Synchronization I

Today

- Race condition, critical regions and locks

Next time

- Condition variables, semaphores and Monitors

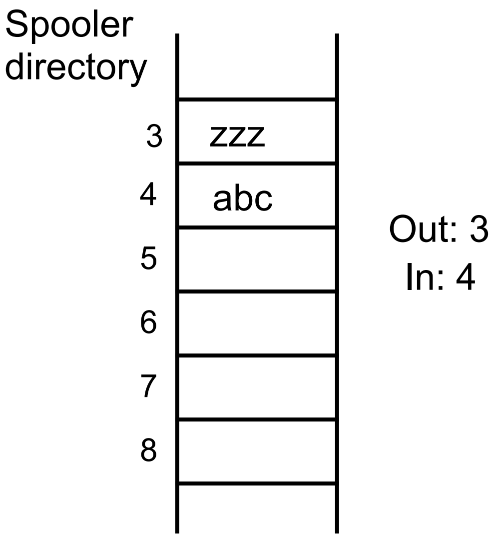
Cooperating processes

- Cooperating processes need to communicate
  - They can affect/be affected by others
- Issues with Inter-Process Communication (IPC)
  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 that B prints – B must wait for A to print
- How about threads?
  - 1. Easy
  - 2 & 3. Pretty much the same

Accessing shared resources

- Many times cooperating threads share memory
- A common example – print spooler
  - A thread wants to print a file, enter file name in a special spooler directory
  - Printer daemon, another thread, 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 threads, A & B, trying to print

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

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

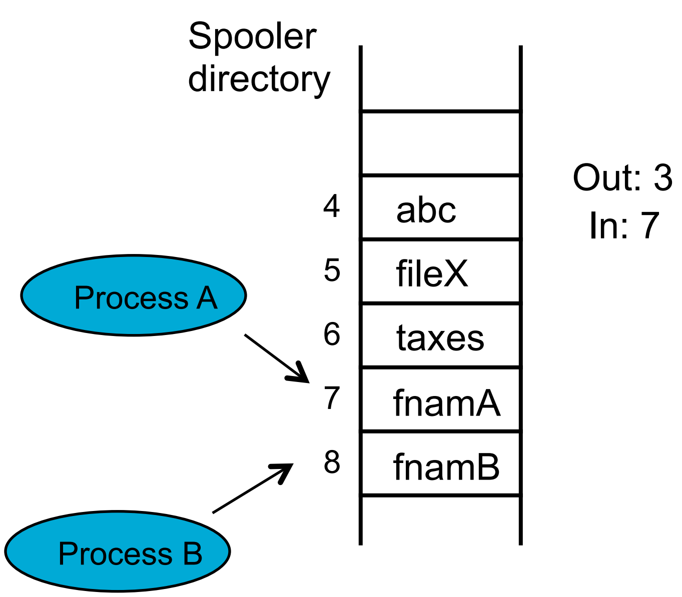
A: $in \leftarrow next\_slot_A + 1$ % 8

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, other times 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

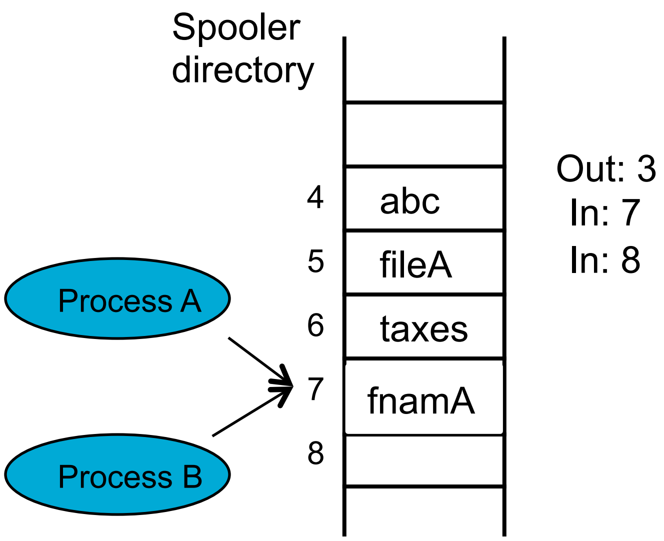
B: $in \leftarrow next\_slot_B + 1$ % 8

Switch

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

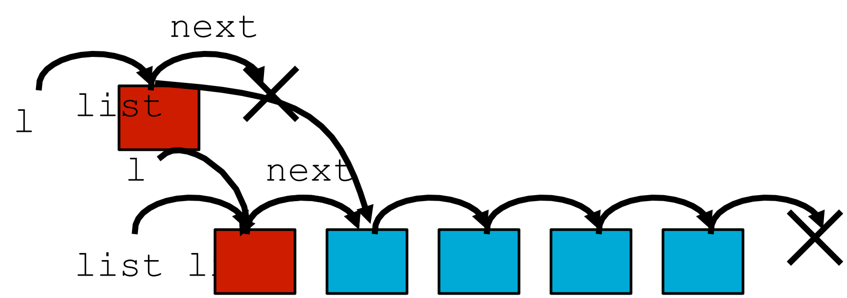
A: $in \leftarrow next\_slot_A + 1$ % 8

B's printout?!



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 threads A and B, what would happen if B executes line 15 before A executes 16?

Race conditions and critical regions

- Problem – the threads 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 thread, 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 thread is using a shared item, others 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

1. No two threads simultaneously in CR
   - Mutual exclusion, at most one thread in
2. No assumptions on speeds or numbers of CPUs
3. No thread outside its CR can block another one
   - Ensure progress; a thread outside the CR cannot prevent another one from entering
4. No thread 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 taking turns?

- Strict alternation
  - `turn` keeps track of whose turn it is to enter the CR

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

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

- Problems?
  - What if thread 0 sets turn to 1, but it gets around to just before its critical region before process 1 even tries?
    - Turn is 1 and both process are in their noncritical region
  - Violates conditions 3

Locks

- Using locks
  - It’s a variable so you have to declare it
  - Threads check lock when entering CR, and free it after
    - Lock is either available (free) or acquired
    - Can hold other information such as which thread holds the lock or a queue of lock requests
  - `lock()`: if available, go on; else don’t return until you have it
  - `unlock()`: if threads are waiting, they will (eventually) find out (or be told)

```
lock_t mutex;

void
insert(int data)
{
  struct list *l;

  lock(&mutex);
  l = malloc(sizeof *l);
  l->data = data;
  l->next = list;
  list = l;
  unlock(&mutex);
}
```

Pthreads locks

- In the POSIX library, a lock is called a **mutex**

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void
insert(int data)
{
  struct list *l;

  Pthread_mutex_lock(&lock); /* a wrapper that checks for errors */
  l = malloc(sizeof *l);
  ...
  Pthread_mutex_unlock(&lock);
}
```

- Note the call passes a variable to lock/unlock
  - To enable fine-grain locking (rather than a single coarse lock)
- Locks must be initialized before used, either this way or dynamically with

```
pthread_mutex_init(&lock, NULL)
```

Implementing locks

- Here it's a simple implementation of `lock()`
- *Are we done?*

```
1 void
2 lock(lock *lk)
3 {
4     while(lk->locked == 1)
5         ; /* spin-wait */
6     lk->locked = 1; /* now set it */
7 }
```

Context switch here and there we go again

- No yet!
  – Correctness problem: Both can concurrently test 4, see it unlocked, and grab it; now both are in the CR
  – Continuously testing a variable for a given value is called *busy waiting*; a lock that uses this is a *spin lock* – spin waiting is wasteful

Implementing locks

- Disabling interrupts
  - Simplest solution – threads disables all interrupts when entering the CR and re-enables them at exit

```
void lock() {
   DisableInterrupts();
}
```

```
void unlock() {
   EnableInterrupts();
}
```

  - No interrupts → no clock interrupts → no other process getting in your way
- Obvious problems
  - Users in control – grabs the CPU and never comes back
  - What about multiprocessors?
  - And yes, it's also inefficient
- Use in the kernel – still multi-core means we need something more sophisticated

TSL(test&set) -based solution

- Atomically test & modify the content of a word – TSL
  - The CPU executing the TSL locks the memory bus to stop other CPUs from accessing memory until it is done
  - In SPARC is `ldstub` (load & store), in x86 is `xchg`

```
int TSL(int *ptr, int new) {
    int old = *ptr;  /* fetch old value at ptr */
    *ptr = new; /*store new value into ptr */
    return old;
}
```

Done atomically

TSL(test&set) -based solution

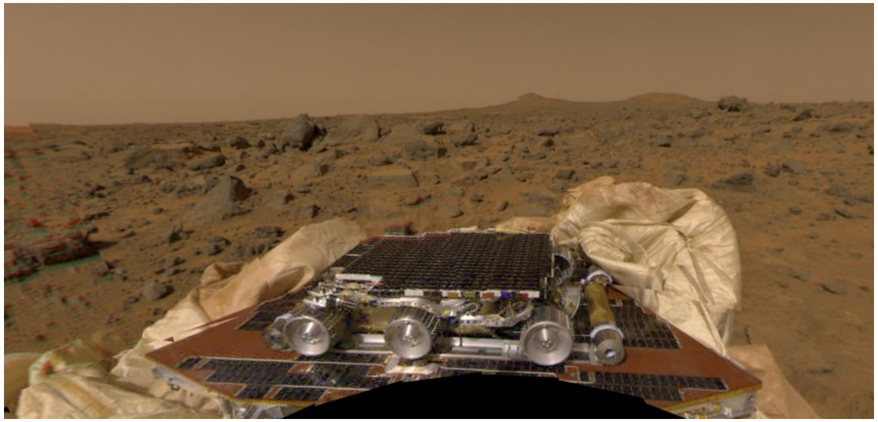
- Entering and leaving CR

```
typedef struct __lock_t {
  int flag;
} lock_t;

 void init(lock_t *loc) {
   lock->flag = 0;
 }

 void lock(lock_t *lock) {
    while (TSL(&lock->flag, 1) == 1)
        ; /* spin-wait */
 }

 void unlock(lock_t *lock) {
    lock->flag = 0;
 }
```

Busy waiting and priority inversion

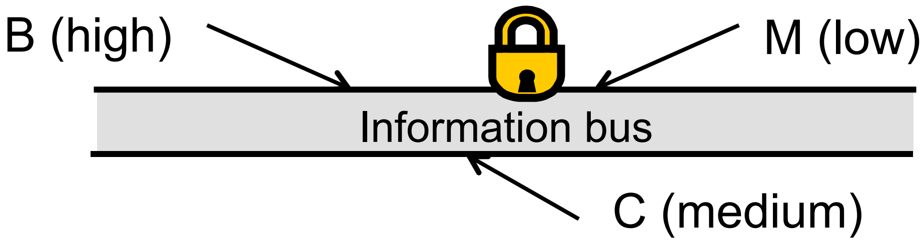
- Problems with TSL-based approach?
  - Waste CPU by busy waiting
  - Can lead to *priority inversion*
    - Two processes, H (high-priority) & L (low-priority)
    - L gets into its CR
    - H is ready to run and starts busy waiting
    - L is never scheduled while H is running …
    - *So L never leaves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

Welcome to Mars!



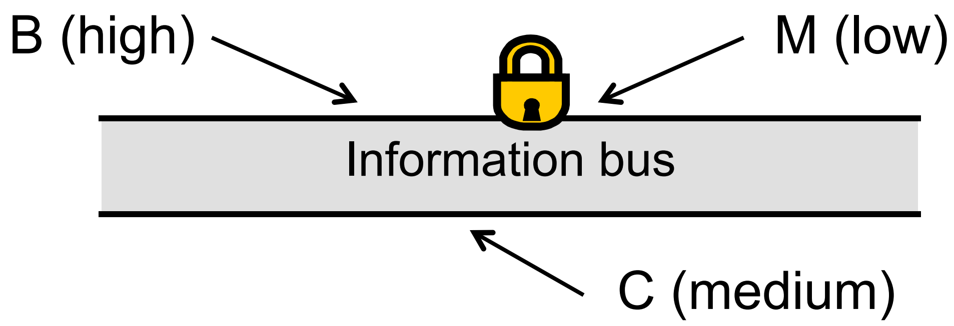
Problems in the Mars Pathfinder*

- Mars Pathfinder
  - Launched Dec. 4, 1996, landed July 4th, 1997
- Periodically the system reset itself, loosing data
- 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 M thread was holding the lock, B 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 (high-priority) was waiting on M (low prioritiy) and**
  - **C (medium priority) 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 disabled!

Yield rather than spin

```
void lock(lock_t *lock) {
   while (TSL(&lock->flag, 1) == 1)
      ; /* spin-wait */
}
```

- Too much spinning
  - Imagine two threads; first one gets the lock and is interrupted
  - Second one wants the lock and have to wait … and wait …
  - Rather than sit in a tight loop, go to sleep
- An alternative – just yield

```
void lock(lock_t *lock) {
   while (TSL(&lock->flag, 1) == 1)
      yield(); /* give up the CPU */
}
```

- Better than spinning but
  - What about the context switching cost?
  - Is there a chance of starvation?

Sleep rather than spin

- What if the *wrong* thread is waken up?
  - The one not holding the lock – wasted context switch
- Too much left to chance
  - The schedule determines who runs next; if it makes a bad choice – yield immediately or sleep
  - Let's get some control over who gets to acquire the lock next

```
typedef struct __lock_t {
  int flag;
  int guard;
  queue_t *q;
} lock_t;
```

An explicit queue to control who gets the lock

```
void init(lock_t *m) {
  m->flag = 0;
  m->guard = 0;
  queue_init(m->q);
}
```

Sleep rather than spin

- Two special calls (from Solaris)
  - `park()`/`unpark()` – put calling thread to sleep / wake one up

```
void lock(lock_t *m) {
   while (TSL(&m->guard, 1) == 1)
     ;
   if (m->flag == 0) {
     m->flag = 1;
     m->guard = 0;
   } else {
     queue_add(m->q, gettid());
     m->guard = 0;
     park();
   }
}
void unlock(lock_t *m) {
   while (TSL(&m->guard, 1) == 1)
      ;
   if (queue_empty(m->q))
      m->flag = 0;
   else
     unpark(queue_remove(m->q);
   m->guard = 0;
}
```

Note the use of guard as a spin-lock around flag and queue manipulation – so not quite avoiding spinning

Here it's where the thread is when woken up

Notice we are not setting flag back to zero when waking up a thread; the one woken up does not have it

Sleep rather than spin

```
void lock(lock_t *m) {
   while (TSL(&m->guard, 1) == 1)
     ;
   if (m->flag == 0) {
     m->flag = 1;
     m->guard = 0;
   } else {
     queue_add(m->q, gettid());
     m->guard = 0;
     park();
   }
}
```

Just curious, what would happen if you park before releasing guard?

Isn't that a race condition? What would happen if the thread about to park is interrupted, the one holding the lock releases it … the parking one will never wakeup!

- One solution uses a third Solaris system call
  - `setpark()` – I am about to park, so be ware
  - After this, if the thread is interrupted and another calls `unpark` before the `park` is called, `parks` returns immediately

```
…
} else {
    queue_add(m->q, gettid());
    setpark();
    m->guard = 0;
    park();
```

Lock-based concurrent data structures

- Making data structures *thread-safe*, i.e., usable by threads
  - Two concerns – correctness, obviously
  - And performance
- A non-concurrent counter

```
typedef struct __counter_t {
   int value;
} counter_t;

void init(counter_t *c) {
   c->value = 0;
}

void increment(counter_t *c) {
   c->value++;
}

void decrement(counter_t *c) {
   c->value--;
}
```

Lock-based concurrent data structures

- A concurrent version

```
typedef struct __counter_t {
   int value;
   pthread_lock_t lock;
} counter_t;

void init(counter_t *c) {
   c->value = 0;
   pthread_mutex_init(&c->lock, NULL);
}
```

```
void increment(counter_t *c) {
   pthread_mutex_lock(&c->lock);
   c->value++;
   pthread_mutex_unlock(&c->lock);
}

void decrement(counter_t *c) {
   pthread_mutex_lock(&c->lock);
   c->value--;
   pthread_mutex_unlock(&c->lock);
}
```

- Still not very scalable; for a better option with sloppy counters
  – S. Boyd-Wickizer et al., “An analysis of Linux Scalability to Many Cores,” OSDI 2010

Lock-based concurrent data structures

- A first concurrent list (part of, actually)

```
typedef struct __node_t {
    int key;
    struct __node_t *next;
} node_t;

typedef struct __list_t {
    node_t *head;
    pthread_mutex_t lock;
} list_t;

void List_Init(list_t *L) {
    L->head = NULL;
    pthread_mutex_init(&L->lock, NULL);
}

int List_Insert(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        perror("malloc");
        pthread_mutex_unlock(&L->lock);
        return -1; /* failure */
    }
    new->key = key;
    new->next = L->head;
    L->head = new;
    pthread_mutex_unlock(&L->lock);
    return 0; /* success */
}
```

```
int List_Lookup(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);
    node_t *current = L->head;
    while(curr) {
        if (curr->key == key) {
            pthread_mutex_unlock(&L->lock);
            return 0; /* success */
        }
        curr = curr->next;
    }
    pthread_mutex_unlock(&L->lock);
    return -1; /* failure */
}
```

Can we simplify this to avoid releasing the lock on the failure path?

Very coarse locking; what's your critical section?

Lock-based concurrent data structures

- And some improvements

```
int List_Insert(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        perror("malloc");
        pthread_mutex_unlock(&L->lock);
        return -1; /* failure */
    }
    new->key = key;
    new->next = L->head;
    L->head = new;
    pthread_mutex_unlock(&L->lock);
    return 0; /* success */
}

int List_Lookup(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);
    node_t *current = L->head;
    while(curr) {
        if (curr->key == key) {
            pthread_mutex_unlock(&L->lock);
            return 0; /* success */
        }
        curr = curr->next;
    }
    pthread_mutex_unlock(&L->lock);
    return -1; /* failure */
}
```

```
int List_Insert(list_t *L, int key) {
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        perror("malloc");
        return -1; /* failur
    }
    new->key = key;
    pthread_mutex_lock(&L->lock);
    new->next = L->head;
    L->head = new;
    pthread_mutex_unlock(&L->lock);
    return 0; /* success */
}
```

Just lock the critical section

```
int List_Lookup(list_t *L, int key) {
    int rv = -1;
    pthread_mutex_lock(&L->lock);
    node_t *current = L->head;
    while (curr) {
        if (curr->key == key) {
            rv = 0;
            break; /*
        }
        curr = curr->next;
    }
    pthread_mutex_unlock(&L->lock);
    return rv;
}
```

A single return path, to avoid potential bugs

Coming up …

- Other mechanisms for synchronization
  - Condition variables
  - Semaphores – slightly higher abstractions
  - Monitors – much better but requiring language support