OPERATING SYSTEMS

Week 5

Much of the material on these slides comes from the recommended textbook by William Stallings

Detailed content

Weekly program

- ✓ Week 1 – Operating System Overview
- ✓ Week 2 – Processes and Threads
- ✓ Week 3 – Scheduling
- ✓ Week 4 – Real-time System Scheduling and Multiprocessor Scheduling
- ❑ **Week 5 – Concurrency: Mutual Exclusion and Synchronization**
- ❑ Week 6 – Concurrency: Deadlock and Starvation
- ❑ Week 7 – Memory Management I
- ❑ Week 8 – Memory Management II
- ❑ Week 9 – Disk and I/O Scheduling
- ❑ Week 10 – File Management
- ❑ Week 11 – Security and Protection
- ❑ Week 12 – Revision of the course
- ❑ Week 13 – Extra revision (if needed)

Key Concepts From Last Lecture

- Synchronization granularity refers to the frequency of synchronization between the process in the system
  - Independent parallelism
  - Coarse and very coarse grained parallelism
  - Medium grained parallelism
  - Fine grained parallelism
- Assignment of processes to processor can be done according to various strategies:
  - Static VS Dynamic
  - Master-Slave VS Peer architecture
- When many processors are available, it is no longer paramount that every single processor be busy as much as possible
  - Rather keeping some processor free is more beneficial

Key Concepts From Last Lecture

- Process scheduling in multiprocessors does not receive much benefit with sophisticated scheduling algorithms.
  - Simple FCFS is a better choice considering the tradeoff between benefit and cost
- Thread scheduling models significantly impact performance.
- Approaches to thread scheduling
  - Load Sharing
  - Gang Scheduling
  - Dedicated processor assignment
  - Dynamic Scheduling

Key Concepts From Last Week

- RT processes are executed in connection with some process or event external to the computer system
- Therefore, RT scheduling must meet one or more deadlines
- Traditional criteria for scheduling does not apply
- RT operating systems are characterized by
  - Determinism, Responsiveness, User Control, Reliability, Fail-soft operation
- Both static and dynamic scheduling is possible in RT system
  - Static
    - Table drive approach VS Priority driven preemptive approach
  - Dynamic
    - Planning base approach VS Best effort approach

Key Concepts From Last Week

- Key factor is meeting the deadlines not processor utilization or etc.
- Algorithms that rely heavily on preemption and on reacting to relative deadlines are appropriate for this purpose
- Two prominent scheduling algorithms are
  - Earliest Deadline First (EDF)
  - Rate Monitoring Scheduling (RMS)
- Priority Inversion and solutions

Week 05 Lecture

Concurrency: Mutual Exclusion and Synchronization

- ❑ Concurrency
- ❑ OS concerns in concurrency
- ❑ Process interaction
- ❑ Requirement of mutual exclusion
- ❑ Mutual exclusion with hardware support
- ❑ Semaphore and its mechanism
- ❑ Mutual exclusion by semaphore
- ❑ Producer/Consumer problems
- ❑ Semaphore implementation
- ❑ Monitors
- ❑ Readers/Writers Problems

Videos to watch before lecture

Multiple Processes

- Operating System design is concerned with the management of processes and threads:
  - Multiprogramming
  - Multiprocessing
  - Distributed Processing

Concurrency Arises in Three Different Contexts:

Multiple Applications

- invented to allow processing time to be shared among active applications

Structured Applications

- extension of modular design and structured programming

Operating System Structure

- OS themselves implemented as a set of processes or threads

Concurrency: key terms

- **Atomic Operation** A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.

- **Critical Section** A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.

- **Deadlock** A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.

Concurrency: key terms

- **Livelock** A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.

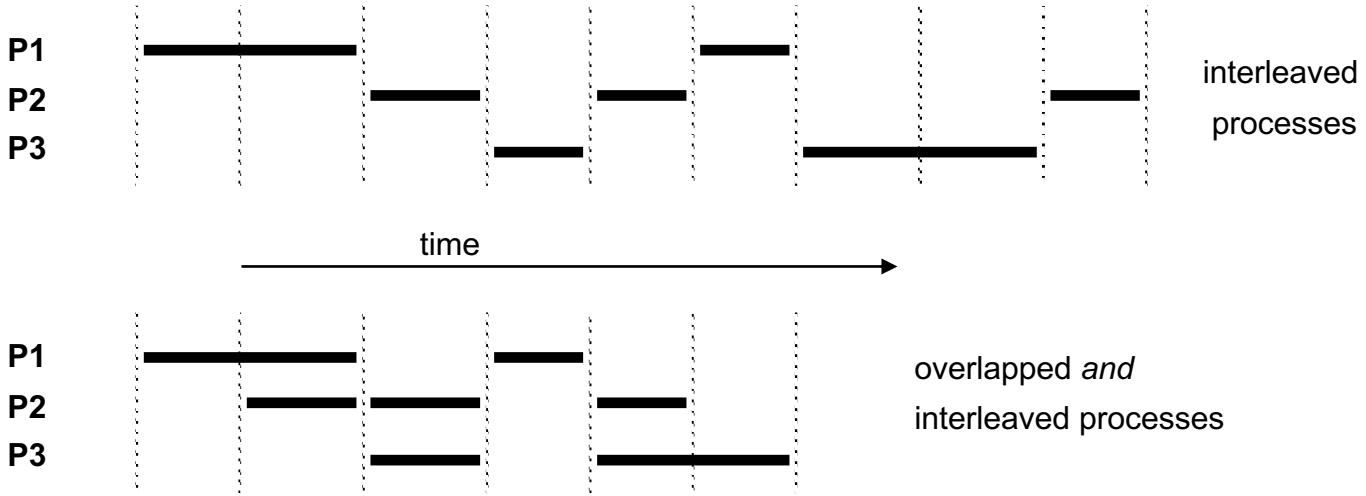
- **Mutual exclusion** The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.

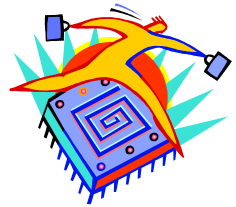
- **Race condition** A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.

- **Starvation** A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

Concurrency

- **Concurrency** is the concept of more than one process (or thread) operating at the same time.
- **Two kinds** of concurrency:





- The basic characteristic of multiprogramming concurrency is that the relative execution speed of processes cannot be predicted. Other processes affect it.

Difficulties of Concurrency

- Sharing of global resources
- Difficult for the OS to manage the allocation of resources optimally
- Difficult to locate programming errors as results are not deterministic and reproducible

Problems from shared resource: example

P1

P2

```
Void echo(){
      chin = getchar();
      chout = chin;
      putchar(chout);
}
```

P1

```
Void echo(){
      chin = getchar();
      chout = chin;
      putchar(chout);
}
```

chin=x

P2

```
Void echo(){
      chin = getchar();
      chout = chin;
      putchar(chout);
}
```

chin=y

P1

```
Void echo(){
      chin = getchar();
      chout = chin;
      putchar(chout);
}
```

chin=y

Protecting shared resource: example

P1

P2

```
Void echo(){
      chin = getchar();
      chout = chin;
      putchar(chout);
}
```

P1

```
Void echo(){
      chin = getchar();
      chout = chin;
      putchar(chout);
}
```

chin=x

P2

```
Void echo(){
      chin = getchar();
      chout = chin;
      putchar(chout);
}
```

P2 blocked entering echo()

P1

```
Void echo(){
      chin = getchar();
      chout = chin;
      putchar(chout);
}
```

chin=x

P2

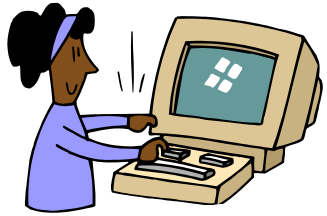
```
Void echo(){
      chin = getchar();
      chout = chin;
      putchar(chout);
}
```

Race conditions

- Occurs when multiple processes or threads read and write data items
- The final result depends on the order of execution
  - the “loser” of the race is the process that updates last and will determine the final value of the variable
- Example 1: P1 and P2 shares a variable ‘a’
  - P1: `a=1`
  - P2: `a=2`
- Example 2: P3 and P4 shares variable ‘b’ and ‘c’
  - Initial values: ‘`b=1`’ and ‘`c=2`’
  - P3: `b=b+c`
  - P4: `c=b+c`

Operating system concerns

- Design and management issues raised by the existence of concurrency:
  - The OS must:
    - be able to keep track of various processes
    - allocate and de-allocate resources for each active process
    - protect the data and physical resources of each process against interference by other processes
    - **ensure that the processes and outputs are independent of the processing speed**



Process interaction

- Processes may interact in three ways:
  - **unaware** of each other but in **competition** for resources,
  - **indirectly aware** of each other by **sharing** some data (e.g. sharing an I/O buffer),
  - **directly aware** of each other by **communicating** for cooperation.

- We now look at each kind of interaction in turn.

Competition

- Many processes may **compete** for access to a resource in the course of their execution:
  - each process is **unaware** of the other;
  - each process should be **unaffected** by the execution of the other(s);
  - **only one** process can safely access the resource at a time, others must wait.

- This situation raises three **control problems**:
  1. Mutual exclusion
  2. Deadlock
  3. Starvation

Sharing and Integrity

Cooperation by data sharing

- Processes may interact **indirectly** by sharing data (such as shared variables, shared files, or shared databases). Such processes may not be **explicitly** aware of each other.
- Since the shared data is held on resources (devices, memory), the cooperating processes **compete** for resources; thus the control problems of **mutual exclusion**, **deadlock**, and **starvation** still exist.
- **More importantly**, they must **cooperate** to ensure the ***integrity*** and ***coherence*** of the shared data.

Sharing and Integrity

P1
```
a:=a+1;
b:=b+1;
```

P2
```
b:=2b;
a:=2a;
```

Concurrent execution
```
a:=a+1;
b:=2b;
b:=b+1;
a:=2a;
```

- For example, suppose that **P1** and **P2** share variables *a* and *b,* and the relation *a=b* must be maintained.
  - note that both **P1** and **P2** preserve the relation *a=b* .
  - however, if statements of **P1** and **P2** are **interleaved**, then the relation is not preserved. In this case, the sequence within each process must be maintained.
- The **integrity** of the shared data can be guaranteed through the use of the functions
  - *entercritical(X),*
  - *exitcritical(X)*
- The parameter *X* may be **data** (file, variable, or database etc) as well as a **resource** or just a critical section **identifier**.

Communication

- Processes need a way to **synchronise** or **coordinate activities**; that is, to **communicate**.
- Communication between processes may be disciplined by a **message passing protocol**.
- Primitives for message passing may be provided by the OS, or as part of a programming language.
- The message passing need not involve shared data or resources, so mutual exclusion is not a big problem for communication.
- However:
  - **Deadlock** may occur, for example, if two processes are blocked by waiting to receive a message from each other.
  - **Starvation** may occur:

P1 is repeatedly trying to communicate with **either P2** or **P3**.

P1 and **P2** exchange messages repeatedly, while **P3** remains blocked waiting for a message from **P1**.

Mutual Exclusion Mechanism


```
        PROCESS 1 */

void P1
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

```
        /* PROCESS 2 */

void P2
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

. . .

```
        /* PROCESS n */

void Pn
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

- Mechanisms for mutual exclusion can be provided by two abstract functions
  - entercritical(R) and
  - exitcritical(R).
- These functions may be implemented by the processes themselves, hardware, or the OS; they are used by the processes.
- The important questions are:
  - **how** are they **implemented**?
  - **how** are they **used**?

Requirements for mutual exclusion

- Any facility to provide support for mutual exclusion should meet the following:
  1. **Mutual exclusion** must be **enforced**: only one process at a time is allowed into its critical section.
  2. A process that **halts** in its noncritical section must do so **without interfering** with other processes.
  3. A process requiring access to a critical section must not be **delayed indefinitely** - **No deadlock, No starvation**
  4. When no process is in a critical section, any process which requests entry to its critical section **must be permitted** to do so without delay.
  5. **No assumptions** are made about the relative process speeds or number of processes.
  6. A process remains inside its critical section for a **bounded time** only.

Note: deadlock and starvation are often discussed under the heading "mutual exclusion". The three concepts overlap a little.

MUTUAL EXCLUSION APPROACHES

- **Responsibility** for mutual exclusion **may** be vested in:
  - The **processes**
    - The software approach
  - The **hardware**
    - Use special purpose atomic (uninterruptable) instructions
  - The **operating system**
    - Use semaphores
    - Use messages

- In any case, we can assume that the hardware provides **elementary mutual exclusion for memory accesses**.
  - Simultaneous access to the same location in main memory is prevented by the hardware.
  - Requests are serialised and granted in an unspecified order.

Hardware support

- **Interrupt disabling**
  - One can ensure mutual exclusion on a **uniprocessor** by **disabling interrupts** during a critical section.
  - This ensures mutual exclusion, but degrades performance because it limits the ability of the OS to interleave programs.

```
While (true){
          /* disable interrupts */;
          /* critical section */;
          /* enable interrupts */;
          /* reminder */;
}
```

- Interrupt disabling does not work on a multiprocessor.

Hardware support

- **Special machine instructions**
  - Remember that access to a memory location excludes any other access to that same location.
  - We can build general mutual exclusion methods by adding new **atomic instructions**.
  - **Atomic instructions cannot** be **interrupted**.

Mutual Exclusion: Hardware Support

- Compare&Swap Instruction
  - also called a “compare and exchange instruction”
  - a **compare** is made between a memory value and a test value
  - if the values are the same a **swap** occurs
  - carried out atomically

```
int compare_and_swap (int *word, int testval, int newval){
          int oldval;
          oldval = *word;
          if (oldval == testval) *word = newval;
          return oldval;
}
```

Compare and swap instruction


```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
  while (true) {
    while (compare_and_swap(&bolt, 0, 1) == 1)
        /* do nothing */;
    /* critical section */;
    bolt = 0;
    /* remainder */;
  }
}
void main()
{
  bolt = 0;
  parbegin (P(1), P(2), . . . ,P(n));

}
```

Compare and swap instruction


```
constant int n = 2;
int bolt;

void main(){
        bolt = 0;
        parbegin(P1(1), P2(2));
}
```

```
void P1 (int 1){
while (true){
    while (compare_and_swap(&bolt,0,1)==0)
        /*do nothing */;
    /*ctitical section */;
    bolt = 0;
    /*remainder */;
  }
}
```

```
void P2 (int 2){
while (true){
    while (compare_and_swap(&bolt,0,1)==0)
        /*do nothing */;
    /*ctitical section */;
    bolt = 0;
    /*remainder */;
  }
}
```

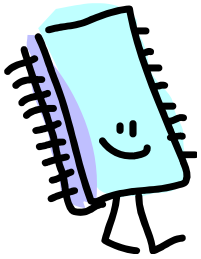
Mutual Exclusion: Hardware Support

- Exchange Instruction
  - Exchanges the contents of a register with that of a memory location
  - carried out atomically

```
void exchange (int *register, int *memory){
          int temp;
          temp = *memory;
          *memory = * register;
          *register = temp;
}
```

Exchange instruction


```
/* program mutualexclusion */
int const n = /* number of processes*/;
int bolt;
void P(int i)
{
  while (true) {
     int keyi = 1;
     do exchange (&keyi, &bolt)
     while (keyi != 0);
     /* critical section */;
     bolt = 0;
     /* remainder */;
  }
}
void main()
{
  bolt = 0;
  parbegin (P(1), P(2), . . ., P(n));
}
```



Hardware methods

- The hardware approach has some advantages:
  - It is applicable to any **number of processors** sharing main memory.
  - It is **simple** and mutual exclusion is easy to verify.
  - It can support **multiple critical sections**, each dependent on its own variable.

Hardware methods

- But there are some serious disadvantages:
  - **Busy-waiting** is employed; that is, while a process is waiting for access, it consumes processor time.
  - **Starvation** is possible: when a process leaves a critical section and more than one process is waiting, the selection of the next process is arbitrary and may indefinitely deny access to some process.
  - **Deadlock** is possible. Suppose that P2 has high priority and P1 has low priority. Suppose that P1 goes critical. Suppose that P1 is interrupted and gives control to P2, then P2 attempts to access the resource controlled by P1. Then P2 will wait for the resource, but P1 will never relinquish it because P1 is never dispatched (since P2 has high priority).

OS and PL approaches to concurrency

- Operating systems and programming languages offer several mechanisms to support concurrency
  - Semaphore
  - Monitor
  - Message Passing

Semaphores

- A set of processes can cooperate using simple signals called **semaphores**.
  - A process **receives** a signal via a semaphore ***s*** using an atomic procedure ***semWait(s)***; if the corresponding signal has not yet been sent, then the process is blocked until transmission takes place.
  - A process **transmits** a signal via a semaphore ***s*** by executing an atomic procedure ***semSignal(s)***.

Semaphores

- More precisely, a semaphore consists of an integer variable with three operations:
  - ***initialise***: **sets** the semaphore to a non-negative value.
  - ***wait***: **decrements** the semaphore value. If the value becomes negative then the process becomes blocked.
  - ***signal***: **increments** the semaphore value. If the value is non-positive, then a process blocked by a wait operation becomes unblocked.
- These three operations are the **only ways** to inspect or manipulate semaphores.
- The operations are atomic, that is, not interruptable.
- A **binary semaphore** is similar, but only uses values 0 and 1.

A Definition of Semaphore Primitives


```
struct semaphore {
     int count;
     queueType queue;
};
void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0) {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
     }
}
```

- `semWait/semSingal` are atomic

Consequences

There is no way to know before a process decrements a semaphore whether it will block or not

There is no way to know which process will continue immediately on a uniprocessor system when two processes are running concurrently

You don't know whether another process is waiting so the number of unblocked processes may be zero or one

A Definition of Binary Semaphore Primitives


```
struct binary_semaphore {
     enum {zero, one} value;
     queueType queue;
};
void semWaitB(binary_semaphore s)
{
     if (s.value == one)
          s.value = zero;
     else {
            /* place this process in s.queue */;
            /* block this process */;
     }
}
void semSignalB(semaphore s)
{
     if (s.queue is empty())
          s.value = one;
     else {
            /* remove a process P from s.queue */;
            /* place process P on ready list */;
     }
}
```

Strong/Weak semaphores

- A queue is used to hold processes waiting on the semaphore

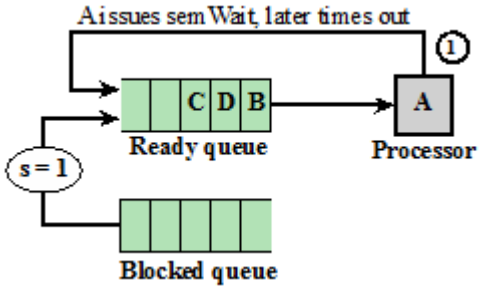
Strong Semaphores

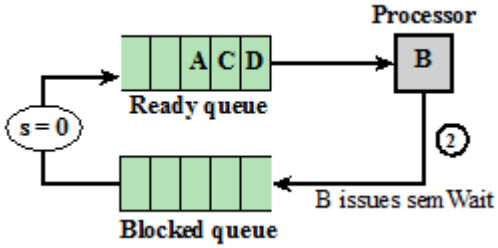
- the process that has been blocked the longest is released from the queue first (FIFO)
- Guarantees freedom from starvation

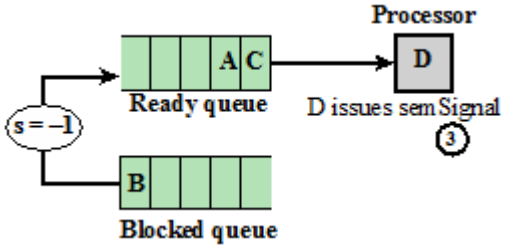
Weak Semaphores

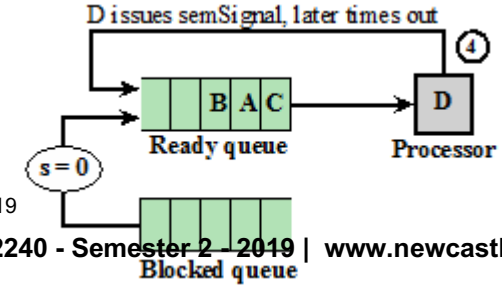
- the order in which processes are removed from the queue is not specified
- Does not guarantee freedom from starvation

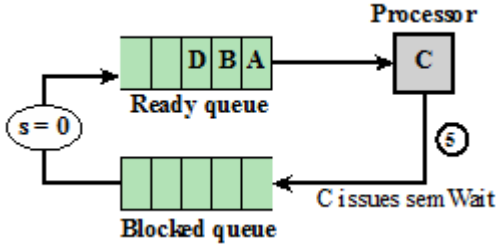
Semaphore Mechanism

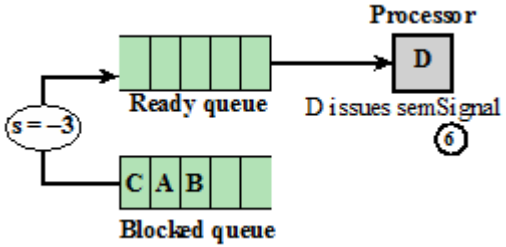


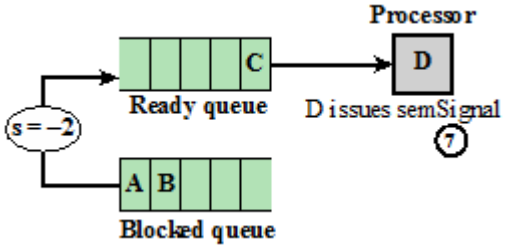












Mutual Exclusion Using Semaphores


```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
     while (true) {
          semWait(s);
          /* critical section   */;
          semSignal(s);
          /* remainder   */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . ., P(n));
}
```

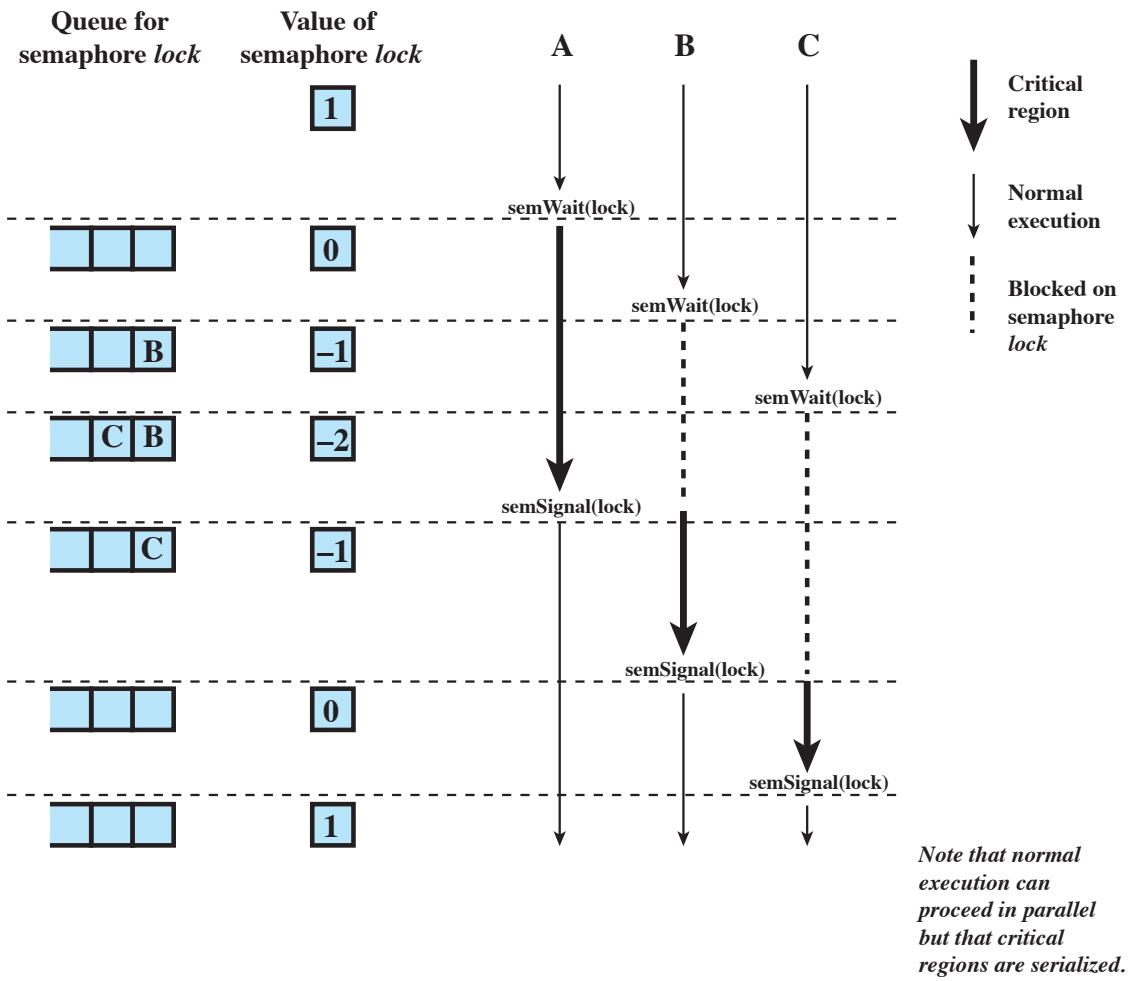
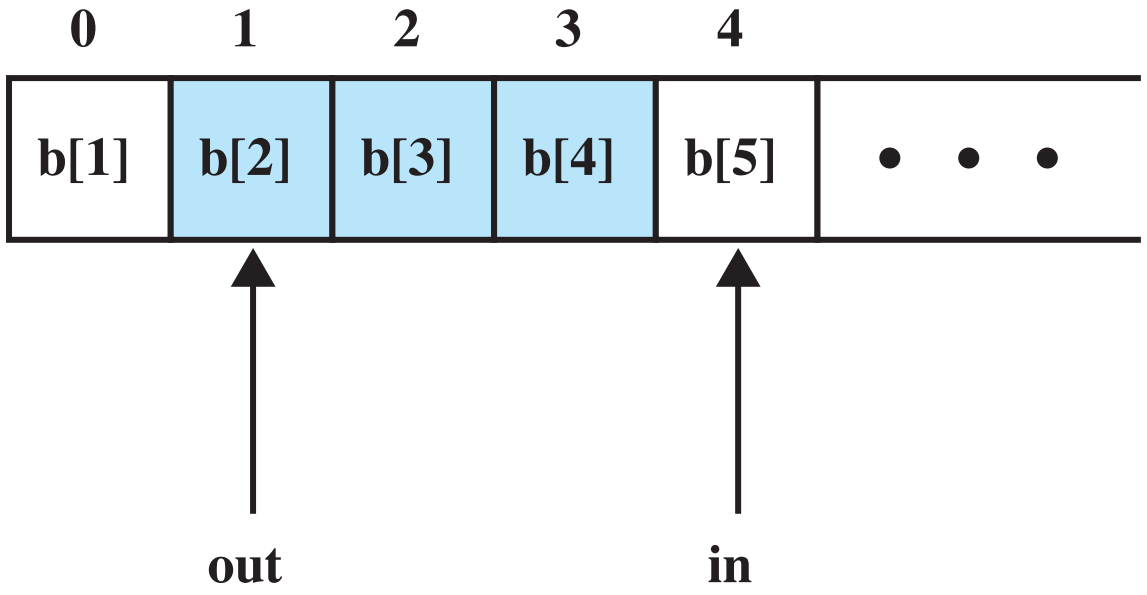


Figure 5.7 Processes Accessing Shared Data Protected by a Semaphore

Producer/Consumer Problem

General Statement:	one or more producers are generating data and placing these in a buffer
	a single consumer is taking items out of the buffer one at a time
	only one producer or consumer may access the buffer at any one time
The Problem:	ensure that the producer can't add data into full buffer and consumer can't remove data from an empty buffer

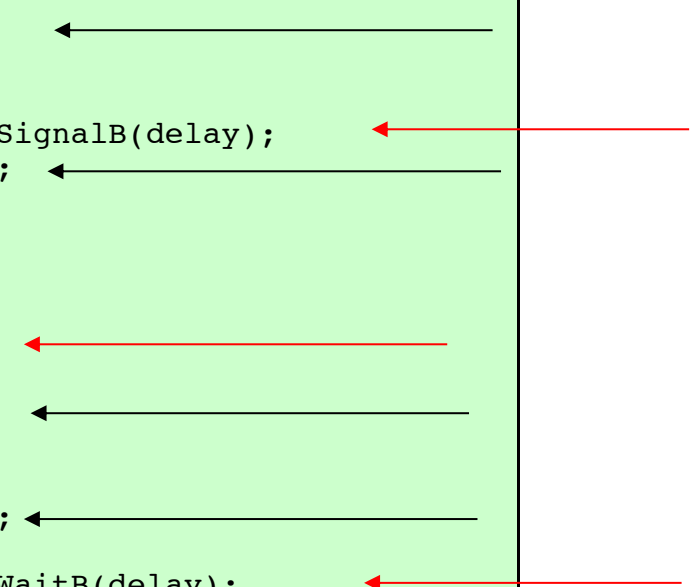


Note: shaded area indicates portion of buffer that is occupied

Figure 5.8 Infinite Buffer for the Producer/Consumer Problem

Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     semWaitB(delay);
     while (true) {
          semWaitB(s);
          take();
          n--;
          semSignalB(s);
          consume();
          if (n==0) semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```



Possible Scenario for the Program

	Producer	Consumer	s	n	Delay
1			1	0	0
2	semWaitB(s)				
3	n++				
4	**if** (n==1) (semSignalB(delay))				
5	semSignalB(s)				
6		semWaitB(delay)			
7		semWaitB(s)			
8		n--			
9		semSignalB(s)			
10	semWaitB(s)				
11	n++				
12	**if** (n==1) (semSignalB(delay))				
13	semSignalB(s)				
14		**if** (n==0) (semWaitB(delay))			
15		semWaitB(s)			
16		n--			
17		semSignalB(s)			
18		**if** (n==0) (semWaitB(delay))			
19		semWaitB(s)			
20		n--			
21		semSignalB(s)			

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     semWaitB(delay);
     while (true) {
          semWaitB(s);
          take();
          n--;
          semSignalB(s);
          consume();
          if (n==0) semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

Note: White areas represent the critical section controlled by semaphore s.

Possible Scenario for the Program

	Producer	Consumer	s	n	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	**if** (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	**if** (n==1) (semSignalB(delay))		0	1	1
13	semSignalB(s)		1	1	1
14		**if** (n==0) (semWaitB(delay))	1	1	1
15		semWaitB(s)	0	1	1
16		n--	0	0	1
17		semSignalB(s)	1	0	1
18		**if** (n==0) (semWaitB(delay))	1	0	0
19		semWaitB(s)	0	0	0
20		n--	0	–1	0
21		semSignalB(s)	1	–1	0

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     semWaitB(delay);
     while (true) {
          semWaitB(s);
          take();
          n--;
          semSignalB(s);
          consume();
          if (n==0) semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

Note: White areas represent the critical section controlled by semaphore s.

Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     semWaitB(delay);
     while (true) {
          semWaitB(s);
          take();
          n--;
          semSignalB(s);
          consume();
          if (n==0) semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

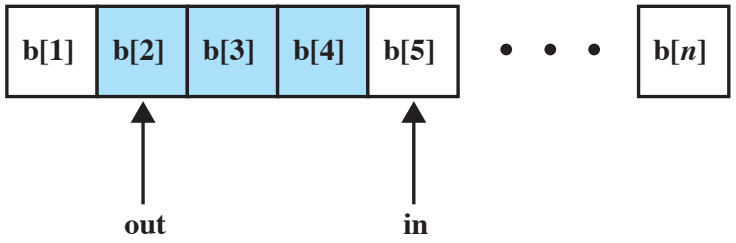

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     int m; /* a local variable */
     semWaitB(delay);
     while (true)  {
          semWaitB(s);
          take();
          n--;
          m = n;
          semSignalB(s);
          consume();
          if (m==0) semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

A Solution to the Infinite-Buffer Producer/Consumer Problem Using Semaphores

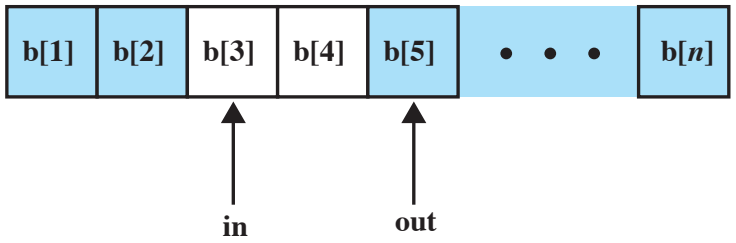
```
/* program  producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
     while (true) {
          produce();
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     int m; /* a local variable */
     semWaitB(delay);
     while (true)  {
          semWaitB(s);
          take();
          n--;
          m = n;
          semSignalB(s);
          consume();
          if (m==0) semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

Producer/Consumer problem with finite buffer



(a)



(b)

Figure 5.12 Finite Circular Buffer for the Producer/Consumer Problem

A Solution to the Bounded-Buffer Producer/Consumer Problem Using Semaphores


```
/* program  producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
     while (true) {
          produce();
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
void producer()
{
     while (true) {
          ?
     }
}
void consumer()
{
     while (true) {
          ?
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

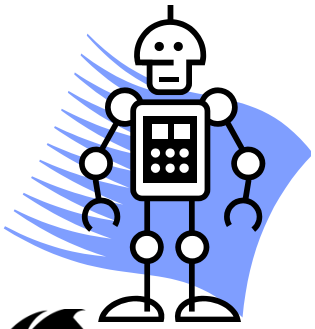
A Solution to the Bounded-Buffer Producer/Consumer Problem Using Semaphores


```
/* program  producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
     while (true) {
          produce();
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
void producer()
{
     while (true) {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Implementation of Semaphores

- Imperative that the `semWait` and `semSignal` operations be implemented as atomic primitives
- Can be implemented in hardware or firmware?
- Software schemes such as Dekker's or Peterson's algorithms can be used
- Use one of the hardware-supported schemes for mutual exclusion



Two Possible Implementations of Semaphores


```
semWait(s)
{
    while (compare_and_swap(s.flag, 0 , 1) == 1)
        /* do nothing */;
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue*/;
        /* block this process (must also set s.flag to 0)
*/;
    }
    s.flag = 0;
}

semSignal(s)
{
    while (compare_and_swap(s.flag, 0 , 1) == 1)
          /* do nothing */;
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
    s.flag = 0;
}
```

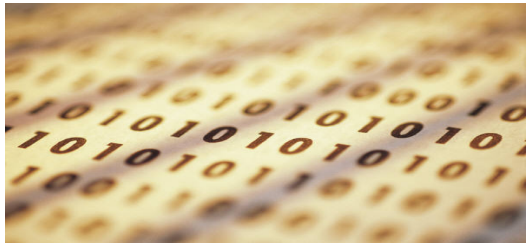
(a) Compare and Swap Instruction

```
semWait(s)
{
    inhibit interrupts;
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process and allow interrupts */;
    }
    else
       allow interrupts;
}

semSignal(s)
{
    inhibit interrupts;
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
    allow interrupts;
}
```

(b) Interrupts

Monitors



- Semaphore is powerful and flexible tool
  - Difficult to use and manage
- Monitor is a programming language construct that provides equivalent functionality to that of semaphores and is easier to control
- Implemented in a number of programming languages
    - including Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, and Java
- Has also been implemented as a program library

Monitor Characteristics

Monitor is a software module consisting of one or more procedures, an initialization sequence, and local data

Local data variables are accessible only by the monitor's procedures and not by any external procedure

Process enters monitor by invoking one of its procedures

Only one process may be executing in the monitor at a time

Synchronization

- Achieved by the use of **condition variables** that are contained within the monitor and accessible only within the monitor
    - Condition variables are operated on by two functions:
        - » **cwait(c)**: suspend execution of the calling process on condition c
        - » **csignal(c)**: resume execution of some process blocked after a cwait on the same condition

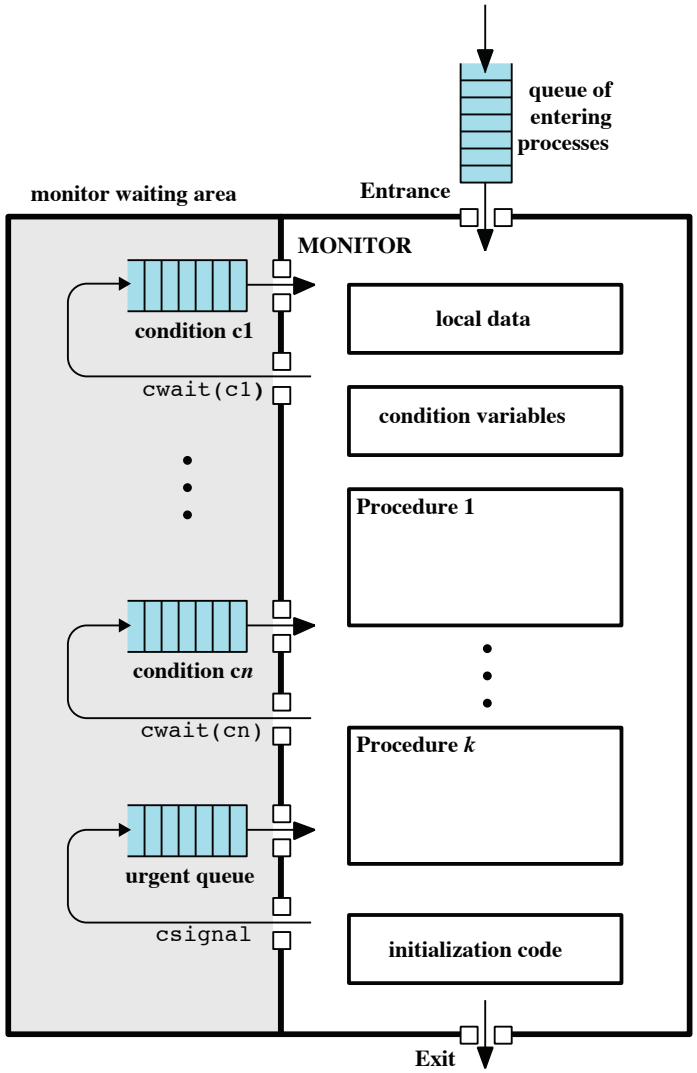


Figure 5.15 Structure of a Monitor

A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor


```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                    /* space for N items */
int nextin, nextout;                                  /* buffer pointers */
int count;                                  /* number of items in buffer */
cond notfull, notempty;          /* condition variables for synchronization */

void append (char x)
{
     if (count == N) cwait(notfull);      /* buffer is full; avoid overflow */
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;
     /* one more item in buffer */
     csignal(notempty);                       /* resume any waiting consumer */
}
void take (char x)
{
     if (count == 0) cwait(notempty);   /* buffer is empty; avoid underflow */
     x = buffer[nextout];
     nextout = (nextout + 1) % N;
     count--;                                    /* one fewer item in buffer */
     csignal(notfull);                        /* resume any waiting producer */
}
{                                                          /* monitor body */
     nextin = 0; nextout = 0; count = 0;          /* buffer initially empty */
}
```

```
void producer()
{
     char x;
     while (true) {
     produce(x);
     append(x);
     }
}
void consumer()
{
     char x;
     while (true) {
       take(x);
       consume(x);
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Monitor VS Semaphore

- Monitors
  - Mutual exclusion: enforced by the construct (monitor) itself
  - Synchronization: Programmers responsibility - to signal (cwait/csignal) appropriately
    - Synchronization is confined to monitor
    - Easy to verify and debug
- Semaphores
  - It is programmers responsibility to
    - Enforce mutual exclusion
    - Synchronization
    - Programming/debugging is difficult

Hoare VS Mesa Monitors

- What should happen when csignal() is called?
  - No waiting threads => the signaler continues and the signal is effectively lost (unlike what happens with semaphores).
  - If there is a waiting thread, one of the threads starts executing, others must wait
- **Hoare-style**: (most textbooks)
  - The thread that signals gives up the lock and the waiting thread gets the lock.
  - When the thread that was waiting and is now executing exits or waits again, it releases the lock back to the signalling thread.
- **Mesa-style**: (Nachos, Java, and most real operating systems)
  - The thread that signals keeps the lock (and thus the processor).
  - The waiting thread waits for the lock.

Bounded Buffer Monitor for Mesa Monitor


```
void append (char x)
{
     while(count == N) cwait(notfull);   /* buffer is full; avoid overflow */
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;                                  /* one more item in buffer */
     cnotify(notempty);                      /* notify any waiting consumer */
}

void take (char x)
{
     while(count == 0) cwait(notempty); /* buffer is empty; avoid underflow */
     x = buffer[nextout];
     nextout = (nextout + 1) % N;
     count--;                                  /* one fewer item in buffer */
     cnotify(notfull);                       /* notify any waiting producer */
}
```

Readers/Writers Problem

- A data area is shared among many processes
  - some processes only read the data area, (readers) and some only write to the data area (writers)
- Conditions that must be satisfied:
  1. any number of readers may simultaneously read the file
  2. only one writer at a time may write to the file
  3. if a writer is writing to the file, no reader may read it

A Solution to the Readers/Writers Problem Using Semaphores: Readers Have Priority


```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true) {
      semWait (x);
      readcount++;
      if (readcount == 1) semWait (wsem);
      semSignal (x);
      READUNIT();
      semWait (x);
      readcount--;
      if (readcount == 0) semSignal (wsem);
      semSignal (x);
    }
 }
void writer()
{
    while (true) {
      semWait (wsem);
      WRITEUNIT();
      semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
```

- **wsem**: enforce mutual exclusion
- **x**: assure the `readcount` is updated properly

A Solution to the Readers/Writers Problem Using Semaphores: Writers Have Priority


```
/*program readersandwriters*/
int  readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {
     semWait (z);
          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1) semWait (wsem);
               semSignal (x);
          semSignal (rsem);
     semSignal (z);
     READUNIT();
     semWait (x);
          readcount--;
          if (readcount == 0) semSignal (wsem);
     semSignal (x);
    }
}
void writer ()
{
    while (true) {
     semWait (y);
          writecount++;
          if (writecount == 1) semWait (rsem);
     semSignal (y);
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
     semWait (y);
          writecount--;
          if (writecount == 0) semSignal (rsem);
     semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

- **`wsem`**: enforce mutual exclusion
- **`x`**: assure the `readcount` is updated properly
- **`rsem`**: inhibits readers while at least on writer desiring to write
- **`y`**: assure the `writecount` is updated properly
- **`z`**: only one reader is allowed to queue on **rsem** others on **z**

A Solution to the Readers/Writers Problem Using Monitors:


```
Monitor ReadersNWriters {
  int WaitingWriters, WaitingReaders,NReaders, NWriters;
  Condition CanRead, CanWrite;

 Void BeginWrite()
  {
        if(NWriters == 1 || NReaders > 0)
        {
              ++WaitingWriters;
             wait(CanWrite);
             --WaitingWriters;
        }
        NWriters = 1;
  }
  Void EndWrite()
  {
         NWriters = 0;
         if(WaitingReaders)
               Signal(CanRead);
         else
               Signal(CanWrite);
  }
```

```
Void BeginRead()
  {
        if(NWriters == 1 || WaitingWriters > 0)
        {
              ++WaitingReaders;
              Wait(CanRead);
             --WaitingReaders;
        }
        ++NReaders;
        Signal(CanRead);
  }

  Void EndRead()
  {
         if(--NReaders == 0)
                 Signal(CanWrite);

  }


}
```

Summary

- Concurrency is a fundamental requirement in modern OS
- As multiple processes are executing concurrently (in multi-processor or uniprocessor system) issues of conflict resolution and cooperation arise
- Concurrent process may interact in a number of ways
  - Compete for resource (unware of other processes)
  - Cooperate by sharing (indirectly aware of other processes)
  - Cooperate by communicating (directly aware of other processes)
- Mutual exclusion is a condition in which only one of a set of concurrent processes is granted access to a non-sharable shared resource
- One approach to supporting mutual exclusion involves the use of special purpose machine instruction

Summary

- OS supports mutual exclusion by providing features
  - Semaphores/Monitors
  - Message Passing
- Semaphore is a special purpose signaling variable that can be operated using two primitives: `semSignal` and `semWait`
- Two types: general semaphore and binary semaphore – essentially have the same expressive power
- Semaphores can be implemented in software or with support from hardware
- Semaphores are powerful and felxible tools for enforcing mutual exclusion but difficult to manage and handle
  - Programmers are responsible to ensure mutual exclusion and synchronization

Summary

- Monitors provide equivalent functionality of semaphores but easier to use
  - Monitor itself ensures mutual exclusion and programmers are responsible to ensure synchronization
- Solution to two classic problems
  - Producer/Consumer problem
  - Readers/Writers problem

References

- **Operating Systems – Internal and Design Principles**
  - By William Stallings
- Chapter 5

Monitor VS Semaphore

- **NOT A DEFINITION – A PARTIAL ANALOGY**

- A monitor is like a public toilet. Only one person can enter at a time. They lock the door to prevent anyone else coming in, do their stuff, and then unlock it when they leave.

- A semaphore is like a bike hire place. They have a certain number of bikes. If you try and hire a bike and they have one free then you can take it, otherwise you must wait. When someone returns their bike then someone else can take it. If you have a bike then you can give it to someone else to return --- the bike hire place doesn't care who returns it, as long as they get their bike back.