Synchronization

Today

- Race condition & critical regions
- Mutual exclusion with busy waiting
- Sleep and wakeup

Next time

- Semaphores and Monitors

Cooperating processes

- Cooperating processes need to communicate
  - They can affect/be affected by others
- Issues
  - 1. How to pass information to another process?
  - 2. How to avoid getting in each other's ways?
    - Two processes trying to get the last seat on a plane
  - 3. How to ensure proper sequencing when there are dependencies?
    - Process A produces data, while B prints it – B must wait for A before starting to print
- How about threads?
  - 1. Easy
  - 2 & 3. Pretty much the same

Accessing shared resources

- Many times cooperating process share memory
- A common example – print spooler
  - A process wants to print a file, enter file name in a special spooler directory
  - Printer daemon, another process, periodically checks the directory, prints whatever file is there and removes the name

A: $next\_slot_A \leftarrow in$ % 7

A: $spooler\_dir[next\_slot_A] \leftarrow file\_name_A$

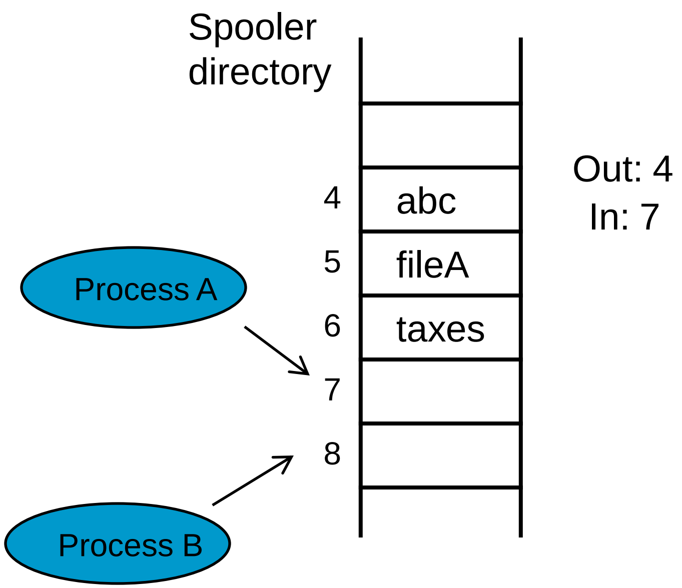
A: $in \leftarrow next\_slot_A + 1$ % 8

Switch -

B: $next\_slot_B \leftarrow in$ % 8

B: $spooler\_dir[next\_slot_B] \leftarrow file\_name_B$

B: $in \leftarrow next\_slot_B + 1$ % 9



Interleaved schedules

- Assumption – preemptive scheduling
- Problem – the execution of the two threads/processes can be interleaved
  - Some times the result of interleaving is OK, others not!

A: $next\_slot_A \leftarrow in$ % 7

Switch

B: $next\_slot_B \leftarrow in$ % 7

B: $spooler\_dir[next\_slot_B] \leftarrow file\_name_B$

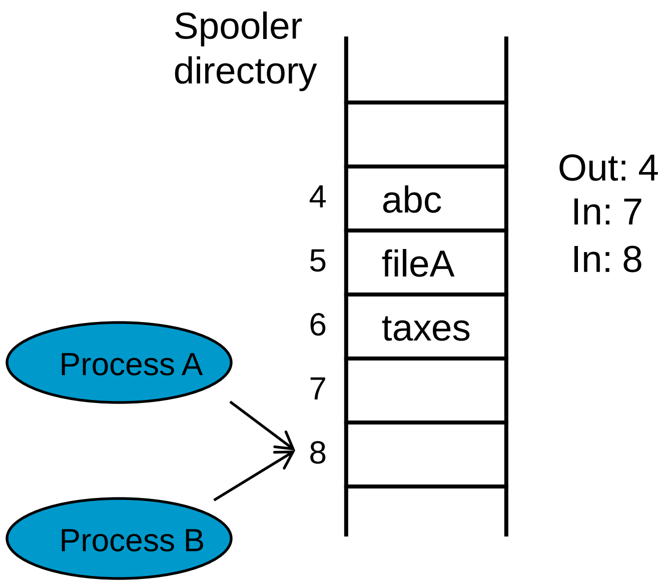
Switch

B: $in \leftarrow next\_slot_B + 1$ % 8

Switch

A: $spooler\_dir[next\_slot_A] \leftarrow file\_name_A$

A: $in \leftarrow next\_slot_A + 1$ % 8



Race conditions and critical regions

- *Race condition*
  – Two or more threads/processes access (r/w) shared data
  – Final results depends on order of execution
- We need mechanisms to prevent race conditions, synchronizing access to shared resources
- Code where race condition is possible – *critical region*
- We need a way to ensure that *if a process is using a shared item (e.g. a variable), other processes will be excluded from doing it*
  – *i.e. only one thread at a time in the critical region (CR)*

Mutual exclusion

Requirements for a solution

- No two processes simultaneously in CR
  - Mutual exclusion, at most one thread in
- No assumptions on speeds or numbers of CPUs
- No process outside its CR can block another one
  - Ensure progress; a thread outside the CR cannot prevent another one from entering
- No process should wait forever to enter its CR
  - Bounded waiting or no starvation
  - Threads waiting to enter a CR should *eventually* be allow to enter

Mutual exclusion with busy waiting

- Lock variable
  - Lock initially 0
  - Process checks lock when entering CR
  - *Problem? Same as before!*
- Disabling interrupts
  - Simplest solution – process disables all interrupts when entering the CR and re-enables them at exit
  - No interrupts → no clock interrupts → no other process getting in your way
  - *Problems?*
    - Users in control – grabs the CPU and never comes back
    - Multiprocessors?
  - Use in the kernel – still multicore means we need something more sophisticated

Strict alternation

- Taking turns
  - `turn` keeps track of whose turn it is to enter the CR

Process 0

```
while(TRUE) {
  while(turn != 0);
  critical_region0();
  turn = 1;
  noncritical_region0();
}
```

Process 1

```
while(TRUE) {
  while(turn != 1);
  critical_region1();
  turn = 0;
  noncritical_region1();
}
```

- Continuously testing a variable for a given value is called *busy waiting*; a lock that uses this is a *spin lock*
- Problems?
  - What if process 0 sets turn to 1, but it gets around to just before its critical region before process 1 even tries?
  - Violates conditions 3

Peterson's solution

Combining locks and turns …

```
#define FALSE 0
#define TRUE 1
#define N 2 /* num. of processes */

int turn;
int interested[N];

void enter_region(int process)
{
  int other;

  other = 1 - process;
  interested[process] = TRUE;
  turn = process;
  while (turn == process &&
         interested[other] == TRUE);
}

void leave_region(int process)
{
  interested[process] = FALSE;
}
```

Template of a process' access to the critical region (process 0):

```
…
enter_region(0);
<CR>
leave_region(0);
…
```

Tracing Peterson’s

Process 0	**Common variables**	**Process 1**
enter_region(0) other = 1 interested[0] = T turn = 0 **(Process 0 in)**	interested[0] = F interested[1] = F, turn = ?	
	interested[0] = T, interested[1] = F, turn = 0	--- --- --- --- ---
--- ---		
		--- --- ---

```
void enter_region(int process)
{
  int other;
  other = 1 – process;
  interested[process] = TRUE;
  turn = process;
  while (turn == process &&
         interested[other] == TRUE);
}
```

Tracing Peterson's

Process 0	Common variables	Process 1
enter_region(0) other = 1	interested[0] = F interested[1] = F, turn = ?	
	interested[0] = F interested[1] = T, turn = ?	enter_region(1) other = 0 interested[1] = T
interested[0] = T turn = 0	interested[0] = T interested[1] = T, turn = 0	
	interested[0] = T Interested[1] = T, turn = 1	turn = 1 <Busy Wait>
turn != 0 *<CR>* leave_region(0) interested[0] = F	interested[0] = F, interested[1] = T, turn = 1	
	interested[0] = F, Interested[1] = F, turn = 1	<CR>

```
void enter_region(int process)
{
  int other;
  other = 1 – process;
  interested[process] = TRUE;
  turn = process;
  while (turn == process &&
        interested[other] == TRUE);
}
```

TSL(test&set) -based solution

- With a little help from hardware – TSL instruction
- Atomically test & modify the content of a word

```
TSL REG, LOCK
```

- REG ← LOCK >> Read the content of variable LOCK into register REG
- LOCK ← non-zero value >> Set lock to a non-zero value

- Entering and leaving CR

```
enter_region:
   TSL REGISTER, LOCK
   CMP REGISTER, #0
   JNE enter_region | non zero, lock set
   RET | return to caller, you're in
```

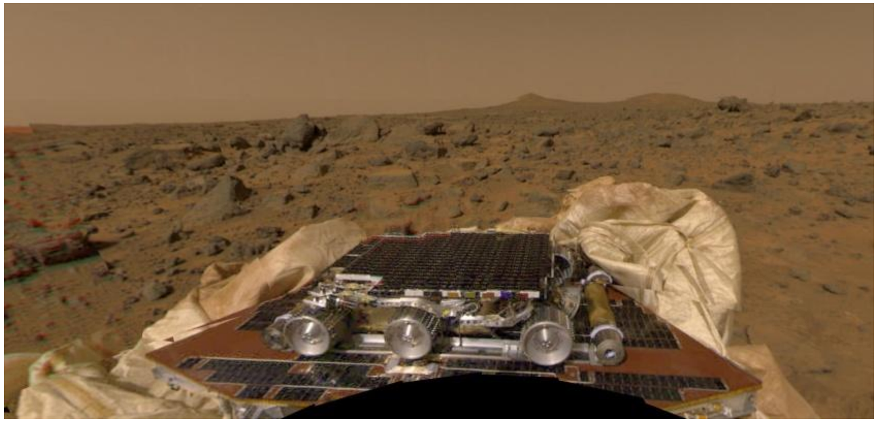
Busy waiting

```
leave_region:
   MOVE LOCK, #0
   RET
```

Busy waiting and priority inversion

- Problems with TSL-based approach?
  - Waste CPU by busy waiting
  - Can lead to *priority inversion*
    - Two processes, H (high-priority) & L (low-priority)
    - L gets into its CR
    - H is ready to run and starts busy waiting
    - L is never scheduled while H is running …
    - *So L never leaves its critical region and H loops forever!*

Welcome to Mars!

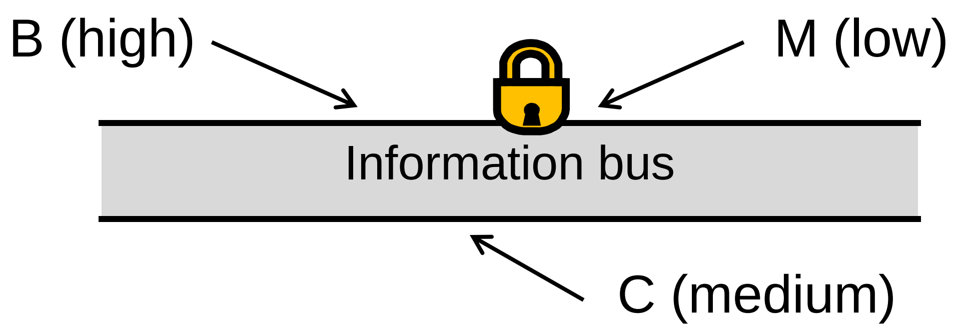


Problems in the Mars Pathfinder*

- Mars Pathfinder
  - Launched Dec. 4, 1996, landed July 4th, 1997
- Periodically the system reset itself, loosing data
- VxWork provides preemptive priority scheduling
- Pathfinder software architecture
  - An information bus with access controlled by a lock
  - A bus management (B) high-priority thread
  - A meteorological (M) low-priority, short-running thread
    - If B thread was scheduled while the M thread was holding the lock, the B thread busy waited on the lock
  - A communication (C) thread running with medium priority

*As explained by D. Wilner, CTO of Wind River Systems, and narrated by Mike Jones

Problems in the Mars Pathfinder*



- Sometimes,
  - **B was waiting on M and**
  - **C was scheduled**
- After a bit of waiting, a watchdog timer would reset the system ☺
- How would you fix it?
  - Priority inheritance – the M thread inherits the priority of the B thread blocked on it
  - Actually supported by VxWork but dissabled!

Sleep & wakeup

- Avoid busy waiting – rather than sit in a tight loop, go to sleep
- An alternative solution
  - Sleep – causes the caller to block, i.e. be suspended until another process wakes it up
  - Wakeup – process passed as parameter is awakened

Producer-Consumer problem

- Also known as *bounded buffer*
  - Two processes & one shared, fixed-size buffer
  - One puts information into the buffer, the other one takes it out

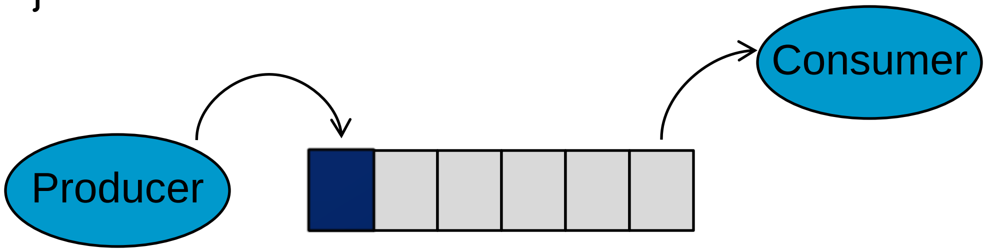
Producer

```
while (TRUE){
    item = produce_item();
    while (count == N);
    insert_item(item);
    ++count;
    if (count == 1)
        wakeup(consumer)
}
```

Consumer

```
while (TRUE){
    while(count == 0);
    item = remove_item();
    --count;
    if (count == (N -1))
        wakeup(producer);
    consume_item(item);
}
```

What if the consumer wants to consume from an empty buffer?



Producer-Consumer problem

- "Simple solution"
  - Producer/consumer goes to sleep if buffer is full/empty

Producer

```
while (TRUE){
   item = produce_item();
   if (count == N) sleep();
   insert_item(item);
   ++count;
   if (count == 1)
      wakeup(consumer)
}
```

Consumer

```
while (TRUE){
   if (count == 0) sleep();
   item = remove_item();
   --count;
   if (count == (N -1))
      wakeup(producer);
}
```

Consumer is not yet logically sleep - producer's signal is lost!

Consumer reads count = 0
scheduler blocks consumer and runs producer
Producer inserts an item, ++count and signals consumer
But consumer is not yet sleep, so signal is lost!
Consumer wakes up, sees count = 0 and goes to sleep
… for ever

- *A piggy bank of waiting bits?*

Coming up ...

- Several mechanisms for synchronization
- Locks are the lowest and require
  - Disabling interrupts or
  - Busy waiting
- Some other alternatives
  - Semaphores – slightly higher abstractions
  - Monitors – much better but requiring language support