

2026

CS332

Subject 11

NJIT

A white curved line that starts under the 'N', goes under the 'J', and then curves upwards and to the right, ending under the 'T'.

New Jersey's Science &
Technology University

THE EDGE IN KNOWLEDGE

Alex. Gerbessiotis

CS Department



The yellowish/brownish notes are copyright the textbook author.

The bluish notes are copyright the instructor and are add-ons and provide more explanation or additional details



Module 1

*Operating
Systems:
Internals
and Design
Principles*

Chapter 5 Concurrency: Mutual Exclusion and Synchronization

Ninth Edition
By William Stallings



Example 1: Process 1 and Process 2 have access to a shared memory variable x;

Need to coordinate read/write access of x

Example 2: Process 10 and its threads 11, 12 and 13 have access to a global variable (or a heap object) y.

Need to coordinate read/write access of y.

Example 3: Process 10 and its threads 11,12, and 13, and Process 20 and its threads 21,22,23 have access to a shared memory variable z.

Need to coordinate read/write access of z.

Issue: Concurrency

Resolution: Synchronization

Method for Synchronization

-> Access to a shared variable/resource/device

-> Synchronize access to shared variable

-> Mutex (Mutual Exclusion)

Concurrency



Concurrent access to a shared variable or a shared resource or shared device by multiple tasks (processes or threads of one or more process)

needs to be coordinated (a.k.a. synchronized)
so that no task interferes with the work of another task.

This is guaranteed by arranging mutual exclusion

Mutual exclusion: when a task has access to a shared variable (read or write)
or uses a common resource

NO OTHER TASK CAN DO SO AT THE SAME TIME

"SAME TIME" : loosely used (to be discussed)

Multiple Processes

- Operating System design is concerned with the management of processes and threads:

- **Multiprogramming**

- The management of multiple processes within a uniprocessor system

- **Multiprocessing**

- The management of multiple processes within a multiprocessor

- **Distributed Processing**

- The management of multiple processes executing on multiple, distributed computer systems
 - The recent proliferation of clusters is a prime example of this type of system

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

Example: Concurrency



shared int i=10;

Proc 0

1. i++;
2. print i;

1.a AC=i;
1.b AC++;

1.c i=AC;
2. print i;

Proc 1

3. i++;
4. print i;

3.a AC=i;
3.b AC++;
3.c i=AC;

4. print i;

++(i) {

a. AC = i;
b. AC ++;
c. i=AC;
}

Both print
11...

Or may
be not!

There is a difference between an interruptible C statement such as i++
Vs the three instructions that can realize i++. Each of the three
instructions (AC=i, ... i=AC) might be un-interruptible but as a group;
We can have an interrupt-ion at the semicolon (end of execution of
instruction

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



C. Concurrency

Examples

ProducerConsumer
ReaderWriter

M. Mutual Exclusion for Synchronization

MS. Software Approaches

suffer from

Busy-wait (aka while() {})

MH. Hardware Approaches

noInterrupt,
Compare&Swap
XCHG

suffer from

Busy-wait (aka while() {})
Starvation
Deadlock

MP. Progr. Language Approaches

Semaphores
Monitors
MessagePassing

Table 5.1

**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.
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.
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.

Race conditions ..theory



P₀ and P₁ both execute echo() ; variables
chin and chout are shared

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

TimeLine

0

1

2

3

4

5.

6.

What

P₀: chin = 'o'P₁: chin = '1'P₁: chout=chin = '1'P₁: out '1'P₀: chout=chin = '1'P₀: out '1'

where is my 'o' ?

See Example 2 on
next page

Producer (getchar)
Consumer (putchar)
Problem



Example 1: (Blackout 2003)

Alarm system in power companies failed (it was working for 10+ years without any issues)

Example 2: Therac-25 (Radiation machine incident)

Race condition led to incorrect dosages
20-100x the correct one
(or 10000x the radiation of a single chest X-ray)

At least 4 deaths reported.

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 (**fairness**)



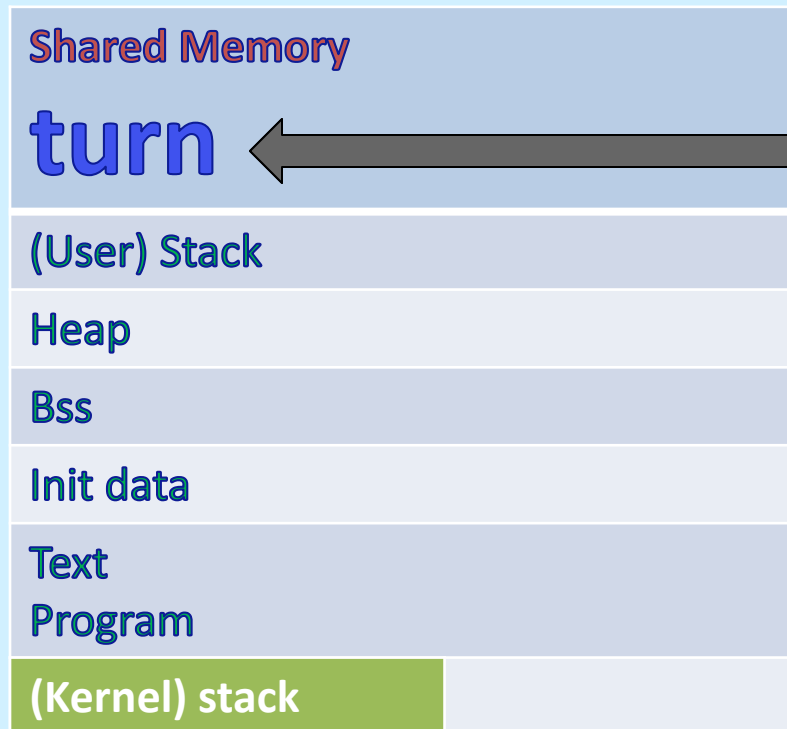
MS. Software Approaches for mutual exclusion

MS.1 Peterson's Algorithm for mutual exclusion

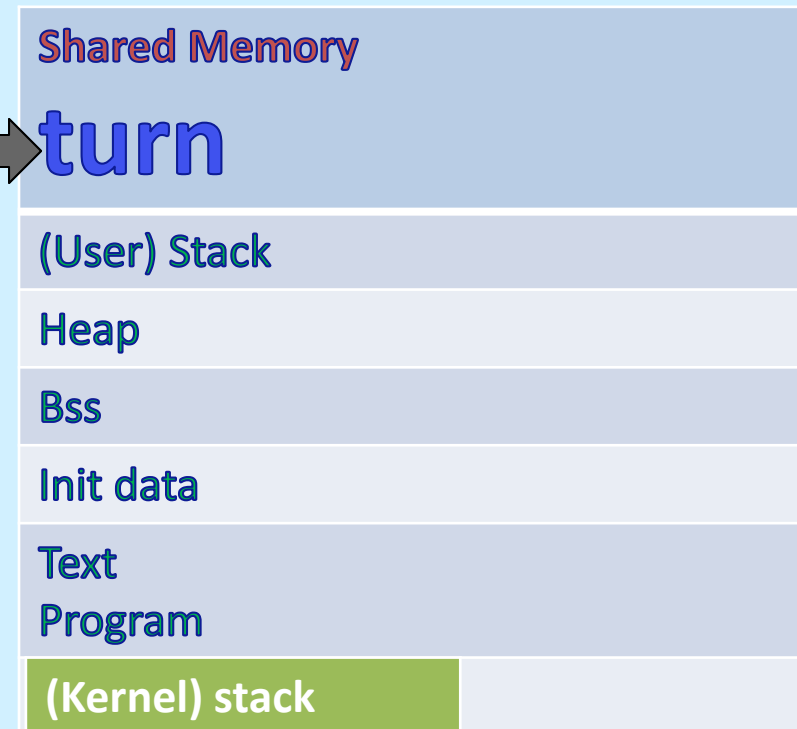
MS1.for.CS Peterson's alg. to solve critical
section problem



Process 0



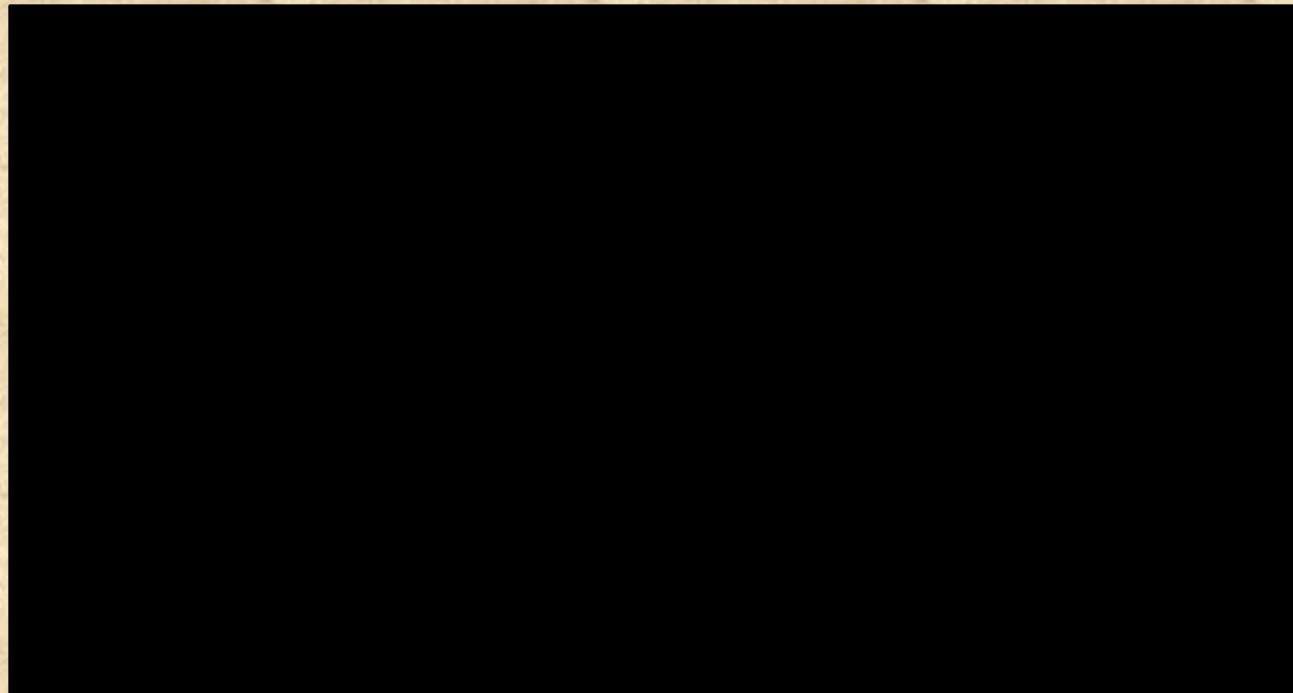
Process 1



turn=1 Back and forth concept turn = 0
vs
is known as **COROUTINE**

/* PROCESS 0 */	/* PROCESS 1 */
<pre>• • while (turn != 0) /* do nothing */ ; /* critical section*/; turn = 1; •</pre>	<pre>• • while (turn != 1) /* do nothing */; /* critical section*/; turn = 0; •</pre>

(a) First attempt



First attempt : issues



1. A process (say 0) fails while $\text{turn} = 0$.
Turn will never become 1 ---> Process 1 starves...
2. Say 0 enters its cs (critical section) Jan 01 (9am) for 10min
Say 1 enters its cs every 2nd of the month at 9:10am for 10min
 - a. 9:00am/01Jan26 Proc 0 in/out cs ($\text{turn}=1$ @9:10)
 - b. 9:00am/02Jan26 Proc 1 in/out cs ($\text{turn}=0$ @9:10)
 - c. wait until 2027?
3. What's the initial value of turn ? (bias)
4. Cosmic Event... (A bit of turn is set to 1... 8 or 9 vs 0 or 1.
5. Does not allow for a process that DOES NOT WANT TO ENTER CRITICAL SECTION: need for $\text{flag}[0]$ and $\text{flag}[1]$...

Faster process that is polite wins!

Polite means If I am 0, I say '1's turn (turn=1)'

If I am 1, I say 'o's turn (turn=0)'

The faster who says so (i.e. your turn) wins!!!

Process 0 sets turn=1 before

Process 1 sets turn=0 ----> turn=0 is last and
Process 0 wins!!

Two sets of variables : **turn** and **flag[0]/flag[1]**

flag[i] : Process i is interested in getting into c.s.
(c.s. : critical section)

MS.1

Peterson's algorithm

```
boolean flag [2];
int turn;
void P0()
{
    while (true) {
        flag [0] = true;
        turn = 1;
        while (flag [1] && turn == 1) /* do nothing */;
        /* critical section */;
        flag [0] = false;
        /* remainder */;
    }
}
void P1()
{
    while (true) {
        flag [1] = true;
        turn = 0;
        while (flag [0] && turn == 0) /* do nothing */;
        /* critical section */;
        flag [1] = false;
        /* remainder */;
    }
}
void main()
{
    flag [0] = false;
    flag [1] = false;
    parbegin (P0, P1);
}
```

Figure 5.3 Peterson's Algorithm for Two Processes



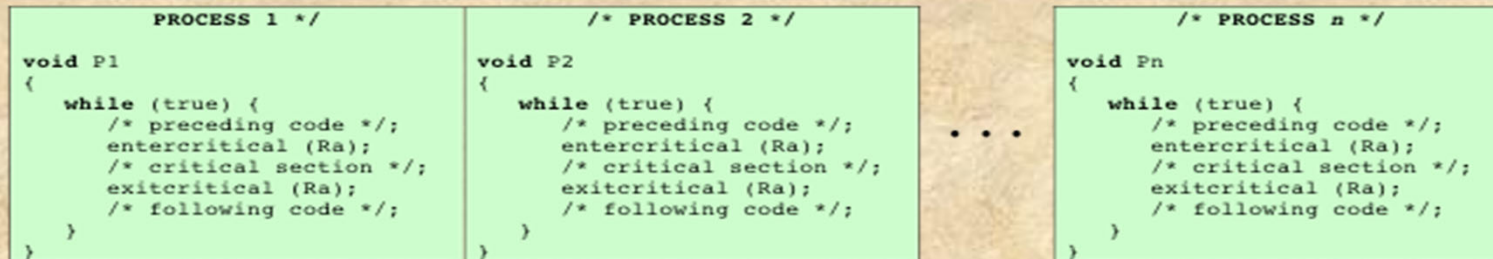
MS.1: Use Peterson's algorithm to solve the **critical section** problem

How ?

MS.1.for.CS Wrap Peterson's code around critical section code

Why? Peterson is extensible to > 2

Issues: Lots of shared `flag[]` variables



```

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 */;
    }
}

```

Figure 5.4 Illustration of Mutual Exclusion

MS.1.for.CS

Issues:

- ... critical section wraparound code
- ... shared variable(s)
- ... while : **Busy Wait!**



MS. Software Approaches

MS.1.for.CS

Issues:

- ... critical section wraparound code
- ... shared variable(s)
- ... while : **Busy Wait!**



MH. Hardware Approaches for mutual exclusion

MH.1 Disable Interrupts (hardware solution)

MH.2 SpinLocks

Spinlocking with hardware support

MH.3 Intel's XCHG

MH.4 Compare and Swap (C&S) (not covered)

MH.5 Test&Set (not covered)



```
Process (int i)
```

```
    while (true) {
```

```
1.    DisableInterrupts;
```

```
2.    CriticalSection; //Usually takes secs
```

```
3.    Enable Interrupts;
```

```
    }
```

Drastic Action

No multiprogramming, multitasking for
duration interrupts are disabled;

If process dies; system is unavailable.



General Setup

`Process (int i) // i-th Process`

General Startup

```
int main(void) {  
    bolt = 0;           // 0 is Unlocked/Unbolted  
                       // 1 is Locked/Bolted  
    in parallel {Process(0), Process(1) , ... Process(n-1)};  
}
```

`bolt 0:` "Door" is Unlocked; no process in c.s.

`bolt 1:` "Door" is locked; process in c.s.



```
Lock    bolt= Unlocked;
```

```
Process (int i)
  while (true) {
1. Acquire(bolt);    //bolt=Locked : 1
2. Critical section;
3. Release(bolt);    //bolt=Unlocked :0
  }
```

Acquire(bolt), Release(bolt)
must be atomic operations or
two processes get the lock at the same time and
both turn bolt to Locked.

Spinlock == Busywait



XCHG (register, memory)
temp = memory;
memory = register;
register=temp;

Assumes XCHG

- (a) is an atomic instruction (hardware)
- (b) thus no interrupt for its operations



```
Process (int i)
  while (true) {
1. int  flag[i] = 1;
2. do XCHG(&bolt,&flag[i]);
3. while  (flag[i] != 0)
4. CriticalSection;    // LOCKED:  bolt is 1
5. bolt = 0;           //UNLOCKED: bolt is 0
  }
```

Atomicity: The only process that finds **bolt == 0** enters CS after setting **bolt=1**; after completion of CS it resets **bolt=0**.

Lines 2-3 : busywait i.e. spinlock using a hardware XCHG



COMPARE&SWAP (old, test, new)

copyold = old;

if old == test old=new;

return(copyold);

Assumes Compare&Swap

(a) is an atomic instruction (hardware)

(b) thus no interrupt for its operations

Mutual Exclusion: Hardware Support

■ Compare&Swap Instruction

MH.3

- 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 (not subject to interruption)



```
Process (int i)
  while (true) {
1. while COMPARE&SWAP (bolt,0,1)==1) {;}
2. CriticalSection;    // LOCKED:  bolt is 1
3. bolt = 0;           //UNLOCKED: bolt is 0
  }
```

Atomicity: The only process that finds `bolt == 0` enters CS after setting `bolt=1`; at completion of CS it resets `bolt=0`.

Line 1: Busywait/Spinlock using a hardware COMPARE&SWAP



```
Process (int i)
  while (true) {
1. while COMPARE&SWAP (bolt,0,1)==1) {block();}
2. CriticalSection;    // LOCKED:  bolt is 1
3. bolt = 0;           //UNLOCKED: bolt is 0
  }
```

Atomicity: The only process that finds `bolt == 0` enters CS after setting `bolt=1`; at completion of CS it resets `bolt=0`.

Line 1: Blocking Lock : causes a context switch!



MH. Hardware Approaches

MH.*.for.CS

Issues:

- ... MH₁: Disabling of interrupts is drastic
- ... MH*: Starvation
- ... MH*: Busy Wait (while, do.while)
- ... MH*: Deadlock

Example: P₁ low priority hold the bolt
P₂ high priority preempts P₁ but blocked
can't get bolt that is held by P₁

Result: Deadlock

Practice 1



- 1.1. What is busy wait?
- 1.2 Name the mechanisms for mutual exclusion that suffer from busy wait.
- 1.3 What is an alternative to busy waiting?



MP. Programming Language approaches for mutual exclusion

MP.1 Semaphores

MP.2 Monitor

MP.3 Message Passing (mailbox: mbox)

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 <i>critical sections</i> . 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.

Table 5.3

Common

Concurrency

Mechanisms

On definitions...part 1



MutualExclusion (**MutEx**) exclusive solutions ...

Lock(s) : Is used to synchronize a process (and its threads) i.e multiple tasks

Locks (here) are spinlocks (busywait)

Software locks ... (Peterson's @Sub11 , Dekker's in textbook)

All are busy-wait aka software-only 'spinlock'

Hardware locks... (XCHG,Cmp&Swap, Test&Set)

As described all are busy-wait aka hardware-supported 'spinlock'

Hardware-based..(Interrupt Enabling and Disabling)

Eliminates the possibility of needing mutual exclusion

Locks in general can be

spinlocks

blocking locks (see next page)

hybrid (spin first then block)

On definitions...part 1



MutualExclusion (**MutEx**) + more solutions ...

Lock(s) : Are used to synchronize multiple processes (and their threads)

“Locking” here is not a spinlock; task gets blocked (‘sleep’)

mutex MutEx with a lock + communication capable ;
tasks get blocked no-spinlock (semaphore-like)

Note: pthreads mutex by default might not be shareable by processes (but can be shared if chosen so)

Rule of thumb: For same address space synch use mutex; for different use a semaphore

Semaphore MutEx no busy-wait/nospinlock, counting semaphore primarily
task communication

Monitor



Blocking operations

```
semaphore    var;
```

```
semWait (var)    // Sleep
semSignal(var)   // Signal
semInit(var,value) // Init
```

```
class semaphore {
private:
    int count;
    queueType queue;

public void semaphore (int v) {           // semInit
    count = v;
    queue = NULL;
}
public void semWait();
public void semSignal();
}
```

count: a negative value means
processes blocked on queue

count: 0 no resource available

count: 1 or more : # resources available

binary semaphore: count 0 or 1 or negative

Binary semaphore is a **lock**!

counting semaphore: count 0,1,2,....

Lock vs Semaphore

Lock

BusyWait

or

Enable/Disable Int

Semaphore

No BusyWait

(process blocks)

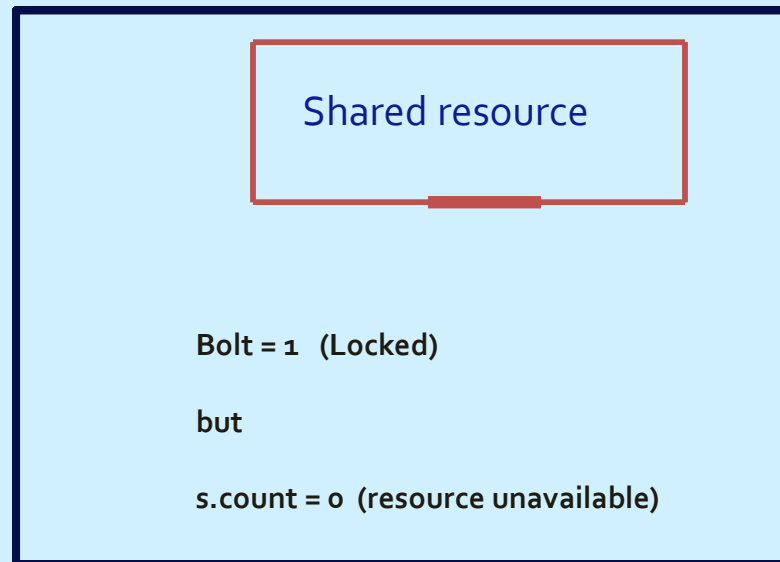
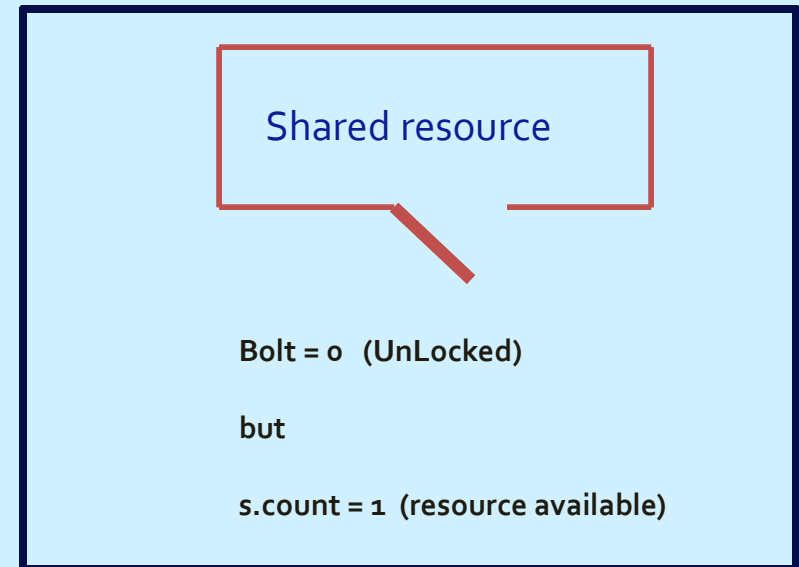
Confusing to use

Strong semaphore

Weak semaphore

queue is FIFO

queue is ?



Above: binary semaphore s (Note : book says $s=0$ instead of $s.count = 0$)

Two values for $s.count$: $s.count=1$ resource is available (door Unlocked aka Bolt=0) else $s.count=0$ resource unavailable (Bolt=1)

Counting Semaphore behavior:

Producer tasks (process or thread) issue `semSignal`

Consumer tasks issue a `semWait`

Binary semaphore:

One process "locks" others get blocked, another may "unlock"

if same process `semWait` x2, second time is blocked unless `semSignal` before

Mutex:

Same process locks others get blocked and one that locks can only unlock

if same process locks twice (recursive lock) must unlock twice before release

Spinlock:

process busywaits and never blocks

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) A **sem**aphore may be **init**ialized to a nonnegative integer value
- 2) The **semWait** operation decrements the semaphore value
- 3) The **semSignal** operation increments the semaphore value

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



```
Process (int i)
```

Use semaphore

```
semaphore s;  
// seminit(s.count,1)  
while (true) {  
1.   semWait(s);  
2.   CriticalSection;  
3.   semSignal(s);  
}
```

s with s.count=1

to protect the
critical section

Note 1: The use of the semaphore above is that of a lock to protect the critical section.

Note 2: In a more general setting one or more tasks generate(s) a resource (semSignal) and one or more tasks consume resources (semWait); think of a linked list or an array being used to accept generated resources but also used for the consumption of such resources.



```
globalid++; myid=globalid;
printf("Hello from Thread
mainID=%2s , gid=%2d of %2d
; PTID=%ld \n",
(char*) ptr, myid,
NTHREADS, pthread_self());
```

Bolt = 1 (Locked)

but

s.count = 0 (resource unavailable)

```
globalid++; myid=globalid;
printf("Hello from Thread
mainID=%2s , gid=%2d of %2d
; PTID=%ld \n",
(char*) ptr, myid,
NTHREADS, pthread_self());
```

Bolt = 0 (UnLocked)

but

s.count = 1 (resource available)

MP.1. Semaphore semantics

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

Figure 5.6 A Definition of Semaphore Primitives

To guarantee **ATOMIC**ness use a mutual exclusion mechanism ie. a lock...or disable interrupts

```
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

Figure 5.17 Two Possible Implementations of Semaphores

MP.1

Semaphore implementation



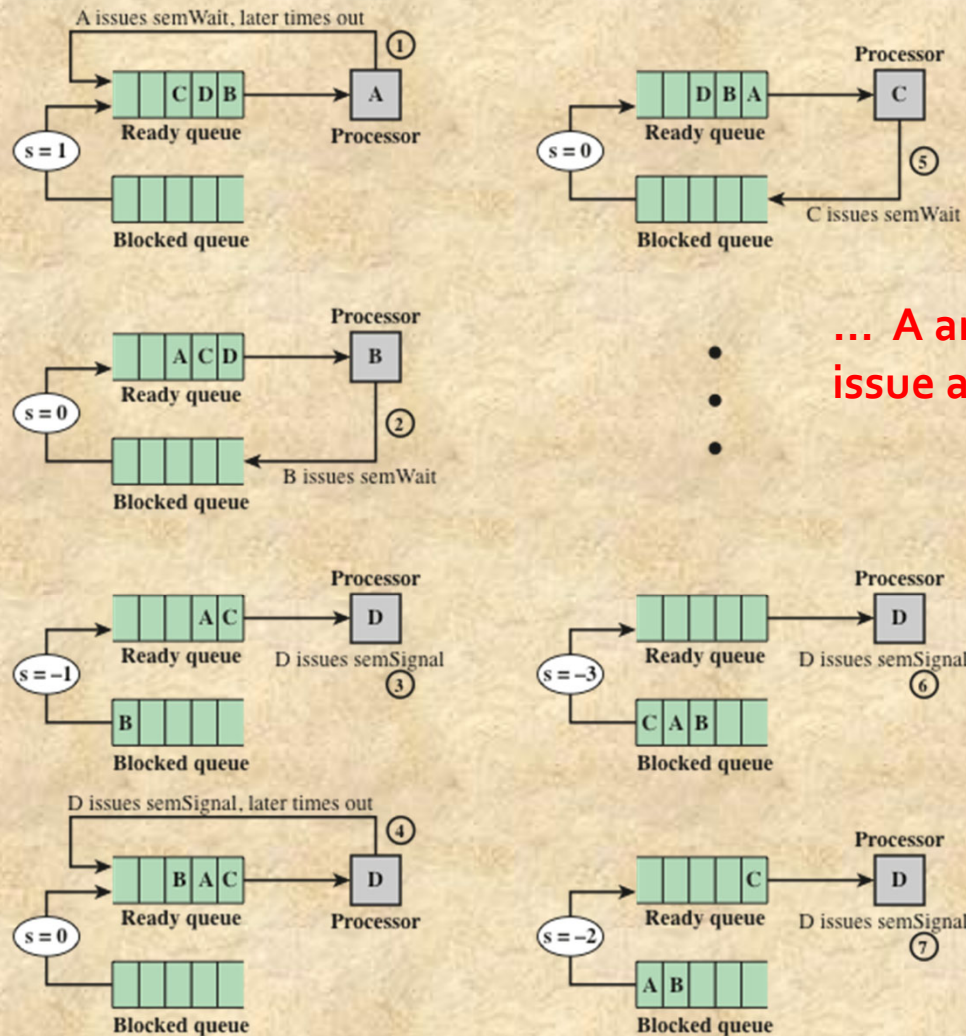
Next page

**Process D is producer (semSignal)
Processes A,B,C are consumers (semWait)**

though

**All might want to access shared variables
for Insert (or Enqueue or Append) or
for Delete (or Dequeue or Take)**

MP.1



... A and B also
issue a semWait

Figure 5.8 Example of Semaphore Mechanism

Producer/Consumer Problem

General Statement:

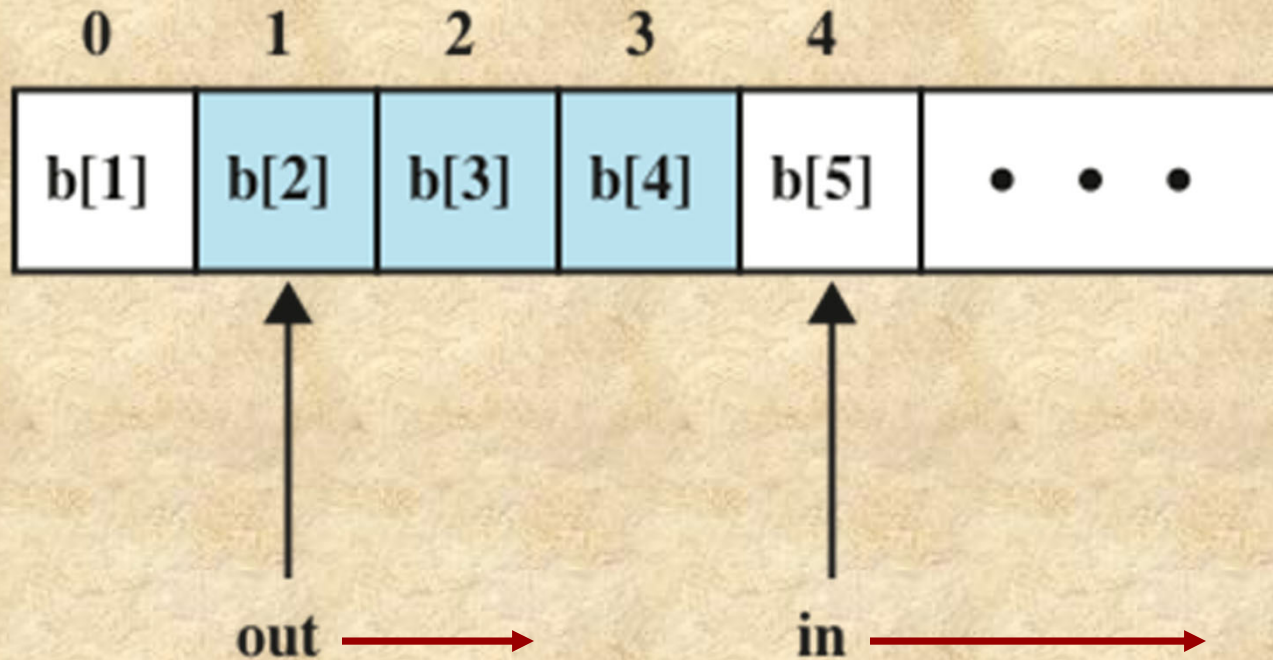
One or more producers are generating data and placing these in a buffer

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 won't try to add data into the buffer if its full, and that the consumer won't try to remove data from an empty buffer



Note: shaded area indicates portion of buffer that is occupied

Figure 5.11 Infinite Buffer for the Producer/Consumer Problem



(previous page: Indexing is C/C++/Java 0,1,2,3 but array contents are indexed Pascal-style b[1],b[2],... !!!!!!!)

Case 1: Infinite sized Queue / Buffer

use two semaphores

- s** to protect the critical section
(of the producer/consumer
append/take insert/delete code)
- n** to protect the queue from an attempt of
underflow (consume an item from empty Q)

Initial values: **s is 1** ; critical section available;
Initial values: **n is 0** ; queue is empty; do not
consume!

```
/* 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 5.14 A Solution to the Infinite-Buffer Producer/Consumer Problem Using Semaphores



Case 2: Finite sized Queue
use three semaphores

s as before
n as before
e to protect queue from an attempt of overflow
 (produce an item and try to insert it
 into a full queue)

Initial values: s, n as before (1 and 0 resp.)

Initial values: **e = lengthofQueue** (#of available slots)

```

/* 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);
}

```

Figure 5.16 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Semaphores



A **monitor** is

a **lock** for mutual exclusion,
a **queue** associated with the lock and
one (implicit) or more **condition variables** each with its own queue

1. Monitor data accessible by monitor functions ONLY (aka class).
2. A process enters a monitor by invoking a monitor function (aka class function)
3. Only one process can enter a monitor and thus execute only one monitor function all other processes are blocked on the queue of the monitor
(monitor enforced synchronization: Java synchronized).

- 1 and 2 : encapsulation : easily implementable using OOP (Java, C++).
3. Monitor provides mutual exclusion facility to its data.
 3. Monitor provides synchronization mechanism with wait and signal

(textbook: cwait and csignal, Java: wait and notify)

Monitors

- Programming language construct that provides equivalent functionality to that of semaphores and is easier to control
- Implemented in a number of programming languages
 - Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, 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

- A monitor supports synchronization by the use of **condition variables** that are contained within the monitor and accessible only within the monitor
 - Condition variables are a special data type in monitors which 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

Monitors



cwait(c)

(sleep)

Suspend execution
of calling process
on condition c;
monitor available
for others

condition variables states

csignal(c)[Java:notify]

(wakeup)

resume execution of
some process waiting
on c; if more than one
pick arbitrary, if none
ignore signal

csignalAll(c)[Java:notifyall]

(wakeup broadcast)

csignal: Hoare monitor signal

cnotify: Mesa or Hansen monitor signal

Monitors: Hoare vs Mesa



Hoare (if statement)

signaling Process give up lock

a waiting Process gets it
waiting Process executes cs
and releases lock

signaling Process gets it back (at some point or Urgent queue is available)

Uses if statement ...

Makes theory easy

Mesa [aka Hansen,Java] (while statement)

signaling Process keeps lock

waiting Process ... keeps on waiting

Uses while statement ...as needed

It's practical (Java)

Infinite buffer case uses notify



```
// FIFO queue GENERIC @CS114
class Q {

    data;

    public void Enq(Object o){

        Append o

    }

    public Object Deq () {

        if not EMPTY(Q) {

            Take o

            return o

        }

    }

}
```

```
// GENERIC + 'THREAD SAFE' at @CS332 mid Subject11
class Q {
    semaphore n=0,s=1;
    data;
    public void Enq(Object o){
        semWait(s);
        Append o
        semSignal(s);
        semSignal(n);
    }

    public Object Deq () {

        if notEMPTY(Q) {
            semWait(n);
            semWait(s);
            Take o
            semSignal(s);

            return o

        }

    }

}
```

```
// JAVA INTERPRETATION of a Monitor solution
class Q {

    private data;
    public void synchronized Enq(Object o){

        Append o

        notify() /*<-some code after notify---->*/;

    }

    public Object synchronized Deq()

        while EMPTY(Q)
            wait();

        Take o

        return o

    }

}
```

Java scenario	CodeLine	RunngQ	ReadyQ	MonitorQueue	Q
t=1: Deq()	while ... wait()	P20	head: P20, P30, P10, P40	-	-
t=2: Deq()	while ... wait()	P30	head: P30,P10,P40		
t=3: Enq(101)	Append: 101	P10	head: P40	P20	
t=3+:Enq(101)	notify()	P10	head: P40	P20,P30	101
t=3++Enq(101)	}	P10	head: P40,P20?,	P20,P30 (who wakes up?)	101
t=4 ??Dequeue??		P40	head: P20, P10	P30 (if P20 wakes up)	101
				P30	??Empty??



```
// FIFO queue GENERIC @CS114
class Q {
```

```
    data ;
    public void Enq(Object o){
```

```
        Append o
```

```
    }
```

```
    public Object Deq () {
```

```
        if not EMPTY(Q) {
```

```
            Take o
```

```
            return o
```

```
        }
```

```
    }
```

SCENARIO 0

```
// GENERIC + 'THREAD SAFE' at @CS332 mid Subject11
```

```
class Q {
    semaphore n=0,s=1;
    data ;
```

```
    public void Enq(Object o){
```

```
        semWait(s);
```

```
        Append o
```

```
        semSignal(s);
```

```
        semSignal(n);
```

```
    }
```

```
    public Object Deq () {
```

```
        if not EMPTY(Q) { // What if block on EMPTY is needed?
```

```
            semWait(n);
```

```
            semWait(s);
```

```
            Take o
```

```
            semSignal(s);
```

```
            return o
```

```
        }
```

```
    }
```

SCENARIO 2

```
// JAVA INTERPRETATION of a Monitor solution
```

```
class Q {
```

```
    private data;
```

```
    public void synchronized Enq(Object o){
```

```
        Append o
```

```
        notifyAll() /*<-some code after notify---->*/;
```

```
    }
```

```
    public Object synchronized Deq()
```

```
        while EMPTY(Q)
```

```
            wait();
```

```
        Take o
```

```
        return o
```

```
    }
```

```
    }
```

SCENARIO 3; generalized SCENARIO 1

Java scenario	CodeLine	RunngQ	ReadyQ	MonitorQueue	Q
t=1: Deq()	while ... wait()	P20	head: P20, P30, P10, P40	-	-
t=2: Deq()	while ... wait()	P30	head: P30,P10,P40	-	-
t=3: Enq(101)	Append: 101	P10	head: P40	P20	-
t=3+:Enq(101)	notifyAll()	P10	head: P40	P20,P30	101
t=3++Eqn(101)	}	P10	head: P40,P20,P30	P20,P30	101
t=4 ??		P40	head: P20, P30,P10	-	101
					?101?



Scenario 1 (simple) (Mesa / Hansen Monitor)

P10 Notify + exits
P20 wakes-up + Deq+exits

1a.1 P20 wakes up as soon as P10 returns from monitor Enq

1a.2 P20 executes while one more iteration and exitswhile, dequeues 101 and returns

1a.3 P10 first completed, leading to completion of P20 of monitor execution

Scenario 2 (simple;not Java) (Hoare)

P10 Notify + not enough time to exit
P20 wakes-up + Deq + exits
P10 exits

1c.1 P20 wakes up as soon as P10 completes notify() but before P10 returns from monitor Enq

1c.2 P20 executes while one more iteration and exitswhile, dequeues 101 and returns
1c.3 P10 takes over to complete the Enq and returns

1c.4 P20 first completed, then P10 immediately after

Scenario 3 (Java execution) (Mesa/Hansen Monitor)

P10 Notify + exits
P20 wakes-up but P40 grabs CPU first
P40 Deq + exits
P20 Deq + runs while loop + wait() again+ sleeps again!

1e.1 P20 wakes us as soon as P10 returns from monitor Enq

1e.2 P40 however executes Deq and returns

1e.3 P20 executes while one more iteration and runs wait() again and sleeps...

NOT POSSIBLE in JAVA

Requirements

1. while statement around wait()
2. notify() last statement in function

Problems

1. Timing issues
2. No unnecessary context switches
3. Unpredictability

1. if statement around wait suffices
- 2 notify() must be last statement in function to avoid problems
3. **Precise timing + tracing possible;**

Assumes no P40 interference:theoretical!

1. No timing issues
2. Context switches: P10 switches out, and later on switches in after P20 switches out!
3. No unpredictability

1. Explains why a while is needed (P20 sleeps again)

**// Non-multithreaded****JAVA CLASS**

```

class Q {
    data ;

    public void Enq(Object o){
        Append o
    }

    public Object Deq () {
        if not EMPTY(Q) {

            Take o
            return o;
        }
    }
}

```

// Java monitor : GOOD JAVA**JAVA MONITOR**

```

class Q {
    private data;
    // ? condition variable(s) ?
    public void synchronized Enq(Object o){
        Append o
        notifyAll();
    }

    public Object synchronized Deq()
        while EMPTY(Q)
            wait();
        Take o
        return o;
    }
}

```

Condition variables



Java monitor : GOOD Java

```
class Q {
    private data;

    public void synchronized Enq(Object o){
        Append o
        notifyAll();
    }

    public Object synchronized Deq()
    while EMPTY(Q)
        wait();
    Take o
    return o
    }
}
```

Java monitor+condition variables: BETTER JAVA

```
class Q {
    private data;
    condition empty;

    public void synchronized Enq(Object o){
        Append o
        empty.notifyAll();
    }

    public Object synchronized Deq()
    while EMPTY(Q)
        empty.wait();
    Take o
    return o
    }
}
```



Java monitor + Condition variables BEST JAVA

```
class Q {
    private data;
    condition empty;
    condition full;
    public void synchronized Enq(Object o){
```

```
    Append o
    empty.notifyAll();
}
```

```
    public Object synchronized Deq()
    while EMPTY(Q)
        empty.wait();
```

```
    Take o
```

```
    return o
}

}
```

Monitors vs Semaphores



Semaphore

Signal with no Process waiting:
s.count++

Mutual Exclusion

by User (Process)

Each semaphore has its queue

Synchronization

User: SemWait

User: SemSignal

Monitor

Signal with no Process waiting
ignored

Mutual Exclusion

by Monitor

Each monitor + condition var has its
own queue

Synchronization

	Generic	Java version
User:	cwait	or wait
User:	csignal	or notify (notifyAll)

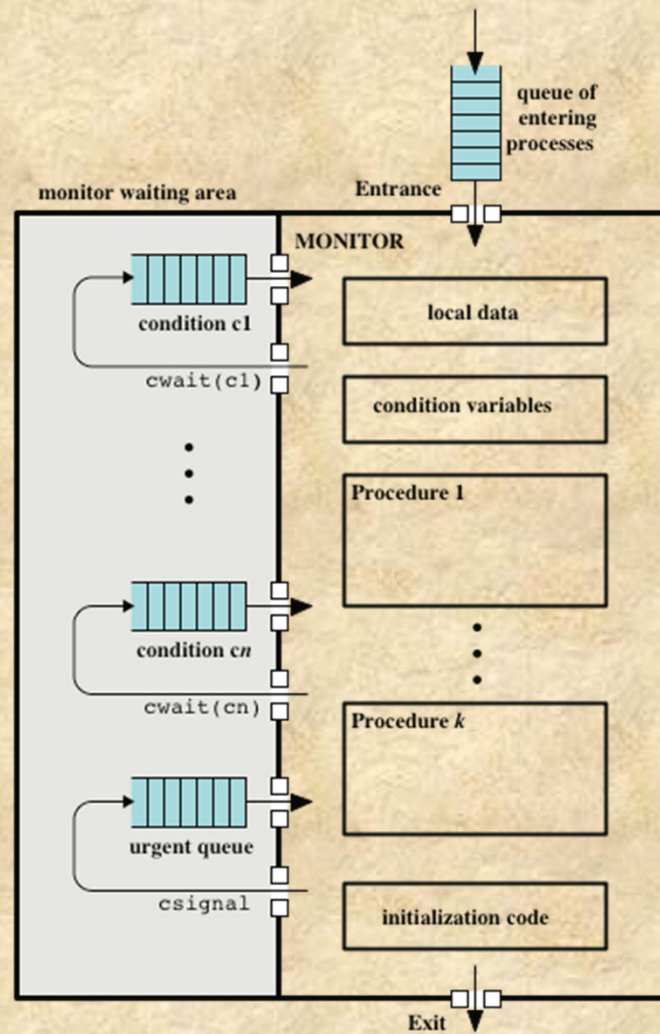


Figure 5.18 Structure of a Monitor

```

/* 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);
}

```

```

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 */
}

```

Figure 5.20 Bounded Buffer Monitor Code for Mesa Monitor

Figure 5.19 A Solution to the Bounded-Buffer Producer/Consumer Problem
Using a Monitor

MP.3 Message Passing



Placeholder

Skip this page

Message Passing

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

Synchronization

- To enforce mutual exclusion

Communication

- 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*

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

Table 5.5
Design Characteristics of Message Systems for
Interprocess Communication and Synchronization

Synchronization

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

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

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

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:

pid1@src:
send(pid2@dest, message)

pid2@dest
receive(pid1@src, message)

(direct addressing)

Same computer : src is dest

Direct
addressing

Indirect
addressing

pid1:
send(dest:mailbox , message)
or
send(dest:socket25, message)

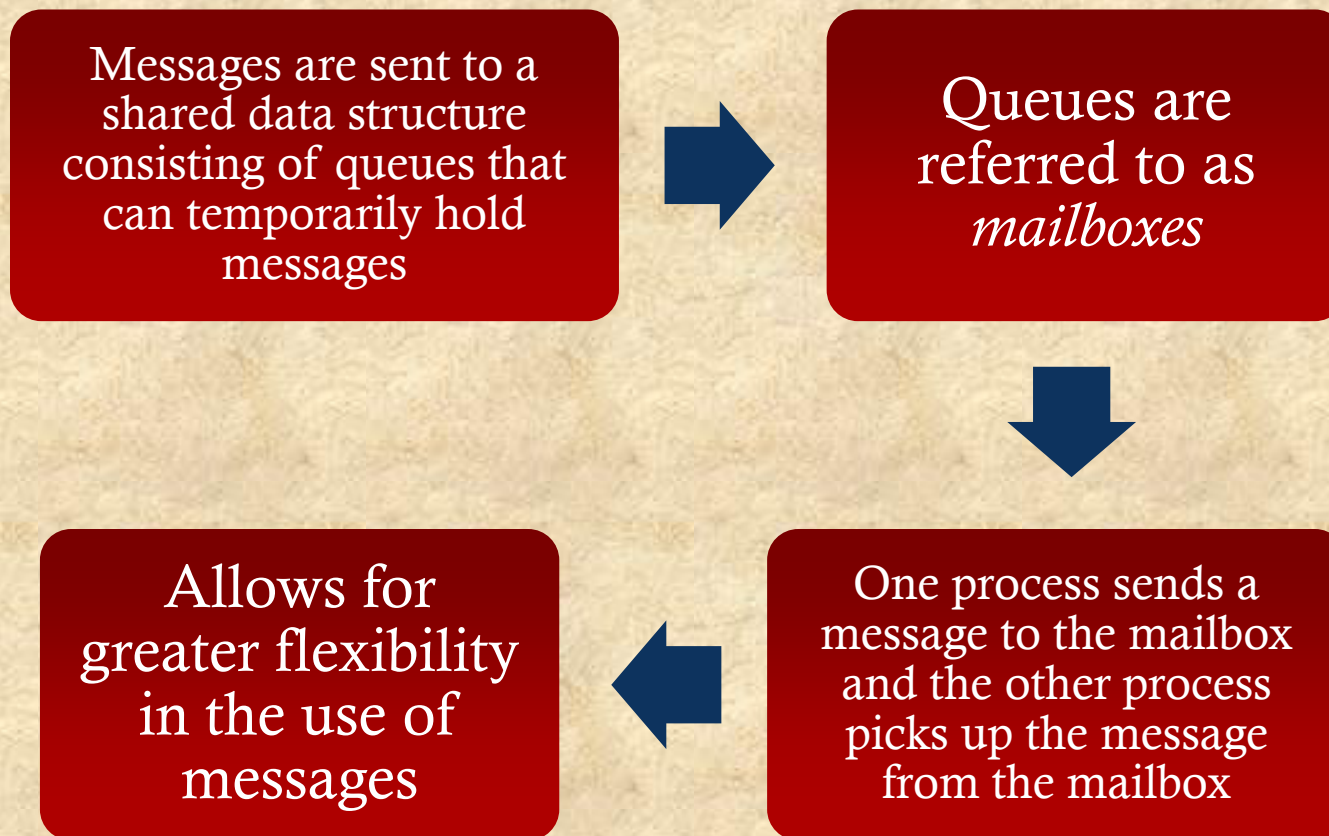
dest:
receive(mailbox,message)
or
receive(socket25, message)

(indirect addressing)

Direct Addressing

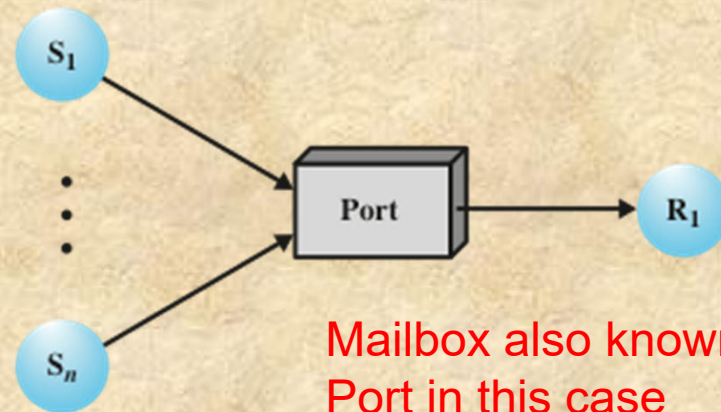
- Send primitive includes a specific identifier of the destination process
- Receive primitive can be handled in one of two ways:
 - Require that the process explicitly designate a sending process
 - Effective for cooperating concurrent processes
 - Implicit addressing
 - Source parameter of the receive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

Indirect Addressing



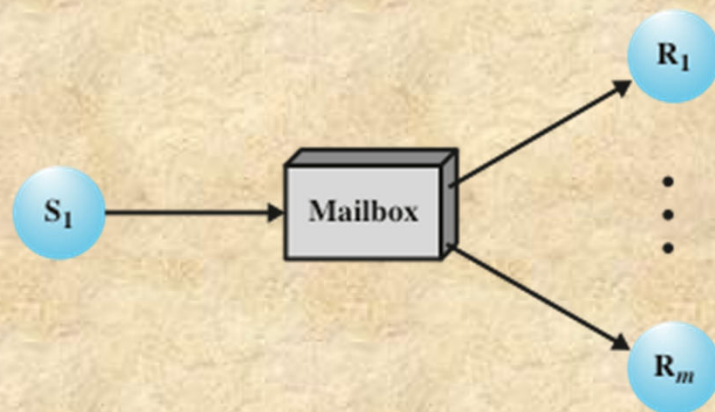


(a) One to one

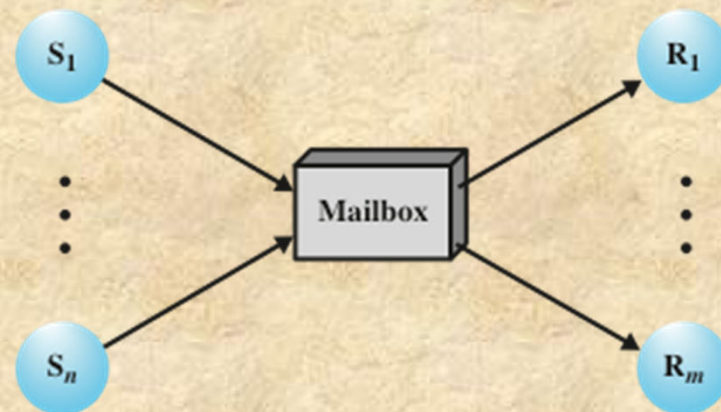


(b) Many to one

Mailbox also known as
Port in this case



(c) One to many



(d) Many to many

Figure 5.21 Indirect Process Communication

Queueing Discipline

- The simplest queueing discipline is first-in-first-out
 - This may not be sufficient if some message are more urgent than others
- Other alternatives are:
 - To allow the specifying of message priority, on the basis of message type or by designation by the sender
 - To allow the receiver to inspect the message queue and select which message to receive next

MP.3.for.CS Message Passing

```
/* 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 5.23 Mutual Exclusion Using Messages

```

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



MP. Progr. Languages

- MP.1** Atomic or MH.* approach for mutual exclusion,
no BusyWait,
confusing to use
- MP.2** Synchronization issues but limited to Monitor
Hoare monitors have scheduling issues
- MP.3** Enforce mutual exclusion easily
Means for interprocess communication

Practice 2



- 2.1. What is the difference between a semaphore and a mutex?
- 2.2 What are the three operations of a semaphore?
- 2.3 How many processes can be active in a monitor at any time ? 0, 1, 2, or more than 2?

NJIT

A white curved line that starts under the 'N', sweeps under the 'J' and 'I', and ends under the 'T'.

New Jersey's Science &
Technology University

THE EDGE IN KNOWLEDGE

Alex. Gerbessiotis

CS Department



Peterson n procs (fair)

Level 0 -> 1 -> 2 -> ... -> n-1 -> wrap around 0

level[*] initialized to -1;

P(i)

while (1) {

for(j=0; j < n ; j++) {

level[i] = j;

last [j] = i;

while ((last[j]==i) && (k: k!=i level[k] >= j)) { ; }

}

CriticalSection();

level[i] = -1;

}