Concurrency: mutual exclusion and synchronization

Slides are mainly taken from «*Operating Systems: Internals and Design Principles", 8/E* William Stallings (Chapter 5).

Sistemi di Calcolo 2

Instructor: Riccardo Lazzeretti

Special thanks to: Daniele Cono D'Elia, Leonardo Aniello, Roberto Baldoni

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

Some Key Terms Related to Concurrency

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.
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.

Some Key Terms Related to Concurrency

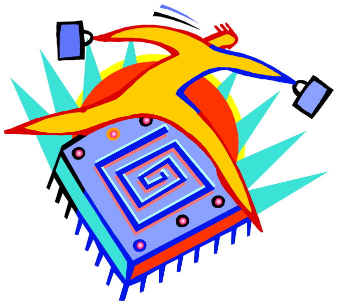
Deadlock	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
Livelock	A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work
Starvation	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

Multiprogramming Concerns

- **Output of process must be independent of the speed of execution of other concurrent processes**

Principles of Concurrency

- Interleaving and overlapping
  - can be viewed as examples of concurrent processing
  - both present the same problems
- [Uniprocessor] The relative speed of execution of processes cannot be predicted; depends on:
  - activities of other processes
  - the way the OS handles interrupts
  - scheduling policies of the OS



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

Race Condition

- Occurs when:
  - multiple processes or threads read and write data items
  - AND 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

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

Table 5.2 Process Interaction

Degree of Awareness	Relationship	Influence that One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	• Results of one process independent of the action of others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation • Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Deadlock (consumable resource) • Starvation

Resource Competition

- Concurrent processes come into conflict when they are competing for use of the same resource
  - for example: I/O devices, memory, processor time, clock

In the case of competing processes, three control problems must be faced:

- **the need for mutual exclusion**
- **deadlock**
- **starvation**

Illustration of Mutual Exclusion

```
              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 */;
   }
}
```

Requirements for Mutual Exclusion

- Must be **enforced**
- A process that halts in its noncritical section must do so without interfering with other processes
- No deadlock or starvation
- A process must not be denied access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a *finite time* only

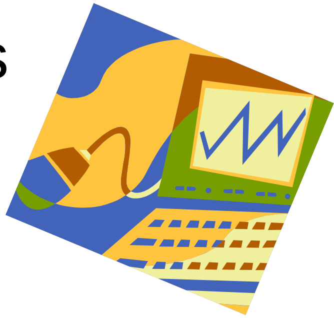
Mutual Exclusion: Hardware Support

- **Interrupt Disabling**
  - uniprocessor system
  - disabling interrupts guarantees mutual exclusion

- **Disadvantages:**
  - the efficiency of execution could be noticeably degraded
  - this approach will not work in a multiprocessor architecture

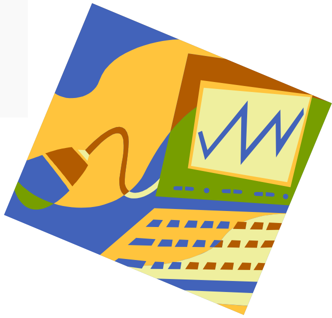
Mutual Exclusion: Hardware Support

- 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** with the new value occurs in memory
  - carried out atomically
  - Available in x86, IA64, sparc, IBM architectures



Mutual Exclusion: Hardware Support

```
int compare_and_swap(int* reg, int oldval, int newval)
{
  ATOMIC();
  int old_reg_val = *reg;
  if (old_reg_val == oldval)
     *reg = newval;
  END_ATOMIC();
  return old_reg_val;
}
```



Hardware Support for Mutual Exclusion

```c
int compare_and_swap(int* reg, int oldval, int newval)
{
  ATOMIC();
  int old_reg_val = *reg;
  if (old_reg_val == oldval)
     *reg = newval;
  END_ATOMIC();
  return old_reg_val;
}
```

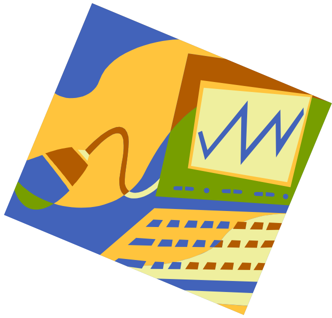
```c
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
  while (true) {
    while (compare_and_swap(&bolt, 0, 1) == 1)
         /* do nothing */;
    /* critical section */;
    bolt = 0;
    /* remainder */;
  }
}
void main()
{
  bolt = 0;
  parbegin (P(1),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

}
```

(a) Compare and swap instruction

Mutual Exclusion: Hardware Support

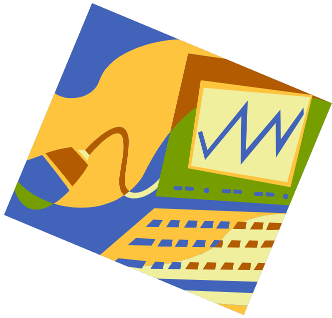
- Exchange instruction
  - exchange the content of a register with that of a memory location
  - Implemented in Pentium and Itanium (IA64) architectures



Mutual Exclusion: Hardware Support

- Exchange instruction

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



Hardware Support for Mutual Exclusion

- Exchange instruction

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

```
/* 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));
}
```

(b) Exchange instruction

Special Machine Instruction: Advantages

- Applicable to any number of processes on either a single processor or multiple processors sharing main memory
- Simple and easy to verify
- It can be used to support multiple critical sections; each critical section can be defined by its own variable

Special Machine Instruction: Disadvantages

- **Busy-waiting** is employed: while a process is waiting for access to a critical section it continues to consume processor time
- Starvation is possible when a process leaves a critical section and more than one process is waiting
- Deadlock is possible *(e.g., because of process priority)*
- Not available in any architecture
  - *Developer must be aware of the architecture*

Common Concurrency Mechanisms

Semaphore	A construct based on 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
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

Common Concurrency Mechanisms

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 variable that has an integer value upon which only three operations are defined:

- There is no way to inspect or manipulate semaphores other than these three operations

1) May be initialized to a nonnegative integer value
2) The semWait operation decrements the value
3) The semSignal operation increments the value

Consequences

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

There is no way, on a uniprocessor system, to know which process will continue immediately after a semWait 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

Figure 5.3 – 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 */;
     }
}
```

Figure 5.4 - 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

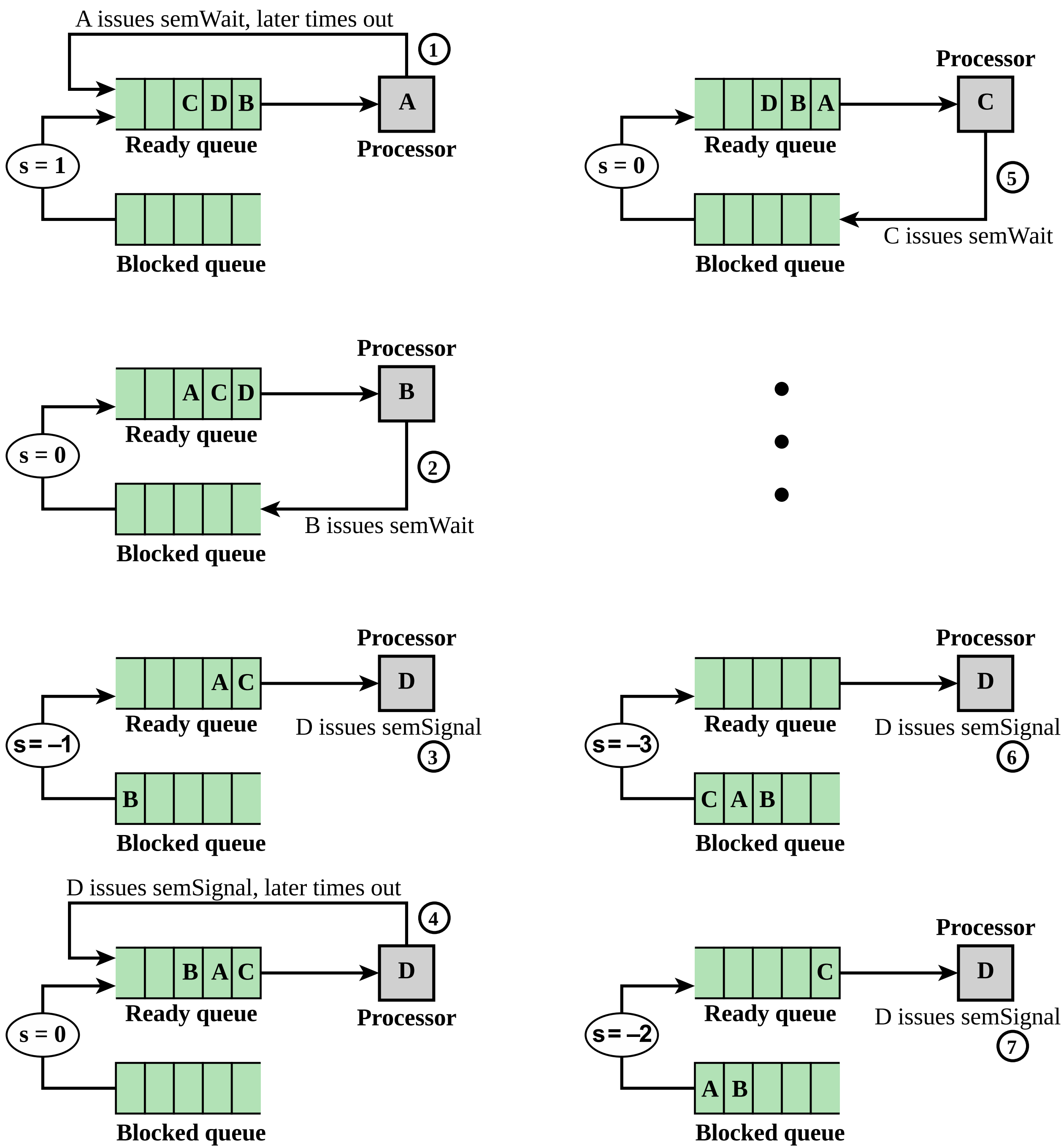
A **queue** is used to hold processes waiting on the semaphore

Strong Semaphores

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

Weak Semaphores

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


A issues semWait, later times out
1
C D B
Ready queue
A
Processor
s = 1
Blocked queue
Processor
D B A
C
Ready queue
s = 0
5
Blocked queue
C issues semWait
Processor
A C D
B
Ready queue
s = 0
2
Blocked queue
B issues semWait
Processor
A C
D
Ready queue
D issues semSignal
s = –1
3
B
Blocked queue
Processor
D
Ready queue
D issues semSignal
s = –3
6
C A B
Blocked queue
D issues semSignal, later times out
4
B A C
D
Ready queue
Processor
s = 0
Blocked queue
Processor
C
D
Ready queue
D issues semSignal
s = –2
7
A B
Blocked queue

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));
}
```

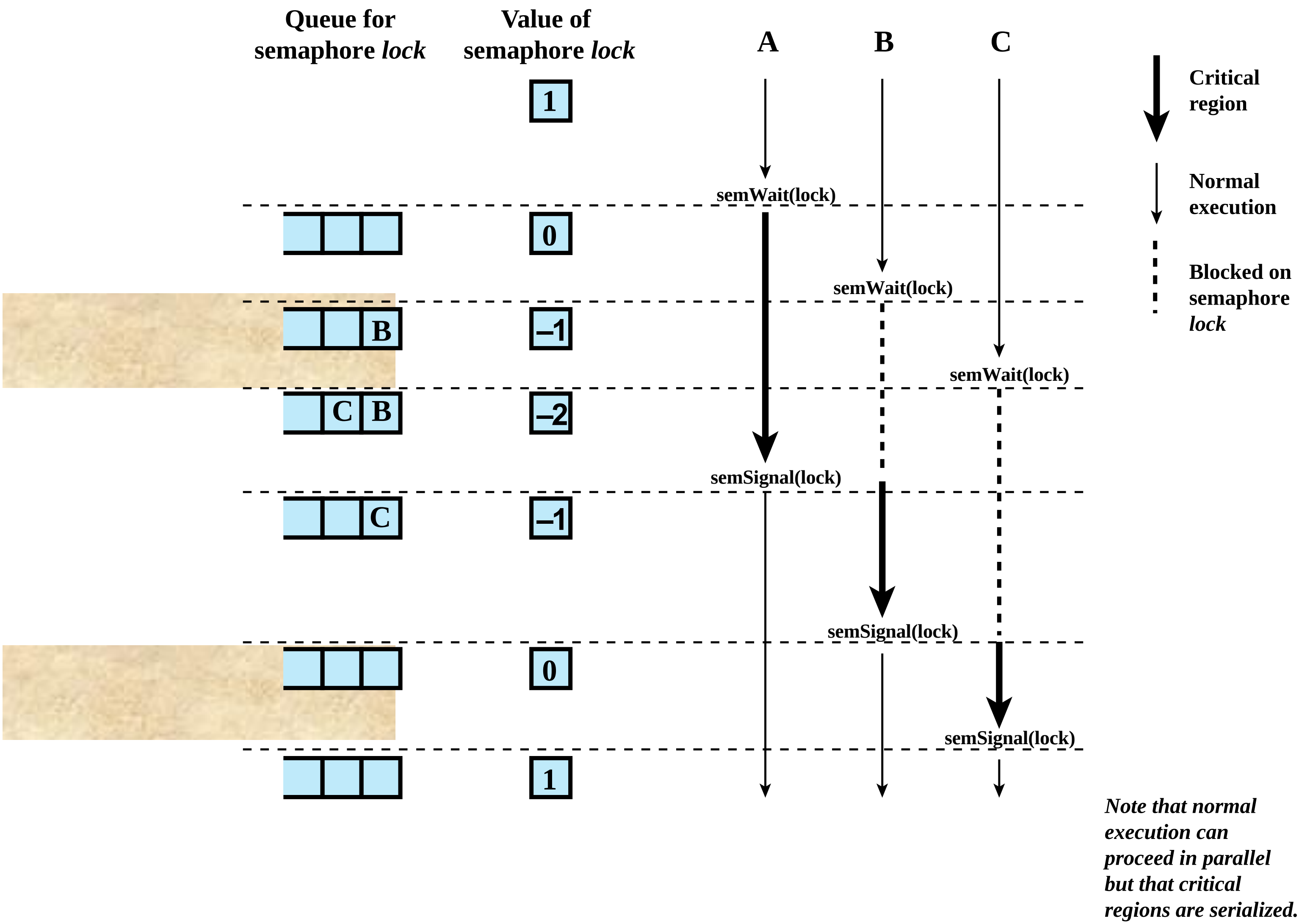
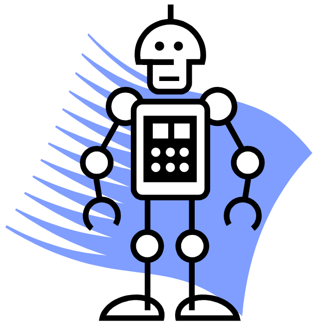


Figure 5.7 Processes Accessing Shared Data Protected by a Semaphore

Implementation of Semaphores

- Imperative that the `semWait` and `semSignal` operations be implemented as atomic primitives
  - Manipulating a semaphore is a mutual exclusion problem
- 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

Figure 5.14
Two Possible Implementations of Semaphores

```
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

Posix semaphores (c)

- `int sem_init(sem_t * sem, int pshared, unsigned value); //initialization`
- `int sem_wait(sem_t *sem); //wait`
- `int sem_post(sem_t *sem); //signal`
- `int sem_destroy(sem_t *sem); //destruction`

- Parameters:
  - `sem`: the semaphore
  - `pshared`: 0 if semaphore is shared among threads, 1 if shared among processes
  - `value`: the starting value (how many resources we can share)
- return -1 in case of error, 0 otherwise
  - Set errno to allow identification of the error type

Example: semaphore

```
...
#include <semaphore.h>

sem_t sem;

void* thread_fun(void* arg){
   sem_wait(&sem);
   //critical section
   sem_post(&sem);
}

int main (){
   sem_init(&sem,0,1);
   pthread_t t1,t2;
   pthread_create(&t1,NULL,thread,NULL);
   pthread_create(&t2,NULL,thread,NULL);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
   sem_destroy(&sem);
   return 0;
}
```

Java semaphores

- `Semaphore(int value) //initialization`
- `Semaphore(int value, boolean how) //initialization`
- `acquire() //wait`
- `release() //signal`

- Parameters:
  - `value`: the starting value (how many resources we can share)
  - `How`: if set to True we are defining a strong semaphore, otherwise a weak semaphore (default)

Example: semaphore

```
...
import java.util.concurrent.*;

class MyThread extends Thread
{
    Semaphore sem;
    String threadName
    public MyThread(Semaphore sem, String threadName) {
          super(threadName);
          this.sem = sem;
          this.threadName = threadName;
     }
    public void run(){
       sem.acquire();
       //critical section
       sem.release();
    }
}
public class SemaphoreDemo{
    int main (){
       Semaphore sem = new Semaphore(1);
       MyThread mt1 = new MyThread(sem, "A");
       MyThread mt2 = new MyThread(sem, "B");

       mt1.start();
       mt2.start();

       mt1.join();
       mt2.join();
    }
}
```

Python semaphores

- `threading.Semaphore([value]) //initialization`
- `acquire([blocking]) //wait`
- `release() //signal`

- Parameters:
  - `value`: the starting value (how many resources we can share)
    - If omitted: 1 (default value)
  - `blocking`: permits the code to avoid blocking on the semaphore
    - If set to True (default), the process is blocked if current value is not positive
    - If set to False, the procedure return True if current value is positive, False otherwise (do not block)

Example: semaphore

```
import threading
sem = threading.Semaphore()

def fun():
    while True:
        sem.acquire()
        # critical section
        sem.release()

t1 = threading.Thread(target = fun)
t1.start()
t2 = threading.Thread(target = fun)
t2.start()
```

Monitors

- 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
- 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

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

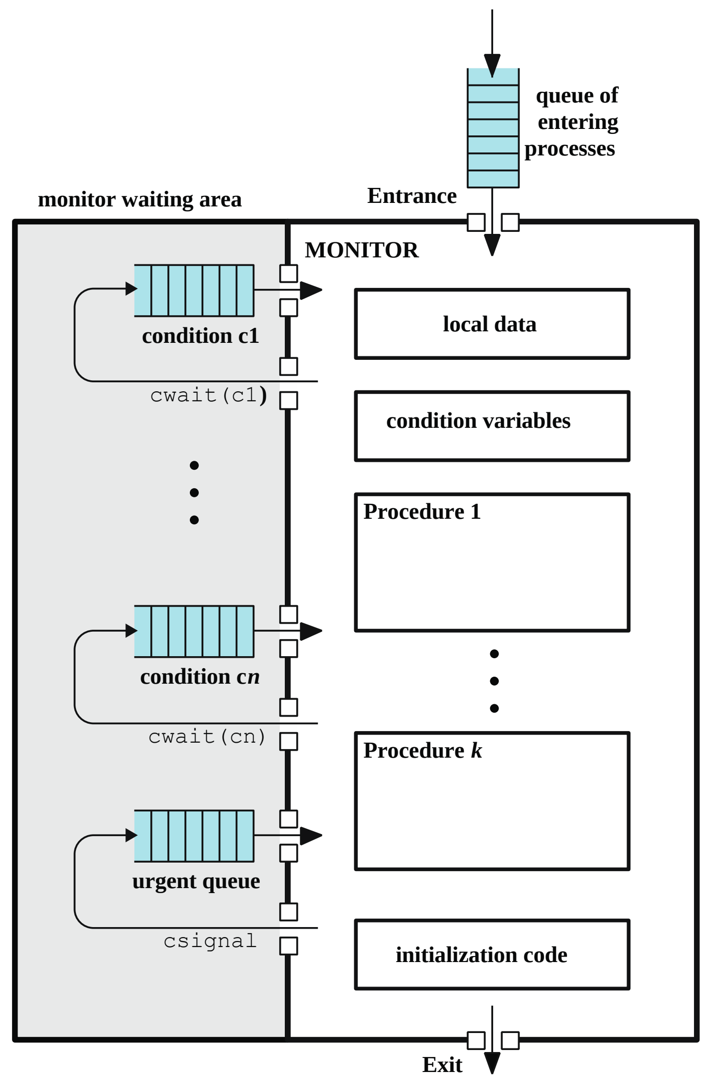


Figure 5.15 Structure of a Monitor

Message Passing

- When processes interact with one another, two fundamental requirements must be satisfied:

synchronization	communication
• to enforce mutual exclusion	• to exchange information

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

Message Passing

- The actual function is normally provided in the form of a pair of primitives:

  send (destination, message)

  receive (source, message)

» 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*

Design Characteristics of Message Systems for Interprocess Communication and Synchronization

Synchronization
- Send
  - blocking
  - nonblocking
- Receive
  - blocking
  - nonblocking
  - test for arrival

Addressing
- Direct
  - send
  - receive
    - explicit
    - implicit
- Indirect
  - static
  - dynamic
  - ownership

Format
- Content
- Length
  - fixed
  - variable

Queueing Discipline
- FIFO
- Priority

Synchronization

Communication of a message between two processes implies synchronization between the two

the receiver cannot receive a message until it has been sent by another process

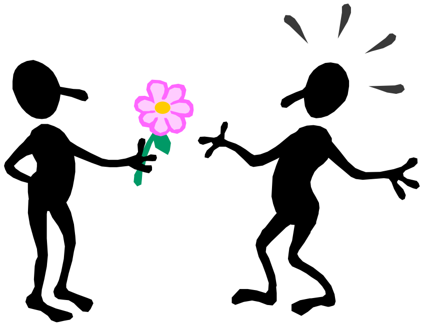
When a receive primitive is executed in a process there are two possibilities:

if there is no waiting message the process is blocked until a message arrives or the process continues to execute, abandoning the attempt to receive

if a message has previously been sent the message is received and execution continues

Blocking Send, Blocking Receive

- Both sender and receiver are blocked until the message is delivered
- Sometimes referred to as a *rendezvous*
- Allows for tight synchronization between processes



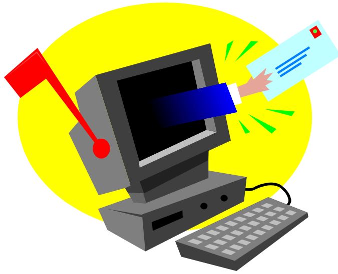
Nonblocking Send

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

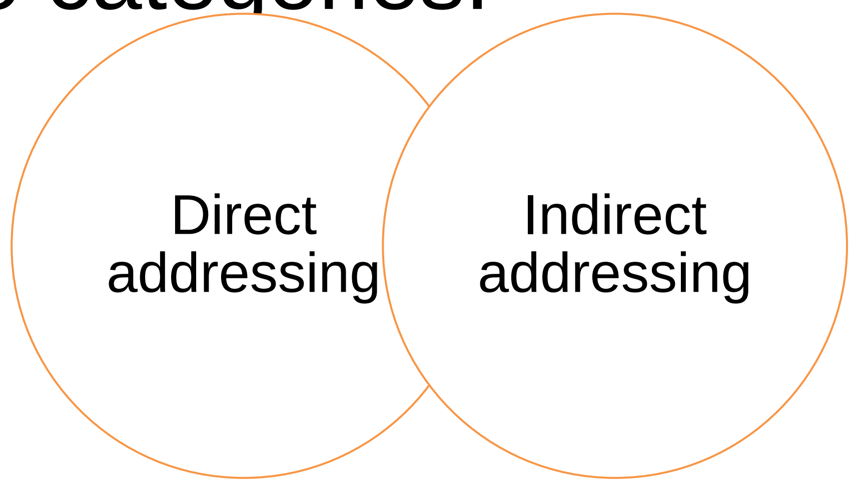
Nonblocking send, nonblocking receive

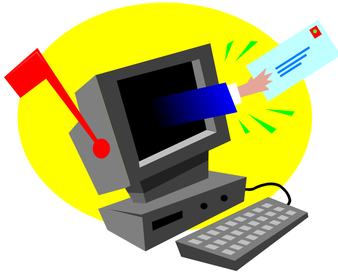
- neither party is required to wait



Addressing

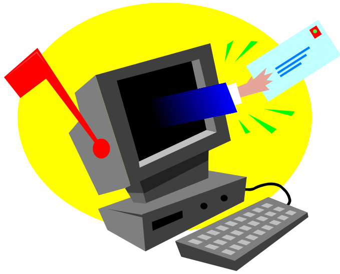
- Schemes for specifying processes in `send` and `receive` primitives fall into two categories:





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

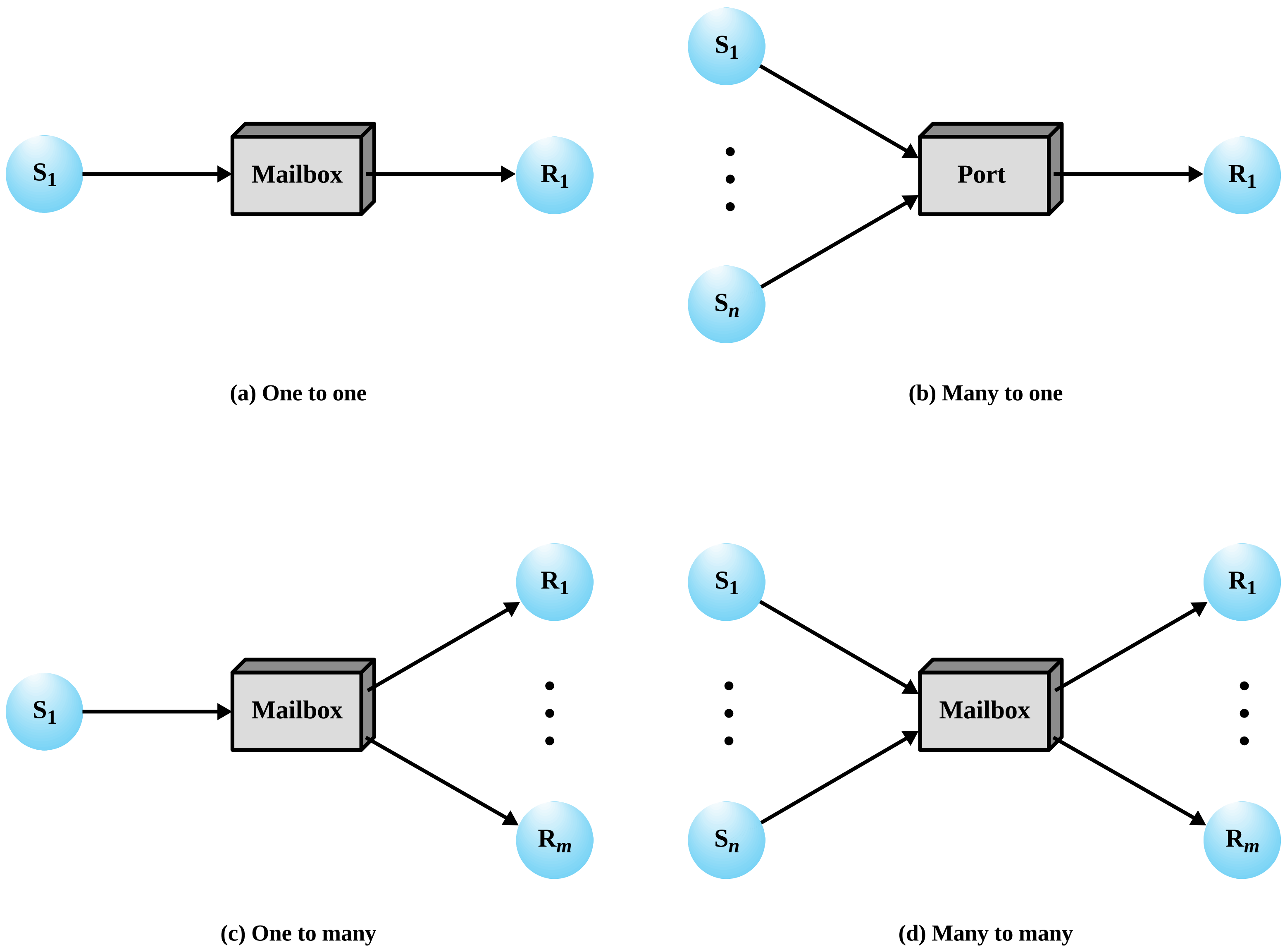


Figure 5.18 Indirect Process Communication

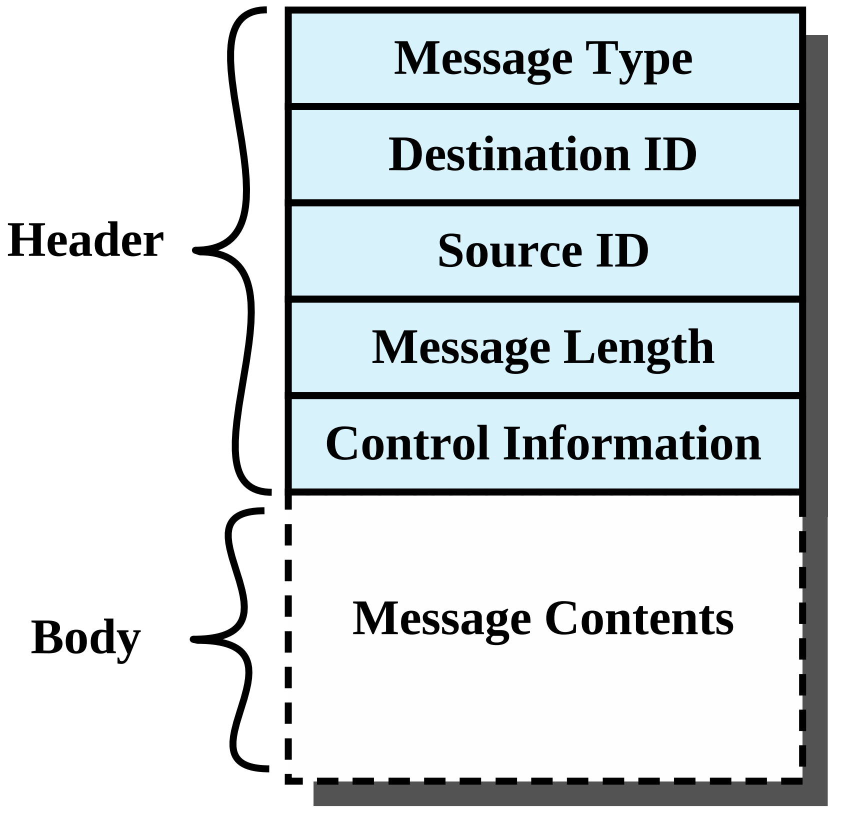


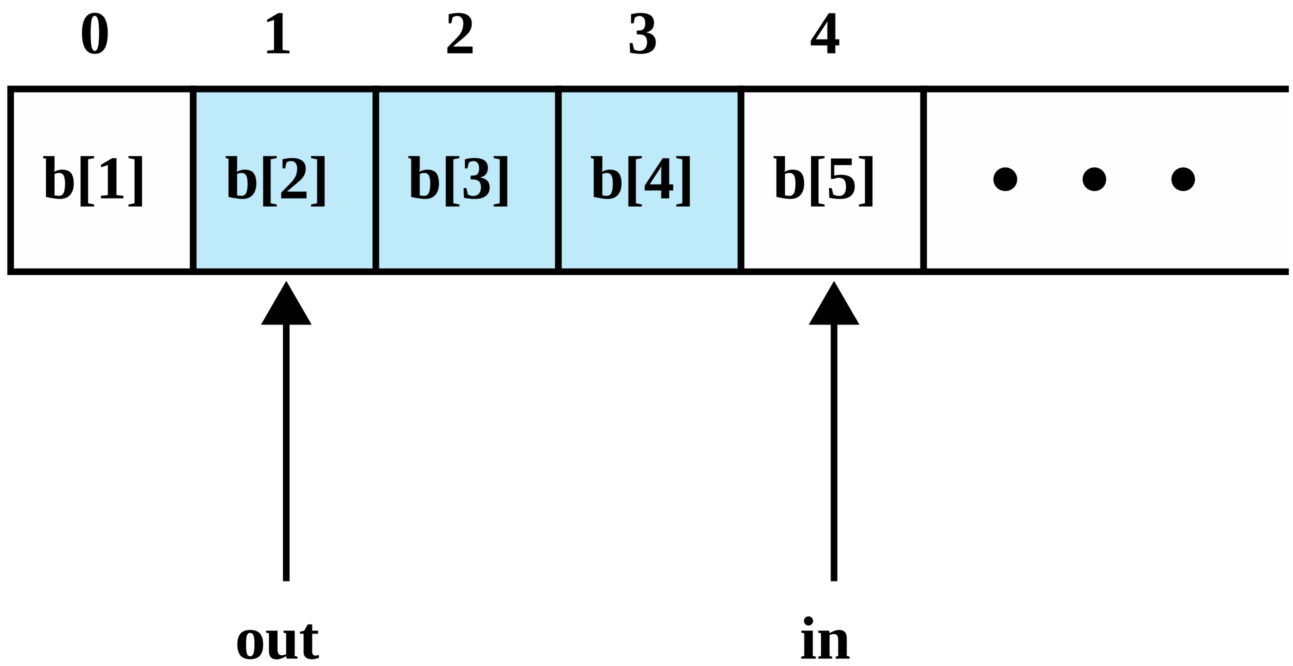
Figure 5.19 General Message Format

```
/* program mutualexclusion */
const int n = /* number of processes  */;
void P(int i)
{
    message msg;
    while (true) {
     receive (box, msg);
     /* critical section   */;
     send (box, msg);
     /* remainder   */;
    }
}
void main()
{
    create_mailbox (box);
    send (box, null);
    parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 5.20
Mutual Exclusion Using Messages

Producer/Consumer Problem

General Statement:	one or more producers are generating data and placing these in a buffer
	one or more consumer are 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

Infinite-Buffer P/C - An incorrect solution

```
/* program producerconsumer */
int n = 0;
binary_semaphore s = 1, delay = 0;
```

```
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
```

Infinite-Buffer P/C - An incorrect solution

```
/* program producerconsumer */
int n = 0;
binary_semaphore s = 1, delay = 0;
```

```
void consumer()
{
     semWaitB(delay);
     while (true) {
          semWaitB(s);
          take();
          n--;
          semSignalB(s);
          consume();
          if (n==0) semWaitB(delay);
     }
}
```

Consuming an item that does not exist...

Producer	Consumer	s	n	Delay
		1	0	0
`semWaitB(s)`		0	0	0
`n++`		0	1	0
if `(n==1)` `(semSignalB(delay))`		0	1	1
`semSignalB(s)`		1	1	1
	`semWaitB(delay)`	1	1	0
	`semWaitB(s)`	0	1	0
	`n--`	0	0	0
	`semSignalB(s)`	1	0	0

Consumer is descheduled before it can test:
if (n==0) semWaitB(delay)
So the producer enters CS and increases n...

Consuming an item that does not exist...

	semSignalB(s)	1	0	0
semWaitB(s)		0	0	0
n++		0	1	0
if (n==1) (semSignalB(delay))		0	1	1
semSignalB(s)		1	1	1
	if (n==0) (semWaitB(delay))	1	1	1

Later on we get **n == -1**: we are reading an invalid element!

```
// consumer
semWaitB(s)
take(), n--
semSignalB(s)
consume()
if (n==0) ...
```

semWaitB(s)	0	1	1
n--	0	0	1
semSignalB(s)	1	0	1
if (n==0) (semWaitB(delay))	1	0	0
semWaitB(s)	0	0	0
n--	0	–1	0
semSignalB(s)	1	–1	0

Consuming an item that does not exist...

	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

Infinite-Buffer P/C - Another incorrect solution

```
/* program producerconsumer */
int n = 0;
binary_semaphore s = 1, delay = 0;
```

```
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
```

Infinite-Buffer P/C - Another incorrect solution

```
/* program producerconsumer */
int n = 0;
binary_semaphore s = 1, delay = 0;
```

```
void consumer()
{
     semWaitB(delay);
     while (true) {
          semWaitB(s);
          take();
          n--;
          consume();
          if (n==0) semWaitB(delay);
          semSignalB(s);
     }
}
```

Deadlock

A possible fix (using binary semaphores only)

```
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);
     }
}
```

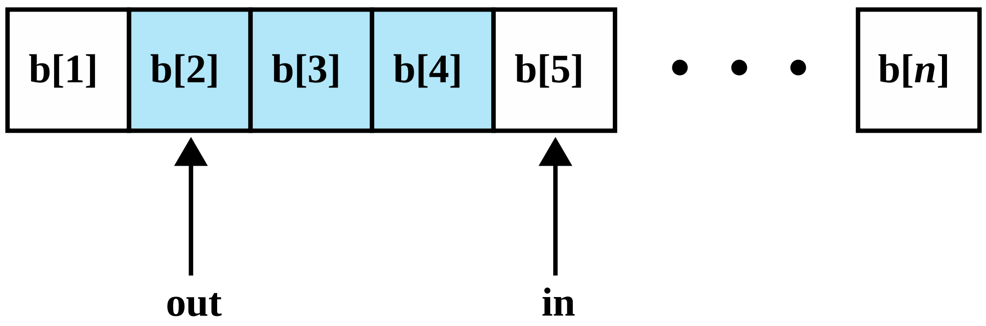
We make a local copy m = n of the variable, so we can read the correct value once we left the CS

Infinite-Buffer P/C – General semaphores

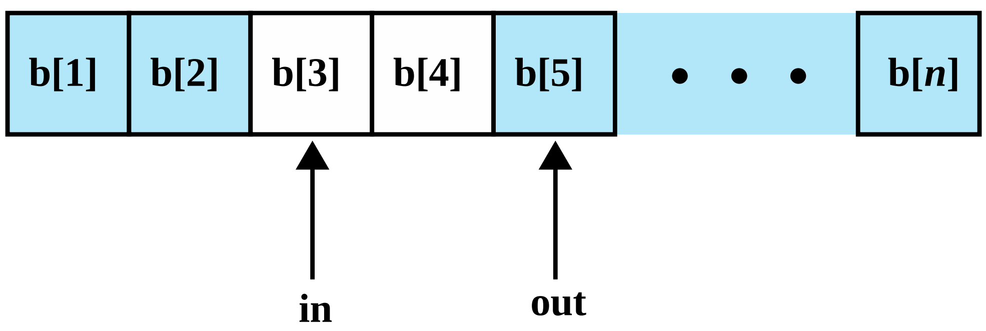
```
/* program producerconsumer */
semaphore n = 0, s = 1;
```

```
void producer()
{
   while (true) {
         produce();
         semWait(s);
         append();
         semSignal(s);
         semSignal(n);
   }
}
```

```
void consumer()
{
   while (true) {
         semWait(n);
         semWait(s);
         take();
         semSignal(s);
         consume();
   }
}
```



(a)



(b)

Figure 5.12 Finite Circular Buffer for the Producer/Consumer Problem

Bounded-buffer 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);
     }
}
```

Bounded-buffer Producer/Consumer

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
```

```
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
```

Bounded-buffer 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();
     }
}
```

What if I swap pairs of adjacent semWait or semSignal operations???
What if I have only a producer or only a reader???

Bounded-buffer P/C using Monitors

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

Monitors:

- local data accessible only by the monitor's procedures
- only one process may be executing in the monitor at a time
- programmer can decide when a procedure running in the monitor should stop or resume depending on a condition predicate

Thus, consume() and produce() do not need to know how take() and append() are implemented inside the monitor...

Bounded-buffer P/C using Monitors

```
/* 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 */
```

Init: `nextin = nextout = count = 0;`

```
void append (char x)
{
     if (count == N) cwait(notfull);
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;
     /* one more item in buffer */
     csignal(notempty);
}
```

Wait if buffer is full, notify pending consumer process on exit (if any)

Bounded-buffer P/C using Monitors

```
/* 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 */
```

Init: `nextin = nextout = count = 0;`

```
void take (char x)
{
    if (count == 0) cwait(notempty);
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;
    csignal(notfull);
}
```

Wait if buffer is empty, notify pending producer process on exit (if any)

(By the way, this algorithm supports *multiple consumers* too...)

```
const int
    capacity = /* buffering capacity */ ;
    null =/* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true) {
     receive (mayproduce, pmsg);
     pmsg = produce();
     send (mayconsume, pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true) {
     receive (mayconsume, cmsg);
     consume (cmsg);
     send (mayproduce, null);
    }
}

void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++) send (mayproduce,
null);
    parbegin (producer, consumer);
}
```

Figure 5.21

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

Init. Mayproduce is full

Init. Mayconsume is empty

Readers/Writers Problem

- A data area is shared among many processes
    - some processes only read the data area (readers)
    - some processes 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

Readers/Writers Problem Using Semaphores

```
/* program readersandwriters */
int readcount = 0;
semaphore x = 1, wsem = 1;
```

Requirements:

- any number of readers may read simultaneously
- only one writer at a time may write
- when a writer is writing, no reader is allowed to read

```
while (true) {
 semWait (wsem);
 WRITEUNIT();
 semSignal (wsem);
}
```

writer()

- acquire exclusive lock using binary semaphore wsem

Readers/Writers Problem Using Semaphores

```
while (true) {
 semWait (x);
 readcount++;
 if (readcount == 1) semWait (wsem);
 semSignal (x);
 READUNIT();
 semWait (x);
 readcount--;
 if (readcount == 0) semSignal (wsem);
 semSignal (x);
}
```

reader()

- binary semaphore x to safely update variable readcount
- when there is only one reader (x==1) lock wsem (no writes!)
- once the read is performed, unlock wsem if no reader is active (i.e., readcount==0) => *writers may starve*, extensions needed!