Chapter 6: Process Synchronization

Synchronization Hardware

- Previously we studied a software based solution to critical section problem (Peterson solution).
- Many systems provide hardware support for critical section code.

- **Uniprocessors** – We could disable interrupts for preemption so when a shared data is being modified no preemption will occur.
  - Generally too inefficient on **multiprocessor** systems. Disabling interrupts on a multiprocessor is very time consuming, as the message is passed to all the processors

- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable

TestAndSet() instruction

- The important characteristic is that this instruction is executed atomically.
- If two TestAndSet C) instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.

```
boolean TestAndSet (boolean *target)
  {
      boolean rv = *target;
      *target = TRUE;
      return rv:
  }
```

TestAndSet() instruction (cntd)

- If the machine supports the TestAndSet () instruction, then we can implement mutual exclusion by declaring a global Boolean variable lock, initialized to false.
- Solution using TestAndSet
  - Shared boolean variable lock is initialized to false.

```
do {
      while ( TestAndSet (&lock ))
              ;   /* do nothing

          //    critical section

      lock = FALSE;

          //      remainder section

  } while ( TRUE);
```

Semaphore

Another synchronization tool is called semaphore. A semaphore S is an integer variable that is accessed only through two standard *atomic operations*: **wait ()** and **signal ()**.

- Wait is represented by **P()** and signal by **V()**.

```
wait (S) {
    while S <= 0
        ; // no-op
      S--;
 }
```

```
signal (S) {
      S++;
    }
```

Semaphore (cntd)

- **Binary semaphore** – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks

- **Counting semaphore** – integer value can range over an unrestricted domain.
  - Used to control access to a given resource consisting of a finite number of instances.
  - Semaphore is initialized to the number of resources available. Each process performs a waitQ operation on the semaphore (thereby decrementing the count).
  - When a process releases a resource, it performs a signal () operation (incrementing the count).
  - When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

Semaphore (cntd)

- **Mutual Exclusion**
- We can use binary semaphores to deal with the critical-section problem for multiple processes. The n processes share a binary semaphore (mutex lock) initialized to 1.

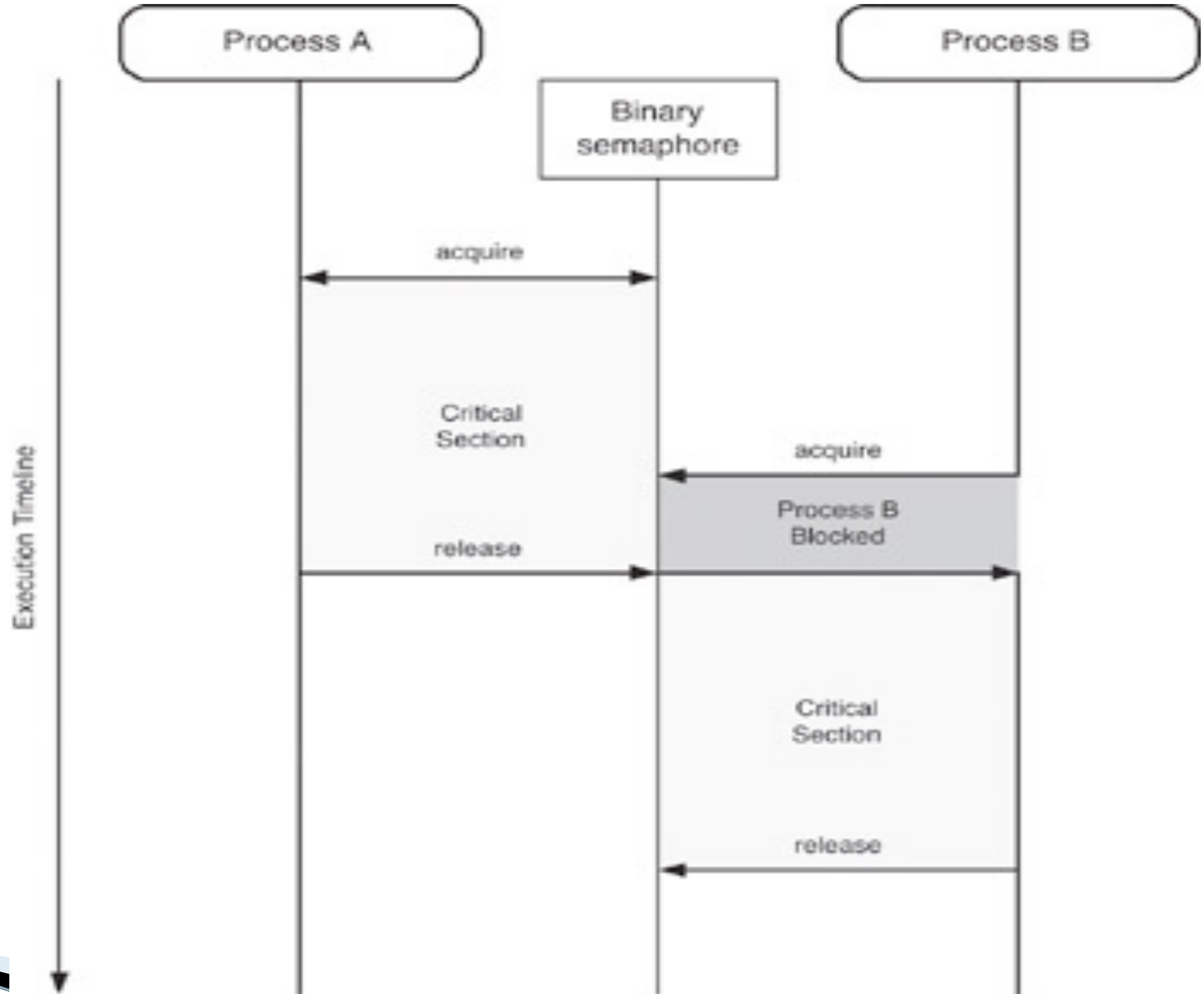
- Each process P, is organized as shown

  - Semaphore S;   // initialized to 1
  - wait (S);

            Critical Section

    signal (S);

Semaphore (cntd)



Semaphore (cntd)

- **Suppose we have 3 printers shared between 5 processes. Write code using semaphore to provide synchronized access to the printers.**

Semaphore (cntd)

- **Implementation**– Main disadvantage of the semaphore definition given previously is that every process waiting for semaphore loops continuously in the entry code.

- Looping wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a **spinlock** (process "spins" while waiting for the lock).

- To eliminate busy waiting when a process has to wait for semaphore it is *blocked*.

- The block operation places a process into a waiting queue associated with the semaphore. Hence a waiting process is not scheduled so no cpu cycles are wasted.

Semaphore (cntd)

- **Implementation**– semaphores under this definition is defined as a "C" struct:

```
typedef struct {
   int value;
   struct process *list;
} semaphore;
```

- **Implementation of wait:**

```
wait (S){
   S->value--;
   if (S->value < 0) {
   add this process to waiting queue
   block();
   }
}
```

Implementation of signal:

```
Signal (S){
      S->value++;
      if (S->value <= 0) {
     remove a process P from the
    waiting queue
     wakeup(P);
      }
}
```

Semaphore (cntd)

- The block() operation suspends the process that invokes it.
- The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

- Note that, although under the classical definition of semaphores with busy waiting the semaphore value is never negative, this implementation may have negative semaphore values.
- If the semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

```
P0                      P1
    wait (S);                wait (Q);
    wait (Q);                wait (S);
    .                        .
    .                        .
    .                        .
    signal (S);              signal (Q);
    signal (Q);              signal (S);
```

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.