Process Synchronization

COMP 3361: Operating Systems I
Winter 2015
http://www.cs.du.edu/3361

1 Quick Solutions for Mutual Exclusion

- Disable interrupts upon entry into critical section; enable when done in critical section
  - currently running code would execute without preemption
  - generally too inefficient on multiprocessor systems
    - operating systems using this not broadly scalable
  - do not know how long will a process stay in critical section

- Use a shared boolean variable to indicate that a process is in its critical section
  - does not work; same race conditions as before

Strict Alternation

Producer

```
while (true) {
   while (turn != 0);
   // critical section:
   // write to buffer
   turn = 1;
   // non-critical section
}
```

← a **spin lock**

Consumer

```
while (true) {
   while (turn != 1);
   // critical section:
   // read from buffer
   turn = 0;
   // non-critical section
}
```

← a **spin lock**

shared boolean variable:
`bool turn = 0;`

Peterson's Solution

- Producer and consumer share two variables
  - int **turn**
    - whose turn is it to enter the critical section
  - boolean **interested[2]**
    - is the process ready to enter its critical section?
    - **interested[i] = *true*** implies that process P_i is ready

Algorithm for Process P_i

P_0 : Producer; P_1 : Consumer

```
do {

    interested[i] = TRUE;
    turn = i;
    while (interested[1-i] && turn == i);

            critical section

    interested[i] = FALSE;

            non-critical section
} while (TRUE);
```

The BTS Instruction

`LOCK BTS [loc], 0`

- copy bit 0 from the byte stored at memory address `loc` into the carry (CF) flag (part of EFLAGS)
- set the bit (make it 1)
- the LOCK prefix ensures that the instruction is executed atomically

- Usually referred to as the Test and Set Lock (TSL) instruction

Solution Using BTS

`lock` is a shared byte initialized to **0x00**

```
do {
    check_lock:
      LOCK BTS [lock],0      ← lock becomes 0x01 after this
      JC check_lock          ← jump if CF=1

        critical section

    lock = 0;

        remainder section

} while (TRUE);
```

a busy waiting loop (spin lock)

Semaphore

- A **semaphore** is an integer variable S
- Can be initialized to non-negative number
- After that it can only be accessed through two standard atomic operations

```
down (S) {
   if (S is 0) {
       add calling process to semaphore's queue
       block calling process
   }
   S = S – 1
}
```

```
up (S) {
   S = S + 1
   if (semaphore's queue is not empty) {
      wake up a process from the queue
   }
}
```

Implementing Semaphores

- How to ensure up and down operations are not interrupted?
- Implemented as system calls
  - OS disables interrupts when running `up` and `down`
- Implemented in thread runtime system
  - must use one of the software synchronization methods to ensure that another `up` or `down` operation is not initiated
- Semaphore value is 0 or 1 → **binary semaphore**
- Any other semaphore → **counting semaphore**

Mutex Locks

- Same as software locks using BTS or XCHG, but with no busy waiting

```
mutex_lock (M) {

      LOCK BTS [M],0
      JC block_process
      RET

      block_process:
          add calling process to mutex queue
          block calling process
}
```

```
mutex_unlock (M) {

      M = 0

      if (mutex's queue is not empty)
          wake up a process from queue
}
```

Implementing Mutex Locks

- Implement in kernel, or in thread runtime system
- What is the difference between a mutex lock and a binary semaphore initialized to 1?
  - none if implemented as shown in the previous slides

- Mutex locks have ownership
  - only the process that owns the lock can unlock it
  - modify `mutex_lock` so that we remember which process owns the lock (ran the function without blocking)
  - modify `mutex_unlock` so that the body executes only if called by the owner

11 Bounded-Buffer Prod.-Cons. Solution

- *N* buffers, each can hold one item
- Mutex mx
  - can also be done using a binary semaphore
- Counting semaphore full initialized to the value 0
  - number of full buffers
- Counting semaphore empty initialized to the value N
  - number of empty buffers

The New Producers-Consumers

Producer

```
while (true) {
   /* Produce an item */
     down(empty);
  /* Add item to buffer */
     up(full);
  }
```

Consumer

```
while (true) {
     down(full);
     mutex_lock(mx);
   /* Remove item from buffer */
     mutex_unlock(mx);
     up(empty);
   /* Consume the item */
  }
```

Why these?

to prevent multiple consumers from reading the buffer concurrently

What Can Happen Here?

S and **Q** are two mutex locks

in process P

```
mutex_lock (S);
mutex_lock (Q);
.
.
mutex_unlock (Q);
mutex_unlock (S);
```

in process T

```
mutex_lock (Q);
mutex_lock (S);
.
.
mutex_unlock (S);
mutex_unlock (Q);
```

DEADLOCK

if P executes mutex_lock(**S**) and then T executes mutex_lock(**Q**); or the other way

Lookout!

up (S)

critical section

down (S)

→ several processes may be executing in their critical section

mutex_lock (mx)

critical section

mutex_lock (mx)

→ deadlock

Condition Variables

- The condition used to block a process in semaphores and mutex locks are simple
  - whether a number is zero or not?
- **Condition variables**: No checks on any variable
  - **wait(C)**
    - a process that invokes this operation is suspended
  - **signal(C)**
    - resumes one of the processes (if any) that invoked wait(C)
- Typical usage
  - user code obtains lock on shared variables
  - **user code checks some condition** on the variables
  - release lock on shared variables
  - calls wait() or signal() depending on result of condition check

Monitors

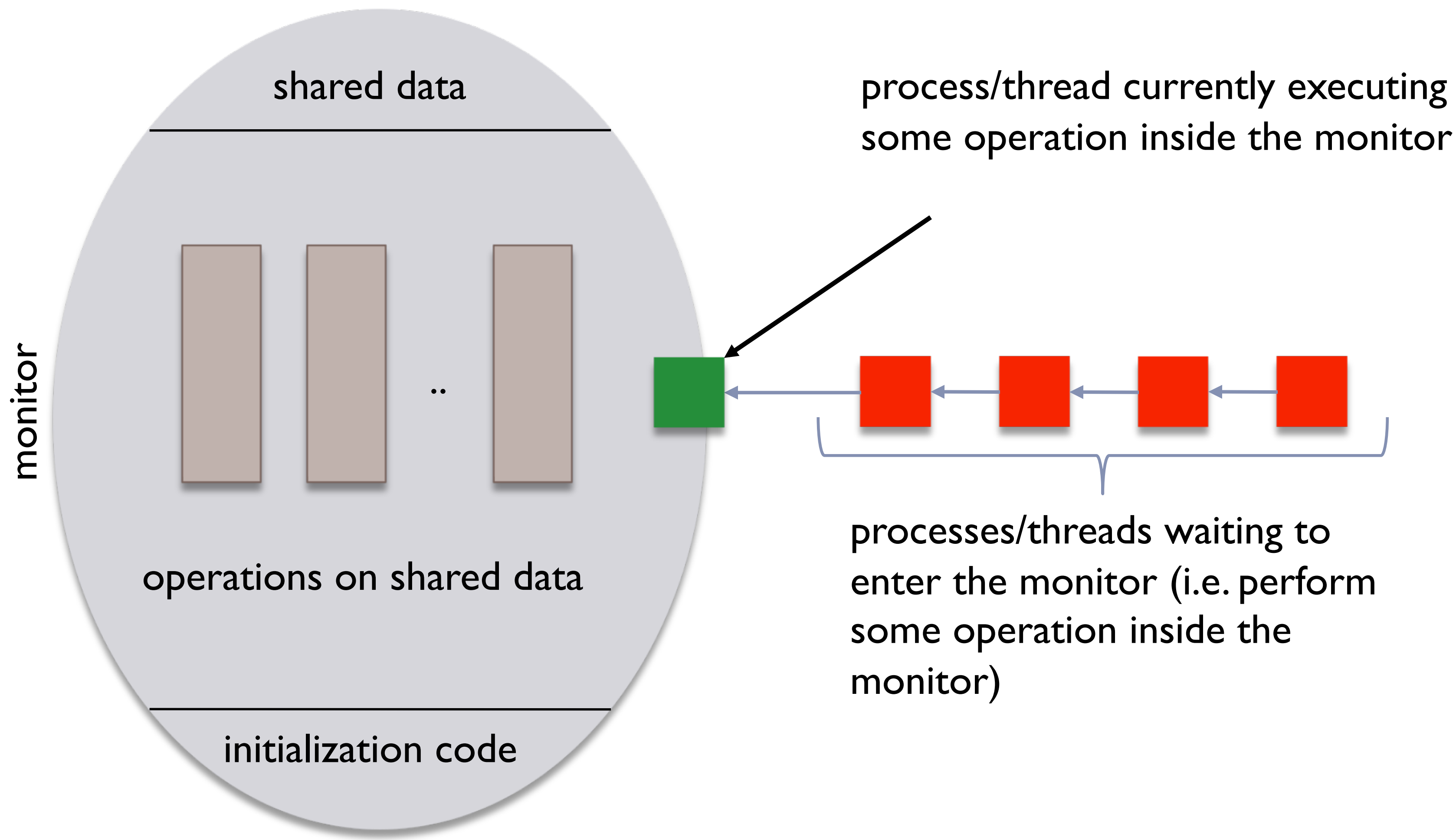
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
  - provided by languages such as Java
- Only one process may be active within the monitor at a time

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

    procedure Pn (…) {……}

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

Schematic View of a Monitor



Mutex Locks Using Pthreads

- `pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;`
  - creates a mutex lock called *mylock* and initializes it
- `pthread_mutex_lock(&mylock);`
  - acquire *mylock*
- `pthread_mutex_unlock(&mylock);`
  - release *mylock*
- `pthread_mutex_destroy(&mylock);`
  - release resources used by mutex *mylock*; in effect uninitializes it

POSIX Semaphores

- **`sem_t my_sem;`**
  - defines a semaphore variable called *my_sem*
- **`sem_init(&my_sem, 0, 5);`**
  - initializes *my_sem* to 5; second arguments says semaphore not to be shared with child processes
- **`sem_wait(&my_sem);`**
  - down operation on semaphore *my_sem*
- **`sem_post(&my_sem);`**
  - up operation on semaphore *my_sem*
- **`sem_destroy(&my_sem);`**
  - destroys the semaphore (release resources associated with *my_sem)*

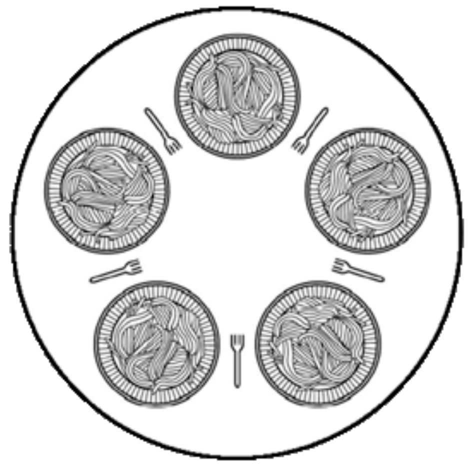
20 Condition Variables Using Pthreads

- `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
  - creates a condition variable called *cond* and initializes it
- `pthread_cond_wait(&cond, &mylock);`
  - releases mutex *mylock* (must have been acquired before calling this function)
  - blocks until a signal on the condition variable *cond* wakes it up
- `pthread_cond_signal(&cond);`
  - unblocks at least one thread (if any) that is blocked on the condition variable *cond*
  - a variant is `pthread_cond_broadcast(&cond)` that unblocks all threads blocked on *cond*
  - thread(s) contend for the mutex used when they called wait()

Some Classical Problems

- Bounded-Buffer Producer-Consumer (PS) Problem
- Dining-Philosophers Problem
- Readers-Writers Problem

Dining-Philosophers Problem



- Can only pick up one ~~fork~~ chopstick at a time
  - pick one, hold, pick another
- Eat when both chopsticks in hand
- Put down both chopsticks and think!!
- Repeat

Dining-Philosophers Solution

- Shared data
  - chopsticks (data, resource, …)
  - an array of binary semaphores chopstick[5]
    - all initialized to 1

Philosopher **i**

```
while (true) {
   down ( chopstick[i] );
   down ( chopstick[(i + 1) % 5] );
   /*  eat  */
   up ( chopstick[i] );
   up ( chopstick[(i + 1) % 5] );
   /*  think  */
}
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - **readers**: only read the data set; they do **not** perform any updates
  - **writers**: can both read and write
- Scenario
  - allow multiple readers to read at the same time
  - when a writer is updating the data, no reader (or another writer) should be accessing the data

Readers-Writer Solution

- Shared data
  - some data set
  - integer readcount initialized to 0
  - mutex lock mx_rc
  - binary semaphore wrt initialized to 1

The Readers-Writer Code

```
do {
    mutex_lock(mx_rc) ;
    readcount ++ ;
    if (readcount == 1)
        down (wrt) ;
    mutex_unlock(mx_rc);

  /* read */

    mutex_lock(mx_rc) ;
    readcount -- ;
    if (readcount  == 0)
         up (wrt) ;
    mutex_unlock(mx_rc) ;
} while (TRUE);
```

Reader

```
while (true) {
        down(wrt);

    /* write */

        up(wrt);
}
```

Writer

A reader always cuts in line in front of writer!

Modify so that new readers wait once writer wants to make an update

References

- Chapter 2.3 and 2.5, Modern Operating Systems, A. Tanenbaum and H. Bos, 4th Edition.