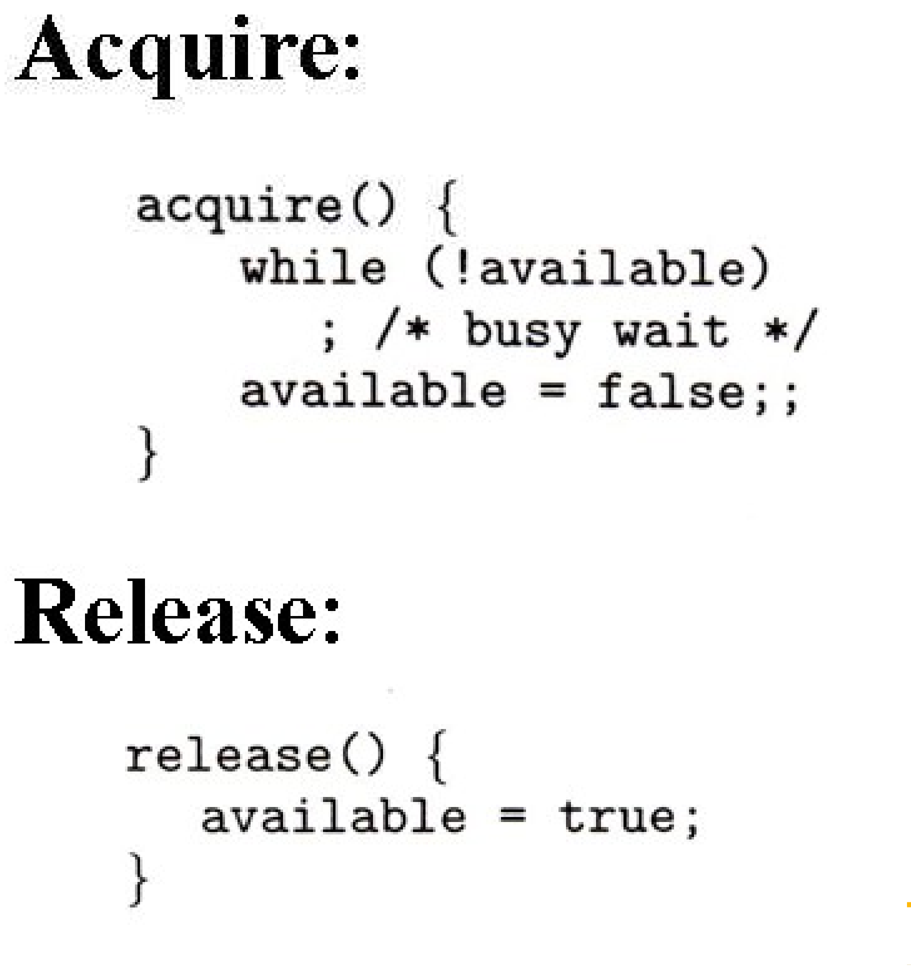
# MUTEX LOCKS FOR CSP

## Mutex Locks

- Hardware solutions discussed are often difficult for ordinary programmers
- More difficult on multi-processor machines, and often platform-dependent. Most systems offer a software API equivalent called ***mutex locks*** or simply ***mutexes.***

```
do {
      acquire lock
          critical section
      release lock
          remainder section
} while (TRUE);
```

- acquire step - block the process if the lock is in use by another process;
- **acquire and release operations are atomic**.
- Acquire and release can be implemented via variable "available":



**Acquire:**

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

**Release:**

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

- One problem - is the busy loop used to block processes in the acquire phase.
- 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. )
- Advantage is - spinlocks do not incur the overhead of a context switch,
- used on multi-threaded machines when it is expected that the lock will be released after a short time.

## Semaphores

✓alternative to simple mutexes is to use ***semaphores***, which are integer variables

✓ only two ( atomic ) operations are defined, the wait and signal operations.

✓Note that not only must the variable-changing steps ( S-- and S++ ) be indivisible,

✓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.

**Wait:**

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

**Signal:**

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

```
do {
   waiting(mutex);

      // critical section

   signal(mutex);

      // remainder section
}while (TRUE);
```

- **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.

- Binary semaphore -- special case of counting semaphores - where the number of resources initially available is just one.
- Semaphores can also be used to synchronize certain operations between processes.
- Assume that process P1 execute statement S1 before process P2 executes statement S2.
  - Create a semaphore named SEM that is shared by the two processes, and initialize it to zero.
  - P1 : S1;
    signal( SEM );
  - P2 :wait( SEM );
    S2;
  - Because synch was initialized to 0, process P2 will block on the wait until after P1 executes the call to signal

## Semaphore Limitations

- Limitation is 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.
- But has  the advantage of avoiding context switches, and so it is sometimes used in multi-processing systems ;
- Short wait times - 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.
- each semaphore has a list of processes waiting for the resource - ,  wake up one when resource becomes free.

## Semaphore Structure:

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

## Wait Operation:

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

## Signal Operation:

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

- value of the semaphore can get negative Bcoz of decrementing the counter before checking its value,
- magnitude is the number of processes waiting for that semaphore.
- **Important that the wait and signal operations be atomic,** 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.
- Uniprocessor env - 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

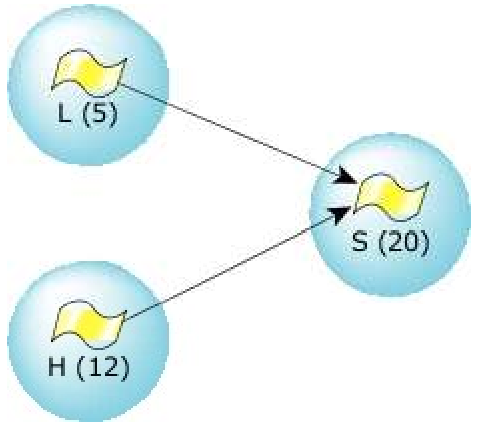
- Major Limitation of Semaphores is that the problem of ***deadlocks***,
- which occur when multiple processes are blocked, each waiting for a resource that can only be freed by one of the other ( blocked ) processes,
- Following example highlights one such scenario

```
      P0               P1

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

- Another problem to consider is that of ***starvation***,
- one or more processes gets blocked forever, and never get a chance to take their turn in the critical section.
- For example if waiting list is based on a FIFO queue, then every process will eventually get their turn,
- If LIFO queue is used it cud result in starvation,
- A loophole - when a high-priority process gets blocked waiting for a resource that is currently held by a low-priority process. Called as **priority inversion**

- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**
- Three processes: L, M, H
  - **Priority: L < M < H**
  - H is waiting for the mutex held by L
  - While L is in its critical section, M arrives and preempts L
  - H process' waiting time is prolonged by process M, which has a lower priority, causing priority inversion
  - PIP – convert to a binary priority setup;



- 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.

Earlier example

- L runs with priority of H
- M comes in mean time H is already waiting;
- So H wud get the resource ahead of M maintainig the priority ordering

## Mars Path Finder -- Priority Inversion

- Pathfinder's applications were scheduled by the VxWorks RTOS
- meteorological data gathering task ran as an infrequent, low priority thread (L) , and used the information bus synchronized with mutual exclusion locks (mutexes).
- very high priority bus management task, which also accessed the bus with mutexes. (H)
- long-running communications task, having higher priority M than the meteorological task, but lower than the bus management task, prevented it from running

## Mars Path Finder -- Priority Inversion

- watchdog timer noticed that the bus management task had not been executed for some time,
- concluded that something had gone wrong, and ordered a total system reset.
- Engineers later confessed that system resets had occurred during pre-flight tests.
- They put these down to a hardware glitch and returned to focusing on the mission-critical landing software!!
- http://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html maintains an excellent detail about this inversion problem

## ANALYSIS AND LESSONS

- First and foremost, diagnosing this problem as a black box would have been impossible. Only detailed traces of actual system behavior enabled the faulty execution sequence to be captured and identified.
- Secondly, leaving the "debugging" facilities in the system saved the day. Without the ability to modify the system in the field, the problem could not have been corrected.
- Finally, the engineer's initial analysis that "the data bus task executes very frequently and is time-critical -- we shouldn't spend the extra time in it to perform priority inheritance" was exactly wrong. It is precisely in such time critical and important situations where correctness is essential, even at some additional performance cost.

## HUMAN NATURE, DEADLINE PRESSURES

David told us that the JPL engineers later confessed that one or two system resets had occurred in their months of pre-flight testing. They had never been reproducible or explainable, and so the engineers, in a very human-nature response of denial, decided that they probably weren't important, using the rationale "it was probably caused by a hardware glitch".

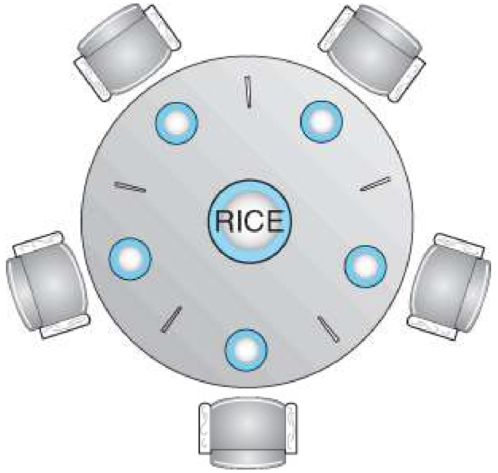
Part of it too was the engineers' focus. They were extremely focused on ensuring the quality and flawless operation of the landing software. Should it have failed, the mission would have been lost. It is entirely understandable for the engineers to discount occasional glitches in the less-critical land-mission software, particularly given that a spacecraft reset was a viable recovery strategy at that phase of the mission.

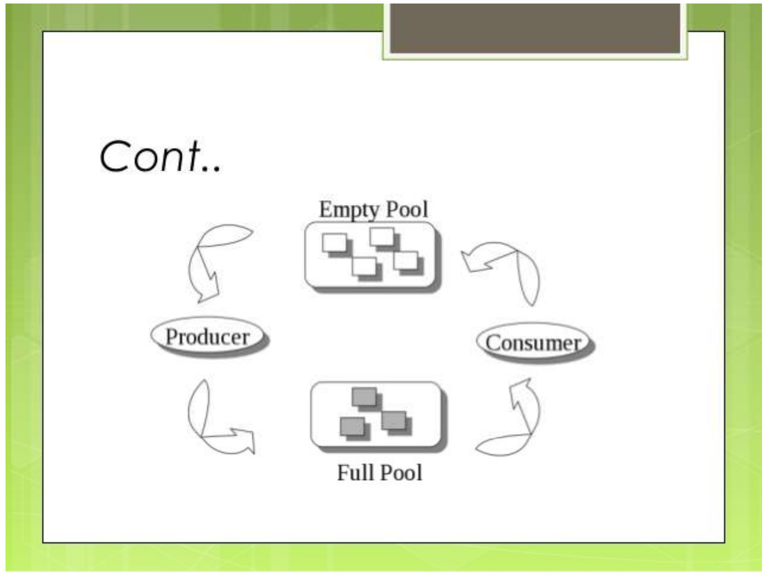
## THE IMPORTANCE OF GOOD  HEORY/ALGORITHMS

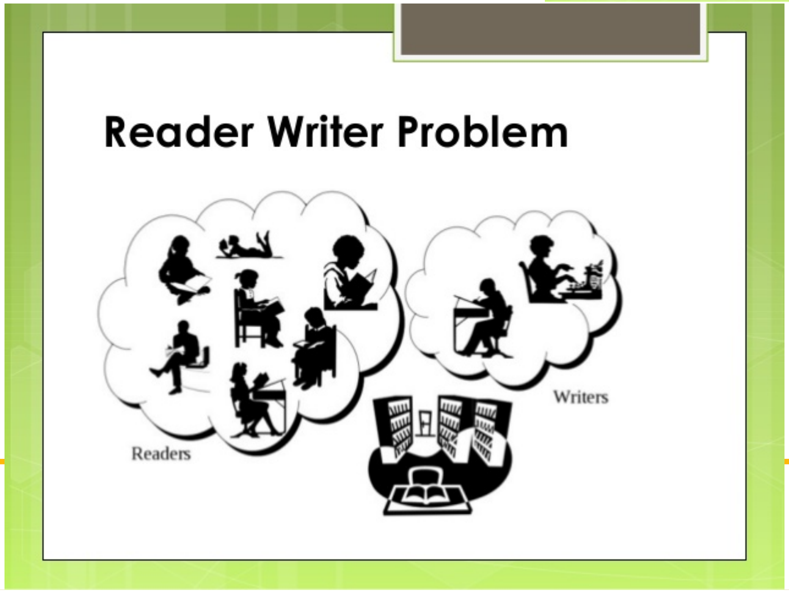
- David also said that some of the real heroes of the situation were some people from CMU who had published a paper he'd heard presented many years ago who first identified the priority inversion problem and proposed the solution. He apologized for not remembering the precise details of the paper or who wrote it.
- **Bringing things full circle**, it turns out that the three authors of this result were all in the room, and at the end of the talk were encouraged by the program chair to stand and be acknowledged. **Lui Sha, John Lehoczky, and Raj Rajkumar**. When was the last time you saw a room of people cheer a group of computer science theorists for their significant practical contribution to advancing human knowledge? :-) It was quite a moment.

## Some Classical CSP Problems

- Dining Philosophers Problem ; Reader Writer Problem
- Producer Consumer Problem




Cont..


Reader Writer Problem