CMSC421: Principles of Operating Systems

Nilanjan Banerjee

Assistant Professor, University of Maryland
Baltimore County
nilanb@umbc.edu
http://www.csee.umbc.edu/~nilanb/teaching/421/

Acknowledgments: Some of the slides are adapted from Prof. Mark Corner and Prof. Emery Berger's OS course at Umass Amherst

Announcements

- Project 1 due on Oct 7th
- Homework 2 is out (due Oct 13th)
- Readings from Silberchatz [6th chapter]

Producer/Consumer Problem using Semaphores

```
semaphore mutex = 1
semaphore full = 0
semaphore empty =
    BUFFER_SIZE

procedure producer() {
    while (true) {
        item = produceItem()
        down(empty)
        down(mutex)
        putItemIntoBuffer(item)
        up(mutex)
        up(full)
    }
}

procedure consumer() {
    while (true) {
        down(full)
        down(mutex)
        item = removeItemFromBuffer()
        up(mutex)
        up(empty)
        consumeItem(item)
    }
}
```

How is a semaphore really implemented

- Implementation of wait or down:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

- Implementation of signal or up:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Example of using Semaphores in linux

Lets look at a demonstration

sem_t * sem = sem_open("filename", flags, mode, initial value)
sem_wait(sem); //decrement
sem_post(sem) //increment

Named semaphore used for synchronization between processes

Unnamed semaphore used for synchronization between threads
Sem_init(sem_t *sem, 0, initial_value)

Example of using pthread_barriers

Barrier impose an ordering in your code

If a barrier is initialized with say 2

you call barrier_wait --- then execution would stop till two threads have called barrier_wait.

```
pthread_barrier_init(barrier);
pthread_barrier_wait(barrier);
```

Reader writer 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
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are treated – all involve priorities

Reader writer problem

thread A

```
lock(&l)
Read data
unlock(&l)
```

thread B

```
lock(&l)
Modify data
unlock(&l)
```

thread C

```
lock(&l)
Read data
unlock(&l)
```

thread A

```
rlock(&rw)
Read data
unlock(&rw)
```

thread B

```
wlock(&rw)
Modify data
unlock(&rw)
```

thread C

```
rlock(&rw)
Read data
unlock(&rw)
```

First solution

- Single lock: safe, but limits concurrency
  - Only one thread at a time, but…
- Safe to have simultaneous readers
  - Must guarantee mutual exclusion for writers

Second solution --- reader/writer locks

- Increases concurrency
- When readers and writers both queued up, who gets lock?
  - Favor readers
    - Improves concurrency
    - Can starve writers
  - Favor writers
  - Alternate
    - Avoids starvation

Exercise: How do you implement reader writer locks?

Shared Data

- Data set
- Semaphore mutex initialized to 1
- Semaphore wrt initialized to 1
- Integer readcount initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
        wait (wrt) ;

            //    writing is performed

        signal (wrt) ;
} while (TRUE);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
                wait (wrt) ;
        signal (mutex)

            // reading is performed

        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)
                signal (wrt) ;
        signal (mutex) ;
    } while (TRUE);
```

Monitors

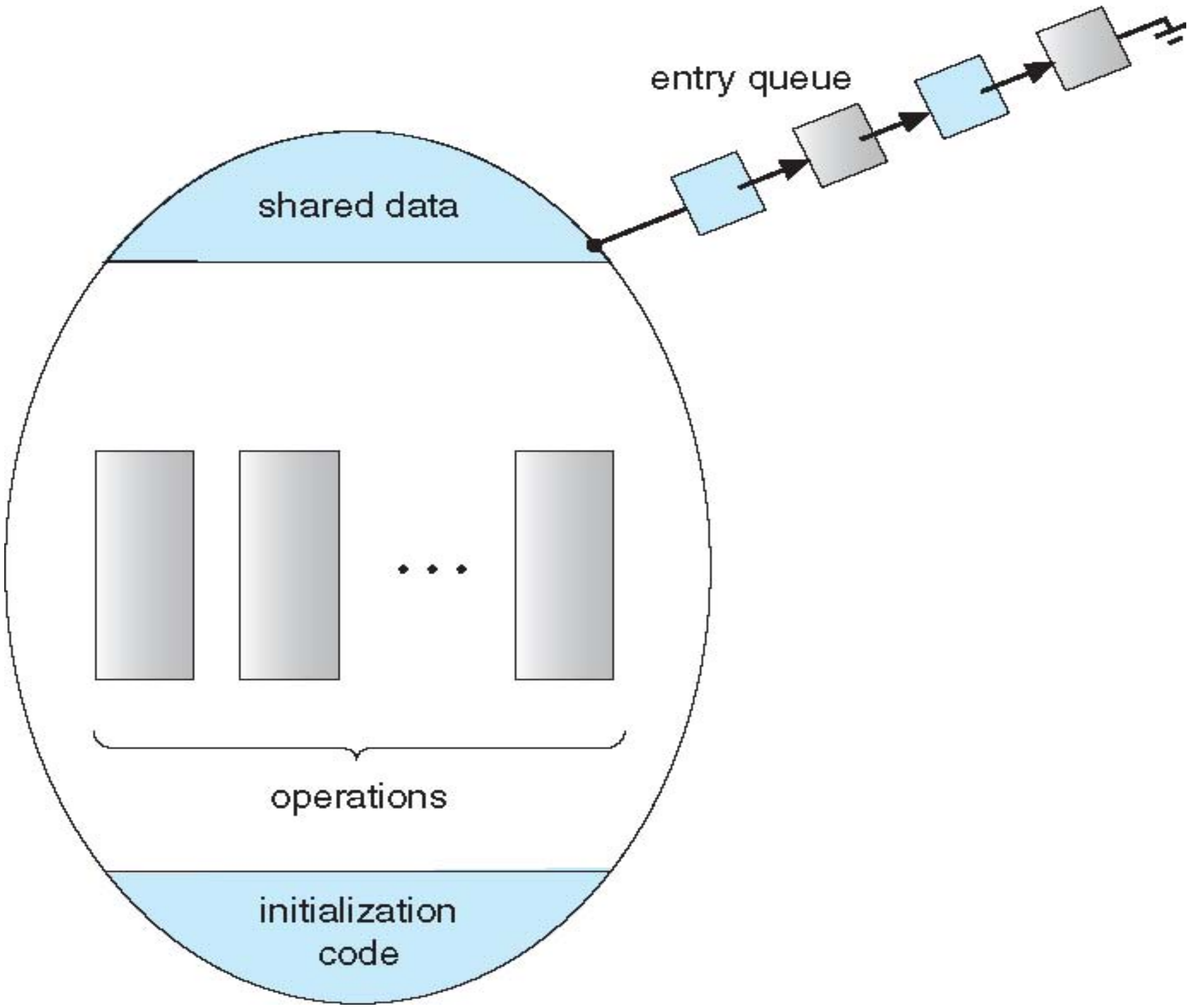
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time

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

   procedure Pn (…) {……}

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

Monitors



Implementing Locks using Swap

```
void Swap (bool *a, bool *b)
        {
                bool temp = *a;
                *a = *b;
                *b = temp:
        }
```

- Shared Boolean variable lock initialized to FALSE;
- Each process has a local Boolean variable key
- Solution:

```
do {
        key = TRUE;
        while ( key == TRUE)
                Swap (&lock, &key );
          //    critical section
         lock = FALSE;

 } while (TRUE);
```

Atomic Transactions (Just a Primer!)

- Assures that operations happen as a single logical unit of work, in its entirety, or not at all
- Related to field of database systems
- Challenge is assuring atomicity despite computer system failures
- Transaction - collection of instructions or operations that performs single logical function
  - Here we are concerned with changes to stable storage – disk
  - Transaction is series of read and write operations
  - Terminated by commit (transaction successful) or abort (transaction failed) operation
  - Aborted transaction must be rolled back to undo any changes it performed

Dining-Philosophers Problem

- Philosophers spend their lives thinking and eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );

            //  eat

        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

            //  think

} while (TRUE);
```

- What is the problem with this algorithm?

An in-class discussion
(surprise : Java swapping)