2.8 PROCESS SYNCHRONIZATION

2.8.1 Background

Cooperating processes (those that can effect or be effected by other simultaneously running processes) with the producer-consumer problem as an example

Producer code :

```
item nextProduced; while( true )
{
/* Produce an item and store it in nextProduced */
nextProduced = makeNewItem( . . . );
/* Wait for space to become available */
while( ( ( in + 1 ) % BUFFER_SIZE ) == out )
; /* Do nothing */
/* And then store the item and repeat the loop. */
buffer[ in ] = nextProduced;
in = ( in + 1 ) % BUFFER_SIZE; }
```

Consumer code :

```
itemnextConsumed;
while( true ) {
/* Wait for an item to become available */
while( in == out ) ;
/* Do nothing */
/* Get the next available item */
nextConsumed = buffer[ out ] ;
out = ( out + 1 ) % BUFFER_SIZE;
/* Consume the item in nextConsumed
( Do something with it ) */
}
```

The only problem with the above code is that the maximum number of items which can be placed into the buffer is BUFFER_SIZE - 1. One slot is unavailable because there always has to be a gap between the producer and the consumer.

- We could try to overcome this deficiency by introducing a counter variable, as shown in the following code segments:
- Unfortunately we have now introduced a new problem, because both the producer and the consumer are adjusting the value of the variable counter, which can lead to a condition known as a **race condition**.
- In this condition a piece of code may or may not work correctly, depending on which of two simultaneous processes executes first, and more importantly if one of the processes gets interrupted such that the other process runs between important steps of the first process. (Bank balance example discussed in class.)
- The particular problem above comes from the producer executing "counter++" at the same time the consumer is executing "counter--". If one process gets part way through making the update and then the other process butts in, the value of counter can get left in an incorrect state.
- But, you might say, "Each of those are single instructions - How can they get interrupted halfway through?" The answer is that although they are single instructions in C++, they are actually three steps each at the hardware level: (1) Fetch counter from memory into a register, (2) increment or decrement the register, and (3) Store the new value of counter back to memory. If the instructions from the two processes get interleaved, there could be serious problems, such as illustrated by the following:

2.8.2 The Critical-Section Problem

- The general idea is that in a number of cooperating processes, each has a critical section of code, with the following conditions and terminologies:
- Only one process in the group can be allowed to execute in their critical section at any one time.
- The code preceding the critical section, and which controls access to the critical section, is termed the entry section.
- The code following the critical section is termed the exit section.
- The rest of the code not included in either the critical section or the entry or exit sections is termed the remainder section.

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

General structure of a typical process Pi

2.8.3 Solution to critical section problem

A solution to the critical section problem must satisfy the following three conditions:

Mutual Exclusion

- Only one process at a time can be executing in their critical section.

Progress

- If no process is currently executing in their critical section, and one or more processes want to execute their critical section, then only the processes not in their remainder sections can participate in the decision, and the decision cannot be postponed indefinitely. (I.e. processes cannot be blocked forever waiting to get into their critical sections.)

Bounded Waiting

- There exists a limit as to how many other processes can get into their critical sections after a process requests entry into their critical section and before that request is granted. (I.e. a process requesting entry into their critical section will get a turn eventually, and there is a limit as to how many other processes get to go first.)

- We assume that all processes proceed at a non-zero speed, but no assumptions can be made regarding the **relative** speed of one process versus another.
- Kernel processes can also be subject to race conditions, which can be especially problematic when updating commonly shared kernel data structures such as open file tables or virtual memory management. Accordingly kernels can take on one of two forms:
- Non-preemptive kernels do not allow processes to be interrupted while in kernel mode. This eliminates the possibility of kernel-mode race conditions, but requires kernel

mode operations to complete very quickly, and can be problematic for realtime systems, because timing cannot be guaranteed.

- Preemptive kernels allow for real-time operations, but must be carefully written to avoid race conditions. This can be especially tricky on SMP systems, in which multiple kernel processes may be running simultaneously on different processors.

2.8.4 Peterson's Solution

- Peterson's Solution is a classic software-based solution to the critical section problem. It is unfortunately not guaranteed to work on modern hardware, due to vagaries of load and store operations, but it illustrates a number of important concepts.
- Peterson's solution is based on two processes, P0 and P1, which alternate between their critical sections and remainder sections. For convenience of discussion, "this" process is Pi, and the "other" process is Pj. (I.e. j = 1 - i)
- Peterson's solution requires two shared data items:
- **int turn** - Indicates whose turn it is to enter into the critical section. If turn = = i, then process i is allowed into their critical section.
- **boolean flag[2]** - Indicates when a process **wants to** enter into their critical section.
- When process i wants to enter their critical section, it sets flag[i] to true.
- In the following diagram, the entry and exit sections are enclosed in boxes.
- In the entry section, process i first raises a flag indicating a desire to enter the critical section.
- Then turn is set to **j** to allow the **other** process to enter their critical section **if process j so desires.**
- The while loop is a busy loop (notice the semicolon at the end), which makes process i wait as long as process j has the turn and wants to enter the critical section.
- Process i lowers the flag[i] in the exit section, allowing process j to continue if it has been waiting.

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

To prove that the solution is correct, we must examine the three conditions listed above:

- **Mutual exclusion** - If one process is executing their critical section when the other wishes to do so, the second process will become blocked by the flag of the first process. If both processes attempt to enter at the same time, the last process to execute "turn = j" will be blocked.
- **Progress** - Each process can only be blocked at the while if the other process wants to use the critical section (flag[j] = = true), AND it is the other process's turn to use the critical section (turn = = j). If both of those conditions are true, then the other process (j) will be allowed to enter the critical section, and upon exiting the critical section, will set flag[j] to false, releasing process i. The shared variable turn assures that only one process at a time can be blocked, and the flag variable allows one process to release the other when exiting their critical section.
- **Bounded Waiting** - As each process enters their entry section, they set the turn variable to be the other processes turn. Since no process ever sets it back to their own turn, this ensures that each process will have to let the other process go first at most one time before it becomes their turn again.

Note that the instruction "turn = j" is **atomic,** that is it is a single machine instruction which cannot be interrupted.

2.8.5 Synchronization Hardware

- To generalize the solution(s) expressed above, each process when entering their critical section must set some sort of **lock**, to prevent other processes from

entering their critical sections simultaneously, and must release the lock when exiting their critical section, to allow other processes to proceed. Obviously it must be possible to attain the lock only when no other process has already set a lock. Specific implementations of this general procedure can get quite complicated, and may include hardware solutions as outlined in this section.

- One simple solution to the critical section problem is to simply prevent a process from being interrupted while in their critical section, which is the approach taken by non preemptive kernels. Unfortunately this does not work well in multiprocessor environments, due to the difficulties in disabling and the re-enabling interrupts on all processors. There is also a question as to how this approach affects timing if the clock interrupt is disabled.
- Another approach is for hardware to provide certain **atomic** operations. These operations are guaranteed to operate as a single instruction, without interruption. One such operation is the "Test and Set", which simultaneously sets a boolean lock variable and returns its previous value, as shown in Figures

booleantest_and_set (boolean *target)

{

booleanrv = *target;

*target = TRUE;

return rv: }

Solution using test-and-set:

```
do
{       while (test_and_set(&lock))
        ; /* do nothing */              /* critical section */
lock = false;
          /* remainder section */
    } while (true);
int compare _and_swap(int *value, int expected, intnew_value)
{       int temp = *value;
if (*value == expected)
*value = new_value;
return temp;
```

```
}
```

Solution using compare-and swap:

```
do {
while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */         /* critical section */
lock = 0;
       /* remainder section */
   } while (true);
```

- The above examples satisfy the mutual exclusion requirement, but unfortunately do not guarantee bounded waiting. If there are multiple processes trying to get into their critical sections, there is no guarantee of what order they will enter, and any one process could have the bad luck to wait forever until they got their turn in the critical section. (Since there is no guarantee as to the relative **rates** of the processes, a very fast process could theoretically release the lock, whip through their remainder section, and re-lock the lock before a slower process got a chance. As more and more processes are involved vying for the same resource, the odds of a slow process getting locked out completely increase.)
- Figure illustrates a solution using test-and-set that does satisfy this requirement, using two shared data structures, boolean lock and boolean waiting[N], where N is the number of processes in contention for critical sections:

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

      /* critical section */

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = false;
   else
      waiting[j] = false;

      /* remainder section */
} while (true);
```

Bounded-waiting mutual exclusion with TestAndSet( ).

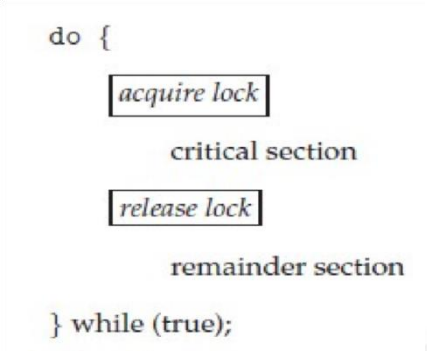
The key feature of the above algorithm is that a process blocks on the AND of the critical section being locked and that this process is in the waiting state. When exiting a critical section, the exiting process does not just unlock the critical section and let the other processes have a free-for-all trying to get in.

Rather it first looks in an orderly progression (starting with the next process on the list) for a process that has been waiting, and if it finds one, then it releases that particular process from its waiting state, without unlocking the critical section, thereby allowing a specific process into the critical section while continuing to block all the others. Only if there are no other processes currently waiting is the general lock removed, allowing the next process to come along access to the critical section.

2.8.6 Mutex Locks

- The hardware solutions presented above are often difficult for ordinary programmers to access, particularly on multi-processor machines, and particularly because they are often platform-dependent.
- Therefore most systems offer a software API equivalent called **mutex locks** or simply **mutexes.** (For mutual exclusion)

- The terminology when using mutexes is to **acquire** a lock prior to entering a critical section, and to **release it when exiting, as shown in Figure :**




```
acquire()
{       while (!available)
        ; /* busy wait */
available = false;;
   }    release() {
available = true;
   }
```

- Just as with hardware locks, the acquire step will block the process if the lock is in use by another process, and both acquire ( ) and release ( ) operations are atomic.
- Acquire and release can be implemented as shown here, based on a Boolean variable "available": .
- One problem with the implementation shown here, (and in the hardware solutions presented earlier), is the busy loop used to block processes in the acquire phase. These types of locks are referred to as **spinlocks**, because the CPU just sits and spins while blocking the process.
- Spinlocks are wasteful of CPU cycles, and are a really bad idea on single-cpu single threaded machines, because the spinlock blocks the entire computer, and doesn't allow any other process to release the lock. (Until the scheduler kicks the spinning process off of the cpu.)
- On the other hand, spinlocks do not incur the overhead of a context switch, so they are effectively used on multi-threaded machines when it is expected that the lock will be released after a short time.

2.8.7 Semaphores

A more robust alternative to simple mutexes is to use **semaphores**, which are integer variables for which only two (atomic) operations are defined, the wait and signal operations, as shown in the following figure.

```
wait(S) {    while (S <= 0)      ; // busy wait    S--; } signal(S) {
  S++;
}
```

Note that not only must the variable-changing steps (S-- and S++) be indivisible, it is also necessary that for the wait operation when the test proves false that there be no interruptions before S gets decremented. It IS okay, however, for the busy loop to be interrupted when the test is true, which prevents the system from hanging forever.

Semaphore Usage

In practice, semaphores can take on one of two forms:

- **Binary semaphores** can take on one of two values, 0 or 1. They can be used to solve the critical section problem as described above, and can be used as mutexes on systems that do not provide a separate mutex mechanism.. The use of mutexes for this purpose is shown in Figure.
- **Counting semaphores** can take on any integer value, and are usually used to count the number remaining of some limited resource. The counter is initialized to the number of such resources available in the system, and whenever the counting semaphore is greater than zero, then a process can enter a critical section and use one of the resources. When the counter gets to zero (or negative in some implementations), then the process blocks until another process frees up a resource and increments the counting semaphore with a signal call. (The binary semaphore can be seen as just a special case where the number of resources initially available is just one.)
- Semaphores can also be used to synchronize certain operations between processes. For example, suppose it is important that process P1 execute statement S1 before process P2 executes statement S2.
- First we create a semaphore named synch that is shared by the two processes, and initialize it to zero.

Then in process P1 we insert the code:

S1; signal(synch);

and in process P2 we insert the code:

wait(synch) ;

S2;

Because synch was initialized to 0, process P2 will block on the wait until after P1 executes the call to signal.

Semaphore Implementation

• The big problem with semaphores as described above is the busy loop in the wait call, which consumes CPU cycles without doing any useful work. This type of lock is known as a **spinlock**, because the lock just sits there and spins while it waits. While this is generally a bad thing, it does have the advantage of not invoking context switches, and so it is sometimes used in multi-processing systems when the wait time is expected to be short - One thread spins on one processor while another completes their critical section on another processor.

• An alternative approach is to block a process when it is forced to wait for an available semaphore, and swap it out of the CPU. In this implementation each semaphore needs to maintain a list of processes that are blocked waiting for it, so that one of the processes can be woken up and swapped back in when the semaphore becomes available. (Whether it gets swapped back into the CPU immediately or whether it needs to hang out in the ready queue for a while is a scheduling problem.)

• The new definition of a semaphore and the corresponding wait and signal operations are shown as follows:

```
typedefstruct{          int value;
struct process *list;
  } semaphore; wait(semaphore *S) {    S->value--;
if (S->value < 0) {
add this process to S->list;      block();
  }
} signal(semaphore *S) {    S->value++;
if (S->value <= 0) {
```

remove a process P from S->list; wakeup(P);

}

}

- Note that in this implementation the value of the semaphore can actually become negative, in which case its magnitude is the number of processes waiting for that semaphore. This is a result of decrementing the counter before checking its value.
- Key to the success of semaphores is that the wait and signal operations be atomic, that is no other process can execute a wait or signal on the same semaphore at the same time. (Other processes could be allowed to do other things, including working with other semaphores, they just can't have access to **this** semaphore.) On single processors this can be implemented by disabling interrupts during the execution of wait and signal; Multiprocessor systems have to use more complex methods, including the use of spinlocking.

Deadlocks and Starvation

- One important problem that can arise when using semaphores to block processes waiting for a limited resource is the problem of **deadlocks**, which occur when multiple processes are blocked, each waiting for a resource that can only be freed by one of the other (blocked) processes, as illustrated in the following example.

- Let ***S*** and***Q***be two semaphores initialized to 1

P_0P_1

wait(S); wait(Q); wait(Q); wait(S);

... ...

signal(S); signal(Q); signal(Q); signal(S);

Another problem to consider is that of **starvation**, in which one or more processes gets blocked forever, and never get a chance to take their turn in the critical section. For example, in the semaphores above, we did not specify the algorithms for adding processes to the waiting queue in the semaphore in the wait( ) call, or selecting one to be removed from the queue in the signal( ) call. If the method chosen is a FIFO queue, then every process will eventually get their turn, but if a LIFO queue is implemented instead, then the first process to start waiting could starve.

Priority Inversion

- A challenging scheduling problem arises when a high-priority process gets blocked waiting for a resource that is currently held by a low-priority process.
- If the low-priority process gets pre-empted by one or more medium-priority processes, then the high-priority process is essentially made to wait for the medium priority processes to finish before the low-priority process can release the needed resource, causing a **priority inversion.** If there are enough medium-priority processes, then the high-priority process may be forced to wait for a very long time.
- One solution is a **priority-inheritance protocol,** in which a low-priority process holding a resource for which a high-priority process is waiting will temporarily inherit the high priority from the waiting process. This prevents the medium-priority processes from preempting the low-priority process until it releases the resource, blocking the priority inversion problem.