CAS CS 552
Intro to Operating Systems

Richard West

Synchronization

Process Synchronization

- Cooperating processes share data (or, more generally, they share resources)
- Concurrent access to shared data may result in data inconsistency (or, more generally, unacceptable interleaved access to resources)

Example: Producer-Consumer Bounded Buffer 1/2

<u>producer</u>

```
while (true) {
	produce item;
	while (counter==n);
	buffer[in] = item;
	in = (in+1)%n;
	counter++;
}
```

<u>consumer</u>

```
while (true) {
	while (counter==0);
	item = buffer[out];
	out = (out+1)%n;
	counter--;
	consume item;
}
```

Example: Producer-Consumer 2/2

- Suppose producer & consumer execute concurrently
  - e.g., let counter = 10 and producer/consumer execute counter++ and counter--, respectively
  - After these statements, counter may be 9,10, or 11, although the correct answer is 10
  - counter = 10 is only guaranteed if producer & consumer execute the instructions that modify counter separately (i.e., atomically!)

Synchronization Principle

- Arbitrary interleaving of the execution of concurrent processes when accessing shared data can lead to data inconsistency
  - What we have is a “data race condition”

- We must ensure that cooperating processes access shared data one at a time using some method that enforces synchronization

Critical Sections

- Critical sections = code segments which access shared data/resources
  - Only <u>one</u> process can execute a critical section at a time
  - Critical sections require <u>mutually exclusive</u> access

Dealing with the Critical Section Problem

- (1) no two processes may be together in a critical section (C.S.)
- (2) no assumptions should be made about the relative speeds of processes
- (3) no process outside its critical section should block (the progress) of other processes
- (4) no process should wait arbitrarily long to enter its critical section

Two-Process Solution to the C.S. Problem

- Strict alternation (method 1)
- Consider two processes, P_i and P_j

```
while (true) {
    non-critical section;
    while (turn!=i);
    critical section;
    turn = j;
}
```

Problem: Does not satisfy rule (3) – P_i cannot enter its critical section if turn!=i, even though P_j may not be in its critical section

Better Solution to C.S. Problem with Two Processes 1/2

- (method 2) introduce a boolean variable “flag”
  - typedef enum {false, true} boolean;
  - boolean flag[2];
  - enum {0,1} turn;

  - Initially, flag[0]=flag[1]=false;
    - turn can be either 0 or 1;

Better Solution to C.S. Problem with Two Processes 2/2

```
while (true) {
    non-critical section;
    flag[i]=true;
    turn=j;
    while ((flag[j]) && (turn==j));
    critical section;
    flag[i]=false;
}
```

- i = local process ID; j = remote/other process ID
- This solution satisfies all four conditions for dealing with critical section problem

Multiple Process Solutions

- Bakery algorithm for n processes
  - Customers assigned ticket numbers
  - Customer with lowest ticket number is serviced next
  - Two customers can have same ticket number, so ties are broken according to some rule:
    - e.g., customer Ci precedes Cj if ID(Ci) < ID(Cj)
    - This requires customers to be numerically and uniquely identified

Bakery Algorithm 1/2

- Common data structures:
  - Boolean choosing[n];
  - Int number[n];
  - Initially, these data structures are set to false and 0, respectively
- Notation:
  - Let, (a,b) < (c,d) if (a<c) || ((a==c) && (b<d))
  - $\max(a_0,\ldots,a_{n-1})$ = largest value in set $\{a_0,\ldots,a_{n-1}\}$

- The following guarantees all four conditions for dealing with critical sections…

Bakery Algorithm 2/2

- Process P_i:

```
while(true) {
      choosing[i]=true;
      number[i]=max(number[0],…,number[n-1])+1;
      choosing[i]=false;
      for (j=0;j<n;j++) {
            while (choosing[j]);
            while ((number[j]!=0) &&
                        ((number[j],j) < (number[i],i)));
      }
      critical section;
      number[i]=0;
      non-critical section;
}
```

Synchronization Hardware

- On uniprocessors, disabling interrupts guarantees safety when a process is executing in a critical section
  - Disabling interrupts is not always practical on a multiprocessor
- Hardware provides <u>atomic</u> instructions
  - Cannot be interrupted but must instead be executed to completion, thereby preventing *interleaved* execution

Test-and-set

- Implemented as an atomic instruction, test-and-set conceptually looks like:

```
atomic boolean test-and-set (boolean *target) {
    boolean b;
    b = *target;
    *target = true;
    return b;
}
```

A test-and-set Mutex

```
while (true) {
    while (test-and-set(&lock));
    critical section;
    lock=false;
    non-critical section;
}
```

- Initially, lock=false;

Test-and-set Usage 1/2

- Using a shared variable “flag” initially 0, here is another solution to the critical section problem:

```
enter_region:
    tsl register, flag   ; copy flag to register and set
                         ; flag to 1
    cmp register, #0 ; was flag 0?
    jnz enter_region  ; if it was non-zero, lock was set
                      ; so loop
    ret               ; return to caller
leave_region:
    mov flag, #0      ; flag=0
    ret               ; return to caller
```

Test-and-set Usage 2/2

- Each process has the following pseudo-code:

```
while (true) {
    enter_region;
    critical section;
    leave_region;
    non-critical section;
}
```

Compare-and-Swap (CAS)

- e.g., CMPXCHG x86 instruction
- Compare contents of memory location w/ specified value
  - if same, change memory location contents to new value

```
atomic int CAS (int *pval, int old, int new) {
  int val = *pval;
   if (val == old)
      *pval = new;
   return val;
}
```

CAS Usage

- Atomic variable increment (fetch-and-add)

```
void incr (int *pval) {
   do {
      int tmp = *pval;
   }
   while (CAS (pval, tmp, tmp+1) != tmp);
}
```

Semaphores

- Used for synchronization
- A semaphore "s" is an integer variable accessed via two atomic operations: <u>wait</u> and <u>signal</u>
  - wait (also known as "p" for "proberen" – Dutch for "to test")
  - signal (also known as "v" for "verhogen" – Dutch for "to increment")

- Conceptually:
  - wait(s): while (s<=0); s--;
  - signal(s): s++;

Semaphores Example

- Let “mutex” be a binary semaphore, initialized to 1
- Process, Pi:

```
while (true) {
        wait(mutex);
        critical section;
        signal(mutex);
        non-critical section;
}
```

- All mutual exclusion problems so far require <u>busy waiting</u>
  - While one process is in its critical section another process must loop through its entry code before entering its own critical section

Spinning versus Blocking Locks

- Spinlocks are essentially busy waiting semaphores
  - A process spins in a tight loop waiting for a lock
- Busy waiting (and, hence, spinlocks) are not good for uniprocessor systems – Why?
  - Waste CPU cycles

- Spinlocks *are* useful in multiprocessor systems (and uniprocessor systems with preemptible kernels)
  - No context-switch is required when a process or thread waits for a lock

- Instead of spinning, a process/thread can be <u>blocked</u> so that the CPU can be used by another process/thread

Blocking Semaphores 1/2

- Blocked processes are moved to a wait queue for a given semaphore
  - Blocked processes are moved to the ready queue by a <u>wakeup</u> operation, when a process executes a <u>signal</u> operation on the corresponding semaphore

Blocking Semaphores 2/2

- wait(s):

```
s--;
if (s < 0) {
        add calling process to wait_queue(s);
        block process;
}
```

- signal(s):

```
s++;
if (s <= 0) {
        remove calling process from wait_queue(s);
        wakeup process;
}
```

Semaphore Semantics

- If a semaphore S is negative…
  - Its magnitude indicates the # of processes waiting on S
- If a semaphore S is positive…
  - Its value indicates the # of instances of the resource guarded by S that can be acquired concurrently
- If S has just two values it is a <u>binary</u> semaphore, else it is a <u>counting</u> semaphore

Deadlocks and Starvation 1/3

- Consider the following:
- P_0:

  wait(s);

  wait(q);

  …

  signal(s);

  signal(q);

- P_1:

  wait(q);

  wait(s);

  …

  signal(q);

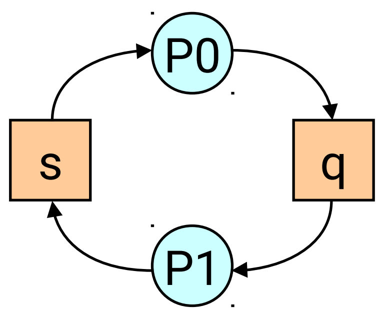
  signal(s);

Can end up with a cycle of waiting processes ⇒ deadlock!

Deadlocks and Starvation 2/3

- Let:
- Pi → R ⇒ process Pi is waiting for a resource guarded by semaphore R
- R → Pi ⇒ a resource guarded by semaphore R is allocated to Pi

- A deadlock occurs when there is a cycle as follows:



Deadlocks and Starvation 3/3

- Related to deadlocks is the notion of starvation
  - Starvation occurs when a process cannot get a resource (e.g., CPU, semaphore) and waits indefinitely while other processes may make progress
    - e.g., a LIFO-ordered semaphore wait queue may cause starvation

Counting Semaphores

- Given two binary semaphores, s1 and s2, we can implement a counting semaphore, S, as follows:

```
binary_semaphore s1=1, s2=0;
int c; // set to initial value for s
```

- wait(S):

```
wait(s1);
c--;
if (c < 0) {signal(s1); wait(s2);}
signal(s1);
```

- signal(S):

```
wait(s1);
c++;
if (c <= 0) signal(s2);
else signal(s1);
```

Classic Problems

- Readers-writers problem:
  - Multiple concurrent processes wish to access a shared data object
    - Some only wish to read – <u>readers</u>
    - Some wish to read & write – <u>writers</u>
- One approach to problem:
  - Allow read access by multiple readers
  - No reader kept waiting unless a writer has already acquired exclusive access to the shared object

  - <u>NOTE</u>: writers may starve if object is always being accessed by readers

Readers-writers Problem

semaphore: mutex, wrt; // initially, both 1

int readcount=0;

- writer

```
wait(wrt);
update_object;
signal(wrt);
```

- reader

```
wait(mutex);
readcount++;
if (readcount==1)
    wait(wrt);
signal(mutex);
read_object;
wait(mutex);
readcount--;
if (readcount==0)
    signal(wrt);
signal(mutex);
```

Monitors

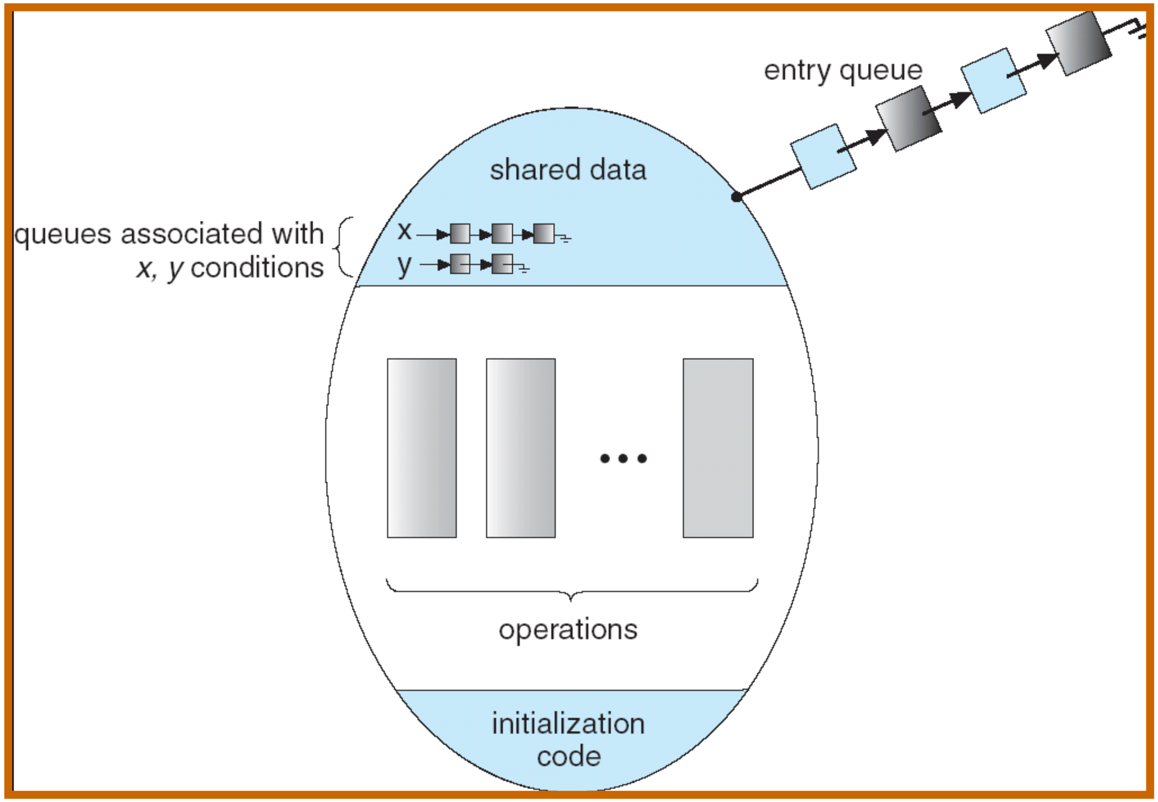
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```
monitor monitor-name {
   // shared variable declarations
   procedure P1 (…) { …. }
     …

   procedure Pn (…) {……}

   Initialization code ( ….) { … }
     …
   }
}
```

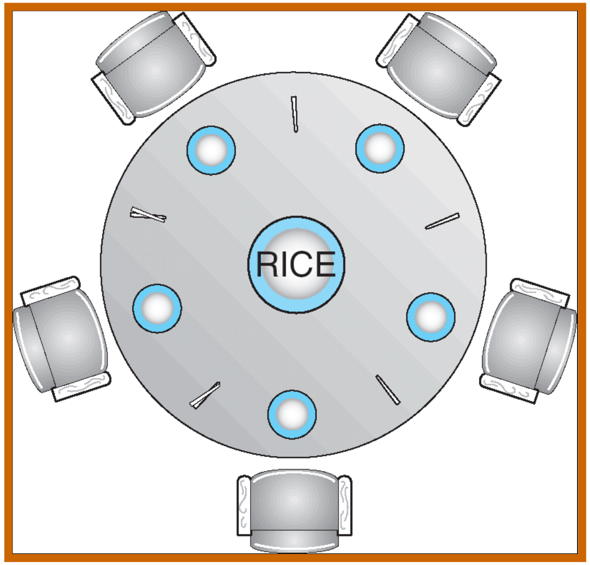
Monitors (continued)

- Procedures defined within a monitor can only access those variables declared locally within the monitor, as well as procedure arguments
- Processes waiting to enter the monitor are queued while one process is active
- Monitors support condition variables
  - e.g., condition_t x,y;
  - Operations:
    - x.wait(); // process is suspended until another process invokes x.signal();
    - x.signal(); // resume one suspended process (if any exist) that previously invoked x.wait()

Monitor with Condition Variables



Dining-Philosophers Problem 1/2



- 5 philosophers, 5 chopsticks and a bowl of rice (data set)

Dining-Philosophers Problem 2/2

- Each philosopher may be in one of 3 possible states
  - thinking, hungry, eating
- A hungry philosopher tries to acquire one chopstick at a time on his/her left- and right-hand sides
- Only if both chopsticks closest to a philosopher are not being used by another philosopher can they both be picked up
  - At such a time, a philosopher may eat
  - After eating, both chopsticks are placed on the table and the philosopher returns to thinking

Dining-Philosophers Solution 1/3

```
monitor DP {
    enum {THINKING, HUNGRY, EATING} state[5];
    condition_t self[5]; //for hungry philosophers waiting to eat

    void pickup(int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self[i].wait;
    }

    void putdown(int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }
```

Dining-Philosophers Solution 2/3

```
void test(int i) {
      if ( (state[(i + 4) % 5] != EATING) &&
         (state[i] == HUNGRY) &&
         (state[(i + 1) % 5] != EATING) ) {
            state[i] = EATING;
            self[i].signal();
      }
}

initialization_code() {
     for (int i = 0; i < 5; i++)
         state[i] = THINKING;
     }
} // end of monitor
```

Dining-Philosophers Solution 3/3

- Each philosopher *I* invokes the operations pickup() and putdown() in the following sequence:

  dp.pickup(i)

  EAT

  dp.putdown(i)

- This solution guarantees no two neighbors are eating simultaneously and no deadlocks will occur…but starvation is still possible

Monitor Implementation Using Semaphores

- For each monitor introduce the following variables

```
semaphore_t mutex;  // (initially  = 1)
semaphore_t next;     // (initially  = 0)
int next-count = 0; // # processes waiting to go next
            in monitor
```

- Each procedure P in monitor will be replaced by

```
wait(mutex);
      P(); // the actual procedure P
if (next-count > 0)
       signal(next)
else
     signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Monitor Implementation (cont.)

- For each condition variable ***x***, we have:

```
semaphore_t x-sem; // (initially  = 0)
int x-count = 0;
```

- The operation x.wait can be implemented as:

```
x-count++;
if (next-count > 0)
      signal(next);    // allow a suspended process to
         // resume
else
      signal(mutex); // allow a process to enter
        // monitor
wait(x-sem);
x-count--;
```

Monitor Implementation (cont.)

- The operation x.signal can be implemented as:

```
if (x-count > 0) {
    next-count++;
    signal(x-sem);
    wait(next);
    next-count--;
}
```

Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
  - An event acts much like a condition variable

Linux Synchronization

- Linux:
  - disables interrupts to implement short critical sections
- Linux provides:
  - semaphores
  - spin locks

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables

- Non-portable extensions include:
  - read-write locks
  - spin locks

Case Study: Synchronization

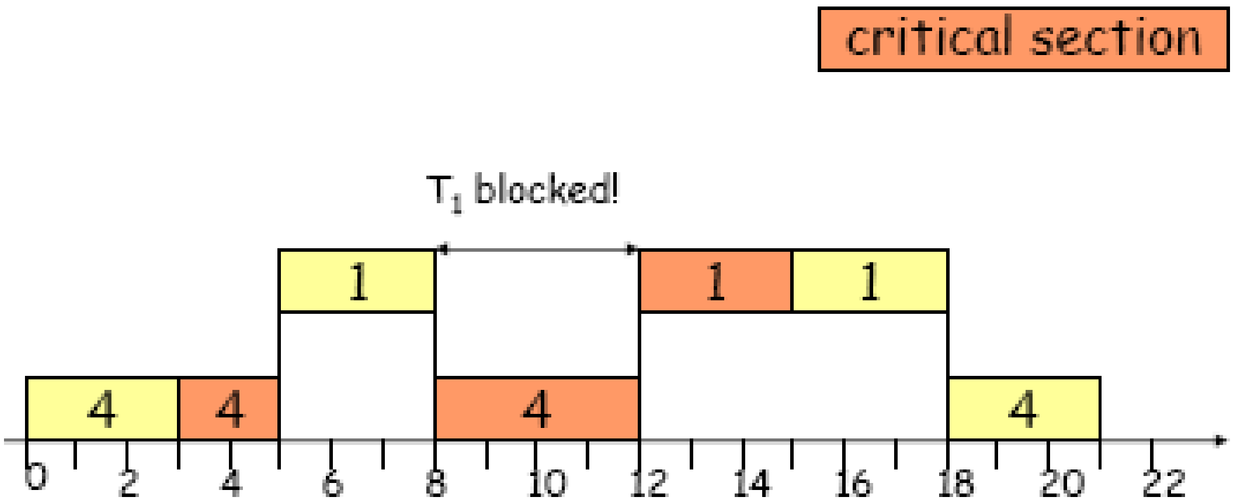
(CPU Protection Problem)

Mutual Exclusion via Semaphores

- Ensure only one task/process is in the critical section
  - wait(S) to get access to semaphore S
  - signal(S) to release S
  - Example: producer consumer

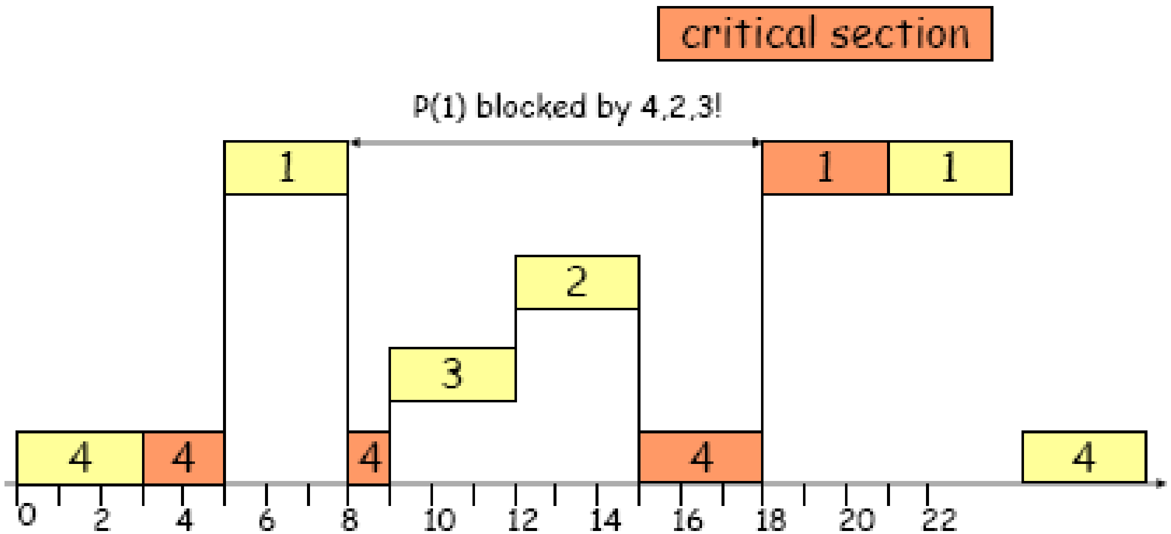
```
Producer()
{
        .
        .
        wait(S)
        critical section /* create data & increment pointer */
        signal(S)
        .
        .
}
```

Priority Inversion



Source of the figure: Chenyang Lu, Washington University Saint Louis

Unbounded Priority Inversion

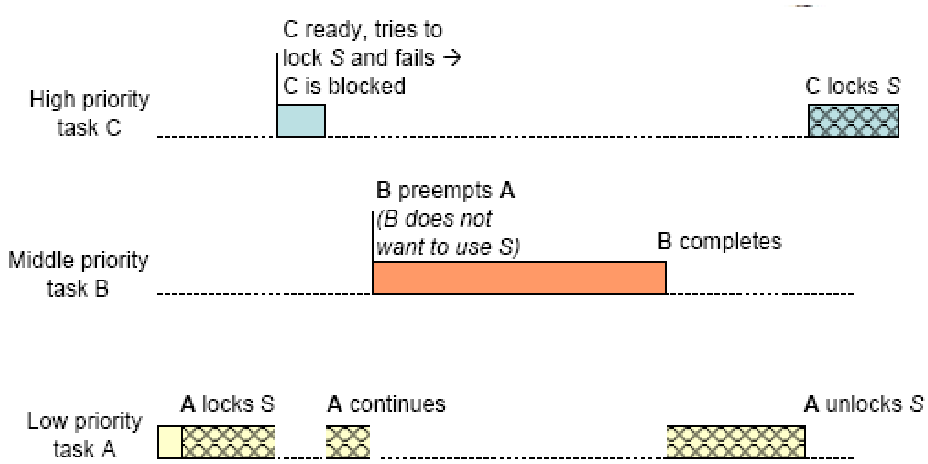


What really happened on Mars?

- Repeated resets in the Mars Pathfinder
  - The Mars Pathfinder mission was widely proclaimed as "flawless" in the early days after its July 4th, 1997 landing on the Martian surface.... But a few days into the mission, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system resets, each resulting in losses of data. The press reported these failures in terms such as "software glitches" and "the computer was trying to do too many things at once"....
  - For a full story, visit http://research.microsoft.com/%7Embj/Mars_Pathfinder/Mars_Pathfinder.html

Pathfinder Incident

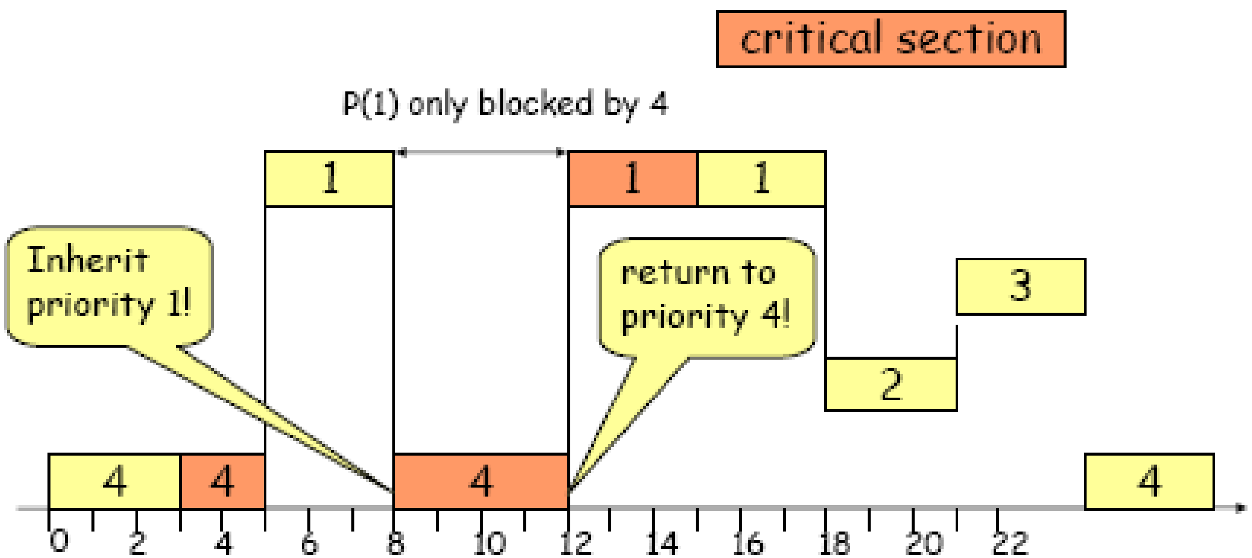
- Classical priority inversion problem due to shared system bus!



Source of the figure: Damir Isovic, Malardaren University, Sweden

Priority Inheritance

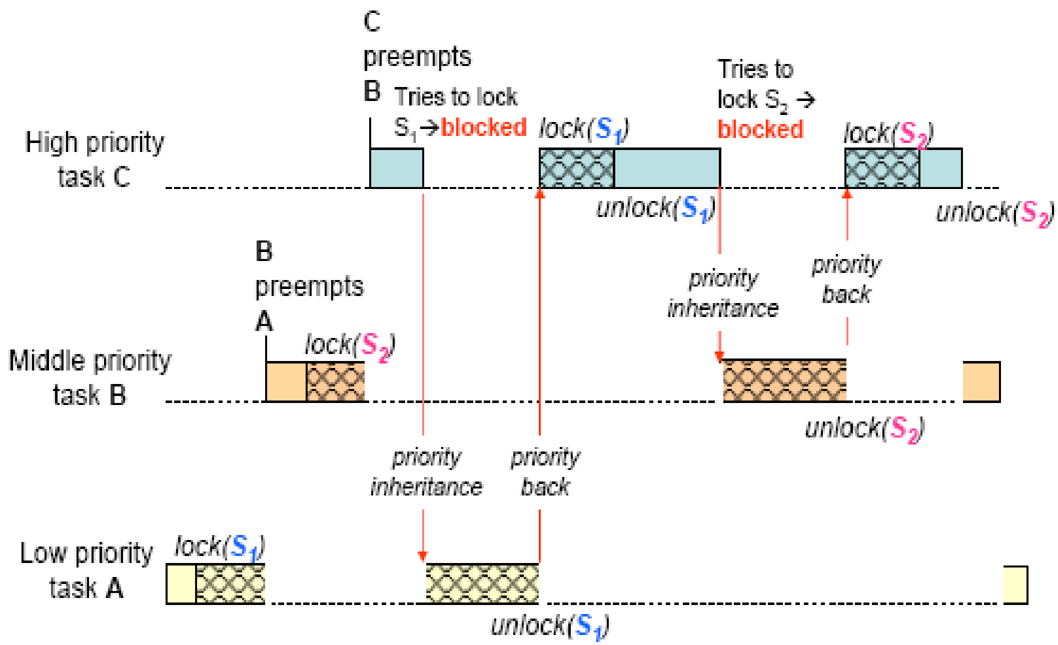
- Inherit the priority of the blocked high priority task



Priority Inheritance Protocol (PIP)

- If T_L blocks a higher proirity task T_H, priority(T_L) $\leftarrow$ priority(T_H)
- When T_L releases a semaphore:
  - Return to its normal priority if it doesn't block any task
  - Otherwise, set priority(T_L) $\leftarrow$ highest priority of the tasks blocking on a semaphore held by T_L
- Transitive
  - T_1 blocked by T_2: priority(T_2) $\leftarrow$ priority(T_1)
  - T_2 blocked by T_3: priority(T_3) $\leftarrow$ priority(T_1)

Chained Blocking: Problem of PIP



In the worst case, the highest priority task T_1 can be blocked by N lower priority tasks in the system when T_1 has to access N semaphores to finish the execution!

Source of the figure: Damir Isovic, Malardaren University, Sweden

Priority Ceiling Protocol (PCP)

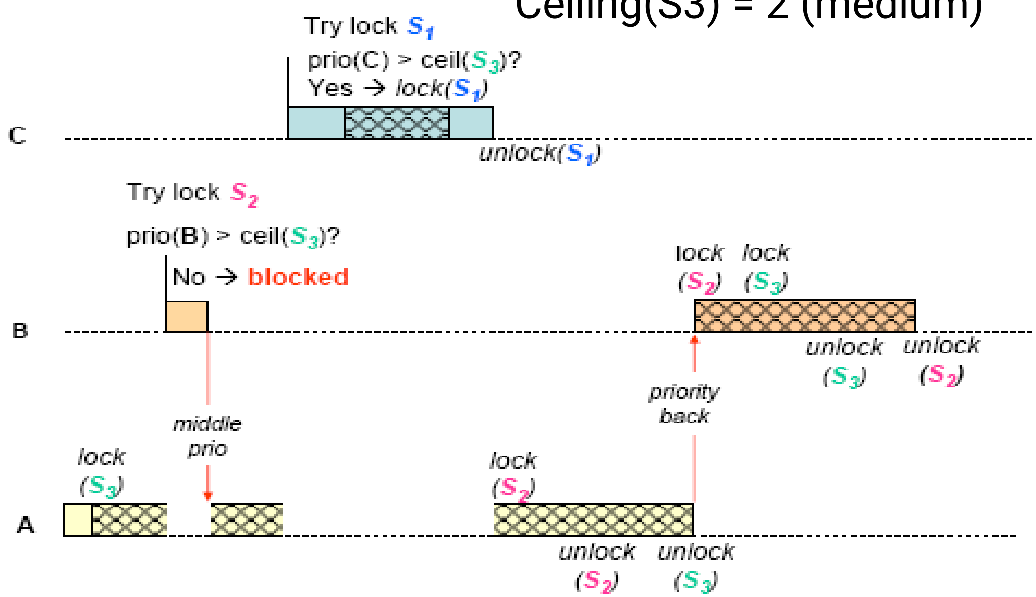
- Avoid chained blocking
  - Guarantee a task is blocked by at most one lower priority task
  - No deadlock
- Assumptions
  - Each task is associated with a fixed priority

PCP

- Each semaphore has a fixed priority ceiling
  - ceiling(S) = highest priority among all the tasks that will request S
- How it works:
  - T_i can access a semaphore S if
    - S is not already allocated to any other task; and
    - Priority of T_i is higher than the current processor ceiling = max(priority ceilings of all the semaphores allocated to tasks other than T_i)

PCP

Ceiling(S1) = 1 (high)
Ceiling(S2) = 2 (medium)
Ceiling(S3) = 2 (medium)



No chained blocking!

Source of the figure: Damir Isovic, Malardaren University, Sweden