Synchronization

Today

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

Next time

- Semaphores and Monitors

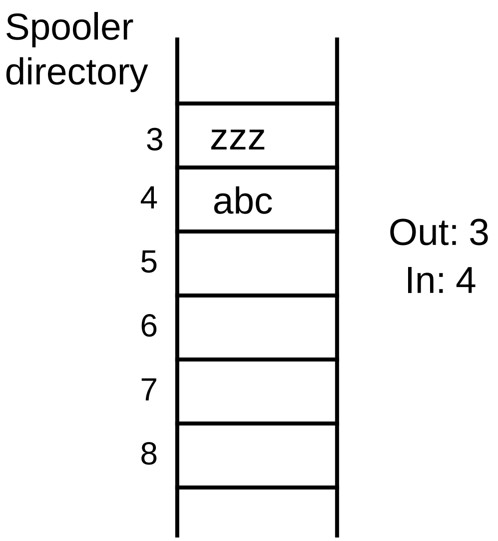
Cooperating processes

- 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

```
next_slot:= in;                          // in = 4
spooler_dir[next_slot] := file_name; // insert "abc"
in++;
```



Accessing shared resources

- Assumption – preemptive scheduling
- Two processes, A & B, trying to print

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

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

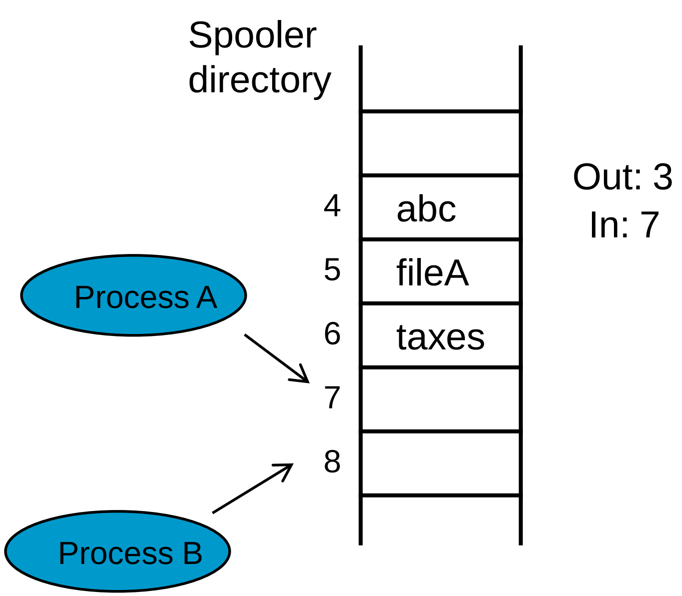
A: $in \leftarrow next\_slot_A + 1$ *% 8*

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

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

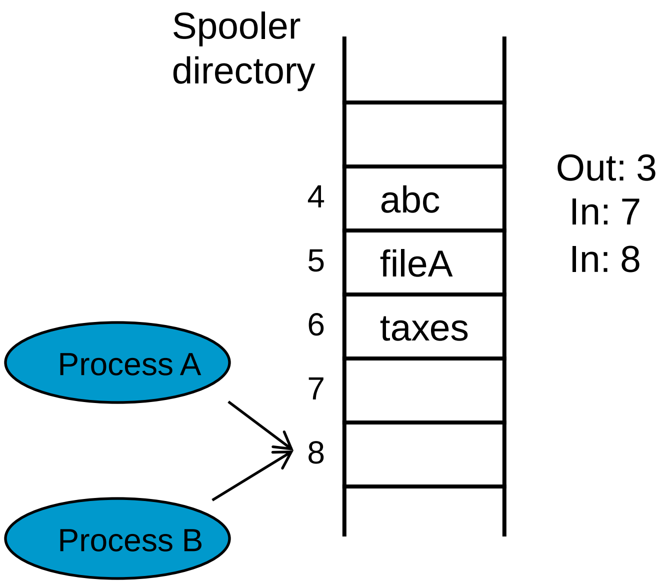
Switch

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

Switch

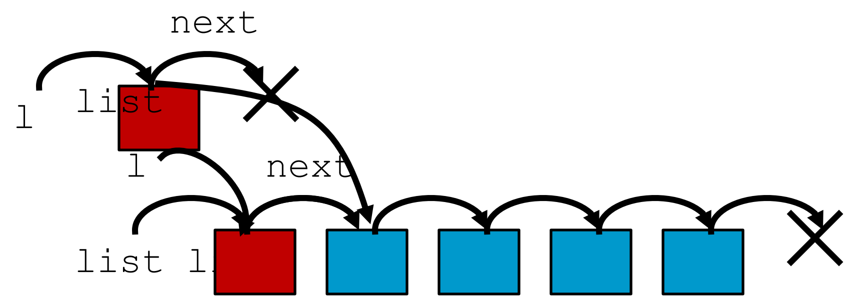
A: $spooler\_dir[next\_slot_A] \leftarrow file\_name_A$

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



Interleaved schedule – another example

```
 1 struct list {
 2   int data;
 3   struct list *next;
 4 };
 5
 6 struct list *list = 0;
 7
 8 void
 9 insert(int data)
10 {
11 struct list *l;
12
13   l = malloc(sizeof *l);
14   l->data = data;
15   l->next = list;
16   list = l;
17 }
```



Two processes, what would happen if one executing line 15 before the other executes 16?

Race conditions and critical regions

- Problem – the process operating on the data assumes certain conditions (invariants) hold
  - For the linked list – `list` points to the head of the list and each element's `next` point to the next element
  - Insert temporarily violates this, but fixes it before finishing
  - *True for a single process, not for two concurrent ones*
- *Race condition*
  - Two or more threads/processes access (r/w) shared data
  - Final results depends on order of execution
- Code where race condition is possible – *critical region*

Race conditions and critical regions

- We need mechanisms to prevent race conditions, synchronizing access to shared resources
  - Some tools try to detect them – `helgrind`
- We need a way to ensure the invariant conditions hold when the process is going to manipulate the share item, i.e. …
- … to ensure that *if a process is using a shared item, 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

How about …?

- Lock variable
  - Lock initially 0
  - Process checks lock when entering CR
  - *Problem? Same as before!*
    - *Both can concurrently test 17, see it unlocked, and grab it; now both are in the CR*

```
 1 void
 2 insert(int data)
 3 {
 4   struct list *l;
 5
 6   acquire(lock);
 7   l = malloc(sizeof *l);
 8   l->data = data;
 9   l->next = list;
10   list = l;
11 }
12
```

```
13 void
14 acquire(lock *lk)
15 {
16    for(;;) {
17       if(!lk->locked) {
18          lk->locked = 1;
19          break;
20       }
21
22    }
23 }
```

How about …?

- 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();
}
```

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

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);
…
```

Peterson's solution

- You can show all conditions hold
  - Mutual exclusion
    - P_1 can only entered if P_2 is not interested or turn is 1
    - If both processes are in their CR, then both have to be interested
    - But they could not have both exited their while statement since turn is either 0 or 1
    - Whoever did not go in, say P_1, have to wait for the other, P_2, to set its interested to false

```
interested[process] = TRUE;
turn = other;
while (interested[other] == TRUE &&
       turn == other);
<CR>
interested[process] = FALSE;
```

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
```

- Continuously testing a variable for a given value is called *busy waiting*; a lock that uses this is a *spin lock*

Synchronization in xv6

- Xv6 uses locks, represented as `struct spinlock`

```
// Mutual exclusion lock.
struct spinlock {
   uint locked;       // Is the lock held?

   // For debugging:
   char *name;        // Name of lock.
   struct cpu *cpu;   // The cpu holding the lock.
   uint pcs[10];      // The call stack (an array of program counters)
                      // that locked the lock.
};


void
acquire(struct spinlock *lk)
{
   pushcli();
   if(holding(lk))
      panic("acquire");

   // The xchg is atomic.
   while(xchg(&lk->locked, 1) != 0)
       ;

   // Record info about lock acquisition for debugging.
   lk->cpu = cpu;
   getcallerpcs(&lk, lk->pcs);
}
```

Synchronization in xv6

```
void
release(struct spinlock *lk)
{
   if(!holding(lk))
      panic("release");
   lk->pcs[0] = 0;
   lk->cpu = 0;

   xchg(&lk->locked, 0);

   popcli();
}
```

What is the difference between `TSL` and `xchg`? Can you implement one with the other?

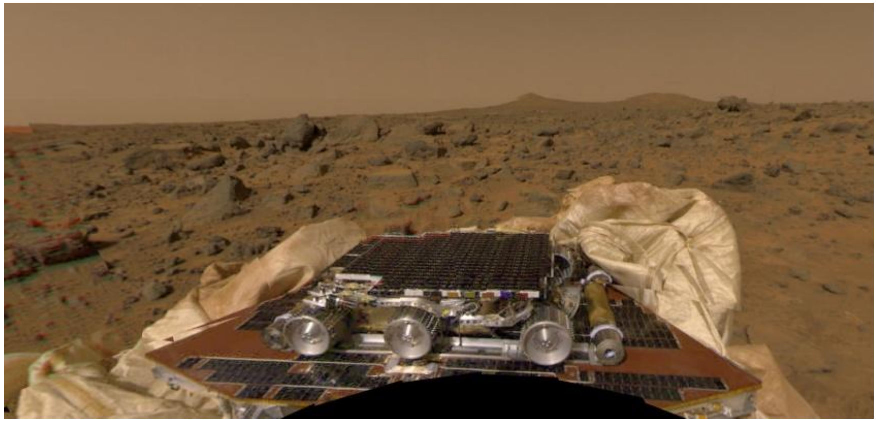
```
static inline uint
xchg(volatile uint *addr, uint newval)
{
   uint result;

   // The + in "+m" denotes a read-modify-write operand.
   asm volatile("lock; xchgl %0, %1" :
                "+m" (*addr), "=a" (result) :
                "1" (newval) :
                "cc");
   return result;
}
```

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!

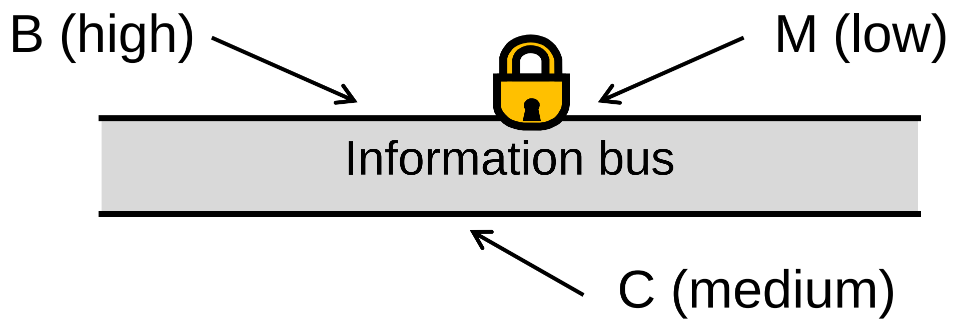


Problems in the Mars Pathfinder*

- Mars Pathfinder
  - Launched Dec. 4, 1996, landed July 4^{th}, 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

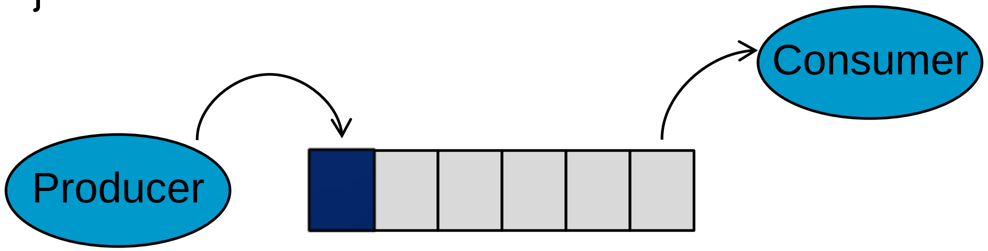
- Also known as *bounded buffer*
  - Two processes & one shared, fixed-size buffer
  - One puts information into the buffer, the other one takes it out

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);
}
```



Producer-Consumer problem

- "Simple solution"
  - If buffer is empty, producer goes to sleep to be awaken when the consumer has removed one or more items
  - Similarly for the consumer

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);
   consume_item(item);
}
```

- Of course, we can still have a race condition!

Producer-Consumer problem

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);
   consume_item(item);
}
```

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

- Possible sequence
  - Consumer reads count = 0; scheduler blocks it, runs producer
  - Producer inserts 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? How many?*

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