Operating Systems
CS220

Lecture 13

Process Synchronization-II

21th June 2021

By: Dr. Rana Asif Rehman

Process Synchronization

Objectives

- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution using TestAndSet

- Shared boolean variable lock., initialized to false

```
while (true) {
  while ( TestAndSet (&lock ))
     ; // do nothing
       //    critical section
      lock = FALSE;
      //     remainder section
}
```

```
int
TestAndSet(boolean
*lockValue)
{
  boolean rv;
  rv = *lockValue;
  *lockValue = true;
  return rv;
}
```

Solution using Swap

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

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

Definition:

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

Test-and-Set Instruction

- Mutual exclusion is assured: if Pi enters CS, the other processes are ***busy waiting***
- Satisfies ***progress*** requirement
- When Pi exits CS, the selection of the next P_j to enter CS is arbitrary
  - No bounded waiting (it is a race!!!)

Operating Systems or Programming Language Support for Concurrency

- Solutions based on machine instructions such as ***test and set*** involve tricky coding
  - For example, the SetAndTest algorithm does not satisfy all the requirements to solve the critical-section problem
  - **Starvation** is possible
- We can build better solutions by providing synchronization mechanisms in the Operating System or Programming Language (This leaves the really tricky code to systems programmers)

Solution to bounded waiting

```
do{
    waiting[ i ] = TRUE;
    key = TRUE;
    while (waiting[ i ] && key)
        key = TestandSet(&lock);
    waiting[ i ] = FALSE;
    //Critical Section
    j=(i+1) % n ;
    while ((j != i) && !waiting[ j ])
        j=(j+1) % n ;
    if (j == i)
        lock = FALSE
    else
        waiting[ j ] = FALSE ;
    //remainder section
}while (TRUE);
```

Semaphores

- A Semaphore S is an integer variable that, apart from initialization, can only be accessed through 2 atomic and mutually exclusive operations:
- wait(S)
  - sometimes called P()
    - Dutch proberen: "to test"
- signal(S)
  - sometimes called V()
    - Dutch verhogen: "to increment"

- The classical definition of **wait** and **signal** is as shown in the following figures
- Useful when critical sections last for a short time, or we have lots of CPUs
- S initialized to positive value (to allow someone in at the beginning)

```
wait(S) {
  while S<=0 do ;
  S- -;  }
```

```
signal(S) {
  S++; }
```

Semaphores in Action

Initialize mutex to 1

```
Process Pi:                    Process Pj:
   repeat                      repeat
 wait(mutex);                     wait(mutex);
       CS                          CS
 signal(mutex);                   signal(mutex);
       RS                          RS
   forever                     forever
```

```
wait(S) {
  while S<=0 do ;
  S- -;  }
```

```
signal(S) {
  S++; }
```

Synchronizing Processes using Semaphores

- Two processes:
  - P_1 and P_2
- Statement S_1 in P_1 needs to be performed **before** statement S_2 in P_2
- We want a way to make P_2 wait
  - Until P_1 tells it is OK to proceed

Define a semaphore "synch"

Initialize synch to 0

Put this in P_2:

```
wait(synch);
S2;
```

And this in P_1:

```
S1;
signal(synch);
```

Semaphores: the Problem of busy waiting

- Semaphore definitions (so far) all require busy waiting
- This type of semaphore is called **spinlock**
- This continual looping is a problem in a multiprogramming setting
- As a solution, modify the definition of the wait and signal semaphores

- Define a semaphore as a record

```
typedef struct {
   int value;
   struct process *L;
} semaphore;
```

- Assume two simple operations:
  - **block** suspends the process that invokes it.
  - **wakeup(*P*)** resumes the execution of a blocked process **P**

Semaphore operations now defined as

wait(*S*):

```
    S.value - -;
    if (S.value < 0) {
                    add this process to S.L;
            block; }
```

signal(*S*):

```
    S.value++;
    if (S.value <= 0) {
            remove a process P from S.L;
            wakeup(P); }
```

Deadlock and Starvation

- An implementation of a semaphore with a waiting queue may result in:
  - **Deadlock:** two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes
    - Let S and Q be two semaphores initialized to 1

P0	**P1**
wait(S);	wait(Q);
wait(Q);	wait(S);
⋮	⋮
signal(S);	signal(Q);
signal(Q)	signal(S);

- **Starvation:** indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended
- If we add or remove processes from the list associated with a semaphore in LIFO manner

Applications of Semaphores

- Binary Semaphores
- Counting Semaphores
- Applications
  - Critical Section Problem
  - Deciding order of execution
  - For Managing Resources
    - e.g. 5 printers

Classical Problems of Synchronization

- Bounded-buffer problem
- Reader-Writer Problem
- Dining Philosophers Problem
- Monitors

```
wait(S) {
  while S<=0 do ;
  S- -;  }
```

```
signal(S) {
  S++; }
```

Classical Problems of Synchronization: Bounded-Buffer Problem

- Shared data: **semaphore full, empty, mutex;**
- Initially: **full = 0, empty = n, mutex = 1**
- We have n buffers. Each buffer is capable of holding ONE item

Consumer

```
do {
    wait(full)
    wait(mutex);
    …
    remove an item from
    buffer to nextc
    …
    signal(mutex);
    signal(empty);
    …
  consume the item in nextc
  …
} while (1);
```

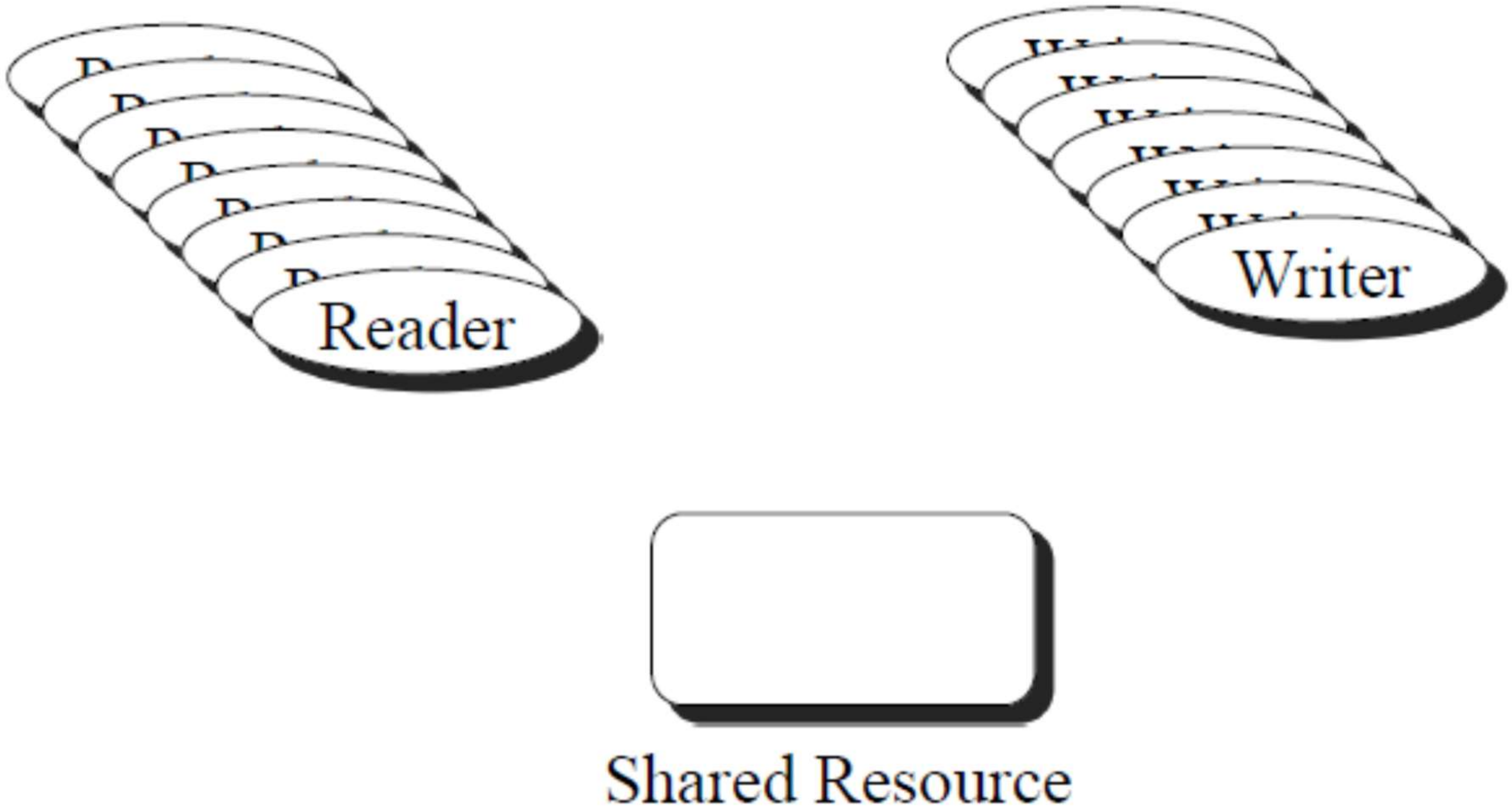
Producer

```
do {
    …
   produce an item in
    nextp
     …
    wait(empty);
    wait(mutex);
    …
  add nextp to buffer
    …
    signal(mutex);
    signal(full);
} while (1);
```

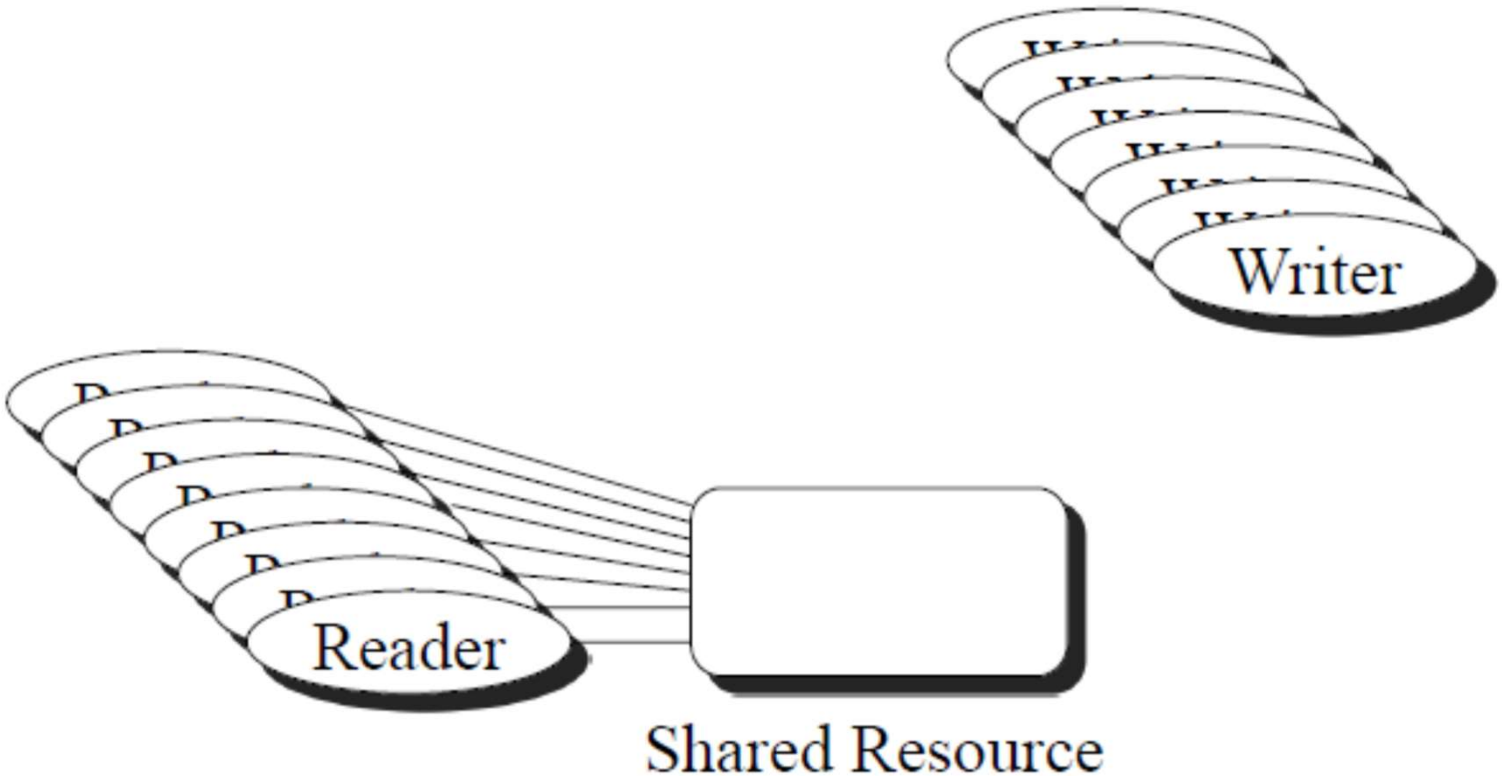
Classical Problems of Synchronization: Readers-Writers Problem

- There is one writer and multiple readers
- The writer wants to write to the database
- The readers wants to read from the database
- We can not allow a writer and a reader writing and reading the database at the same time
- We can allow one or more readers reading from the database at the same time
- Two different versions:
  - **First reader-writers problem**
  - **Second readers-writers problem**

Reader-writers problem (Cont.)

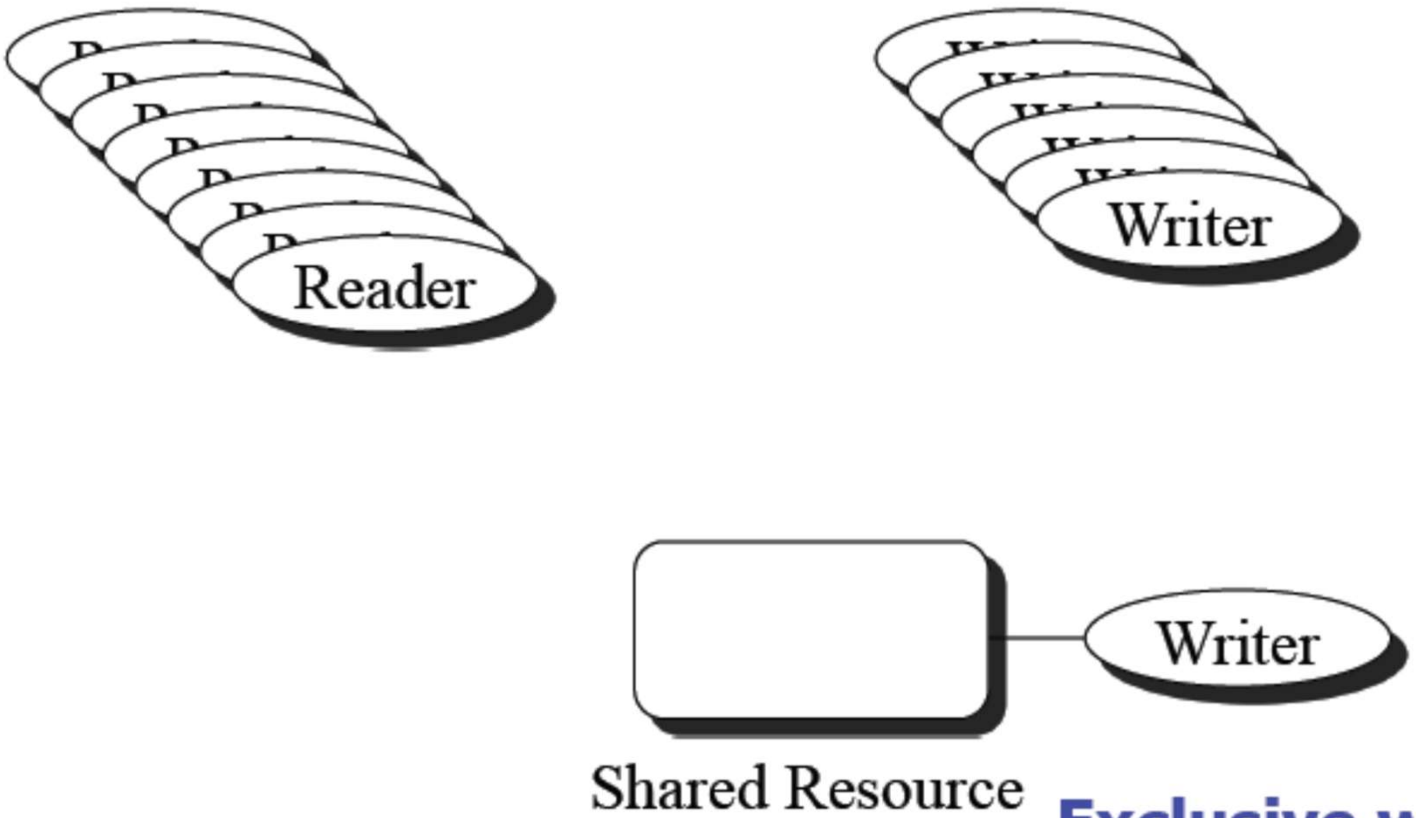


Reader-writers problem (Cont.)



Concurrent readers

Reader-writers problem (Cont.)



Exclusive writer

First Solution: Reader’s precedence

```
Reader() {
    while(TRUE) {
        wait(mutex);
          readCount++;
          if(readCount==1)
              wait(wrt);
        signal(mutex);

        read(resource);

        wait(mutex);
          readCount--;
          if(readCount == 0)
              signal(wrt);
        signal(mutex);
}}
```

```
Writer() {
    while(TRUE) {
        wait(wrt);
          write(resource);
        signal(wrt);
    }
}
```

```
resourceType *resource;
int readCount = 0;
semaphore mutex = 1;
semaphore wrt = 1;
```

- First reader competes with writers
- Last reader signals writers

First Solution: reader's precedence

```
Reader() {
    while(TRUE) {
        wait(mutex);
          readCount++;
          if(readCount==1)
              wait(wrt);
        signal(mutex);

        read(resource);

        wait(mutex);
          readCount--;
          if(readCount == 0)
              signal(wrt);
        signal(mutex);
}}
```

```
Writer() {
    while(TRUE) {
        wait(wrt);

          write(resource);

        signal(wrt);
    }
}
```

- First reader competes with writers
- Last reader signals writers
- Any writer must wait for all readers
- Readers can starve writers
- "Updates" can be delayed forever

Second Solution: Writer's precedence

```
Reader() {
  while(TRUE) {
      wait(rd);
          wait(mutex1);
              readCount++;
              if(readCount == 1)
                  wait(wrt);
          signal(mutex1);
      signal(rd);
      read(resource);
    wait(mutex1);
      readCount--;
      if(readCount == 0)
          signal(wrt);
    signal(mutex1);
  }
}
```

```
writer() {
  while(TRUE) {
      wait(mutex2);
          writeCount++;
          if(writeCount == 1)
              wait(rd);
      signal(mutex2);
      wait(wrt);
          write(resource);
      signal(wrt);
      wait(mutex2)
          writeCount--;
          if(writeCount == 0)
              signal(rd);
      signal(mutex2);
```

int readCount = 0, writeCount = 0;
semaphore mutex1 = 1, mutex2 = 1;
semaphore rd = 1, wrt = 1;

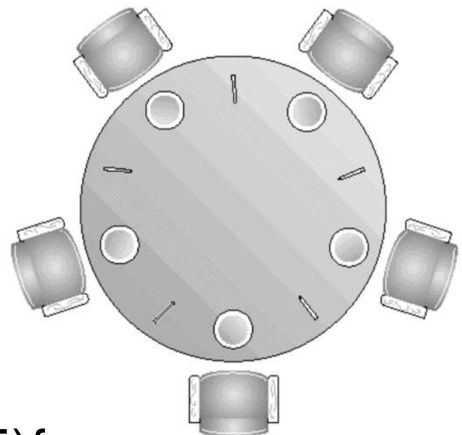
The Dining Philosophers Problem

- A classical synchronization problem
- 5 philosophers who only eat and think
- Each need to use 2 forks for eating
- There are only 5 forks
- Illustrates the difficulty of allocating resources among process without deadlock and starvation

The Dining Philosopher Problem

- Each philosopher is a process
- One semaphore per fork:
  - Fork: array[0..4] of semaphores
  - Initialization:

    fork[i].count:=1 for i:=0..4
- A first attempt:
  - Deadlock if each philosopher starts by picking his left fork!

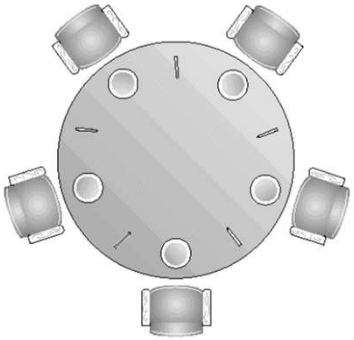


```
Pi() {
  while(TRUE){
    think;
    wait(fork[i]);
      wait(fork[i+1 mod 5]);
        eat;
      signal(fork[i+1 mod 5]);
    signal(fork[i]);
  }
}
```

The Dining Philosophers Problem

- Idea: admit only 4
- philosophers at a time who try to eat
- Then, one philosopher can always eat when the other 3 are holding one fork
- Solution: use another semaphore T to limit at 4 the number of philosophers "sitting at the table"
- Initialize: T.count:=4

```
Pi(){
  while(TRUE){
    think;
    wait(T);
      wait(fork[i]);
        wait(fork[i+1 mod 5]);
          eat;
        signal(fork[i+1 mod 5]);
      signal(fork[i]);
    signal(T);
  }
}
```



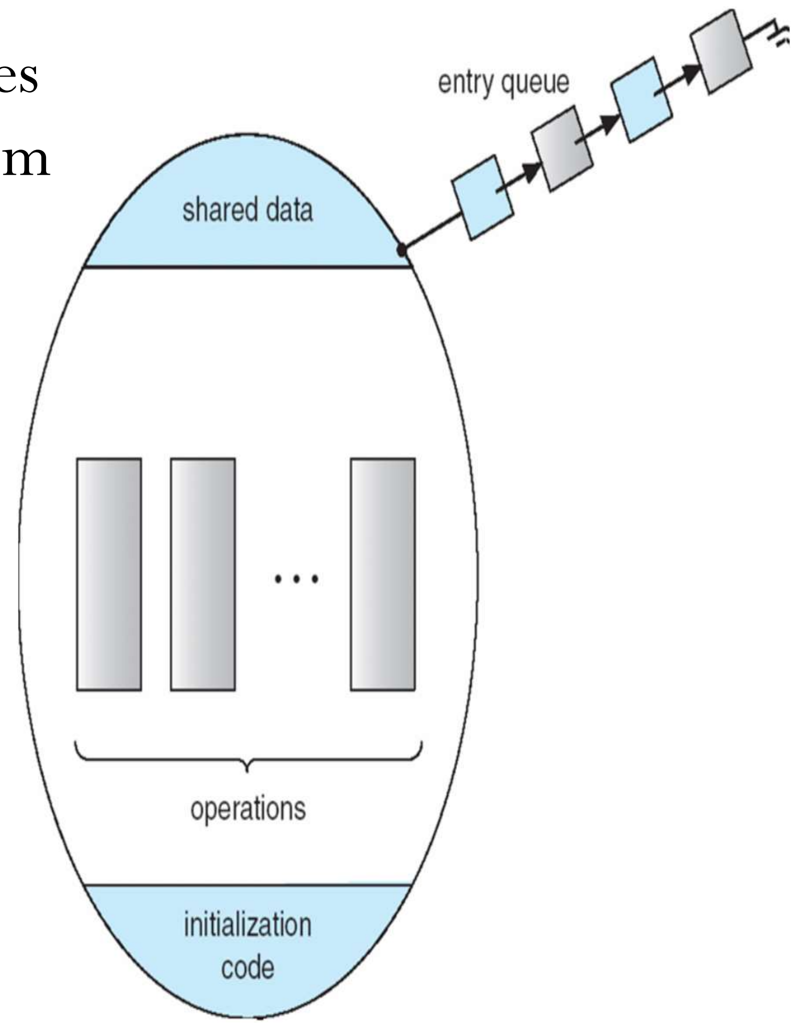
Recall: Problems with Semaphores

- Semaphores are a powerful tool for enforcing mutual exclusion and coordinate processes
- Problem: wait(S) and signal(S) are scattered among several processes
  - It is difficult to understand their effects
  - Usage must be correct in all processes
  - One bad (or malicious) process can fail the entire collection of processes

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- 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

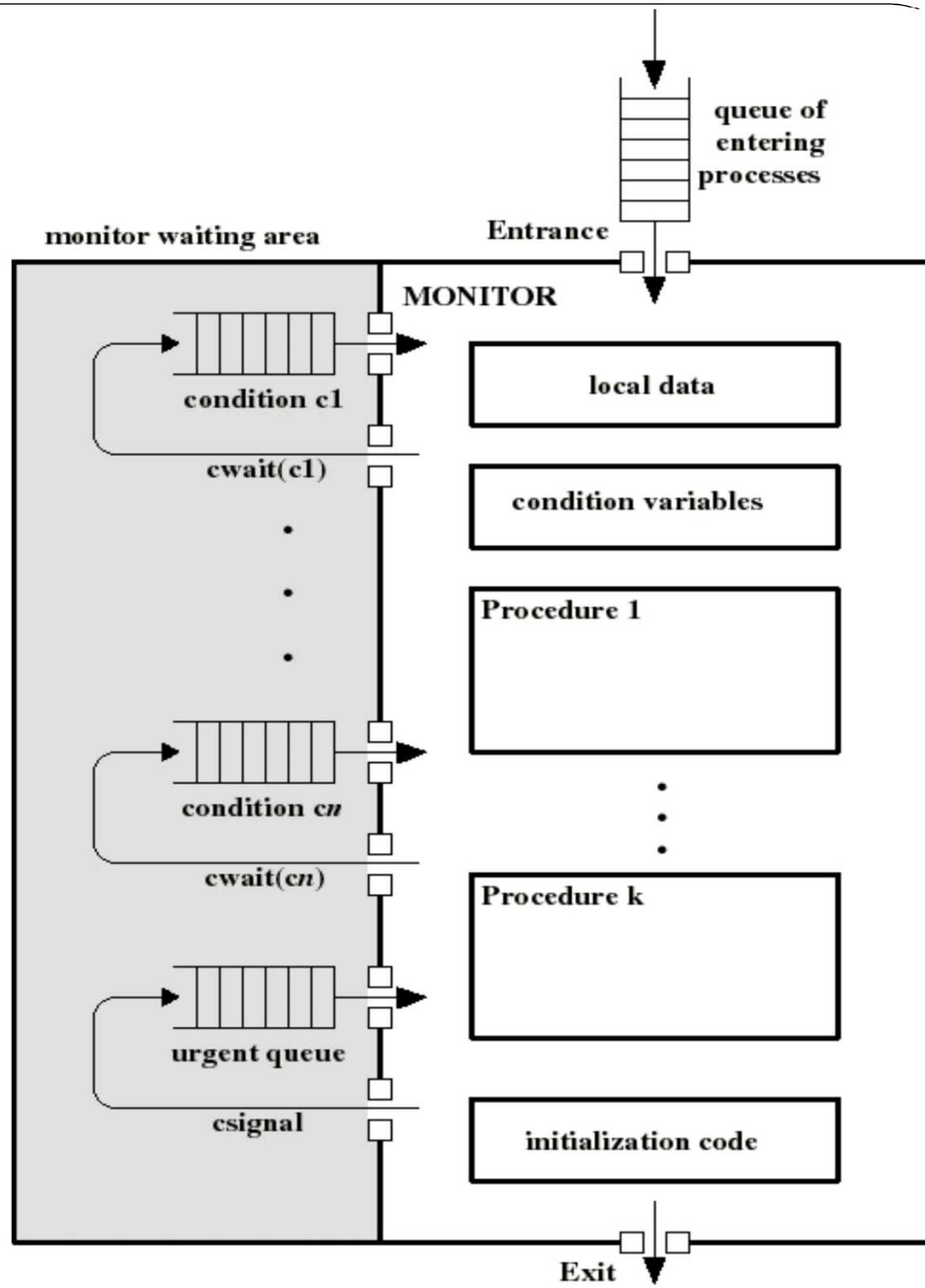
- Is a software module containing:
  - one or more procedures
  - an initialization sequence
  - local shared data variables
- Characteristics:
  - Local shared variables accessible only by monitor's procedures
  - a process enters the monitor by invoking one of it's procedures
  - only one process can be in the monitor at any one time
- The monitor ensures mutual exclusion
  - no need to program this constraint explicitly
- Shared data are protected by placing them in the monitor
  - The monitor locks the shared data on process entry

Condition Variables

- Process synchronization is done using condition variables, which represent conditions a process may need to wait for before executing in the monitor
- condition x, y;
- Local to the monitor (accessible only within the monitor)
- Can be accessed and changed only by two functions:
  - x.wait(): blocks execution of the calling process on condition x
    - the process can resume execution only if another process executes x.signal()
  - x.signal(): resume execution of some process blocked on condition x.
    - If several such processes exists: choose any one
    - If no such process exists: do nothing

Monitors

- Awaiting processes are either in the entrance queue or in a condition queue
- A process puts itself into condition queue cn by issuing cn.wait()
- cn.signal() brings into the monitor one process in condition cn queue
- *signal-and-wait* and *signal-and-continue*



Producer Consumer using Monitor

- Two types of processes:
  - producers
  - consumers
- Synchronization is now confined within the monitor
- append(.) and take(.) are procedures within the monitor: are the only means by which P/C can access the buffer
- If these procedures are correct, synchronization will be correct for all participating processes

```
Producer:
while(TRUE){
        produce item;
        append(item);
}
Consumer:
while(TRUE){
        item=take();
        consume item;
}
```

Monitor for the Bounded P/C Problem

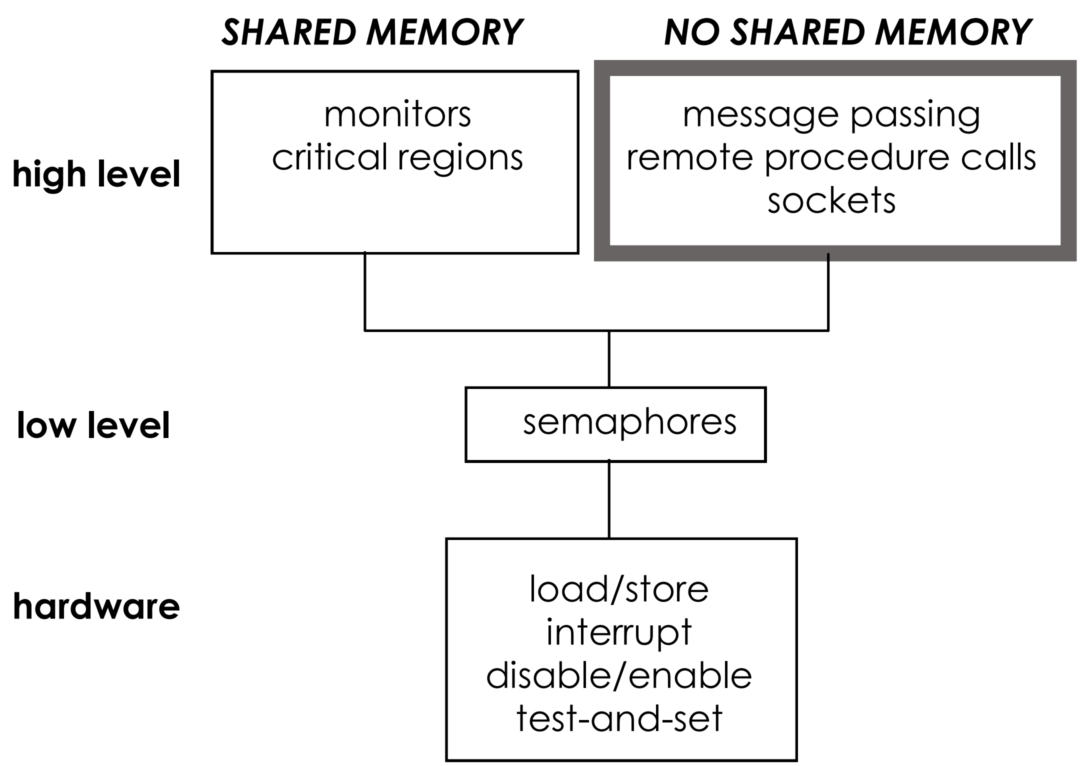
- Buffer:
  - *buffer:* array[0..k-1] of items;
- Buffer pointers and counts:
  - *nextin:* points to next item to be appended
- *nextout:* points to next item to be taken
- *count:* holds the number of items in the buffer
- Condition variables:
  - *notfull:* notfull.signal() indicates that the buffer is not full
  - *notempty:* notempty.signal() indicates that the buffer is not empty

Monitor for bounded buffer problem

```
Monitor boundedbuffer {
  Item buffer[k];
  integer nextin, nextout, count;
  condition notfull, notempty;
  Append(v){
    if (count==k)
      notfull.wait();
    buffer[nextin] = v;
    nextin = (nextin+1) mod k;
    count++;
    notempty.signal();
  }
  initialization_code(){
    nextin=0; nextout=0; count=0;
  }
```

```
   Item Take(){
     if (count==0)
       notempty.wait();
     v = buffer[nextout];
     nextout =
          (nextout+1) mod k;
     count--;
     notfull.signal();
     return v;
   }
}
```

Synchronization Primitives — Summary



References

- Operating System Concepts (Silberschatz, 8th edition) Chapter 6