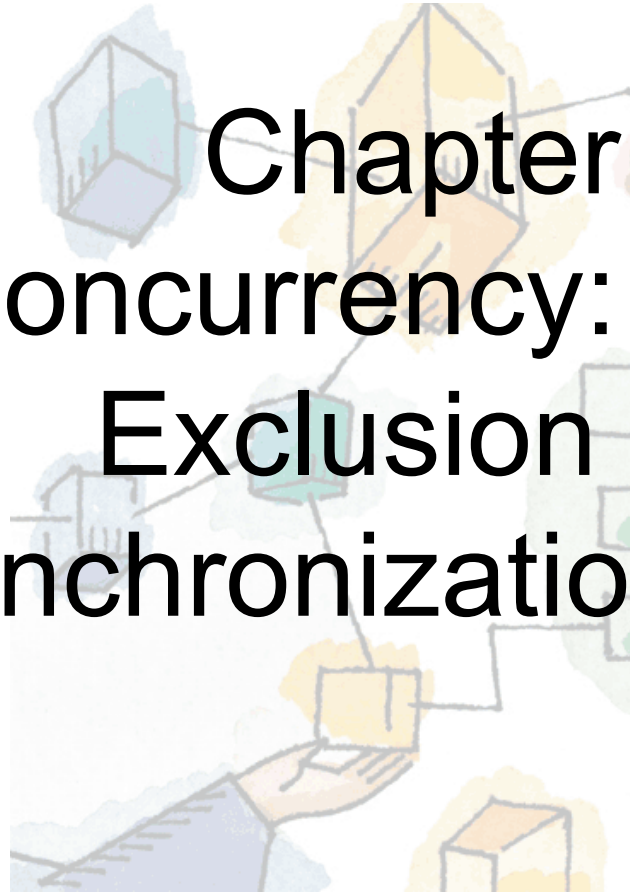
Operating Systems:
Internals and Design Principles
William Stallings



Chapter 5
Concurrency: Mutual Exclusion and Synchronization (cont.)

Outline

- OS synchronization mechanisms
  - Semaphore
    - Binary and general semaphores
    - Implementation
  - Producer/consumer problem
  - Bounded buffer
  - Monitor
    - Condition variables
  - Message passing
  - Reader/Writer problem

Common Concurrency Mechanisms

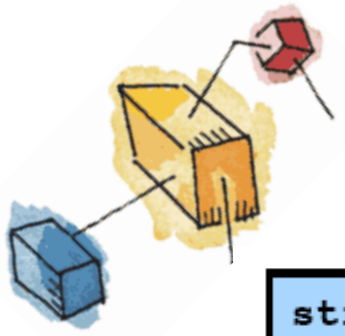
Semaphore	An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a **counting semaphore** or a **general semaphore**
Binary Semaphore	A semaphore that takes on only the values 0 and 1.
Mutex	Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).
Condition Variable	A data type that is used to block a process or thread until a particular condition is true.
Monitor	A programming language construct that encapsulates variables, access procedures and initialization code within an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are *critical sections*. A monitor may have a queue of processes that are waiting to access it.
Event Flags	A memory word used as a synchronization mechanism. Application code may associate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag. The thread is blocked until all of the required bits are set (AND) or until at least one of the bits is set (OR).
Mailboxes/Messages	A means for two processes to exchange information and that may be used for synchronization.
Spinlocks	Mutual exclusion mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability.

Semaphore

- A queue is used to hold processes waiting on the semaphore – eliminating busy-wait
- Semaphore:
  - An integer value used for signalling among processes.
- Only three operations may be performed on a semaphore, all of which are atomic:
  - Initialize the integer,
  - decrement the value (`semWait`),
  - increment the value (`semSignal`)

Semaphore

- The semaphore `s` is initially assigned a zero or positive value
- When a process issues a `semWait`, `s` is decremented
  - The process will be blocked if `s` goes negative
  - The negative value equals the number of blocked processes waiting to be unblocked
- Each `semSignal`, incrementing `s`, unblocks one of the waiting processes when `s` is negative



Semaphore Primitives

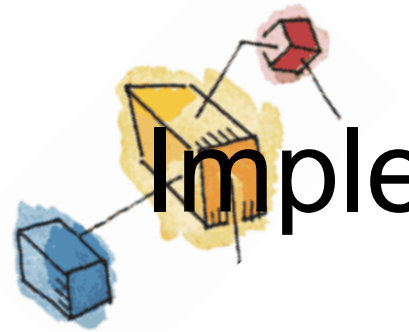
```
struct semaphore {
      int count;
      queueType queue;
};
void semWait(semaphore s)
{
      s.count--;
      if (s.count < 0) {
            /* place this process in s.queue */;
            /* block this process */;
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <=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

Figure 5.3 A Definition of Semaphore Primitives

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 */;
     }
}
```

Figure 5.4 A Definition of Binary Semaphore Primitives



Implementation of Semaphores

- the semWait and semSignal operations themselves are critical sections and thus must be implemented as atomic primitives
- Use one of the hardware-supported schemes for mutual exclusion

Strong/Weak Semaphores

- A queue is used to hold processes waiting on the semaphore – eliminating busy-wait
  - In what order are processes removed from the queue?

Strong Semaphores

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

Weak Semaphores

- the order in which processes are removed from the queue is not specified

Mutual Exclusion Using Semaphores

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

Figure 5.6 Mutual Exclusion Using Semaphores

Mutual Exclusion Using Semaphores

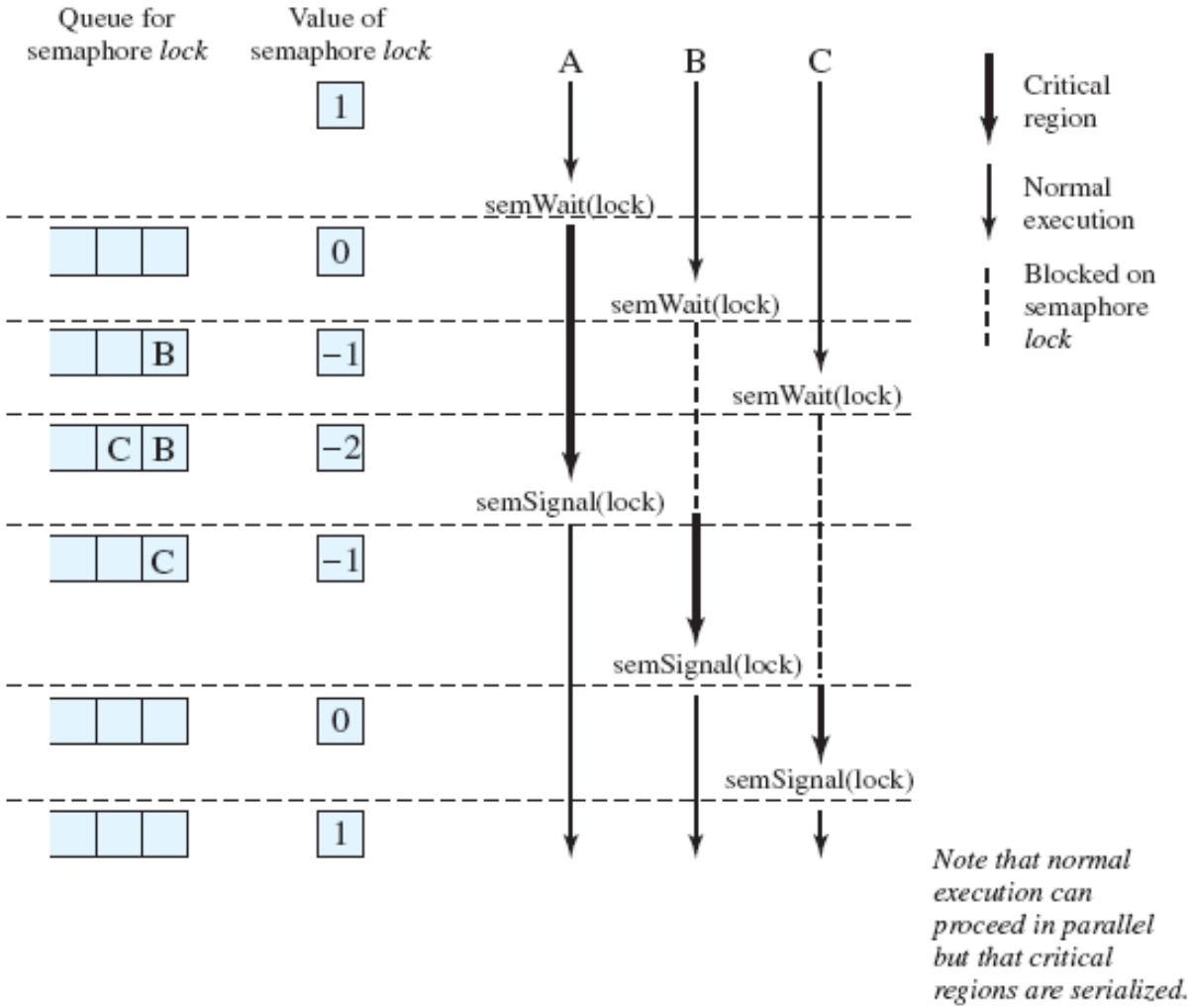


Figure 5.7 Processes Accessing Shared Data Protected by a Semaphore

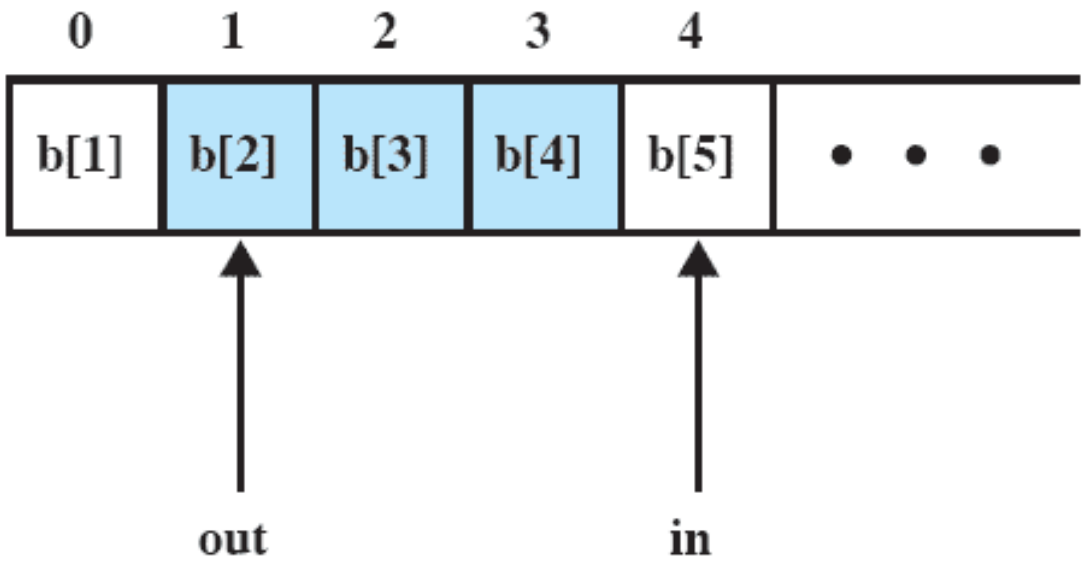
Synchronization Using Semaphores

```
Semaphore s = 0;        /* Initialize s to 0 */
Proc_0() {                  proc_1() {
  . . .                       . . .
  S1;                         semWait (s);
  semSignal (s);              S2;
  . . .                       . . .
}                           }
```

Producer/Consumer Problem

- General Situation:
  - One or more producers are generating data and placing these in a buffer
  - One or more consumers are taking items out of the buffer one at 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 empty buffer

Buffer Structure



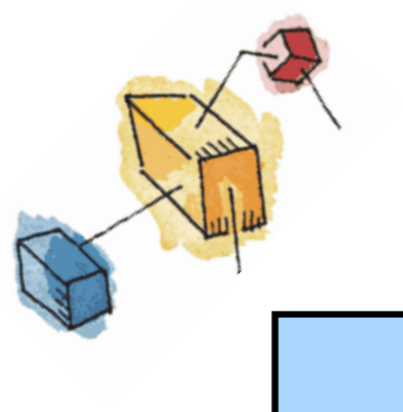
Note: shaded area indicates portion of buffer that is occupied

Figure 5.8 Infinite Buffer for the Producer/Consumer Problem

Binary Semaphores: Incorrect Solution

```
/* 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);
}
```

A producer can update n before if statement!



Binary Semaphores: Correct Solution

```
/* 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);
}
```

Can cause starvation!

Producers continue to append & make n > 1 always!

C1: delay = 0 **Starved here!**

C0: m > 0

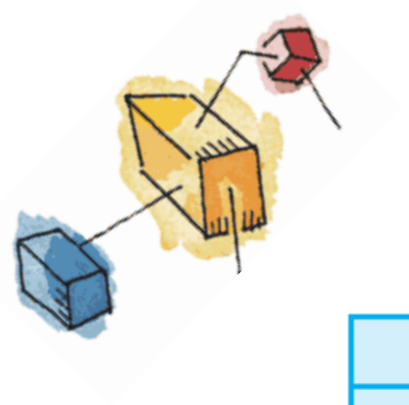
General Semaphores

Ensure consumer not to take items when the buffer is empty

For mutual exclusion

```
/* 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);
}
```

Figure 5.11 A Solution to the Infinite-Buffer Producer/Consumer Problem Using Semaphores



Bounded Buffer

Block on:	Unblock on:
Producer: insert in full buffer	Consumer: item inserted
Consumer: remove from empty buffer	Producer: item removed

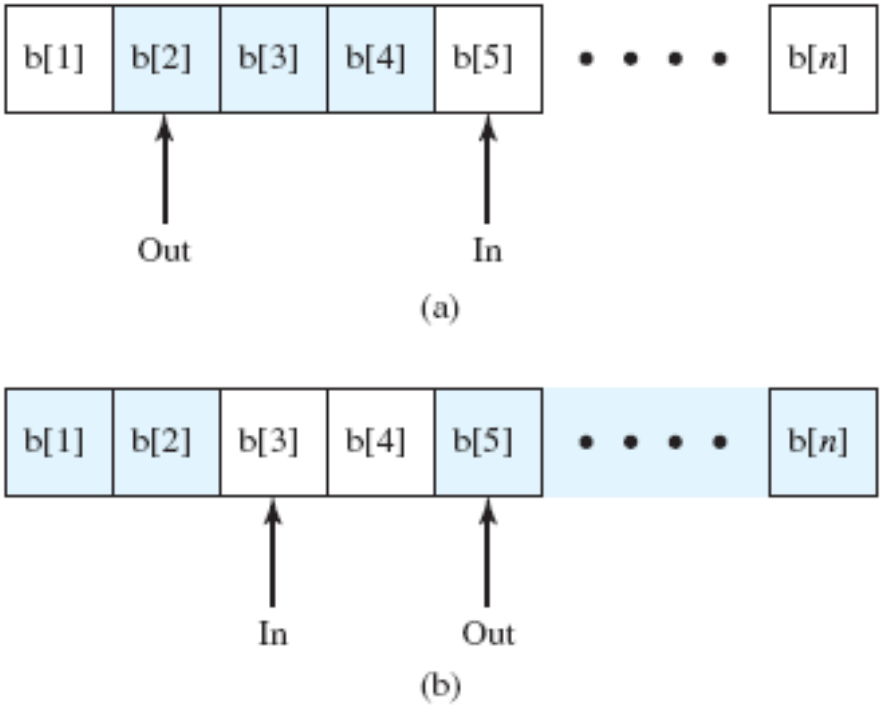
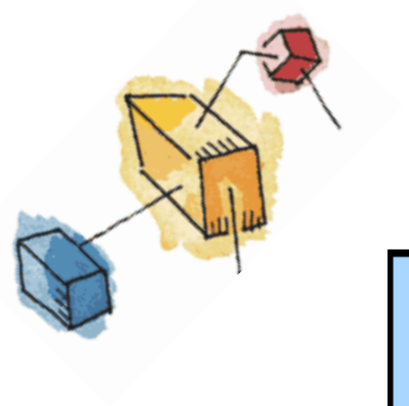


Figure 5.12 Finite Circular Buffer for the Producer/Consumer Problem



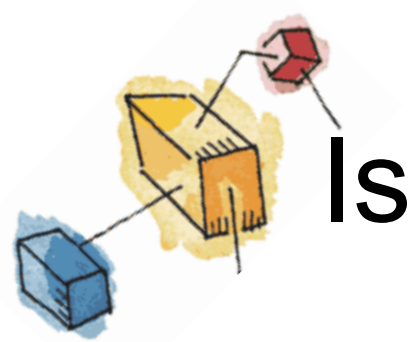
Semaphores

```
/* 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);
}
```

Ensure producer not to add items when the buffer is full

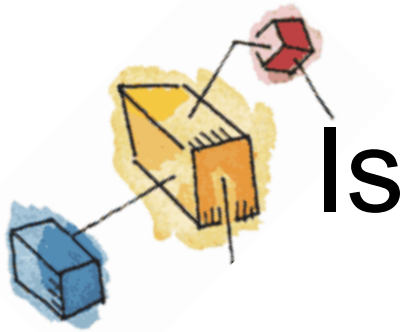
Producers wait here if the buffer is full

Producer informs Consumers that there are data items available



Issues with Semaphores

- Semaphores provide a powerful synchronization tool. However,
  - semSignal() and semWait() are scattered among several processes. Therefore, it is difficult to understand their effects
  - Usage must be correct in all the processes.
  - One bad process (or one programming error) can kill the whole system.



Issues with Semaphores

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
semWait (S);	semWait (Q);
semWait (Q);	semWait (S);
…	…
semSignal (S);	semSignal (Q);
semSignal (Q);	semSignal (S);

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Dining Philosophers Problem

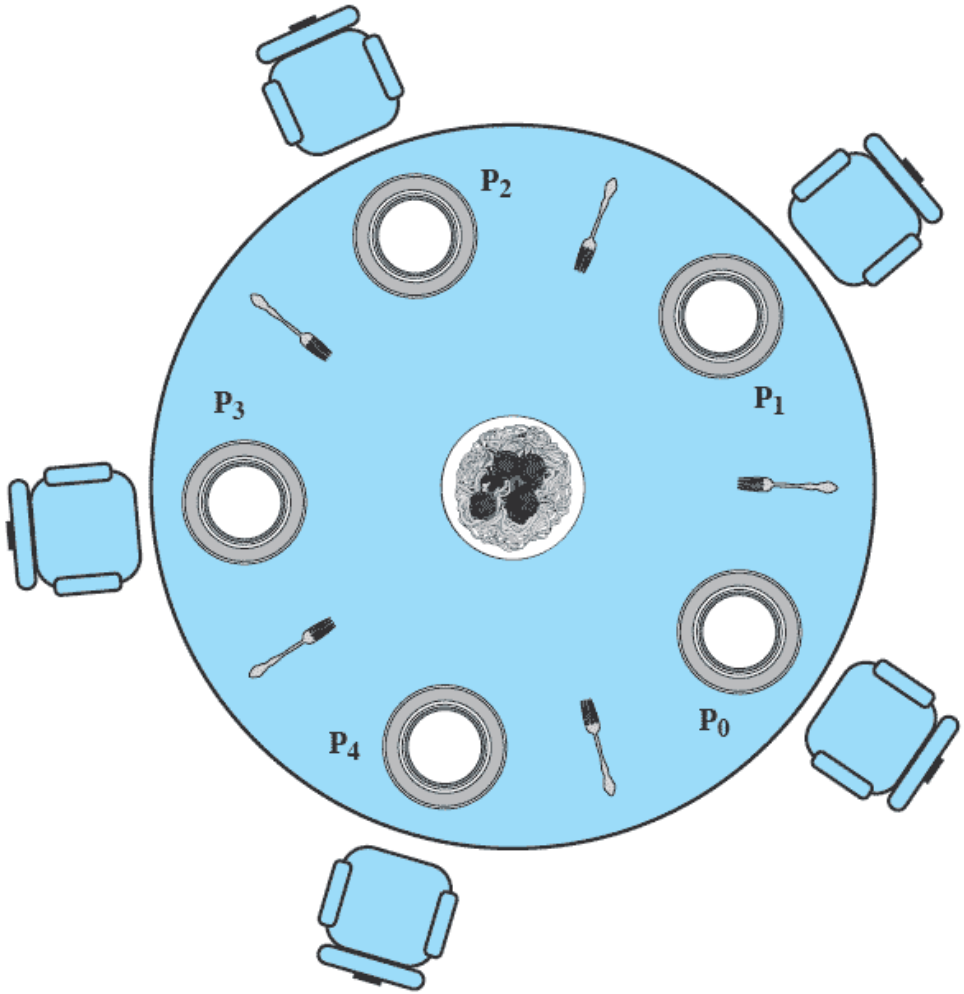


Figure 6.11 Dining Arrangement for Philosophers

Dining Philosophers Problem

```
/* program      diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
     while (true) {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1), philosopher
(2),
          philosopher (3), philosopher (4));
     }
```

Warning: This solution could create a deadlock!

Figure 6.12 A First Solution to the Dining Philosophers Problem

Monitors

- The monitor is a programming-language construct that provides equivalent functionality to that of semaphores and that is easier to control.
- Implemented in a number of programming languages, including
  - Concurrent Pascal, Pascal-Plus,Modula-2, Modula-3, and Java
- Software module consisting of one or more procedures, an initialization sequence, and local data

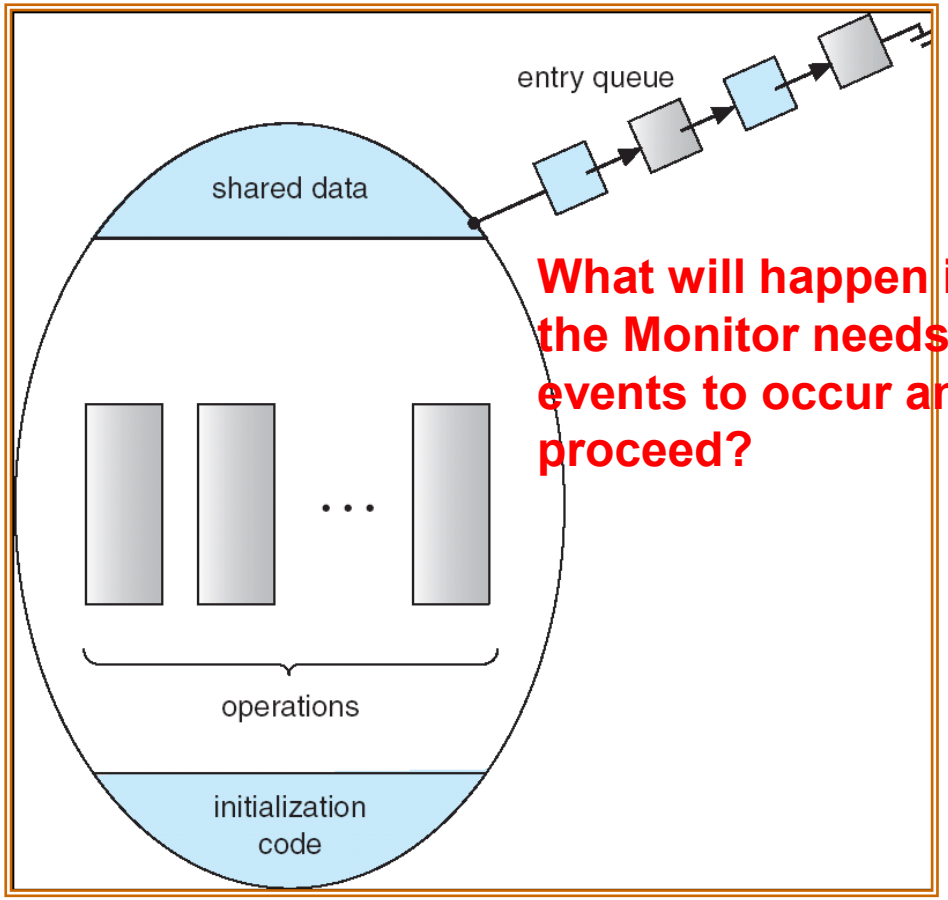
Monitor Characteristics

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

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

Process enters monitor by invoking one of its procedures

Schematic View of a Monitor

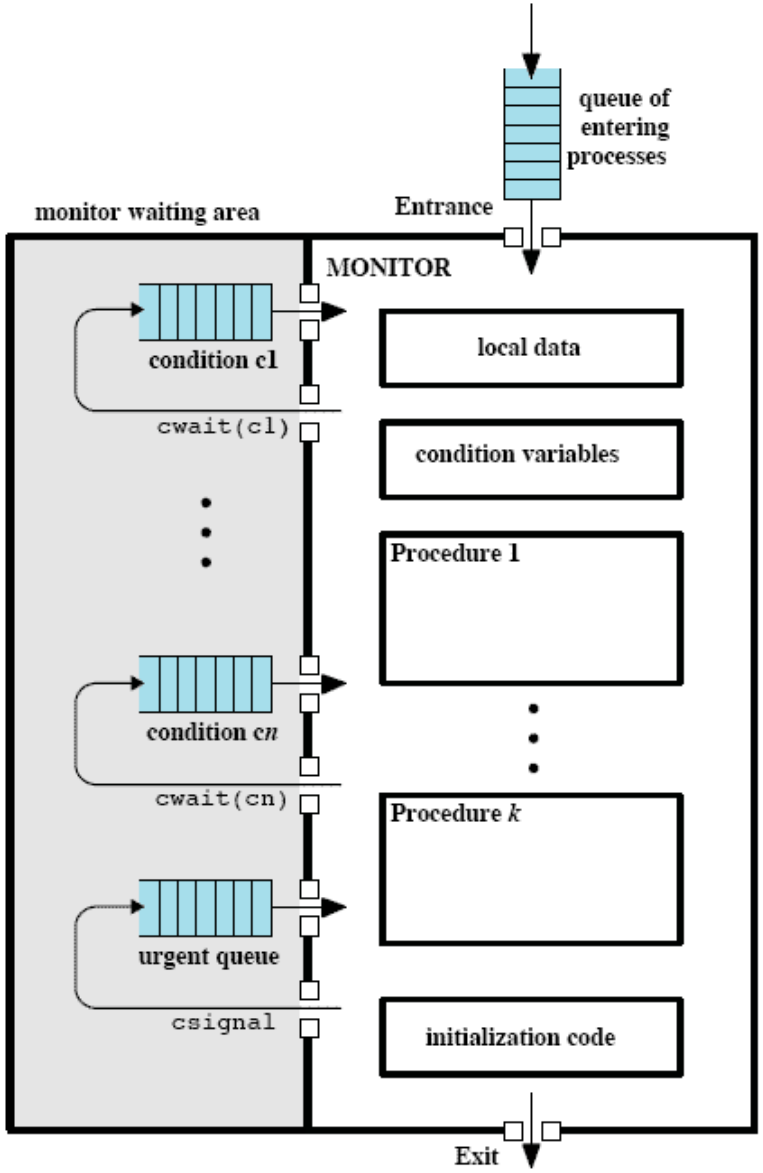


What will happen if one process in the Monitor needs to wait for some events to occur and is unable to proceed?

Synchronization

- Synchronisation achieved by using **condition variables** that are contained within a monitor and only accessible 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(c) on the same condition

Structure of a Monitor

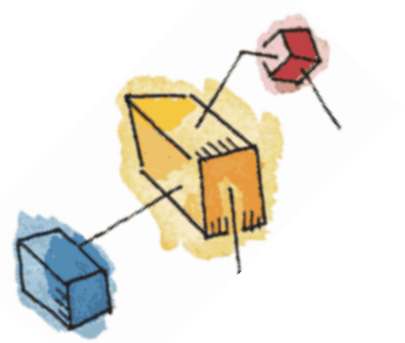


Bounded Buffer Solution Using Monitor

```
/* 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);
}
```



Message Passing

- When processes interact with one another, there are two fundamental requirements:

synchronization	communication
• to enforce mutual exclusion	• to exchange information

- Message Passing is one approach to providing these functions
  - works with distributed systems *and* shared memory multiprocessor and uniprocessor systems

Message Passing

- The actual function of message passing is normally provided in the form of a pair of primitives:
  - send (destination, message)
  - receive (source, message)
- **Exchange information**
  - A process sends information in the form of a *message* to another process designated by a *destination*
  - A process receives information by executing the receive primitive, indicating the *source* and the *message*

Blocking send, Blocking receive

- Both sender and receiver are blocked until message is delivered
- **Synchronization**
  - Receiver cannot proceed until the message is received
  - Sender cannot proceed until the message arrives to the destination

Non-blocking Send, Non-blocking Receive

Nonblocking send, blocking receive

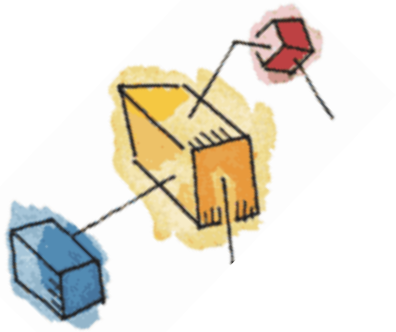
- sender continues on but receiver is blocked until the requested message arrives
- most useful combination
- sends one or more messages to a variety of destinations as quickly as possible
- example -- a service process that exists to provide a service or resource to other processes

Nonblocking send, nonblocking receive

- neither party is required to wait

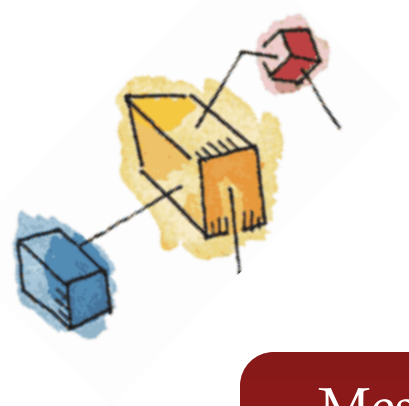
Addressing

- Sending process need to be able to specify which process should receive the message
  - Direct addressing
  - Indirect Addressing



Direct Addressing

- Send primitive includes a specific identifier of the destination process
- Receive primitive can be handled in one of two ways:
  - require that the process explicitly designate a sending process
    - effective for cooperating concurrent processes
  - implicit addressing
    - source parameter of the receive primitive possesses a value returned when the receive operation has been performed



Indirect addressing

Messages are sent to a shared data structure consisting of queues that can temporarily hold messages

Queues are referred to as *mailboxes*

One process sends a message to the mailbox and the other process picks up the message from the mailbox

Allows for greater flexibility in the use of messages

Indirect Process Communication

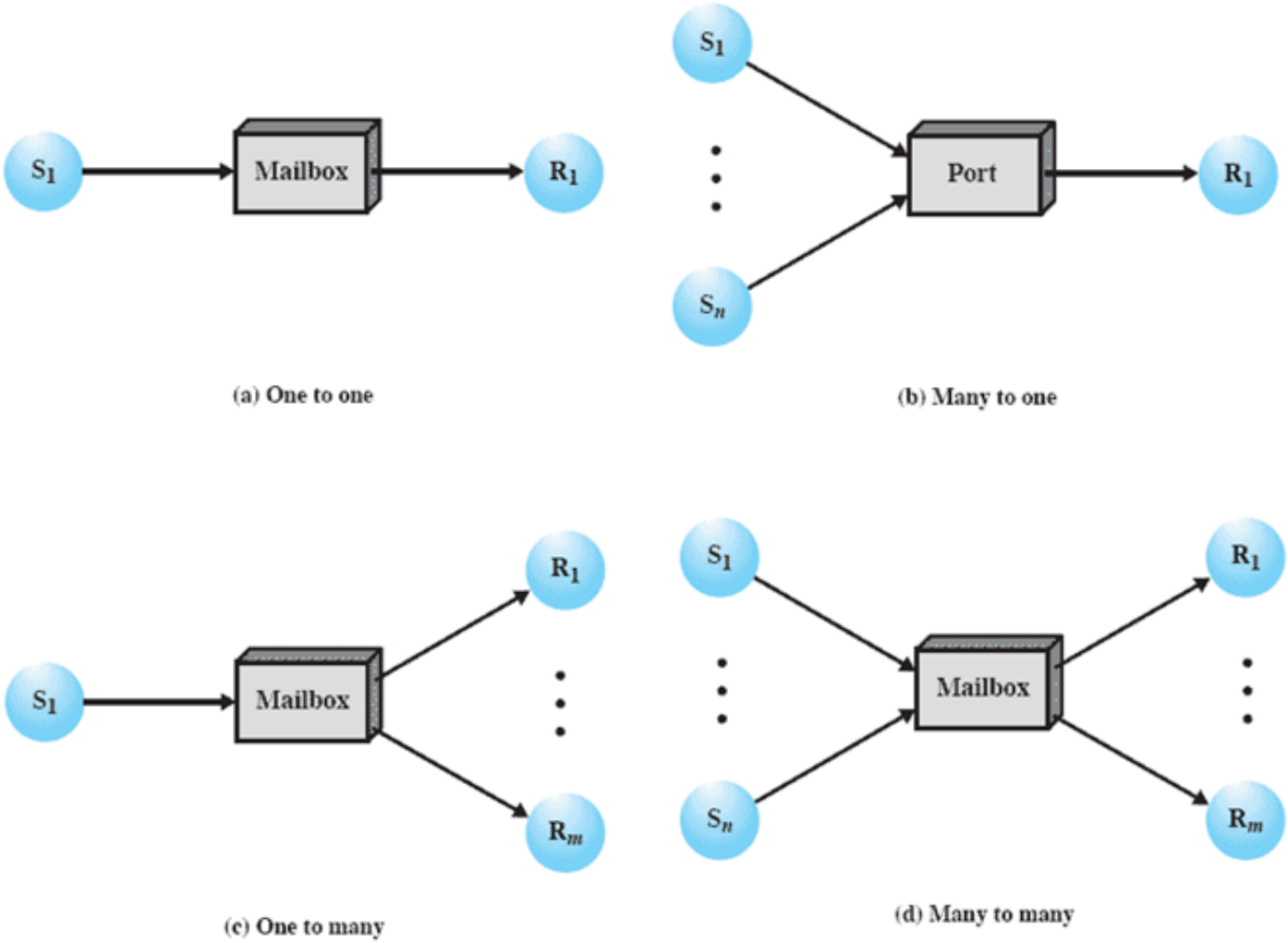


Figure 5.18 Indirect Process Communication

General Message Format

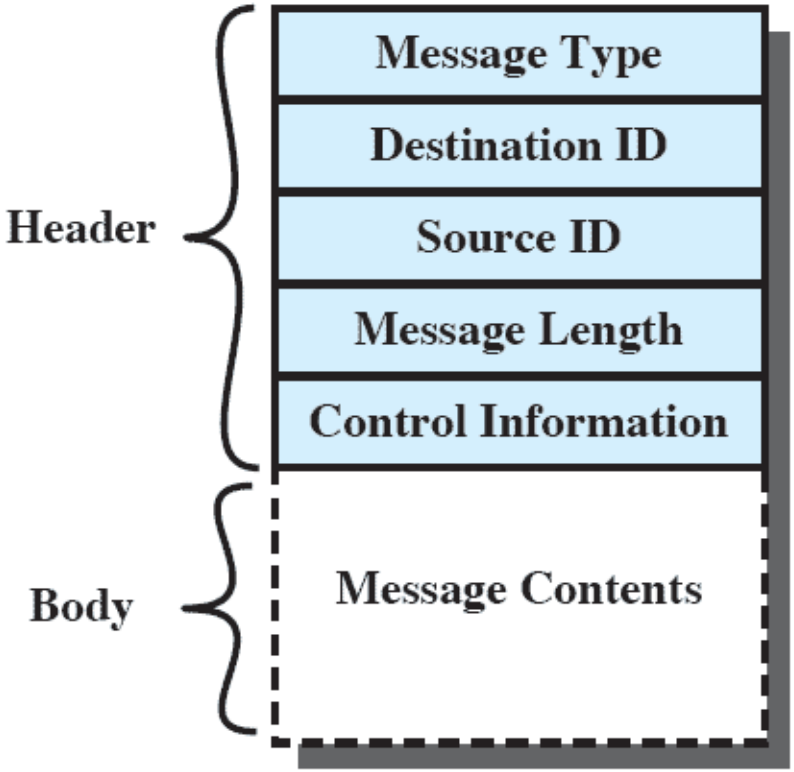


Figure 5.19 General Message Format

Readers/Writers Problem

- A data area is shared among many processes
  - Some processes only read the data area (readers), some only write to the area (writers)
- Conditions that must be satisfied:
  1. Multiple readers may read the file at once.
  2. Only one writer at a time may write
  3. If a writer is writing to the file, no reader may read it.

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);
}
```

To ensure

1. **only one writer can write at a time**
2. **no writer can write when there are readers**

Readers continue to arrive & make readcount > 1 always!

Writers may get starved here!

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);
    }
}
```

Readers and writers compete for "rsem"

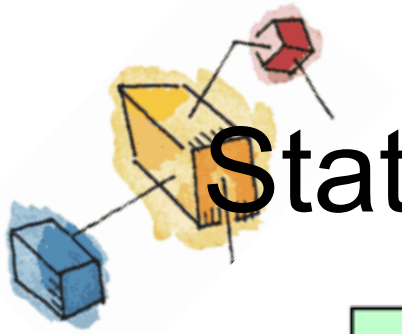
```
void writer ()
{
    while (true) {
     semWait (y);
          writecount++;
          if (writecount == 1) semWait (rsem);
     semSignal (y);
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
     semWait (y);
          writecount--;
          if (writecount == 0) semSignal (rsem);
     semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

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);
    }
}
```

"z" is used to make sure only one reader is competing "rsem" with writers

```
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);
}
```



State of the Process Queues

Readers only in the system	•*wsem* set •no queues
Writers only in the system	•*wsem* and *rsem* set •writers queue on *wsem*
Both readers and writers with read first	•*wsem* set by reader •*rsem* set by writer •all writers queue on *wsem* •one reader queues on *rsem* •other readers queue on *z*
Both readers and writers with write first	•*wsem* set by writer •*rsem* set by writer •writers queue on *wsem* •one reader queues on *rsem* •other readers queue on *z*

Summary

- Operating system themes are:
  - Multiprogramming, multiprocessing, distributed processing
  - Fundamental to these themes is concurrency
    - Issues of conflict resolution and cooperation arise
    - Effective solution: mutual exclusion, no deadlock and starvation
- Mutual exclusion
  - Condition in which there is a set of concurrent processes, only one of which is able to access a given resource or perform a given function at any time
  - One approach involves the use of special purpose machine instructions – may not be efficient as it uses busy waiting

Summary

- Semaphores
  - Used for signalling among processes and can be readily used to enforce a mutual exclusion discipline
- Monitors
  - A programming-language construct that provides equivalent functionality to that of semaphores and that is sometimes easier to control
- Messages
  - Useful for the enforcement of synchronization discipline
  - Exchange information