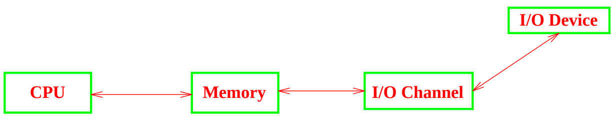
Similarly, the incrementing of num happens in the following sequence:

```
load num, R
add R, 1
store R, num
```

The memory system guarentees that each of the load and store operations happen atomically, but in general with process execution interleaved, the operations of incrementing and decrementing *num* can be interleaved in fairly arbitraty ways. By considering two different interleavings of these operations shown in Figure 4, it is easy to see that depending on how these operations are interleaved, the updating of num is different. The reason this happens is that atomicity at the level of individual load and store operations is not enough. It seems as though entire statements or program fragments need to be executed "atomically". This condition is known as the **critical section problem** and the region of code that needs to be executed atomically is known as the **critical section**.

However, there are some added complications even in this simple example. For instance, we need to ensure that

> CPU needs to be delayed if the buffer is empty, i.e. $num = 0$



buf[0..N-1]: Storage for circular queue
first: First element (in the range 0..N-1)
num: Number of elements (in the range 0..N)

Figure 1: : Motivation for Concurrency

Reading Procedure
{
1: readval = buf[first];
2: first = first $\oplus$ 1;
3: num = num - 1;
}

Figure 2: :Reading code from bounded buffer

Reading Procedure
{
1: buf[first$\oplus_N$ num] = val;
2: num = num+1
}

Figure 3: : Reading code from bounded buffer

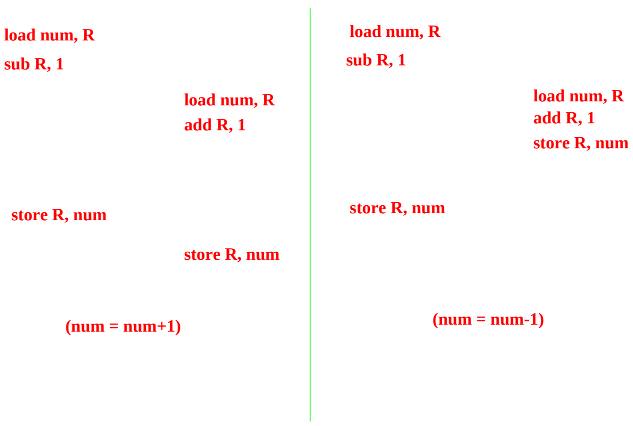


Figure 4: Possible Interleavings of increment and decrement instructions

Reading Procedure in a loop

```
{
    while (num == 0) loop;
    readval = buf[first];
    first = first ⊕ + 1
    num = num - 1;
}
```

Figure 5: : Reading code from bounded buffer

I/O channel needs to be delayed if the buffer is full, i.e. $num = \mathrm{N}$

Each of these conditions can be ensured by coding the reading and writing functions inside of a while loop. For example, the reading could be as in Figure 5. The purpose of the while loop is to ensure that if *buf* is empty, the reader process blocks until the writer (I/O channel) deposits something into *buf*. A similar loop can be used by the writer to hold off from writing into *buf* if the buffer is full (i.e. num = N).

An abstract formulation of this particular problem is as follows. A **abstract process** is an independent entity that we deal with. It is basically a program under execution. There is a one-to-one correspondence between an abstract process and a process on a machine. The **finite progress** assumption is that each process executes at some finite, but undetermined speed. The **finite** progress assumption is the strongest that we are prepared to make.

There are two important reasons that we seek to model program execution as multiple processes.

- This is an accurate characterization of reality. There is a real parallelism in usual program execution provided by I/O and interrupts.
- This analysis provides ways to structure a program to have high throughput by restructuring it as multiple processes. One piece of computation in a program can proceed while another is delayed.

Both reasons require processes to cooperate. Cooperation requires communication. Communication and a finite progress assumption **requires** synchronization. This is because when a process reads a shared state, it must know

- The state is meaningful, (critical section requirement)
- It can proceed further (conditional synchronization)

So, we are faced with the problems:

- How to specify processes
- How to specify synchronization