# Concurrency: Mutual Exclusion and Synchronization

Chapter 5

# Concurrency

- ✓ Communication among processes
- ✓ Sharing resources
- ✓ Synchronization of multiple processes
- ✓ Allocation of processor time

# Difficulties with Concurrency

- ✓ Sharing global resources
- ✓ Management of allocation of resources
- ✓ Programming errors difficult to locate

# A Simple Example

**Process 1**

```
input(in, keyboard);
...
out = in;
...
output(out, display)
...
...
```

**Process 2**

```
...
input(in,keyboard);
...
out = in;
...
output(out, display);
...
```

- ***What is the problem here?***

# Operating System Concerns

- Keep track of active processes
- Allocate and deallocate resources
  - processor time
  - memory
  - files
  - I/O devices
- Protect data and resources
- Result of process execution must be independent of the speed of execution and timings of events, since other processes share the processor time

# Process Interaction Types

- ✓ Processes *completely unaware* of each other
- ✓ Processes *indirectly aware* of each other
- ✓ Process *directly aware* of each other

# Competition Among Processes for Resources

- ✓ Execution of one process may affect the behavior of competing processes
- ✓ If two processes wish to access the same *non-sharable* resource, one process will be allocated the resource and the other will have to wait
- ✓ Non-sharable means: can’t be used simultaneously by different processes
- ✉ The blocked process *might* never get access to the resource and never terminate

# Control Problems

- ***Mutual Exclusion***
  - achieved using *critical sections*
    - only one program at a time is allowed in its critical section
    - example only one process at a time is allowed to send command to the printer
- ***Deadlock***
- ***Starvation***

# Cooperation Among Processes by Sharing ***Non-sharable*** Resources

- ✓ Processes use and update shared data such as shared variables, files, and data bases (Note: *as resources*, these items are treated as non-sharable!)
- ✓ Writing must be mutually exclusive
- ✓ Critical sections are used to provide mutual exclusion on data

# Cooperation Among Processes by Communication

- ✓ Communication provides a way to synchronize, or coordinate, the various activities
- ✓ Possible to have *deadlock*
  - each process might be waiting for a message from the other process
- ✓ Possible to have *starvation*
  - two processes sending message to each other while another process waits for a message

# Requirements for Mutual Exclusion

- ✓ Only one process at a time is allowed in the critical section for a resource
- ✓ If a process halts in its critical section, it must not lock out other processed forever
- ✓ A process requiring the critical section must not be delayed indefinitely---no deadlock or starvation

# Requirements for Mutual Exclusion

- A process must not be delayed 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 amount of time only

# Busy-Waiting

Example:

Igloo has small entrance so only one process at a time may enter to check a value written on the blackboard.  If the value on the blackboard is the same as the process, the process may proceed to the critical section.

If the value on the blackboard is not the value of the process, the process leaves the igloo to wait.  From time to time, the process reenters the igloo to check the blackboard.

# Igloo in Formal Terms

```
binary turn;   // binary type is enum{0,1}
```

**Process 0**

```
⋮
while turn != 0 do { }
<critical section>
turn = 1;
...
```

**Process 1**

```
⋮
while turn != 1 do { }
<critical section>
turn = 0;
...
```

# Busy-Waiting Problems

- ✓ Processes must strictly alternate in their use of their critical section
- ✓ If one process fails, the other process is permanently blocked
- ✓ Each process should have its own key to the critical section so if one process is eliminated, the other can still access its critical section

# Busy-Waiting: Second Attempt

- Each process can examine the other's status but cannot alter it
- When a process wants to enter the critical section is checks the other processes first
- If no other process is in the critical section, it sets its status for the critical section

# Busy Waiting: Second Attempt

boolean `flag[2];   // initially all false`

Process 0

```
:
while flag[1] do { }
flag[0] = true;
<critical section>
flag[0] = false;
...
```

Process 1

```
:
while flag[0] do { }
flag[1] = true
<critical section>
flag[1] = false;
...
```

- This method does not guarantee mutual exclusion:
  *Each process can check the flags and then proceed to enter the critical section at the same time*

# Busy-Waiting Third Attempt

- Set flag to enter critical section before checking other processes
- If another process is in the critical section when the flag is set, the process is blocked until the other process releases the critical section

# Busy Waiting: Third Attempt

boolean `flag[2];`
`(initially all false)`

Process 0

```
:
:
flag[0] = true;
while flag[1] do { }
 <critical section>
flag[0] = false;
...
```

Process 1

```
:
:
flag[1] = true
while flag[0] do { }
<critical section>
flag[1] = false;
...
```

- **Deadlock is possible**
  *when two process set their flags to enter the critical section. Now each process must wait for the other process to release the critical section*

# Busy-Waiting Fourth Attempt

- A process sets its flag to *true* to indicate the desire to enter its critical section, but is prepared to reset the flag
- Other processes are checked.  If they are in the critical region, the flag is reset back to *false* and later is set to indicate the desire to enter the critical region.
- This is repeated until the process can enter the critical region.

# Busy Waiting: Fourth Attempt

Process 0

```
:
flag[0] = true;
while flag[1] do {
   flag[0]= false;
   <random delay>
   flag[0]=true;
 }
<critical section>
flag[0] = false;
...
```

Process 1

```
:
flag[1] = true
while flag[0] do {
 flag[1]= false;
  <random delay>
  flag[1]=true;
 }
<critical section>
flag[1] = false;
...
```

- *It is possible for each process to set their flag, check other processes, and reset their flags. Neither process waits for the other to finish -- it is not a deadlock. It is a livelock ! Still, undesirable*

# Busy-Waiting: Correct Solution

- ✓ Each process gets a <u>turn</u> at the critical section
- ✓ If a process wants the critical section, it sets its <u>flag</u> and may have to wait for its turn
- ✓ Must combine the `turn` and `flag` variables

✉ Read about Decker's and Peterson's algorithms (which provide the correct solutions) in the textbook!

# Mutual Exclusion - Interrupt Disabling

- A process runs until it invokes an operating-system service or until it is interrupted
- Disabling interrupts guarantees mutual exclusion
- Problems:
  - Limits the processor in its ability to interleave programs
  - Efficiency of execution could be noticeably degraded
  - Multiprocessing:
    - disabling interrupts on just one processor will *not* guarantee mutual exclusion (Why?)

# Mutual Exclusion Machine Instructions

- One special machine instruction is used to update a memory location so other instructions cannot interfere
- This can be used for single and multiple processors
- Can be used for multiple critical sections

# Test and Set Machine Instruction

## Test and Set:

```
bool testNset (int i)
{
   if i == 0 then {
     i = 1;
      return(true)
   }
   return (false)
}
```

- This must be done *atomically* !

## Each process:

```
int mutex = 0; // shared var

repeat {} until testNset(mutex)
<critical section>
mutex = 0;
<rest of the process>
```

A process can enter critical section only when `mutex` becomes `0` and that process gets to execute `testNset`

(Only one process can do that!)

# Mutual Exclusion Machine Instructions

- *Disadvantages*
  - Other processes must use busy-waiting while trying to execute test-and-set
  - Starvation is possible: when a process leaves a critical section and more than one process is waiting, the next processor to execute test-and-set is chosen randomly by hardware
  - Deadlock is possible: If a low priority process P1 is in the critical region and a higher priority process P2 needs CPU, P2 will obtain the processor. If P2 now wants to enter the critical section, it'll be blocked because P1 is still in its critical region

# Semaphores

- Special variable called a *semaphore* is used for signaling
- If a process is waiting for a signal, it is suspended until that signal is sent
- `wait` and `signal` operations cannot be interrupted
- Queue is used to hold processes waiting on the semaphore

# Semaphores

✓ **General** semaphore:

// Let P denote current process

```
struct{ int count,
        queue procqueue} s
wait(s):
 s.count--;
 if (s.count < 0){
  <put P in s.procqueue>
  <block P>
 }
signal(s):
 s.count++;
 if (s.count =< 0){
  <remove P from s.procqueue>
   <put P in ready queue>
 }
```

✓ **Binary** semaphore:

```
struct{ binary value,
        queue procqueue} s
wait(s):
if (s.value == 1)
   s.value=0;
else {
    <put P in s.procqueue>
    <block P>
  }

signal(s):
if (!empty(s.procqueue)){
   <remove P from s.procqueue>
   <put P in ready queue>
}
s.value=1
```

# Use of Binary Semaphores

Each process:

```
binary semaphore s = 1;
```

repeat

```
    wait(s);
    <critical section>
    signal(s);
    <rest>
```

forever

# Implementing Semaphores Using Test and Set

```
wait(s):
 repeat {}
   until testNset(s.flag);
 s.count--;
 if (s.count<0) {
   <put current process
             in s.procqueue>
   s.flag=0
   <block this process>
 }
 //enable other testNset ops
 if (s.flag != 0) s.flag=0
```

```
signal(s):
 repeat {}
   until testNset(s.flag);
 s.count++;
 if (s.count=<0) {
   <remove some process P
              from s.procqueue>
    s.flag=0
   <place P in ready queue>
 }
 //enable other testNset ops
 if (s.flag != 0) s.flag=0
```

Why is it OK to busy-loop?

# Implementing Semaphores Using Interrupts

```
Wait(s):
 // Let P be current process
 inhibit interrupts;
 s.count--;
 if s.count<0 {
   <put P in s.procqueue>
   allow interrupts
   <block P>
 }
allow interrupts;
```

```
Signal(s):

 inhibit interrupts;
 s.count++;
 if s.count=<0 {
   <remove some process P
             from s.procqueue>
   allow interrupts
   <place P in ready queue>
 }
 allow interrupts;
```

*Should really disable interrupts on all processors

# Producer/Consumer Problem
## (binary & general semaphores)

- ✓ One or more producers are generating items and place them in a buffer
- ✓ A single consumer is taking items out of the buffer, one at time
- ✓ Only one producer or consumer can access the buffer at any one time
- ✓ Two semaphores are used:
  - one represents the # of items in the buffer (to guard against over/under-flow)
  - one signals that it is all right to use the buffer (to ensure mutual exclusion when producer & consumers access the buffer)

# Producer & Consumer Functions

**producer**:

```
repeat
   <produce item v>
    in++;
   buffer[in] = v;
forever;
```

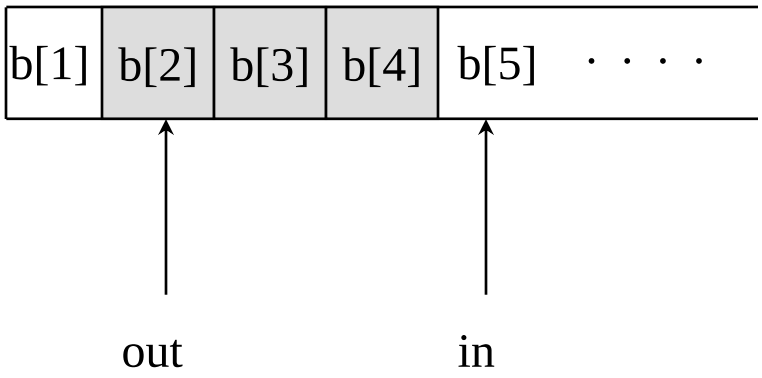
**consumer**:

```
repeat
   while in <= out do {};
   w = buffer[out];
   out++;
   <consume item w>
forever;
```

**Main program:**

```
Parbegin
      producer;
      consumer;
parend
```

# Infinite Buffer




Note: shaded area indicates the portion of the buffer that is occupied

# Producer / Consumer Synchronization With Infinite Buffer

binary semaphore mutex;
semaphore numOfItems

int in, out = 0

**producer**:

```
repeat
   <produce item v>
   wait(mutex);
    in++;
    buffer[in] = v;
   signal(mutex);
   signal(numOfItems);
forever;
```

**consumer**:

```
repeat
   wait(numOfItems);
   wait(mutex);
   w = buffer[out];
   out++;
   signal(mutex);
   <consume item w>
forever;
```

# Producer & Consumer with Circular Buffer

**Producer:**

```
repeat
   <produce item v>
   while(in+1 mod n==out)do{};
   buffer[in] = v;
   in =(in+1)mod n;
forever;
```

**Consumer:**

```
repeat
   while (in==out)do {};
   w = buffer[out];
   out = (out+1) mod n;
   <consume item w>
forever;
```

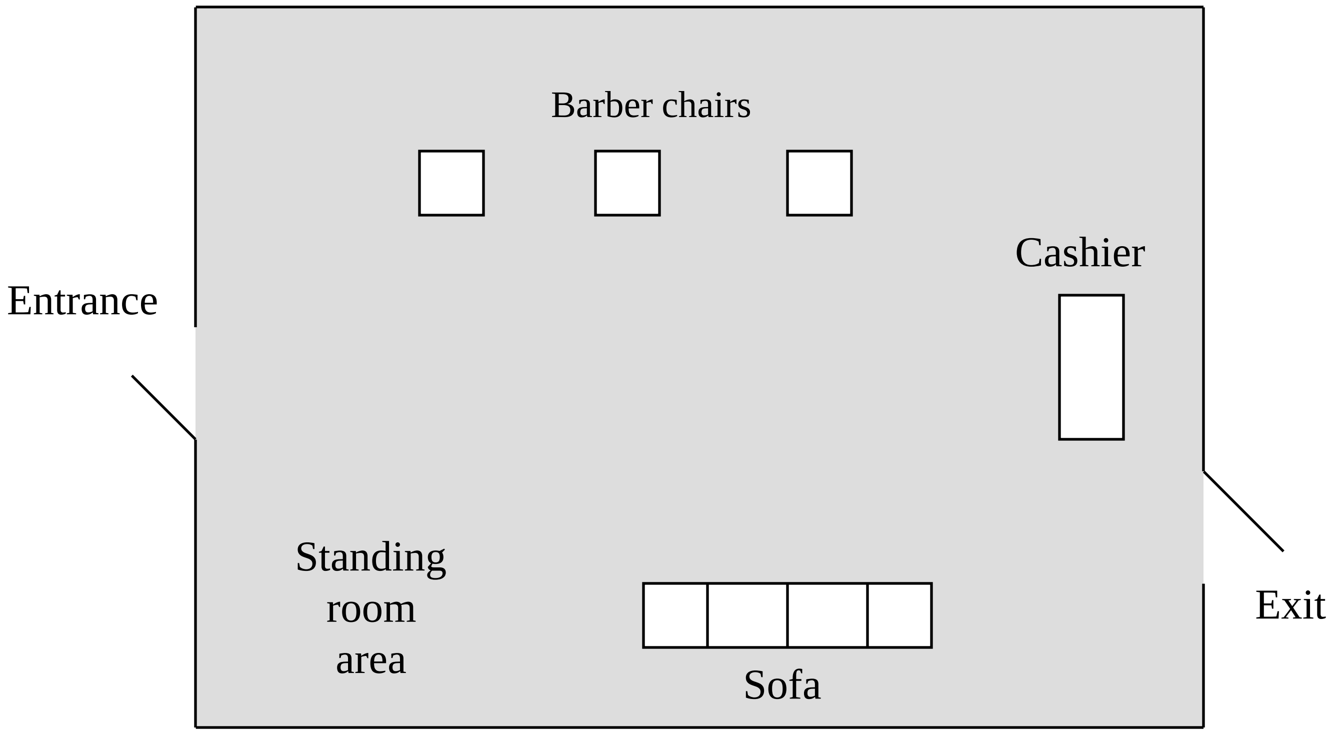
One cell in circular buffer always stays unused

**Main program:**

```
Parbegin
  producer; … producer;
  consumer; … consumer;
parend
```

- To synchronize, we need to sprinkle this with semaphores. *Solution is in the textbook*.

# The Barbershop




3 barbers
1 cashier

4 seats on sofa
20 standing places

# Customer Process

```
semaphore  standing = 20;
semaphore sofa = 4;
semaphore barbChair = 3;
binary semaphore
          custReady = 0,
          barbDone = 0,
          leaveChair = 0,
          payment = 0,
          receipt = 0;
```

Customer:

```
wait(standing);
 <enter shop>
wait(sofa);
 <sit on sofa>
signal(standing);
wait(barbChair);
 <get up from sofa>
signal(sofa);
 <sit in barber chair>
signal(custReady);
wait(barbDone);
  <leave barber chair>
signal(leaveChair);
signal(payment);
  <pay>
wait(receipt);
  <exit shop>
```

# Barber & Cashier

Barber:

```
wait(custReady);
  <cut hair>
signal(barbDone);
wait(leaveChair);
signal(barbChair);
```

Cashier:

```
wait(payment);
  <accept pay>
signal(receipt);
```

**Main program:**

```
parbegin
  customer(1); customer(2); ...  ; customer(100);
  barber(1); barber(2); barber(3);
  cashier;
parend
```

# Monitors

- ✓ Much higher-level synchronization constructs than semaphores.
- ✓ Only one process is allowed to execute inside the monitor at any given time. Other processes are suspended while waiting for the monitor
- ✓ Processes can be suspended while in the monitor. In this case, the process "temporarily leaves" the monitor, so other processes can enter the monitor.

# Monitors

- ✓ Process enters a monitor by calling one of the monitor's procedures
- ✓ Process leaves a monitor when done or when suspended on `condvar.wait`

  (`condvar` is some ***conditional variable*** used in monitors)
- ✓ Process can issue `condvar.signal` before leaving, which wakes up some process previously suspended on `condvar` (if several waiting, one is chosen according to some scheduling strategy)

# Producer/Consumer with Infinite Bufer and Monitors

Consumer:

```
take(x);
<consume x>
```

Producer:

```
<produce x>
append(x);
```

`take`( ) and `append`( ) are procedures defined in the monitor (shown next)

# A Monitor for Infinite Buffers

```
monitor infiniteBuff;
  int in, out = 0;
  int count = 0;
  char *buffer;
  condition notEmpty;

  append(char *x)
  {
    buffer[in] = x;
    in++; count++;
    notEmpty.signal;
  }

take(char *x)
{
  if (count==0)
      notEmpty.wait;
  x = buffer[out];
  out++;
  count--;
}

end monitor
```

Conditional variable

# Message Passing

- ✓ Used to:
  - Enforce mutual exclusion
  - Exchange information

```
send (destination, message)
receive (source, message)
```

# Message Passing - Synchronization

- Sender and receiver may or may not be blocking (waiting for message). For instance:
- ***Rendezvous***: Blocking send, blocking receive
  - both sender and receiver are blocked until message is delivered

# Message Passing - Synchronization

- ✓ *Nonblocking send, blocking receive*
  - sender continues processing such as sending messages as quickly as possible
  - receiver is blocked until the requested message arrives
- ✓ *Nonblocking send, nonblocking receive*

# Addressing

✓ *Direct addressing*

- *send*-primitive includes a specific identifier of the destination process: `send(dest,msg)`
- *receive*-primitive could know ahead of time from which process a message is expected:

  `receive(sourceProcess123, MsgVar)`
- *receive*-primitive could use source parameter to return a value when the receive operation has been performed:

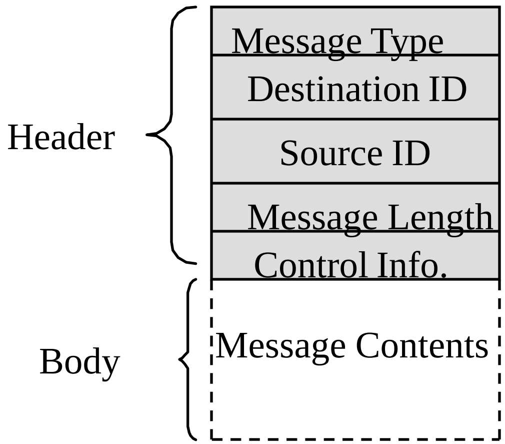
  `receive(SourceProcVar, MsgVar)`

# Addressing

✓ *Indirect addressing*

- messages are sent to a shared data structure consisting of queues
- queues are called *mailboxes*
- one process sends a message to the mailbox and the other process picks up the message from the mailbox
- *port* is a mailbox assigned to a specific process statically (e.g., ftp port, telnet port)

# General Message Format



# Readers/Writers Problem

- ✓ Any number of readers may simultaneously read the file
- ✓ Only one writer at a time may write to the file
- ✓ When a writer is writing to the file, no reader may read it

# Readers/Writers Using Message Passing

Assume each reader & writer has a *unique id* and its *own mailbox*

## Reader with id = I

```
send(msg(readrequest,I),
                coordinator);
receive(mailbox(I),Msg);
<read unit>
send(msg(done,I),coordinator);
```

## Writer with id = J

```
send(msg(writerequest,J),
                coordinator);
receive(mailbox(J), Msg);
<write unit>
send(msg(done,J),coordinator);
```

We also need a *coordinating* process to receive all these requests and to decide who gets a reply and when.
(E.g., it can give priority to readers or to writers, etc.)

# A Coordinator (writers have priority)

```
int readerNum = 0;
Boolean writer = NULL;
while (true) {
    case (writer == NULL):
            if (!empty(queue-of-done-msgs-from-readers)) {      // Ack readers' done's
                receive(done,Id);                              // Blocking receive
                readerNum--;
            }
            else if (!empty(queue-of-writerequests))
                receive(writerequest,writer);                  // Now writer != NULL
            else if (!empty(queue-of-readrequests)) {
                receive(readrequest,Id);                       // Blocking receive
                readerNum++;
                send(Id,ok);    // Non-blocking send: grant a reader. No pending writers, done's
            };
    case (readerNum > 0  &&  writer != NULL):    // If a writer is pending, don't take new requests
            receive(done,Id);
            readerNum--;
    case (readerNum == 0  &&  writer != NULL):  // Grant the pending writer
            send(writer,ok);                   //  Non-blocking send
            receive(done,writer);              // Blocking  receive
            writer=NULL;
}
```