CMP325 Operating Systems Lecture 16

Synchronization using Monitor

Muhammad Arif Butt, PhD

Note:

Some slides and/or pictures are adapted from course text book and Lecture slides of

- Dr Syed Mansoor Sarwar
- Dr Kubiatowicz
- Dr P. Bhat
- Dr Hank Levy
- Dr Indranil Gupta

For practical implementation of operating system concepts discussed in these slides, students are advised to watch and practice following video lectures:

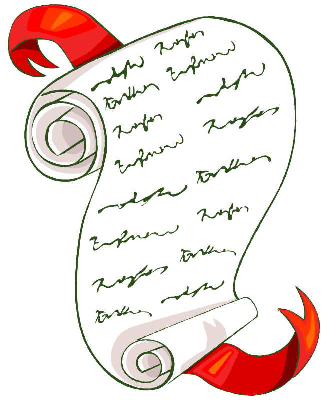
OS with Linux:

https://www.youtube.com/playlist?list=PL7B2bn3G_wfBuJ_WtHADcXC44piWLRzr8

System Programming:

https://www.youtube.com/playlist?list=PL7B2bn3G_wfC-mRpG7cxJMnGWdPAQTViW

Today's Agenda



- Review of Previous Lecture
- Compiler Based Solutions
  - Critical Regions/Statement
    - Producer Consumer Problem using Critical Region
  - Monitors
    - Why monitors
    - Introduction to Monitors
    - Condition variables
    - Producer Consumer Problem using Monitors
    - Dining Philosopher Problem using Monitors

High Level Language Constructs

- Semaphores are good synchronization primitives since they don't waste CPU time through busy waiting
- However, the correctness of a program which uses semaphores depends crucially on correct calls to `wait()` and `signal()` operations. The responsibility is completely on the programmer to issue these calls correctly
- We have seen that wrong initialization/placement of `wait()` and `signal()` in semaphores may cause:
    - Violation of mutual exclusion
    - Dead lock
    - Starvation
- **Solution of above problem is:**
    - Shift the responsibility of enforcing mutual exclusion from programmer to compiler
        - Critical Regions
        - Monitors

A C.R/Monitor are High Level Synchronization construct that allows safe sharing of an ADT among concurrent cooperating processes

Critical Regions

- A **region statement** is a high level synchronization construct that allows safe sharing of an ADT among concurrent cooperating processes
- Critical section solution using region statement has two parts
- Variables that must be accessed under mutual exclusion
- A new language statement(construct) that identifies a critical region in which the variables are accessed

```
v: shared T;
region v when B do S;
```

- Variable **v** is a shared variable of type **T**
- The region statement says that the shared variable **v** can be accessed by a process inside the critical region **S**, iff the condition B is evaluated to true and there is no other process executing inside the critical region **S**. If condition **B** is false or there is another process executing inside **S**, the process is delayed until **B** becomes true and no other process is in the region associated with **v**

Bounded Buffer Problem (using region statement)

```
#define BUFFER_SIZE 5
typedef struct{ ---- } item;
item buffer[BUFFER_SIZE];
int in = 0;   //points to location where next item will be placed, will
              be used by producer process
int out = 0;  //points to location from where item is to be consumed,
              will be used by consumer process
int ctr = 0;
```

```
               Producer Process
item nextProduced;
while(1)
{
  region buffer when (ctr<BUFFER_SIZE)
  {
       buffer[in] = nextProduced;
       in = (in + 1) % BUFFER_SIZE;
       ctr++;
  }
}
```

```
               Consumer Process
item nextConsumed;
while(1)
{
   region buffer when (ctr>0)
   {
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        ctr--;
   }
}
```

Introduction to Monitors

- Monitors can be thought of as an object-oriented extension of the idea of a semaphore
- A monitor is **similar** to a language class that ties data and operations together
- A monitor is **similar** to a class in the sense that its private data can only be accessed by its methods
- A monitor is **different** from a class in the sense that it allows only a single process at a time to execute its procedure
- It contains:
  - Procedures
  - Initialization code
  - Shared data

Introduction to Monitors (…)

- The monitor construct has been implemented in a number of programming languages, including Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3 and Java
- Main characteristics of a monitor are:
  - The local data variables are accessible only by the monitor's procedures and not by any other external procedure
  - A process enters the monitor by invoking one of its procedures
  - Only one process may be executing in the monitor at a time; any other processes that have invoked the monitor are blocked, waiting for the monitor to become available

Monitors & Mutual Exclusion

- By enforcing the discipline of one process at a time, the monitor is able to provide a mutual exclusion facility. The data variables in the monitor can be accessed by only one process at a time
- Thus, a shared data structure can be protected by placing it in a monitor
- Compiler implements the mutual exclusion on monitor. Programmer does not have to be aware of how the compiler arranges for mutual exclusion
- Turn critical sections into monitor procedures, no two processes shall execute their critical sections at same time

Monitors & Java

- The Java synchronized construct implements a limited form of monitor
- In order to turn a java class into a monitor:
  - Make all data private
  - Make all methods synchronized

```
Class MyQueue{
   private ….; // queue data
   public void synchronized addToQueue(Item a){
     put the item a in the queue;
   }
   public item synchronized removeFromQueue(){
     if (queue is not empty){
          remove item;
          return item;
     }
   }
}
```

Monitors in General

```
Lock lock;
Queue queue;

addToQueue(item) {
   lock.Acquire();          // Lock shared data
   queue.enqueue(item);     // Add item
   lock.Release();          // Release Lock
}

removeFromQueue() {
   lock.Acquire();          // Lock shared data
   item = queue.dequeue();// Get next item or null
   lock.Release();          // Release Lock
   return(item);            // Might return null
}
```

Lets suppose the queue is empty, and a process calls removeFromQueue method, acquires lock and starts waiting until something is there in queue?

Monitors & Java (cont…)

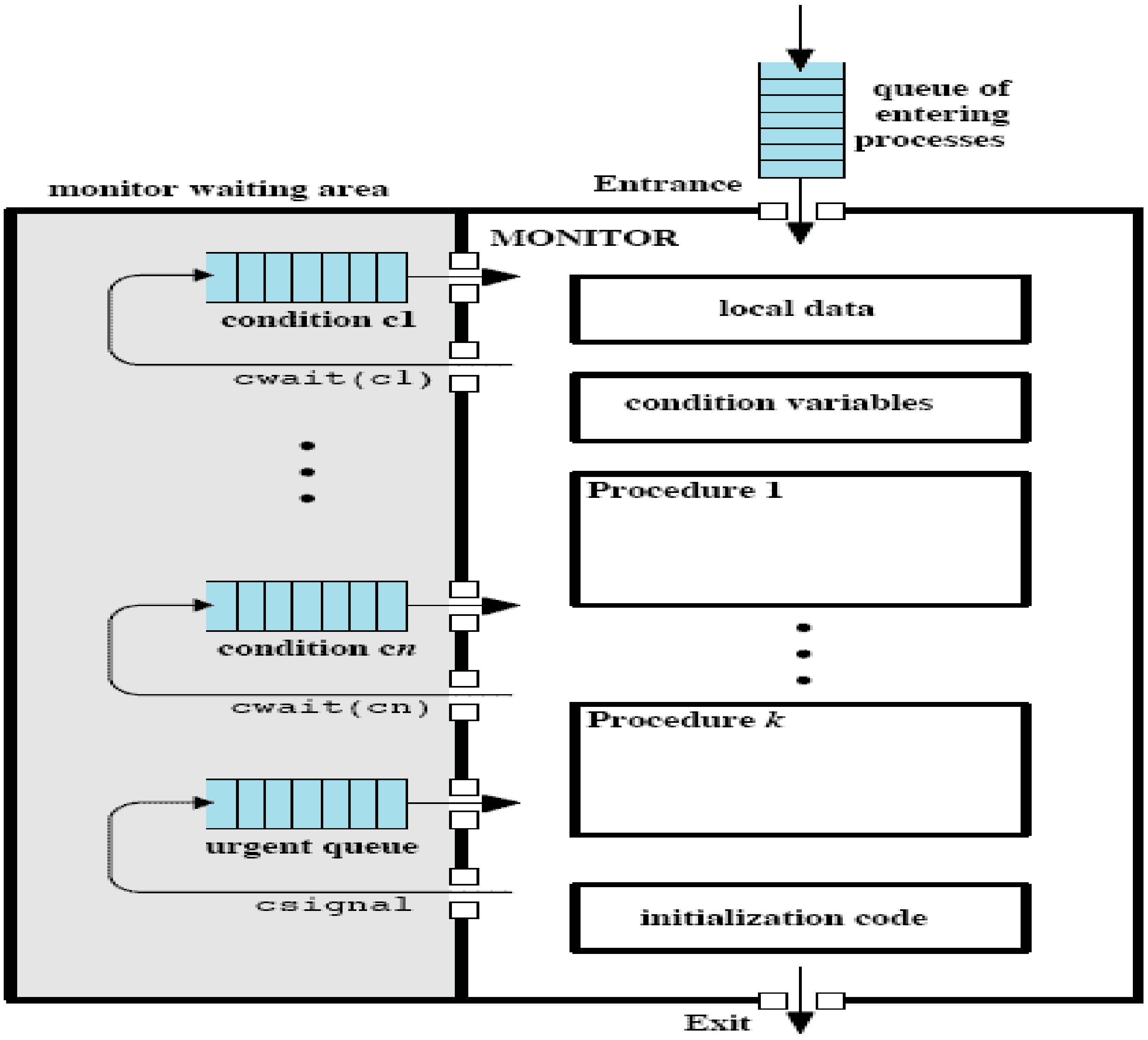
100$ Question: How can we change removefromQueue() to wait until something is in queue?

- Logically, we want to go to sleep inside of the monitor
- But if the process go to sleep inside the monitor, then other threads cannot access the queue. So no thread can call the addToQueue() method. So the thread that has called the removeFromQueue() method could sleep forever
- **Solution:**
  - Use condition variables
    - Condition variables enable a thread to sleep inside a critical section
    - Any lock held by the thread is atomically released when the thread is put to sleep

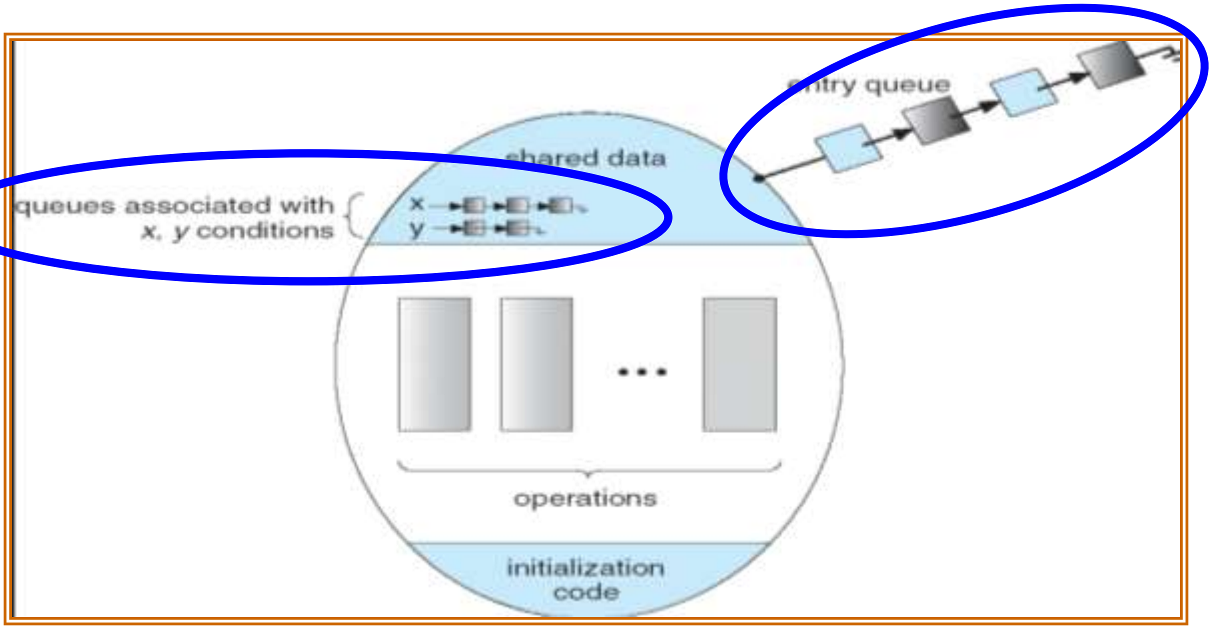
Condition Variables

- A monitor supports synchronization by the use of condition variables that are contained within the monitor and accessible only within the monitor
- A condition variable is a queue of threads waiting for something inside a critical section
- Condition variables are a special data type in monitors, which are operated on by two functions
  - **cwait(x)** The process that called the monitor procedure containing this statement is suspended on a FIFO queue associated with **x**. The monitor is now available for use by another process. The process invoking this operation remain suspended until another process invokes csignal(x)
  - **csignal(x)** Resumes exactly one suspended process at the head of the queue associated with **x**. If no process is suspended, this operation has no effect. (This is unlike the signal() operation on a semaphore, where a signal operation always increments value of semaphore by one)

Structure of a Monitor



Monitors Architecture



- **Lock**: the lock provides mutual exclusion to shared data
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
  - Lock initially free
- **Condition Variable**: a queue of threads waiting for something *inside* a critical section
  - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can't wait inside critical section

What happens after a Signal

- **Hoare-Style Monitor.** Most text books discuss Hoare-Style monitors. If there is a process in a condition queue, a process from that queue runs immediately when another process issues a `csignal` for that condition. Thus, the process issuing the `csignal` must either immediately exit the monitor or be blocked on the monitor.
- **Draw backs of Hoare Monitor**

1. If the process issuing the `csignal` has not finished with the monitor, then two additional process switches are required:
   a) One to block this process
   b) Other to resume it when the monitor becomes available
2. Process scheduling associated with a signal must be perfectly reliable. When a `csignal` is issued, a process from the corresponding condition queue must be activated immediately and the scheduler must ensure that no other process enters the monitor before activation. Otherwise, the condition under which the process was activated could change

What happens after a Signal (…)

- **Mesa-Style Monitor.** Lampson and Redell developed a different definition of monitors for the language Mesa. Java and most real OS follow Mesa-style monitors. `csignal` primitive is replaced by `cnotify`. When a process executing in a monitor executes `cnotify(x)`, it causes the **x** condition queue to be notified, but the signaling process continues to execute. The result of the notification is that the process at the head of the condition queue will be resumed at some convenient future time when the monitor is available.
- One useful refinement that can be associated with the **`cnotify(x)`** primitive is a **watchdog** timer associated with each condition primitive. *A* process that has been waiting for the maximum timeout interval will be placed in a Ready state regardless of whether the condition has been notified. When activated, the process checks the condition and continues if the condition is satisfied. The time out prevents the indefinite starvation of a process in the event that some other process fails before signaling a condition.

Producer Consumer Problem using Monitor

Producer Consumer (Bounded Buffer)

```
monitor ProducerConsumer{
    condition full, empty;
    int count;
void placeItem(item){
    if(count == BUF_SIZE)   wait(full);
    PLACE ITEM IN THE BUFFER;
    count++;
    if(count == 1)  signal(empty);
}
item takeItem(){
    if (count == 0)   wait(empty);
    REMOVE ITEM FROM THE BUFFER;
    count--;
    if(count == BUF_SIZE -1)  signal(full);
}
end monitor;
```

Producer Consumer (…)

Producer

```
do{
    item = produceItem();
    ProducerConsumer.placeitem(item);
}while(1);
```

Consumer

```
do{
    item = ProducerConsumer.takeItem();
    consumeItem(item);
}while(1);
```

Producer Consumer (…)

- Only one process is allowed to execute a monitor procedure at any time. In this case, the producer may execute placeItem() and the consumer may execute takeItem(), but not both at the same time. This ensure mutually exclusive access to the shared variables.
- This solution is more structured compared to semaphores due to two reasons.
  - The data and procedures are encapsulated in a single module.
  - Mutual exclusion is provided automatically and without user involvement in the code.
- The Producer and Consumer processes only see the abstract implementation of the procedures takeItem() and placeItem() and need not know the details of the implementations or data.

Dining Philosopher Problem using Monitor

Dining Philosopher (using Monitor)

```
monitor dp{
   enum {thinking, hungry, eating} state[5];
   condition self[5];
void pickup(int i){
   state[i] = hungry;
   test(i);
   if(state[i] != eating)
      self[i].wait();
}
void putdown(int i){
   state[i] = thinking;
   test((i+4)%5);
   test((i+1)%5);
}
void init(){
   for(int i = 0 ; i < 5 ; i++)
      state[i] = thinking;
}
```

```
void test(int i){
 if((state[(i+4)%5]!=eating)&&
   (state[i] == hungry)    &&
   (state[(i+1)%5] != eating)){
   state[i] = eating;
   self[i].signal();
 } }
```

P_i

```
do{
   think();
   pickup(i);
   eat;
   putdown(i);
}while(1);
```

Dining Philosopher (cont…)

1. **state** is an array of enum type, that can have one out of these three values. Every philosopher can in one of these three states. A philosopher can set **state[i] = eating** iff her two neighbors are not eating.
2. Five condition variables, one for each philosopher. A philosopher uses his condition variable to delay himself inside monitor when he is hungry, but is unable to obtain the chopsticks he needs.
3. **Pickup function.** Every philosopher call the pickup function to pick up the chopsticks. First of all it change its state to hungry, and then call the test function, which may set the state of the philosopher to eating. After calling test function, if the state of the process has not been set to eating, then the calling process i, will suspend itself on condition variable self[i].
4. **Putdown function.** After eating the philosopher calls the putdown function. First of all it changes its state back to thinking, call test on its left and then right neighbors.
5. **init** function sets the state of the philosopher to thinking mode.
6. **Test** function first checks if my left philosopher is not eating and if I am hungry and if my right philosopher is not eating then set the state of the calling philosopher process **(i)** to eating and send a signal to this process.

Reader Writer Problem using Monitor

Reader Writer (using Monitor)

```
Monitor ReadersWriters{
   condition OKtoRead, OKtoWrite;
   int readerCount = 0;
   boolean busy = false;

void startReading(){
   if (busy)//if db is not free, block
      OKtoRead.wait();
   readerCount++;//inc readcount
   OKtoRead.signal();
}
void endReading(){
    readerCount--;//dec readcount
    if (readerCount == 0)
         OKtoWrite.signal();
}
void startWriting (){
   if (busy || readerCount != 0)
      OKtoWrite.wait();
   busy = true;
}
void endWriting(){
   busy = false;
   if(OKtoRead.Queue)
      OktoRead.signal();
   else
      OKtoWrite.signal();
}
```

```
void reader(){
   while(1){
    ReadersWriters.startReading();
    readDatabase();
    ReadersWriters.endReading();
   }
}
void writer(){
   while(1){
     make_data(&info);
     ReaderWriters.startWriting();
     writeDatabase();
     ReaderWriters.endWriting();
   }
}
```

SUMMARY

We're done for now, but Todo's for you after this lecture…

- Go through the slides and Book Sections: 6.7
- Devise pseudo code for the Barber Shop Problem and Smokers Problem using Region statements and Monitors
- Google out the system calls available in POSIX and System-V for the use of condition variables to achieve synchronization