

Mutex Locks

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

release() {
   available = true;
}
```

Problem: spinlock - process "spins" while waiting for the lock.





Semaphore

- \square Semaphore S integer variable
- ☐ Two standard operations modify S: wait() and signal()
 - \square Originally called P() \rightarrow P (from the Dutch proberen, "to test");
 - $V() \rightarrow$ (from verhogen, "to increment").
- Less complicated
- ☐ Can only be accessed via two indivisible (atomic) operations

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

2 versions of semaphore exists, 1st version with busy waiting, 2nd version with blocking call





Semaphore as General Synchronization Tool

- Binary semaphore integer value can range only between 0 and 1; can be simpler to implement
 - Also known as mutex locks
- Mutual-exclusion implementation with Binary semaphore

```
Semaphore mutex; // initialized to 1 do { wait (mutex); // Critical Section signal (mutex); // remainder section } while (TRUE);
```





- Use a binary semaphore when the goal is to protect a critical section. The binary semaphore ensures that only one thread or process can execute in the critical section at any given time.
- Use a counting semaphore when you have a fixed number of resources that can be shared among multiple threads or processes, but the access to these resources does not constitute a critical section that requires mutual exclusion.
 - Managing access to multiple resources. Counting semaphores allow multiple threads or processes to access shared resources concurrently, up to a specified limit. They are not typically used to enforce mutual exclusion in critical sections, as they allow more than one thread to proceed if the count is greater than 1.
 - Eg: DB connections





- Counting semaphore integer value can range over an unrestricted domain
- □ Case 1: To control access to a given resource
- Semaphore is initialized to the number of resources
- □ Semaphore Count =0;all resources are utilized
- ☐ Case 2 : process synchronization
- ☐ (if we want the restriction on the order of execution: s2 is executed only after s1 completes)
- □ synch=0

```
S_1; signal(synch); Disadvantage: Busy waiting thus wastes CPU cycles wait(synch); S_2;
```





- □ While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.
- This type of semaphore is also called spinlock a because the process "spins" while waiting for the lock.
- Busy waiting wastes CPU cycles that some other process might be able to use productively.
- □ To overcome the need for busy waiting, we can modify the definition of the wait() and signal() semaphore operations.
- rather than engaging in busy waiting, the process can *block* itself. The block operation places a process into a waiting queue associated with the semaphore





Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
 - value (of type integer)
 - pointer to next record in the list

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

- Two operations:
 - block place the process invoking the operation on the appropriate waiting queue
 - wakeup remove one of processes in the waiting queue and place it in the ready queue





Semaphore Implementation with no Busy waiting (Cont.)

Synchronization tool that does not require busy waiting

Implementation of wait:

```
wait(semaphore *S) {
       S->value--:
                                                    Entry
       if (S->value < 0) {
              add this process to S->list;
              block();
```

Here, semaphore values may be negative, whereas semaphore values are never negative under the classical definition of semaphores with busy waiting.

Semaphore value -ve indicates the processes waiting to acquire that semaphore

Implementation of signal:

```
signal(semaphore *S) {
      S->value++:
      if (S->value <= 0)
              remove a process P from S->list;
             wakeup(P);
                                 6.8
```

Put the processes from the blocked list to ready queue... not into CS

Exit



Silberschatz, Galvin and Gagne ©2011

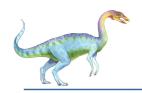


Deadlock and Starvation

- □ Deadlock two or more processes are waiting indefinitely for an event
- Let S and Q be two semaphores initialized to 1

- Starvation indefinite blocking
 - occur if we remove processes from the list associated with a semaphore in LIFO order





Priority Inversion

- Priority Inversion Scheduling problem when lower-priority process holds a lock needed by higher-priority process
- Solved via priority-inheritance protocol

L and H(not M) share resource R

- L and H share/modify kernel data(CS)
- □ Assume we have three processes, L, M, and H, whose priorities follow the order L < M < H. Assume that process H requires resource R, which is currently being accessed by process L. Ordinarily, process H would wait for L to finish using resource R.</p>
- □ *M* becomes runnable, thereby preempting process
- □ a process with a lower priority—process *M*—has affected how long process *H* must wait for *L* to relinquish resource *R*.
- This problem is known as priority inversion.
- priority-inheritance protocol would allow process L to temporarily inherit the priority of process H, thereby preventing process M from preempting its execution. When process L had finished using resource R, it would relinquish its inherited priority from H and assume its original priority. Because resource R would now be available, process H—not M—would run next.