Process Synchronization

Dr. Manmath N. Sahoo
Dept. of CSE, NIT Rourkela

Race Condition

- A condition when several processes access and manipulate the same data-item; and final result depends on the order of access.

Producer Algo (ctr++):

P1: reg1 = ctr;
P2: reg1 = reg1 + 1;
P3: ctr = reg1;

Consumer Algo (ctr--):

C1: reg2 = ctr;
C2: reg2 = reg2 - 1;
C3: ctr = reg2;

- Let ctr = 5 (initially).
- Consider the execution order: P1, P2, C1, C2, P3, C3
- Finally, ctr = 4 (wrong value)

Critical Section

- Critical Section is a section of code where some shared variable(s) is/are modified.

```
do {
        ...
        ...
     entry section

        critical section

     exit section

        reminder section

} while (TRUE);
```

Fig: General structure of a typical process with critical section

Solution to Critical-Section Problem

Criteria

- Mutual Exclusion – No two processes may be simultaneously inside their critical sections
- Progress – If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
- Bounded Waiting – A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section

Two Processes: Solution – 1

- `int turn`
  - shared between processes P_1 and P_2
  - initialized to **1** or **2**

P_1	P_2
`do {` `    while(turn ≠ 1);` `        critical section` `    turn = 2;` `        reminder section` `} while(TRUE)`	`do {` `    while(turn ≠ 2);` `        critical section` `    turn = 1;` `        reminder section` `} while(TRUE)`

Two Processes: Solution – 1

- Mutual Exclusion: Satisfied.
- Progress: Not Satisfied.
  - `turn=1` and P_1 enters to its critical section.
  - P_1 makes `turn=2` in its exit section.
  - P_1 wishes to enter to critical section but can't.
- Bounded waiting: Satisfied

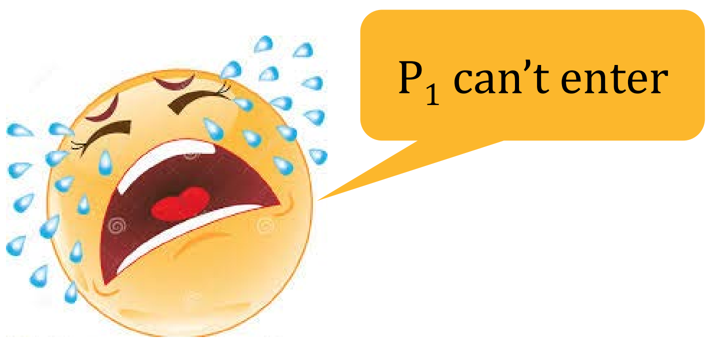
Two Processes: Solution – 2

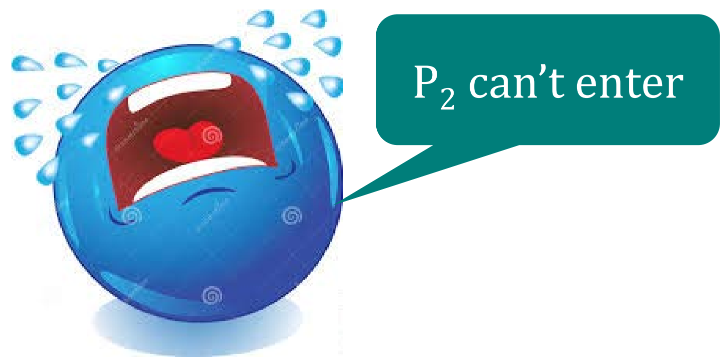
- `int flag[2]` //initialized to **FALSE**

P_1	P_2
`do {` `flag[1] = TRUE;` `while(flag[2]);` `critical section` `flag[1] = FALSE;` `reminder section` `} while(TRUE)`	`do {` `flag[2] = TRUE;` `while(flag[1]);` `critical section` `flag[2] = FALSE;` `reminder section` `} while(TRUE)`

Two Processes: Solution – 2

- Mutual Exclusion: Satisfied.
- Progress: Not Satisfied.
  - P_1 makes `flag[1]=TRUE`
  - P_2 makes `flag[2]=TRUE`





- Bounded waiting: Satisfied

Two Processes: Solution – 3

P_1	P_2
<pre>do { while(flag[2]); flag[1] = TRUE; critical section flag[1] = FALSE; reminder section } while(TRUE)</pre>	<pre>do { while(flag[1]); flag[2] = TRUE; critical section flag[2] = FALSE; reminder section } while(TRUE)</pre>

Two Processes: Solution – 3

- Mutual Exclusion: Not Satisfied.
  - while(flag[2]); -- Pass
  - while(flag[1]); -- Pass
  - P_1 makes flag[1] = TRUE; and enters into critical section
  - P_1 makes flag[2] = TRUE; and enters into critical section
- Progress: Satisfied.
- Bounded waiting: Not Satisfied
  - P_1 may continuously enter into the critical section even when P_2 is waiting in its while loop

Two Processes: Solution – 4 [Peterson's Solution]

P_1	P_2
`do{` `flag[1] = TRUE;` `turn = 2;` `while(flag[2] && turn==2);` `critical section` `flag[1] = FALSE;` `reminder section` `} while(TRUE)`	`do{` `flag[2] = TRUE;` `turn = 1;` `while(flag[1] && turn==1);` `critical section` `flag[2] = FALSE;` `reminder section` `} while(TRUE)`

Two Processes: Solution – 4 [Peterson’s Solution]

- Mutual Exclusion: Satisfied.
- Progress: Satisfied.
- Bounded waiting: Satisfied

Multiple Processes Solution [Bakery Algorithm]

- On a bakery, there are 2 ticket generating machines, suppose. If mutual exclusive access is not given to the number then more than two processes (customers) may have same number. In case of tie, process with lowest pid will be served first.
- (num1, pid1) < (num2, pid2)
  - True; if num1 < num2
  - True; if num1==num2 && pid2 < pid2

Multiple Process Solution [Bakery Algorithm]

P_i

```
do{
    choosing[i] = TRUE;
    number[i]=max(number[0:n-1])+1
    choosing[i] =  FALSE;
    for j = 0 to n-1 {
       while(choosing[j]);
       while( number[j] && (number[j],j) < (number[i],i) );
    }

        critical section

    number[i]=0;

        reminder section

} while(TRUE)
```

Hardware Solutions to Critical Sections

- Enable and Disable Interrupt
- Test-and-set instruction
- Swap instruction

Enable and Disable Interrupt

```
                 Pi
do{
     DI
          critical section
     EI
          reminder section
} while(TRUE)
```

- May miss out some important system interrupts
- Not suitable for multi-processor systems

Test-and-Set instruction

- Test-and-set instruction - defined below as if it were a function

```
boolean Test-and-Set (boolean *target){
  boolean rv = *target;  // return value
  *target = true;  // set value of target
  return(rv);
}
```

Test-and-Set instruction: Solution1

P_i
`` do{   while(Test-and-Set(&lock));   critical section   lock = FALSE;   reminder section } while(TRUE) ```

- lock initialized to FALSE.
- Mutual exclusion: YES
- Progress: YES
- **Bounded waiting: NO**

Test-and-Set instruction: Solution2

- Variables used:
  - global boolean waiting[n] //initialized to FALSE
  - global boolean lock //initialized to FALSE
  - local key

Test-and-Set instruction: Solution2

```
P_i
do{
    waiting[i] = TRUE; key = TRUE;
    while(waiting[i] && key)
      key = Test-and-Set(&lock);
    waiting[i]=FALSE;

       critical section

    j = (i+1) % n;
    while(j!=i && waiting[j]==FALSE)
      j = (j+1) % n;
    if(j==i)
      lock = FALSE;
    else waiting[j] = FALSE;

       reminder section

} while(TRUE)
```

- Mutual exclusion: YES
- Progress: YES
- Bounded waiting: YES

Swap Instruction

- Swap instruction - defined below as if it were a function

```
boolean Swap (boolean *a, *b){
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

Swap Instruction: Solution1

- global boolean lock;
- local boolean key;
- lock & key initialized to FALSE

- Mutual exclusion: YES
- Progress: YES
- **Bounded waiting: NO**

P_i

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

P_j

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

Swap instruction: Solution2

P_i

```
do{
    waiting[i] = TRUE; key = TRUE;
    while(waiting[i] && key)
      Swap(&lock, &key);
    waiting[i]=FALSE;

      critical section

    j = (i+1) % n;
    while(j!=i && waiting[j]==FALSE)
      j = (j+1) % n;
    if(j==i)
      lock = FALSE;
    else waiting[j] = FALSE;

      reminder section

} while(TRUE)
```

- Mutual exclusion: YES
- Progress: YES
- Bounded waiting: YES

Semaphore

- A synchronization primitive proposed by Dijkstra in 1968.
- Consist of a positive integer value
- Two operations
  - P(S) or wait(S): waits for semaphore to become positive
  - V(S) or signal(S): increments semaphore by 1
- P(S) and V(S) operations are atomic.
- Two Types: (i) Binary (ii) Counting

Binary Semaphore: Spin-Lock/Busy-Wait Solution

- Can take two values: 1 or 0 (initialized to 1)

```
Struct Semaphore{ int value; };
Semaphore mutex;
mutex.value=1;
```

```
Pi
do{
    while(mutex.value == 0);
    mutex.value=0;
        critical section
    mutex.value++;
        reminder section
} while(TRUE)
```

P(mutex)

V(mutex)

- Mutual exclusion: YES
- Progress: YES
- **Bounded waiting: NO**

Binary Semaphore: Solution with Waiting List

```
Struct Semaphore{
   int value;
   Struct Process *List;
};
Semaphore mutex;
mutex.value=1;
```

- Mutual exclusion: YES
- Progress: YES
- Bounded waiting: YES

P_i

```
do{
  if(mutex.value == 0){
     Add Pi to mutex->List;
     block();
  }
  else mutex.value=0;

     critical section

  if(mutex->List is nonempty){
     Remove Pj from mutex->List;
     wakeup(Pj);
  }
  else mutex.value++;

     reminder section
} while(TRUE)
```

Counting Semaphore

- Useful when we have multiple instances of same shared resource.
- Initialized to the number of instances of the resource. (e.g. printer)

Counting Semaphore: Spin-Lock/Busy-Wait Solution

```
Semaphore countSem;
countSem.value=3;
```

P_i

```
do{
    while(countSem.value == 0);
    countSem.value--;

        critical section

    countSem.value++;

        reminder section

} while(TRUE)
```

- Mutual exclusion: YES
- Progress: YES
- **Bounded waiting: NO**

Counting Semaphore:
Solution with Waiting List

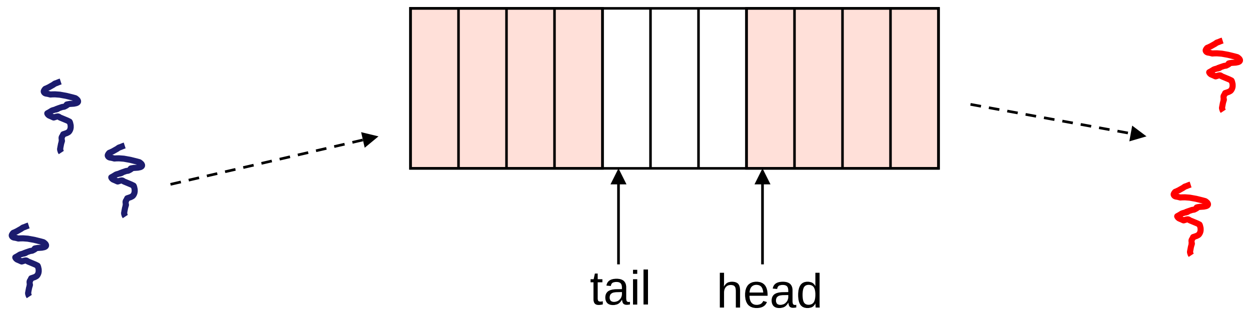
P_i

```
do{
    countSem.value--;
    if(countSem.value < 0){
        Add Pi to countSem->List;
        block();
    }

        critical section

    countSem.value++;
    if(countSem.value <= 0){
        Remove process Pj from countSem->List;
        wakeup(Pj);
    }

        reminder section

} while(TRUE)
```

Mutual exclusion: YES
Progress: YES
Bounded waiting: YES

Bounded Buffer Problem

- *AKA* "producer/consumer" problem
  - there is a buffer in memory with N entries
  - producer threads insert entries into it (one at a time)
  - consumer threads remove entries from it (one at a time)
- Threads are concurrent
  - so, we must use synchronization constructs to control access to shared variables describing buffer



Bounded Buffer Problem

- Constraints
  - The consumer must wait if buffers are empty (synchronization constraint)
  - The producer must wait if buffers are full (synchronization constraint)
  - Only one thread can manipulate the buffer at a time (mutual exclusion)

Bounded Buffer Problem: Developing a Solution

- Each constraint needs a semaphore

```
Semaphore mutex = 1;
Semaphore nFreeBuffers = N;
Semaphore nLoadedBuffers = 0;
```

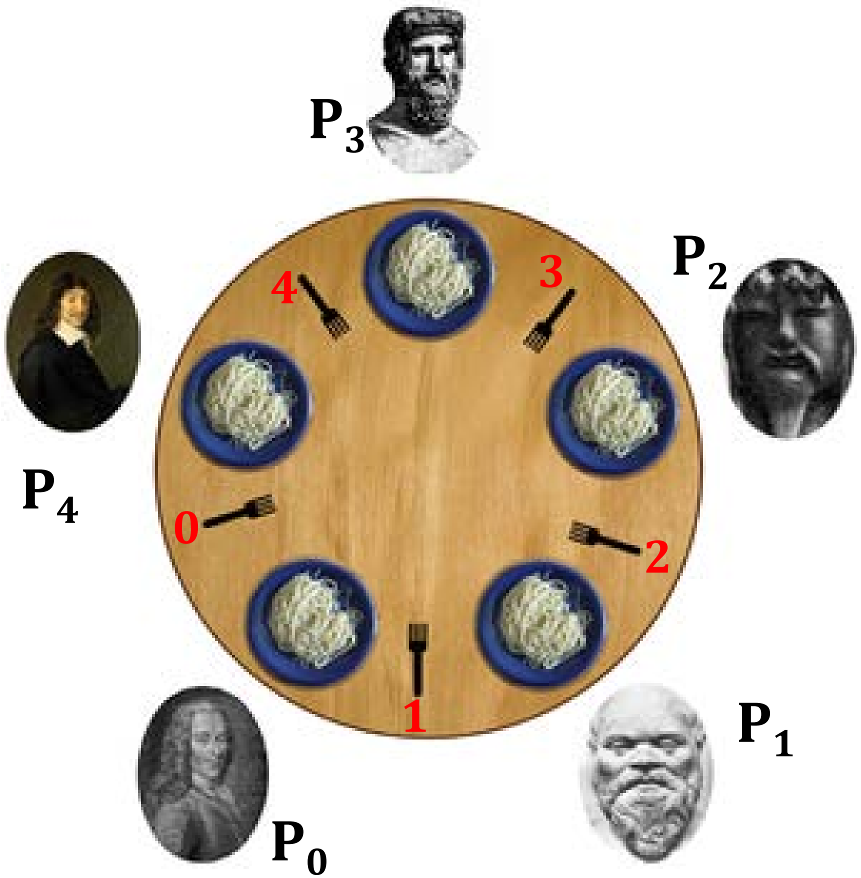
Producer

```
P(nFreeBuffers);
P(mutex);
// put 1 item in the buffer
V(mutex);
V(nLoadedBuffers);
```

Consumer

```
P(nLoadedBuffers);
P(mutex);
// take 1 item from buffer
V(mutex);
V(nFreeBuffers);
```

Dining Philosophers Problem



P_i
do{
P(Chopstick[i])
P(Chopstick[(i+1)%5])
EAT
V(Chopstick[(i+1)%5])
V(Chopstick[i])
THINK
} while(TRUE)

- What if all the philosophers grab their left chopsticks!!

Dining Philosophers Problem: Solution to deadlock

- Allow at most *n-1* philosophers to seat.
- Allow a philosopher to grab the chopsticks only if both are available.

```
DI
P(Chopstick[i])
P(Chopstick[(i+1)%5])
EI
```

- An odd philosopher grabs his left chopstick first then the right chopstick. An even philosopher grabs his right chopstick then the left chopstick.

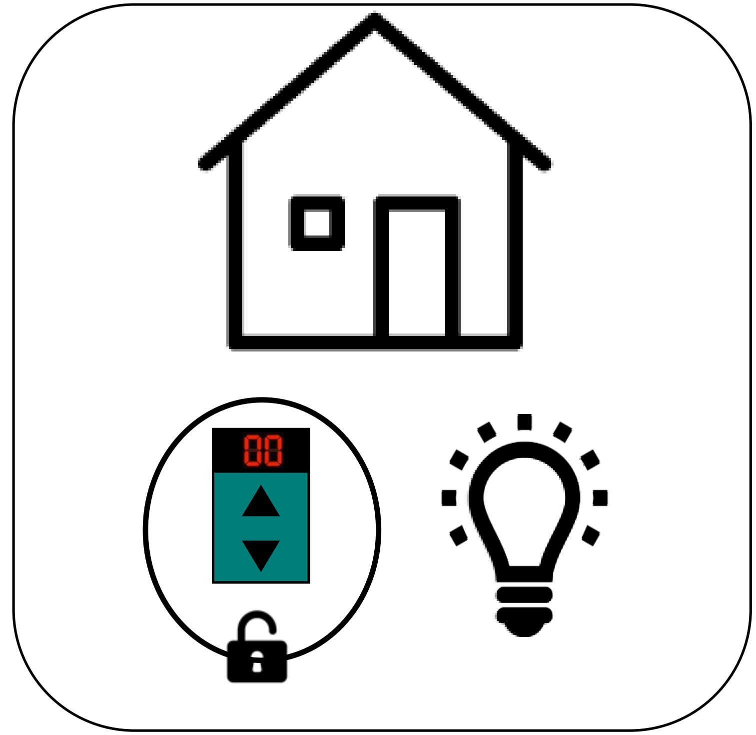
P_0	P_1	P_2	P_3	P_4
P(C[1])	P(C[1])	P(C[3])	P(C[3])	P(C[0])
P(C[0])	P(C[2])	P(C[2])	P(C[4])	P(C[4])

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.
- Conditions
  - Any number of readers may simultaneously read the file.
  - If a writer is writing to the file, no reader/writer is allowed to access the file.

Readers Writers Problem: Solution 1 – Preference to Readers

- Semaphore mutex initialized to 1. 
- Integer rdcount initialized to 0. 
- Semaphore wSem initialized to 1. 



R_i

```
do{
    P(mutex);
     rdcount++;
     if (rdcount==1)  P(wSem);
    V(mutex);

      READ

    P(mutex);
     rdcount--;
     if (rdcount==0)  V(wSem);
    V(mutex);

} while(TRUE)
```

W_i

```
do{
    P(wSem);

      WRITE

    V(wSem);
} while(TRUE)
```

Readers Writers Problem: Solution 1 – Preference to Readers

- Readers only
  - All readers are allowed to READ
- Writers only
  - One writer at a time
- Both readers and writers with read first
  - Writer has to wait on P(wrt)
- Both readers and writers with write first
  - Reader has to wait on P(wrt)

Writers may starve !

Readers Writers Problem: Solution 2 – Preference to Writers

- Integer **rdcount** – keeps track of number of readers.
- Integer **wrtcount** – keeps track of number of writers.
- Semaphore **mutex1** – controls the updating of rdcount.
- Semaphore **mutex2** – controls the updating of wrtcount.
- Semaphore **rSem** – inhibits all readers while there is at least one writer desiring access to critical section.
- Semaphore **wSem** – inhibits all writers while there is at least one reader desiring access to critical section.
- Semaphore **mutex3** – to avoid long queue on rSem. So that waiting writer processes get preference.

Readers Writers Problem: Solution 2 – Preference to Writers

R_i

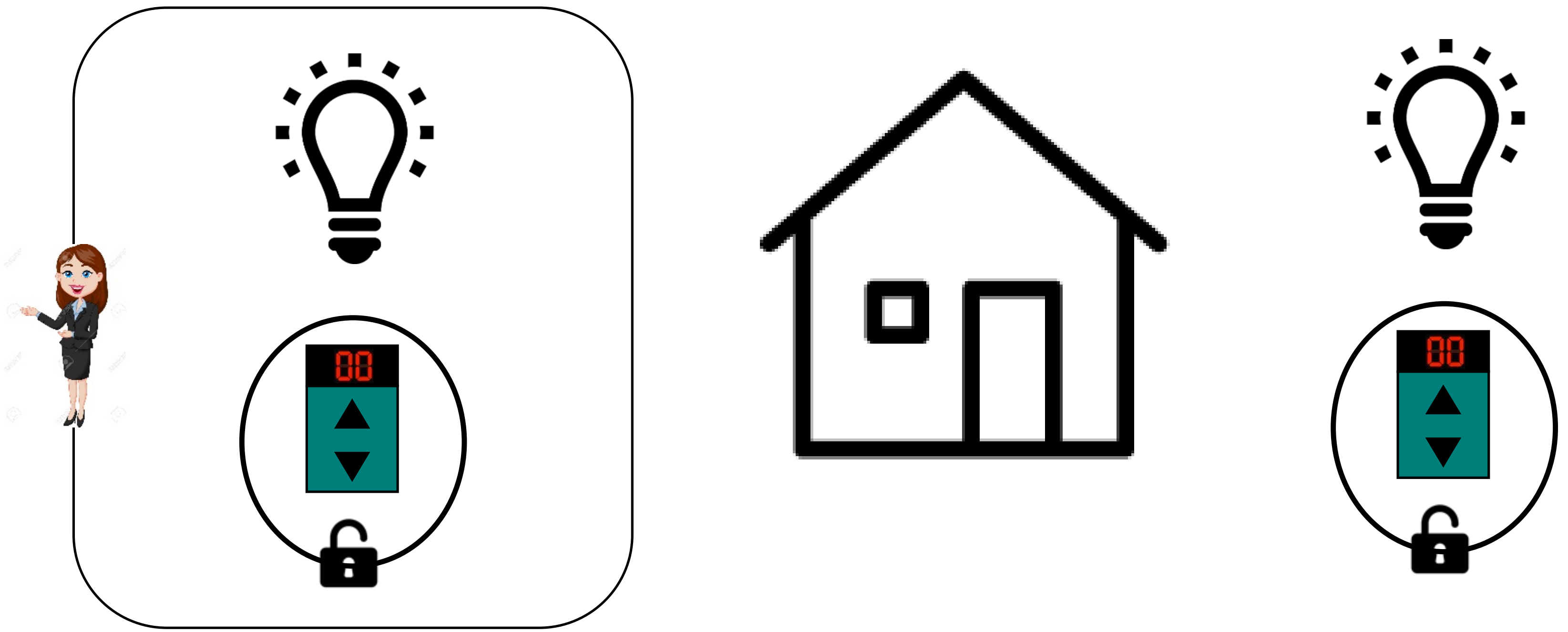
```
do{
    P(mutex3);
     P(rSem);
      P(mutex1);
       rdcount++;
       if(readcount == 1)  P(wSem);
      V(mutex1);
     V(rSem);
    V(mutex3);
      READ
    P(mutex1);
     readcount--;
     if (readcount == 0)  V(wSem);
    V(mutex1);
      reminder section
} while(TRUE)
```

W_i

```
do{
    P(mutex2);
     wrtcount++;
     if(writecount == 1) P(rSem);
    V(mutex 2);

    P(wSem);
      WRITE
    V(wSem);

    P(mutex2);
     writecount--;
     if (writecount == 0) V(rSem);
    V(mutex 2);
      reminder section
} while(TRUE)
```

Readers may starve ! 

Readers Writers Problem: Solution 3 – Based on the arrival order



Entry **Exit**

- Integer **rdcount**
- Semaphore **rdcount_mutex**
- Semaphore **access_mutex**
- Semaphore **order_mutuex**

Readers Writers Problem: Solution 3 – Based on the arrival order

R_i

```
do{
    P(order_mutex);
     P(rdcount_mutex);
      rdcount++;
      if (rdcount==1)
         P(access_mutex));
     V(rdcount_mutex);
    V(order_mutex);

      READ

    P(rdcount_mutex);
     rdcount--;
     if (rdcount==0)
        V(access_mutex);
    V(rdcount_mutex);

} while(TRUE)
```

W_i

```
do{
    P(order_mutex);
     P(access_mutex));
    V(order_mutex);

     WRITE

    V(access_mutex);

} while(TRUE)
```

Problems with semaphore

- Suppose that a process interchanges the order of wait() and signal() operations – violates mutual exclusion.

```
signal(mutex);
  //critical section
wait(mutex);
```

- Suppose that a process replaces signal() with wait() – results in deadlock.

```
wait(mutex);
  //critical section
wait(mutex);
```

Problems with semaphore

- Suppose that a process replaces wait() with signal() or viceversa – results in deadlock – violates mutual exclusion.

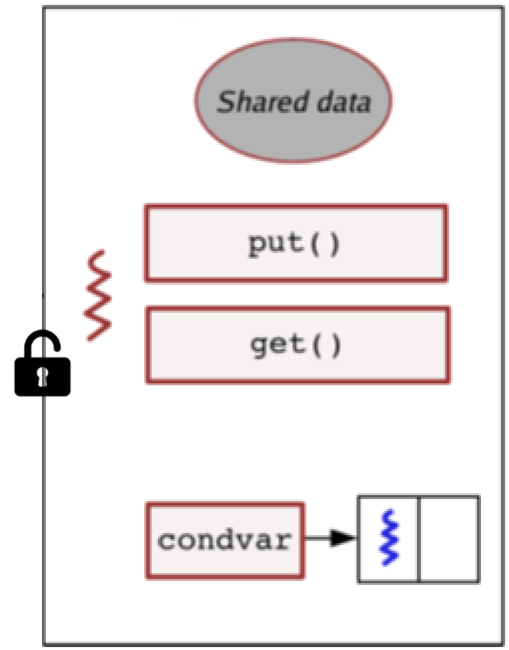
```
signal(mutex);                wait(mutex);
  //critical section            //critical section
signal(mutex);                wait(mutex);
```

- Suppose that a process omits wait(), signal() or both– results in deadlock or violation of mutual exclusion.

```
//critical section              wait(mutex);
signal(mutex);                    //critical section
```

Monitor: A structured synchronization tool

- A monitor is a module that encapsulates:
  - some shared data
  - some atomic procedures
  - a set of condition variables



- Only one process at a time may be active in a monitor

Monitor: A structured synchronization tool

- Condition variables allow for blocking and unblocking.
  - If a thread/process has to wait/block for some event to occur by some other thread/process, then it waits in the queue of the corresponding condition variable. e.g., **condvar.wait()**.

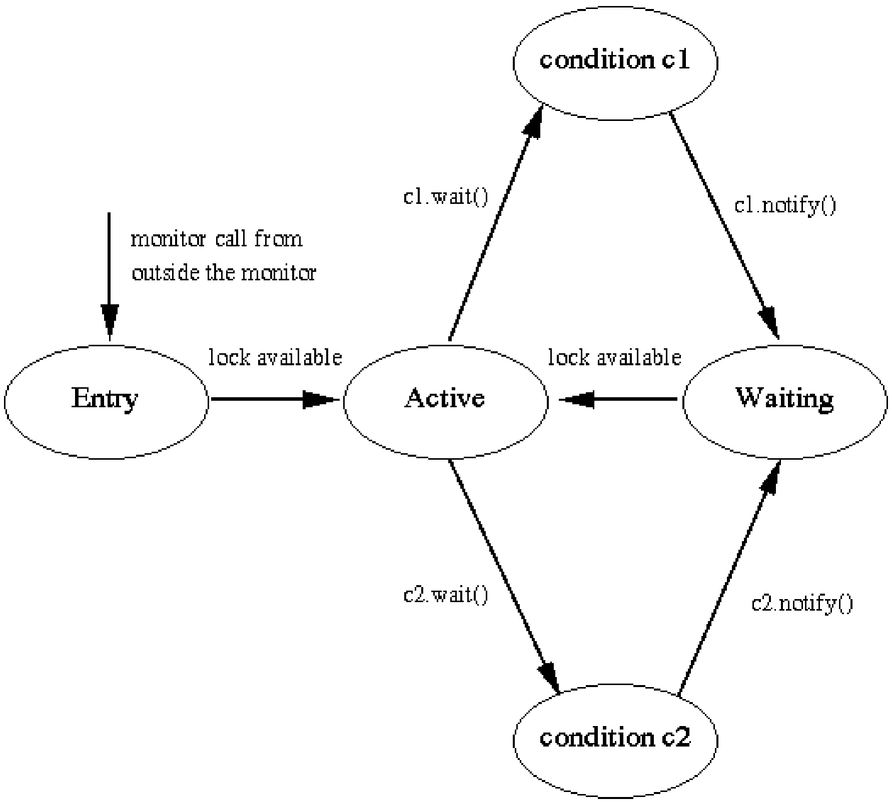
    (Note: it immediately releases the monitor lock)
  - If a thread/process has generated the event, it can indicate this by executing **condvar.signal() or condvar.notify()**.
    - This wakes up a waiting thread/process from condvar queue.
    - If condvar queue is empty then it cond.signal() doesn't do anything.
    - What happens to the signaler thread/process?
      - Implementation 1: They wait in the signaler queue
      - Implementation 2: They are active (no signaler queue)

Queues in monitor implementation

- The **entry queue** contains processes attempting to call a monitor procedure from outside the monitor.
  - Each monitor has one entry queue.
- The **waiting queue** contains processes that have been awakened by a notify operation.
  - Each monitor has one waiting queue.
- **Condition variable queues** contain processes that have executed a condition variable wait operation.
  - There is one such queue for each condition variable.
- The **signaler queue** contains processes that have executed a notify/signal operation.
  - Each monitor has at most one signaler queue.
  - In some implementations, a notify leaves the process active and no signaler queue is needed.

Monitor state transition:
Without signaler queue

- The lock becomes available when the active process executes a wait or leaves the monitor.



Solution to Producer Consumer problem using Monitor

```
monitor PC {
    int DATA[10];
    int R, F;
    Condition FULL, EMPTY;
    Produce(v) {
        if ( (R+1)%10== F ) then FULL.Wait();
        put v into DATA array;
        EMPTY.Signal();
    }
    Consume() {
        if( head == 0 ) then EMPTY.Wait();
        consume the next DATA array value;
        FULL.Signal();
    }
   init(){ R=F=0; }
}
```

Producer$_i$

```
…
…
DATA.Produce(25);
…
…
```

Consumer$_i$

```
…
…
DATA.Consume(25);
…
…
```

Monitor in Java

One modern language that uses monitors is Java.

- Each object has its own monitor.
- Methods are put in the monitor using the synchronized keyword.
- Each monitor has one condition variable.
- The methods on the condition variables are: wait(), notify(), and notifyAll().
- Since there is only one condition variable, the condition variable is not explicitly specified.