

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first acquire() a lock then release() the lock
 - Boolean variable indicating if lock is available or not
- Calls to acquire() and release() must be atomic
 - Usually implemented via hardware atomic instructions
- But this solution requires busy waiting
 - This lock therefore called a spinlock

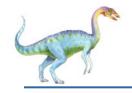




acquire() and release()

```
acquire() {
     while (!available)
         ; /* busy wait */
     available = false;;
  release() {
     available = true;
  do {
  acquire lock
      critical section
  release lock
    remainder section
} while (true);
```





Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore **S** integer variable
- Can only be accessed via two indivisible (atomic) operations
 - wait() and signal()Originally called P() and V()
- Definition of the wait() operation

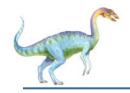
```
wait(S) {
    while (S <= 0)
    ; // busy wait
    S--;
}</pre>
```

Definition of the signal() operation

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



Silberschatz, Galvin and Gagne ©2013



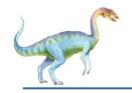
Semaphore Usage

- Counting semaphore integer value can range over an unrestricted domain
- Binary semaphore integer value can range only between 0 and 1
 - Same as a mutex lock
- Can solve various synchronization problems
- Consider P₁ and P₂ that require S₁ to happen before S₂
 Create a semaphore "synch" initialized to 0

```
P1:
S<sub>1</sub>;
signal(synch);
P2:
wait(synch);
S<sub>2</sub>;
```

Can implement a counting semaphore S as a binary semaphore

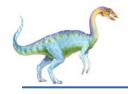




Semaphore Implementation

- Must guarantee that no two processes can execute the wait() and signal() on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the wait and signal code are placed in the critical section
 - Could now have busy waiting in critical section implementation
 - But implementation code is short
 - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution



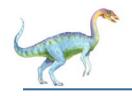


Semaphore Implementation with no Busy waiting

- 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
- 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

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

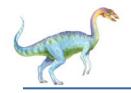




Implementation with no Busy waiting (Cont.)

```
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) {</pre>
      remove a process P from S->list;
      wakeup(P);
```





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

```
P_0 P_1 wait(S); wait(Q); wait(Q); ... signal(S); signal(Q); signal(S);
```

- Starvation indefinite blocking
 - A process may never be removed from the semaphore queue in which it is suspended
- Priority Inversion Scheduling problem when lower-priority process holds a lock needed by higher-priority process
 - Solved via priority-inheritance protocol

