Chapter - two

Process Management

Process and Threads...

Lecture 2.2

Outline

- Inter process communication
- Race conditions
- Mutual exclusion
- Algorithms to avoid critical region problem
  - ✓ Disabling Interrupts
  - ✓ Lock Variables
  - ✓ Strict Alternation
  - ✓ Semaphores
  - ✓ Message passing

Inter Process Communication

- Process need to communicate with other processes.
- Three issues are there:

  1- How one process can pass information to another process.

  2- Making sure that two or more processes do not get into each other's way when engaging in critical activities.

  3- Proper sequencing when dependencies are present.

- Example: If Process A produces data and process B prints them, B has to wait until A has produced some data.

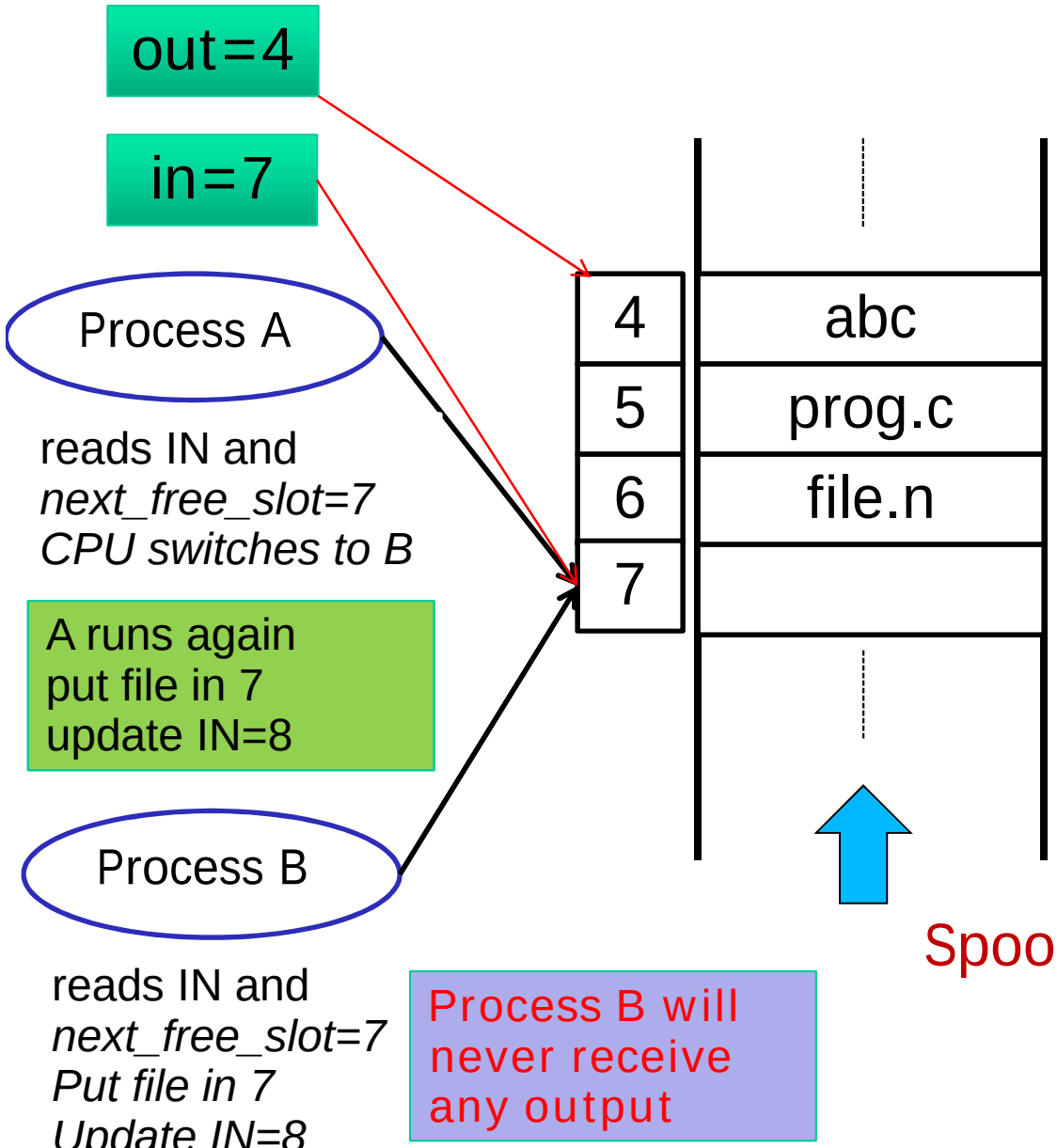
Inter Process Communication...

Race Condition

- The situation where several processes access – and manipulate shared data concurrently.
  - (E.g, Main memory, printer spooler,...)
- The final value of the shared data depends upon which process finishes last.

- To prevent race conditions, concurrent processes must be synchronized.

IPC: Race condition, print spooler



- If process wants to print a file, it enter the file name in spooler directory
- Another process printer daemon prints and remove their name

Assumption

- Slot 0-3 are printed out
- 4-6 are ready to be printed
- next_free_slot is 7

Inter Process Communication...

The Critical-Section Problem

- How do we avoid race conditions? (i.e. shared variables, files, memory)
- n processes all competing to use some shared data
- Sometimes a process has to access shared memory or files that can lead to races.
  - That point of the program where the shared memory access is called critical region.

Problem

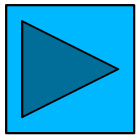
- How do we ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section?

Inter Process Communication...

Mutual Exclusion

Solution to Critical-region problem

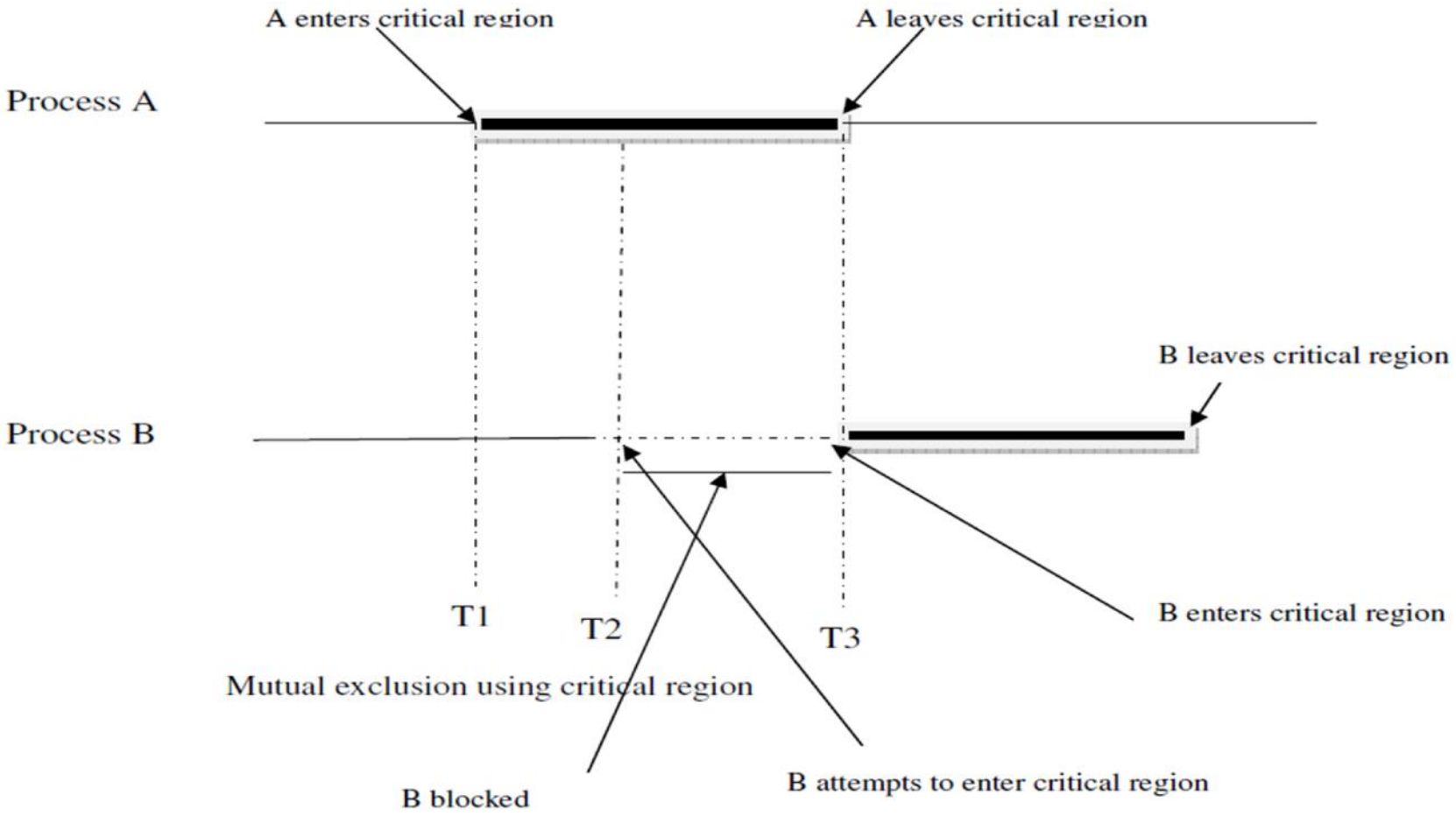
- To avoid races we need four conditions to hold to have a good solutions:
  - No two processes may be simultaneously inside the critical regions.
  - No assumptions may be made about speeds or the number of CPUs.
  - No process running outside its critical region may block other processes.
  - No process should have to wait forever to inter its critical region.



Inter Process Communication...

Mutual Exclusion

Solution to critical-region problem



Inter Process Communication...

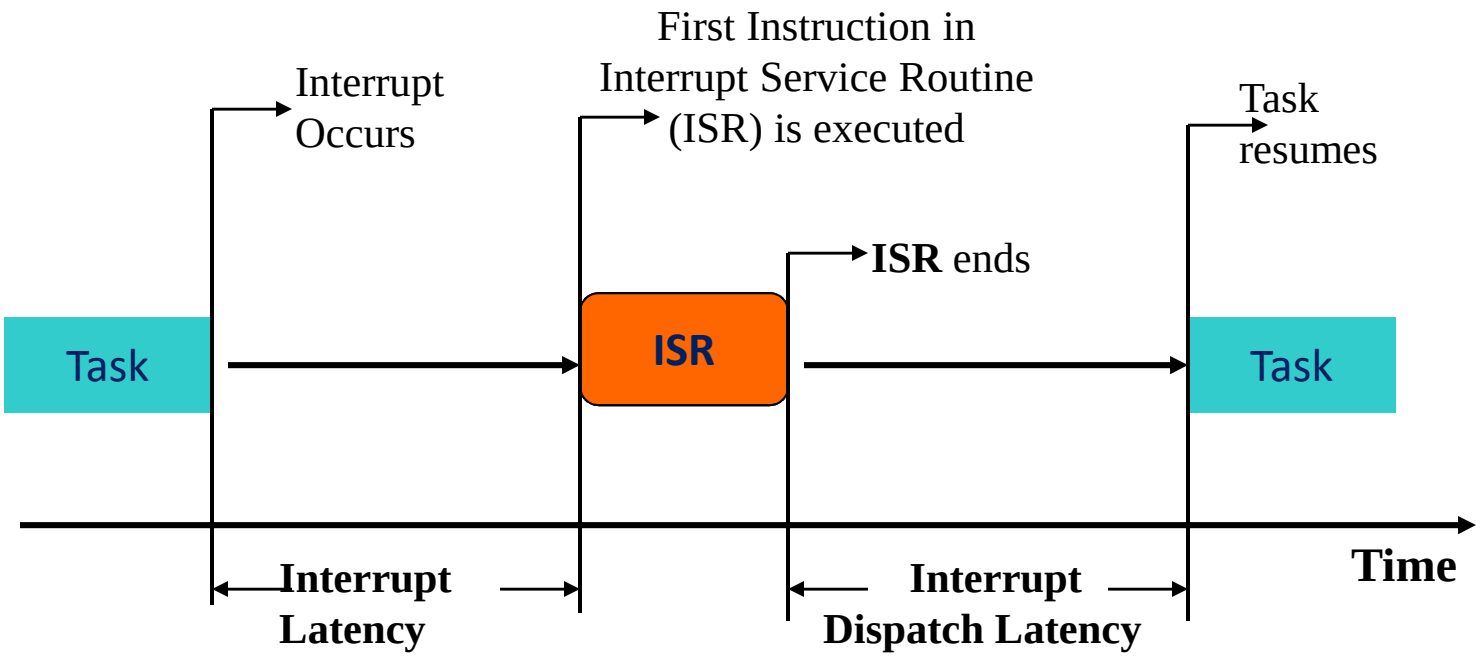
Mutual Exclusion

Algorithms to avoid critical region problem

- Disabling Interrupts
- Lock Variables
- Strict Alternation
- Semaphores
- Message passing

Disabling Interrupts

- Each process disable all interrupts just after entering its critical region and re-enable them just before leaving it.
- Thus, once a process has disabled interrupts, it can update the shared memory with out problem.



Interrupt Latency should be very small
Kernel has to respond to real time events

Disabling Interrupts...

```
while (true)
{
 disable_interrupts();
 critical_section();
 enable_interrupts();
}
```

Disabling Interrupts...

- Drawbacks
  - It is unwise to give user processes the power to turn off interrupts.
  - Suppose that one of them did it, and never turned them on again? That could be the end of the system.
  - For a multiprocessor system, with two or more CPUs, disabling interrupts affects only the CPU that executed the disable instruction.
  - The other ones will continue running and can access the shared memory.

Lock Variables

– Consider having a single shared variable X.

– The following code performs the lock operation:

```
B:  if LOCK (X) = 0 (*item is unlocked*)
    then LOCK (X) ← 1 (*lock the item*)
    enter critical region
    else begin
      wait (until lock (X) = 0) and
      the OS wakes up the process;
    goto B
    end;
```

Lock Variables...

– The following code performs the unlock operation:

LOCK (X) $\leftarrow$ 0 (*unlock the item*)
if any processes are waiting then
 wake up one of the waiting processes;

Strict Alternation

- the integer variable turn, initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory.

p_0

```
while (TRUE) {      /* loop */
while (turn != 1)
    critical_region( );
        turn =1;
noncritical_region( );
          }
```

p_1

```
while (TRUE) {      /*loop*/
   while (turn != 0)
     critical_region( );
    turn =0;
   noncritical_region( );
     }
```

Strict Alternation...

- Problem:
  - busy waiting: Continuously testing a variable until some value appears, which wastes CPU time
    - If P_0 set turn 0, P_1 will continuously check turn till it becomes 1.
  - Suppose turn is set to 1 and P_1 is in non-critical region, if P_0 wants to enter in critical-region, not possible.
    - Violets condition 3, P_0 is being block by a process not in its critical region

Sleep and wakeup

- is IPC primitives that block instead of wasting CPU time when they are not allowed to enter critical regions.

- Sleep is a system call that causes the caller to block.
  - be suspended until another process wakes it up.

- The wakeup call has one parameter, the process to be awakened.

Sleep and wakeup...

The Producer-Consumer Problem (the bounded_buffer problem)

- Two processes share a common, fixed size buffer.
- One of them, the producer, puts information into the buffer, and
- the other one, the consumer, takes it out.
- Case 1: when the producer wants to put a new item in the buffer, but it is already full.
  - go to sleep to be awakened when the consumer has removed one or more items.
- Case 2: if the consumer wants to remove an item from the buffer and sees that the buffer is empty,
  - it goes to sleep until the producer puts something in the buffer and wakes it up.

Code for producer

```
#define N 100   /* number of slots in the buffer */
int count = 0   /* number of items in the buffer */
void producer(void) {
int item;
while (TRUE) {          /* repeat forever */
item = produce_item();  /* generate next item */
if (count == N) sleep(); /* if buffer=full, go to sleep */
insertitem(item);        /* put item in buffer */
count = count + 1; /* increment count of items in buffer */
if (count == 1) wakeup(consumer); /* was buffer empty? */
     }
}
```

Code for consumer

```
    void consumer(void)
          {
     int item;
   while (TRUE)  {    /* repeat forever */
 if (count == 0) sleep(); /* if buffer= empty,got to sleep */
 item = remove_item();    /* take item out of buffer */
count= count - 1; /* decrement count of items in buffer*/
 if (count == N - 1)wakeup(producer);/* was buffer full? */
        consume_item(item);        /* print item */
```

The producer-consumer: problem

Race condition.

- The buffer is empty and the consumer has just read count=0 .
- At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer.
- Case 1:
  - Producer sees count = 0, insert item, inc. count =1, and it wakes up the consumer. However, consumer is not asleep.
  - Problem: wakeup signal is lost.
- Case 2:
  - next when consumer runs, count =0 and go to sleep...
- Case 3:
  - producer will fill buffer and will go to sleep.
  - Both will sleep forever.

The producer-consumer: problem...

Problem

- The problem here is that a wakeup sent to a process that is not (yet) sleeping is lost.

Solution

- Semaphore

Semaphore

- Semaphore was proposed by Dijkstra to manage concurrent processes by using a simple integer value, which is known as a semaphore.

- Semaphore is simply a variable which is non-negative and shared between threads/processes.

- This variable is used:
  - to solve the critical section problem and
  - to achieve process synchronization in the multiprocessing environment.

Semaphore...

- Dijkstra suggested using an integer variable to count the number of wakeups saved for future use.
- In his proposal, a new variable type, called a semaphore, was introduced.
- A semaphore could have the value 0, indicating that no wakeups were saved, or
- some positive value if one or more wakeups were pending.
- Dijkstra proposed having two operations,
  - down – sleep
  - up - wakeup,

Semaphore...

TWO OPERATIONS

Case 1

- The down operation on a semaphore checks to see if the value is greater than 0.
- If so, it decrements the value (i.e., uses up one stored wakeup) and just continues.
- If the value is 0, the process is put to sleep without completing the down for the moment.

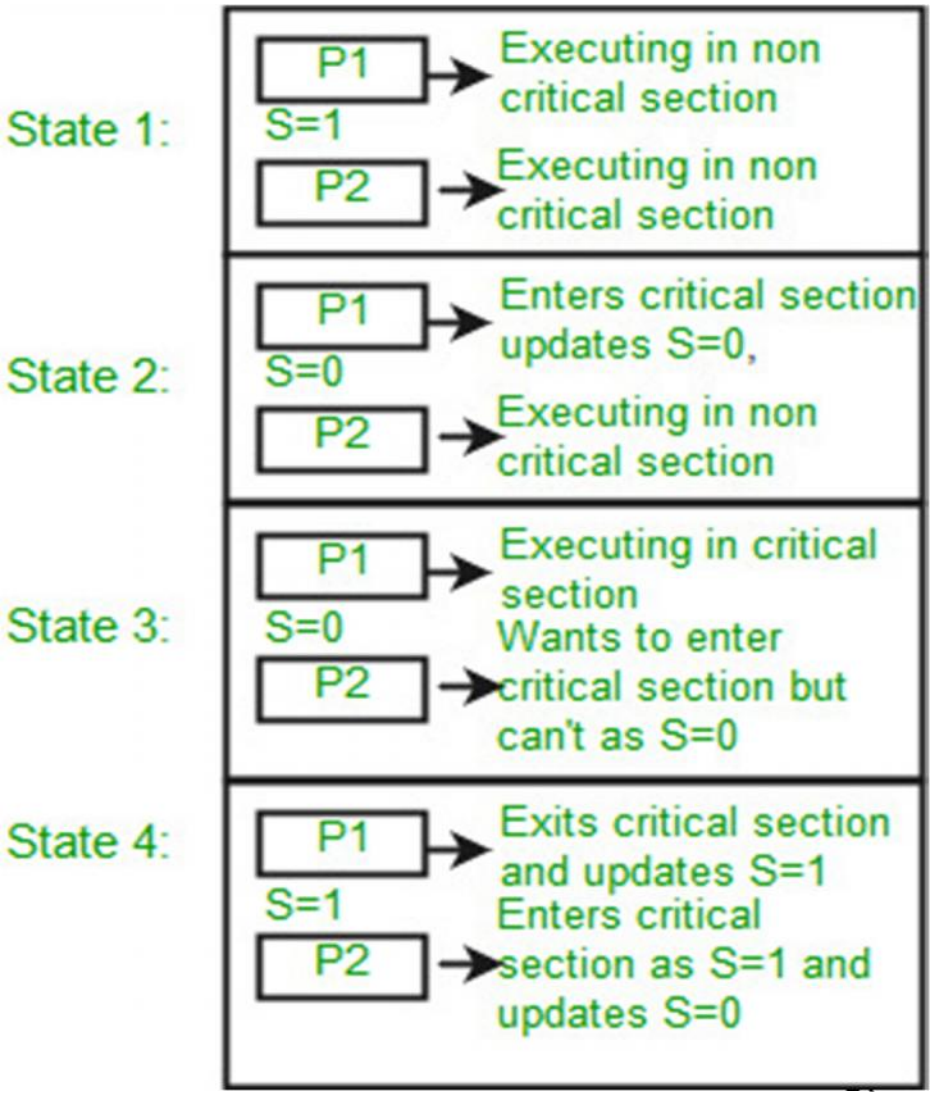
Case 2

- The up operation increments the value of the semaphore
- If one or more processes were sleeping on that semaphore, unable to complete an earlier down operation, one of them is chosen by the system (e.g., at random) and is allowed to complete its down.

Solving the producer consumer problem using Semaphore

- Semaphores that are initialized to 1 and used by two or more processes to ensure that only one of them can enter its critical region at the same time are called binary semaphores.

- If each process does a down just before entering its critical region and an up just after leaving it, mutual exclusion is guaranteed.



Solving the producer consumer problem using Semaphore

- Semaphores solve the lost-wakeup problem.
- This solution uses three semaphores:
  1. full for counting the number of slots that are full,
  2. empty for counting the number of slots that are empty,
  3. mutex to make sure that producer and consumer do not access the buffer at the same time

- full is initially 0,
- empty is initially equal to the number of slots in the buffer,
- mutex is initially 1.

```
#define N 100                    /* number of slots in the buffer */
typedef int semaphore;           /* semaphores are a special kind*/
semaphore mutex = 1;             /* controls access to critical region*/
semaphore empty = N;             /* counts empty buffer slots */
semaphore full = 0;              /* counts full buffer slots */

void prducer(void) {
int item;
while (TRUE) {                   /* TRUE is the constant 1 */
item =produce_item();/*generate something to put in the buffer*/
down(&empty);                 /* decrement empty count */
down(&mutex);                   /* enter critical region */
insertitem(item);               /* put new item in buffer */
up(&mutex);                     /* leave critical region */
up(&full);                    /* increment count of full slots */
```

```
void consumer(void)
  {
   int item;
   while (TRUE) {          /* infinite loop */
  down(&full);             /* decrement full count */
  down(&mutex);            /* enter critical region */
item = remove_item();      /* take item from buffer */
 up(&mutex);               /* leave critical region */
 up(&empty);              /* increment count of empty slots*/
consume_item(item);      /* do something with the item */
```

Semaphore: Java implementation

Gentle reminder

- As we have seen, the producer-consumer problem is an example of a multi-process synchronization problem.

- The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.
  - The producer's job is to generate data, put it into the buffer, and start again.
  - At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

Semaphore: Java implementation...

Problem

To make sure that:

- the producer won't try to add data into the buffer if it's full and
- the consumer won't try to remove data from an empty buffer.

Solution

- The producer is to go to sleep if the buffer is full.
- The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again.
- In the same way, the consumer can go to sleep if it finds the buffer to be empty.
- The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

Semaphore: Java implementation...

- Producer-consumer problem can be solved using semaphore to control synchronization.

- The program consists of four classes:
  - Buffer : the buffer that you're trying to synchronize
  - Producer : the producer thread that is producing item
  - Consumer : the consumer thread that is consuming item from the buffer
  - Producer-Consumer : the main class that creates the single Buffer, Producer, and Consumer.

Semaphore, java...

```
import java.io.*;
import java.util.concurrent.Semaphore;
  class Buffer {
      int item;
//semConsumer is initialized to 0 to ensure put executes first
 static Semaphore semConsumer=new Semaphore(0);
 static Semaphore semProducer=new Semaphore(1);
//get item from buffer
public void get() {
            try {
// Before consumer can consume an item, it must acquire a permit fromsemConsumer
            semConsumer.acquire();
                    }
    catch(InterruptedException e) {
  System.out.println("Unable to enter CS!");
        }
```

Semaphore, java...

```
//consumer consumes item
System.out.println("Consumer consumed item:"+ item);
//after consumer consumes item, it should release semProducer to notify producer
 semProducer.release();
                }
//put item in buffer
    public void put(int item) {
                        try {
 //Before producer can produce an item,  it must acquire a permit from semProducer
                semProducer.acquire();
                        }
            catch(InterruptedException e) {
System.out.println("Unable to put item on buf");
                        }
```

Semaphore, java...

```
//producer producing item
        this.item=item;
System.out.println("Producer produces item"+
item);

// After producer produces the item, it releases semConsumer to notify
consumer
          semConsumer.release();

                              }
                }
```

Semaphore, java...

```
//Producer class
class Producer implements Runnable {
        Thread myThread;
            Buffer b;
    Producer(Buffer b) {
        this.b=b;
   myThread=new Thread(this, "Producer");
    myThread.start();
              }
    public void run() {
    for(int i=0; i<10;i++)
     //producer puts item
        b.put(i);
              }
          }
```

Semaphore, java...

```
//Consumer class
class Consumer implements Runnable {
      Thread myThread;
      Buffer b;
Consumer(Buffer b) {
       this.b=b;
 myThread=new Thread(this, "Consumer");
      myThread.start();
                 }
    public void run() {
    for(int i=0; i<10;i++)
       //consumer get item
                 b.get();
                } }
```

Semaphore, java...

```
  //main class
 public class ProducerConsumer {
  public static void main(String[] args) {
//creating buffer queue
 Buffer b=new Buffer();

//starting consumer thread
       Consumer cons=new Consumer(b);

//starting producer thread
    Producer prod=new Producer(b);
                }
            }
```

Processes communication using message passing

Process: program running within a host.

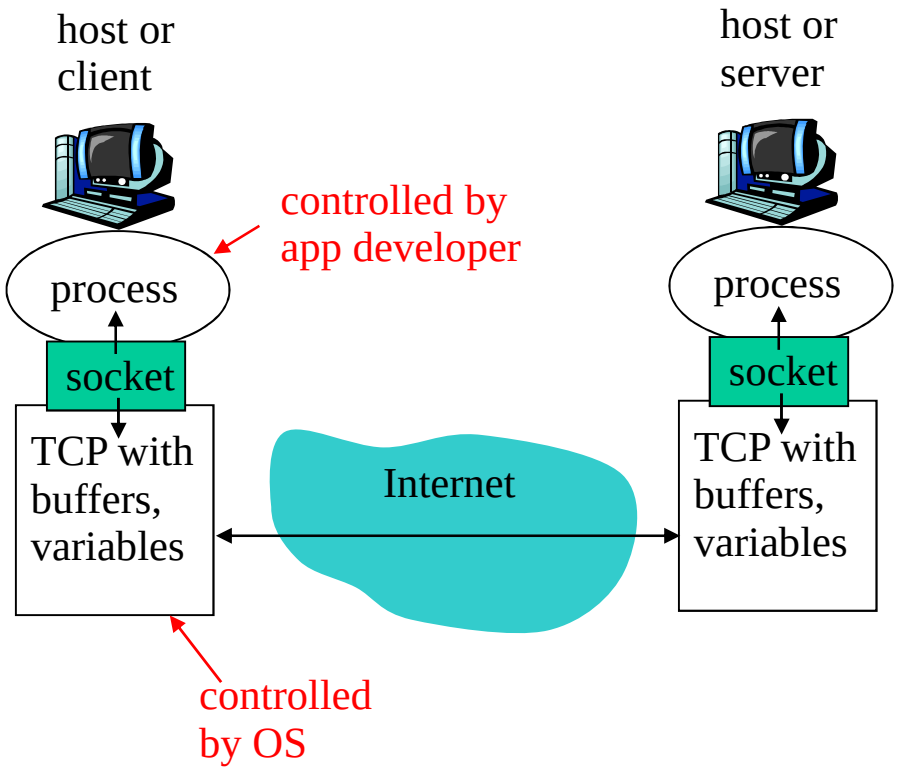
- within same host, two processes communicate using inter-process communication (defined by OS).
- processes in different hosts communicate by exchanging messages

Client process: process that initiates communication

Server process: process that waits to be contacted

Sockets

- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process gives message out door
  - sending process relies on transport infrastructure on other side of door which brings message to socket at receiving process

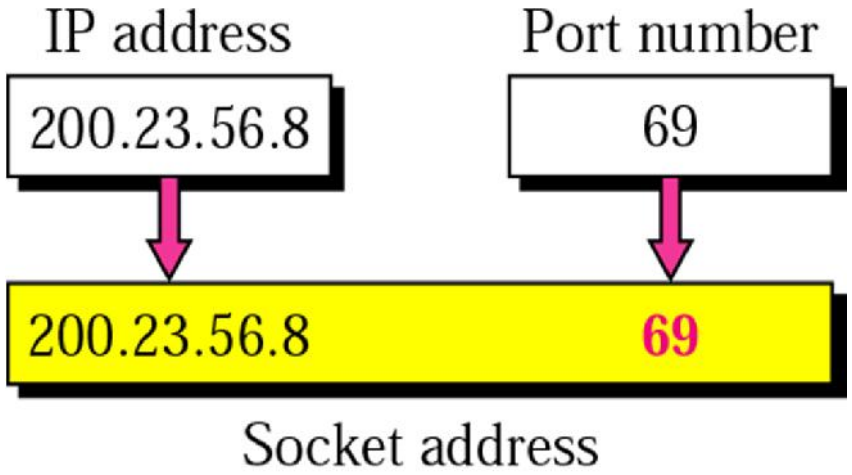


Addressing processes

- to receive messages, process must have identifier
- host device has unique 32-bit IP address
- Q: does IP address of host on which process runs suffice for identifying the process?
  - A: No, many processes can be running on same host
- identifier includes both IP address and port numbers associated with process on host.
- Example port numbers:
  - HTTP server: 80
  - Mail server: 25
- to send HTTP message to a web server with IP address:217.110.45.12, we use:
  - 217.110.45.12:80

Socket Addresses

- process-to-process delivery needs two addresses:
  - IP address and
  - port number at each end
- the combination of an IP address and a port number is called a socket address
- a transport-layer protocol needs a pair of socket addresses:
  - the client socket address
  - the server socket address
- the IP header contains the IP address; the UDP or TCP header contains the port number



some of the well-known ports used by TCP

Port	Protocol	Description
20	FTP, Data	File Transfer Protocol (data connection)
21	FTP, Control	File Transfer Protocol (control connection)
23	TELNET	Terminal Network
25	SMTP	Simple Mail Transfer Protocol
53	DNS	Domain Name Server
67	BOOTP	Bootstrap Protocol
79	Finger	Lookup information about a user
80	HTTP	Hypertext Transfer Protocol

Socket programming

Goal: learn how to build client/server application that communicate using sockets

Socket API

- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by apps
- client/server paradigm
- two types of transport service via socket API:
  - reliable, byte stream-oriented TCP
  - unreliable datagram UDP

socket

a host-local, application-created, OS-controlled interface (a "door") into which application process can both send and receive messages to/from another application process

Socket programming using TCP

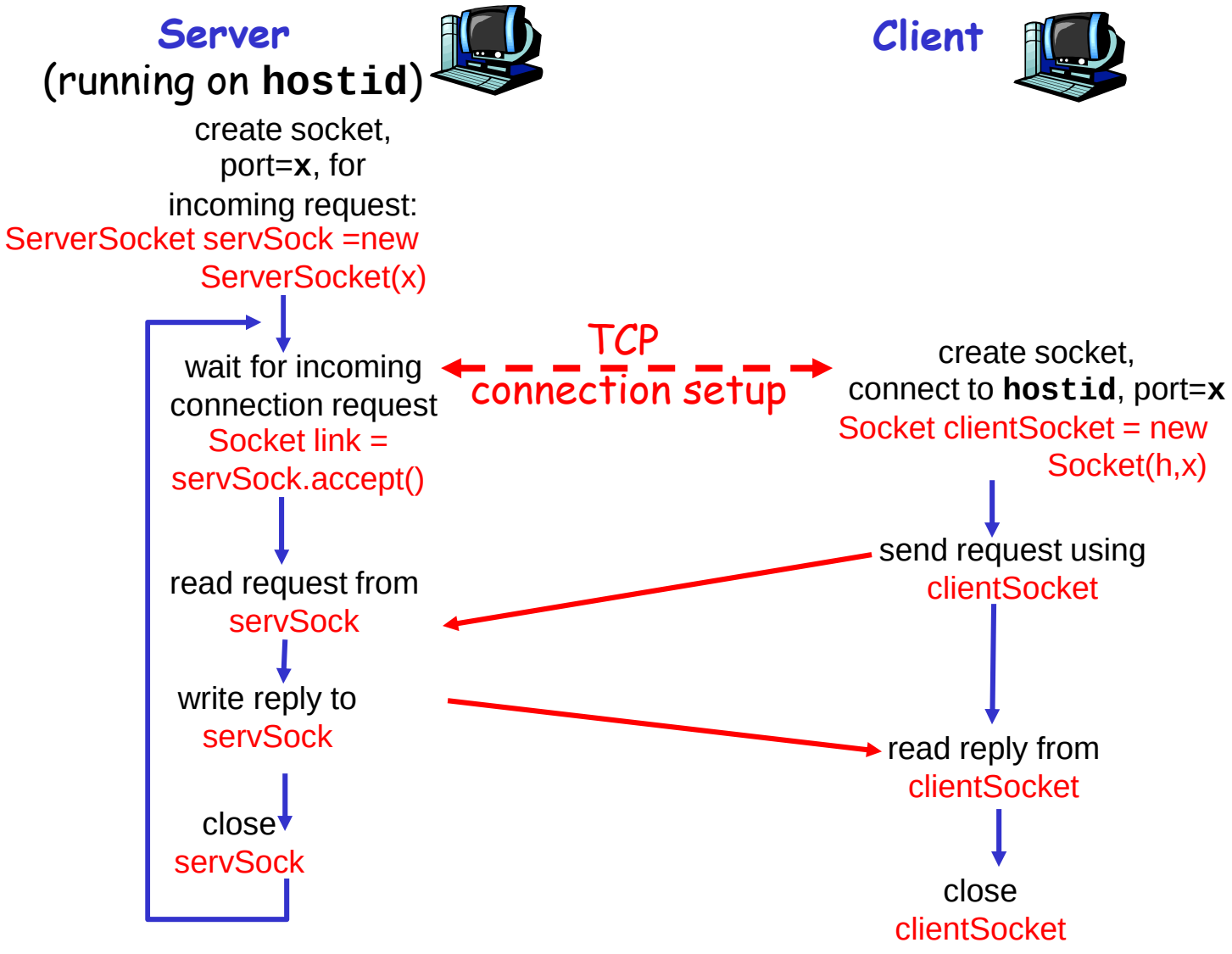
Client must contact server

- server process must first be running
- server must create socket (door) that welcomes client's contact

Client contacts server by:

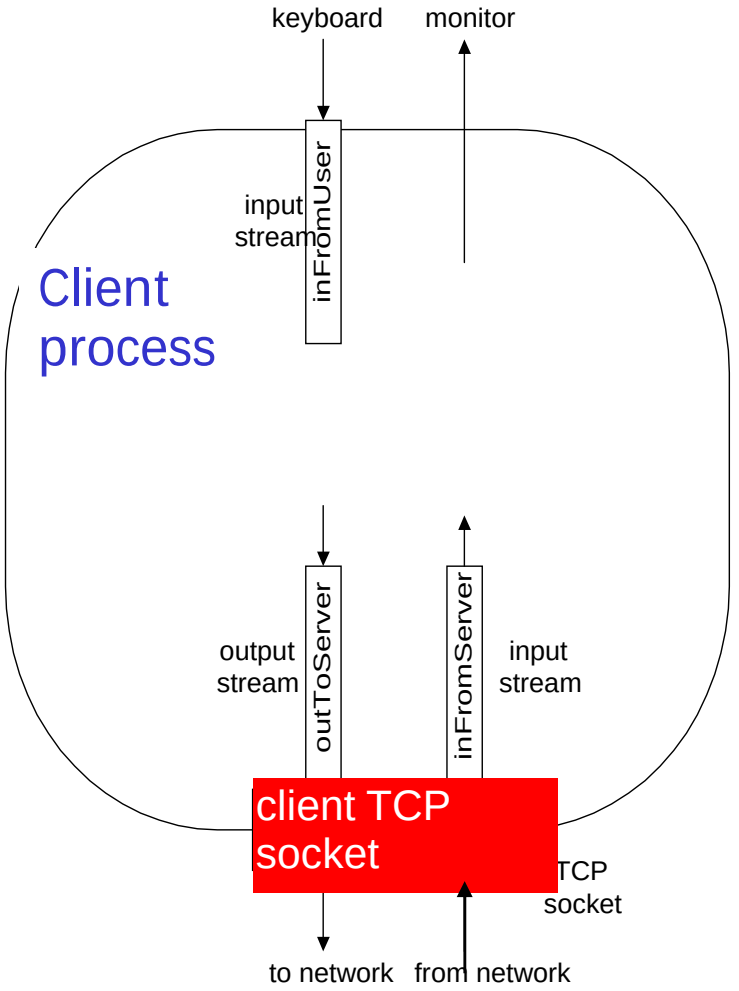
- creating client-local TCP socket
- specifying IP address, port number of server process
- When client creates socket: client TCP establishes connection to server TCP
- When contacted by client, server TCP creates new socket for server process to communicate with client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients

Client/server socket interaction: TCP



Stream terminology

- A stream is a sequence of characters that flow into or out of a process.
- An input stream is attached to some input source for the process, e.g., keyboard or socket.
- An output stream is attached to an output source, e.g., monitor or socket.



Server-Socket programming

- The java.net package provides ServerSocket and DatagramSocket objects for servers at the TCP/IP socket level.

Establishing a stream server involves five steps:

1- Create a new Socket with port number.

2- Set a Socket connect for a client using the accept() method.

3- Create inputstream and outputstream objects.

4- Process stream data.

5- Close the streams.

Server-Socket programming cont'd...

- The server program establishes a socket connection on Port 1234 in its listenSocket method.
- It reads data sent to it and sends that same data back to the client in its listenSocket method.

ListenSocket Method

- The listenSocket method creates a ServerSocket object with port number on which the server program is going to listen for client communications.

Server-Socket Programming cont'd...

- The port number must be an available port, which means the number cannot be reserved or already in use.
- For example, Unix systems reserve ports 1 through 1023 for administrative functions leaving port numbers greater than 1024 available for use.

```
  public static void listenSocket(){
try   {
  ServerSocket servSock = new ServerSocket(1234);
       }
     catch(IOException e){
  System.out.println("Unable to create port!");
        System.exit(-1);
     }
```

Server-Socket Programming cont'd...

- Next, the listenSocket method creates a Socket connection for the requesting client.
- This code executes when a client starts up and requests the connection on the host and port where this server program is running.
- When the connection successfully established, the servSock.accept() method returns a new Socket object.

Server-Socket Programming cont'd...

```
Socket link=null;
try
      {
         link = servSock.accept();
           }
      catch(IOException e)
   {
     System.out.println("Accept failed: 1234");
 }
```

Server-Socket Programming cont'd...

- Then, the listenSocket method creates a BufferedReader object to read the data sent over the socket connection from the client program.
- It also creates a PrintWriter object to send the data received back to the client.

```
BufferedReader in = new BufferedReader(new
 InputStreamReader(link.getInputStream()));

 PrintWriter out = new
 PrintWriter(link.getOutputStream(), true);
```

Server-Socket Programming cont'd...

- Lastly, the listenSocket method loops on the input stream to read data as it comes in from the client and writes to the output stream to send the data back.

```
    int numMessages = 0;
       String message = in.readLine();
while (!message.equals("close"))   {
   System.out.println("Message received.");
          numMessages++;
out.println("Message " + numMessages + ": " +
message);
          message = in.readLine();
              }
```

Lab: Server -Socket source code

```
import java.io.*;
import java.net.*;
 public class Server {
  private static ServerSocket servSock;
  private static final int PORT=1234;
  public static void main(String[] args)
  throws IOException {
  System.out.println("Openning port.....");
  while(true)
   {
    listenSocket();
      }
    }
```

Lab: Server-Socket source code ...

Create server socket at port 1234

Create socket connection with the client

```
public static void listenSocket()  {
     try {
     servSock=new ServerSocket(PORT);
     }  catch(IOException e)   {
System.out.println("Unable to create
socket with port no:1234!");
     System.exit(-1); }
   Socket link=null;
   try {
      link=servSock.accept();
     }  catch(IOException e){
   System.out.println("Accept failed:
Port 1234");
      }
```

Lab: Server-Socket source code ...

Create input stream, attached to socket

Create output stream, attached to socket

Read in line from socket

```
try {

BufferedReader in=new BufferedReader(new
 InputStreamReader(link.getInputStream(
 )));


 PrintWriter out=new
PrintWriter(link.getOutputStream(),tru
e);

  int numMessages=0;

  String message=in.readLine();
while(!message.equals("close")) {

    System.out.println("Message
recieved.");

          numMessages ++;
```

Lab: Server-Socket source code...

Write out line to socket

```
out.println("Message" + numMessages+
  ":" + message);
   message=in.readLine();
     }
   }

catch(IOException e)
              {
System.out.println("Message is not
  recieved");

End of while loop,
loop back and wait for
another client connection
          }
        }
      }
```

Client-Socket Programming

- Establishing a stream client involves four steps:

1- Create a new Socket with a server IP address and port number.

2- Create input stream and output stream objects.

3- Process stream data and

4- Close the streams.

Client-Socket Programming cont'd...

- The client program establishes a connection to the server program on a particular host and port number in its listenSocket method, and
- It then sends data entered by the end user to the server program.
- The listenSocket method also receives the data back from the server and prints it to the command line.

Client-Socket Programming cont'd...

listenSocket Method

- The listenSocket method first creates a Socket object with the IP address ("local host") and port number (1234) where the server program is listening for client connection requests.

```
Socket link= new Socket(host,PORT);
```

- It then provide a place where the data shall be stored by creating BufferedReader object to read the streams sent by the server back to the client.

```
BufferedReader in=new BufferedReader(
        new
InputStreamReader(link.getInputStream()));
```

Client-Socket Programming cont'd...

- Next, it creates a PrintWriter object to send data over the socket connection to the server program.

```
PrintWriter out=
        New
    PrintWriter(link.getOutputStream(),true);
```

- It also creates a BufferedReader object Set up stream for keyboard entry...

```
BufferedReader userentry=new BufferedReader(
      new InputStreamReader(System.in));
```

Client-Socket Programming cont'd...

- This listenSocket method code gets the input streams and passes it to the PrintWriter object, which then sends it over the socket connection to the server program.

- Lastly, it receives the input text sent back to it by the server and prints the streams out.

```
String message, response;
      do {
           System.out.print("Enter message:");
           message=userentry.readLine();
           out.println(message);
           response=in.readLine();
           System.out.println("\nSERVER>" + response);
         } while(!message.equals("close"));
```

Lab: Client-Socket Programming

```
import java.io.*;
 import java.net.*;
 public class Client {
 private static InetAddress host;
 private static final int PORT=1234;
 public static void main(String[] args) throws
IOException {
 try {
    host=InetAddress.getLocalHost();
  }
 catch(UnknownHostException e)
  {
   System.out.println("Host id not found!");
   System.exit(-1);
  }
  listenSocket();
  }
```

Server is local

Lab: Client-Socket Programming

```
public static void listenSocket() {
    Socket link=null;
   try {
    link=new Socket(host,PORT);}
  catch(IOException e){
 System.out.println("Unable to connect");
System.exit(-1);}
try {
  BufferedReader in=new BufferedReader(new
   InputStreamReader(link.getInputStream()));
PrintWriter out=new
   PrintWriter(link.getOutputStream(),true);

 BufferedReader userentry=new
   BufferedReader(new
   InputStreamReader(System.in));

    String message, response;
```

Create client socket, connect to server

Create input stream attached to socket

Create output stream attached to socket

Create input stream for user entry

Client-Socket Programming ...

```
do {
        System.out.print("Enter message:");
        message=userentry.readLine();
        out.println(message);
        response=in.readLine();
        System.out.println("\nSERVER>" + response);
      } while(!message.equals("close"));
      }
       catch(IOException e)
         {
         System.out.println("Message is not sent.");
          }
       }
      }
```

Project

- Make the client-server program as multithreading client-server for any type of application:
- E.g. it may be chat room, or client –server, where the server can be any server application, web server, ftp server, scientific calculator,...