


# Comp 310
# Computer Systems and Organization

## Lecture #6
## The Process and Communication
## (Programming with Processes)

Prof. Joseph Vybihal



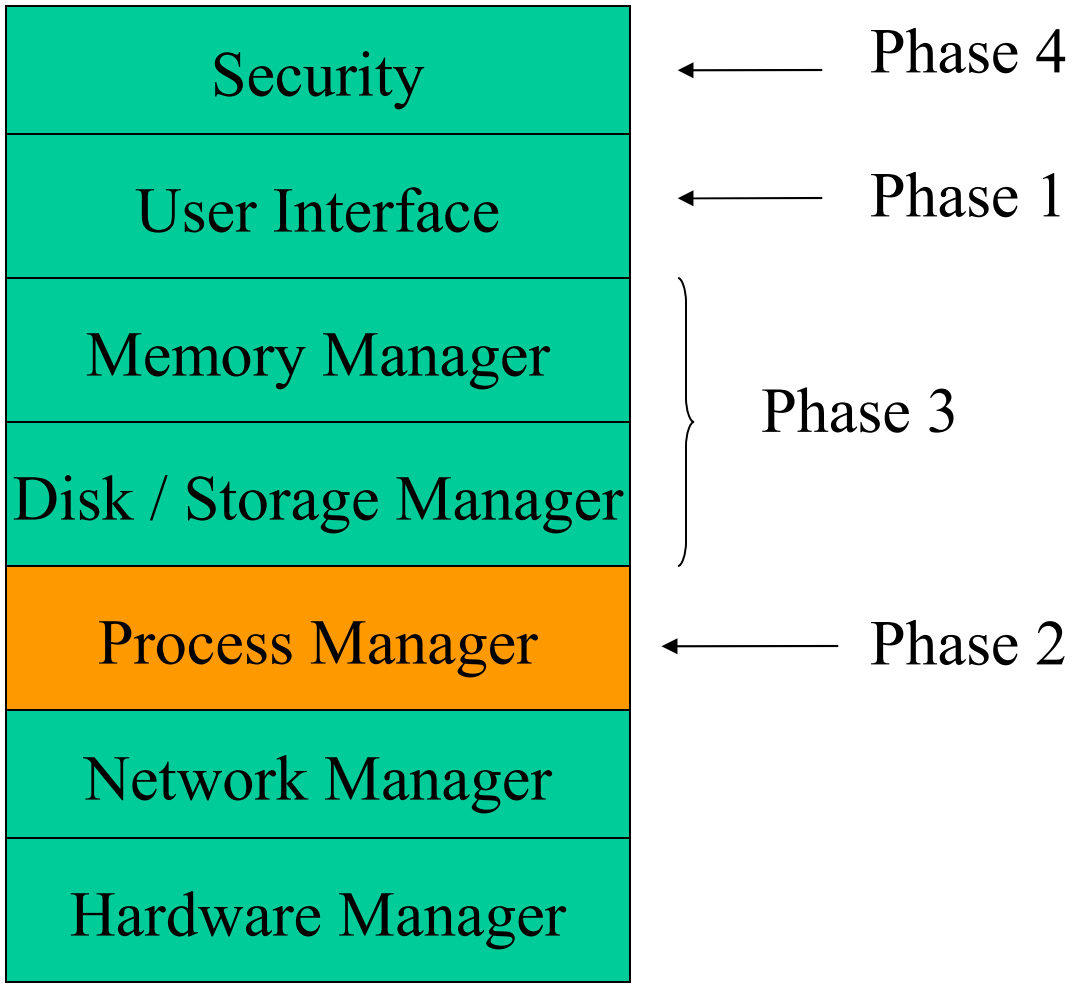

# Announcements

- C Programming Tutorial...




# Basic OS Architecture

(Course Table of Contents)






# Part 1

## Process Scheduling




# Implementing Processes

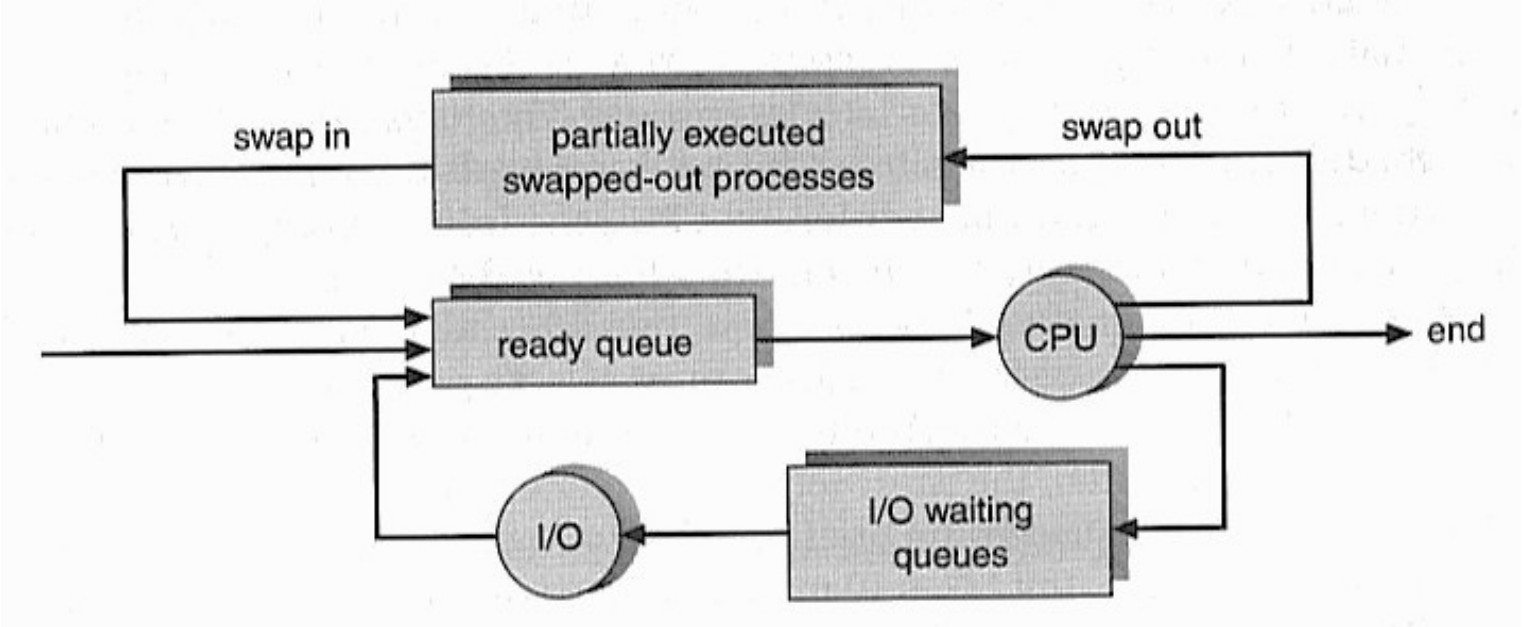
- OS is responsible for:
  - Dynamically selecting the next process to run
  - Rescheduling performed by dispatcher
- Dispatcher Algorithm:

```
Loop forever {
        run the process for a while (quanta).
        stop process (quanta, I/O, interrupt) and save its state.
        load state of another process.
}
```



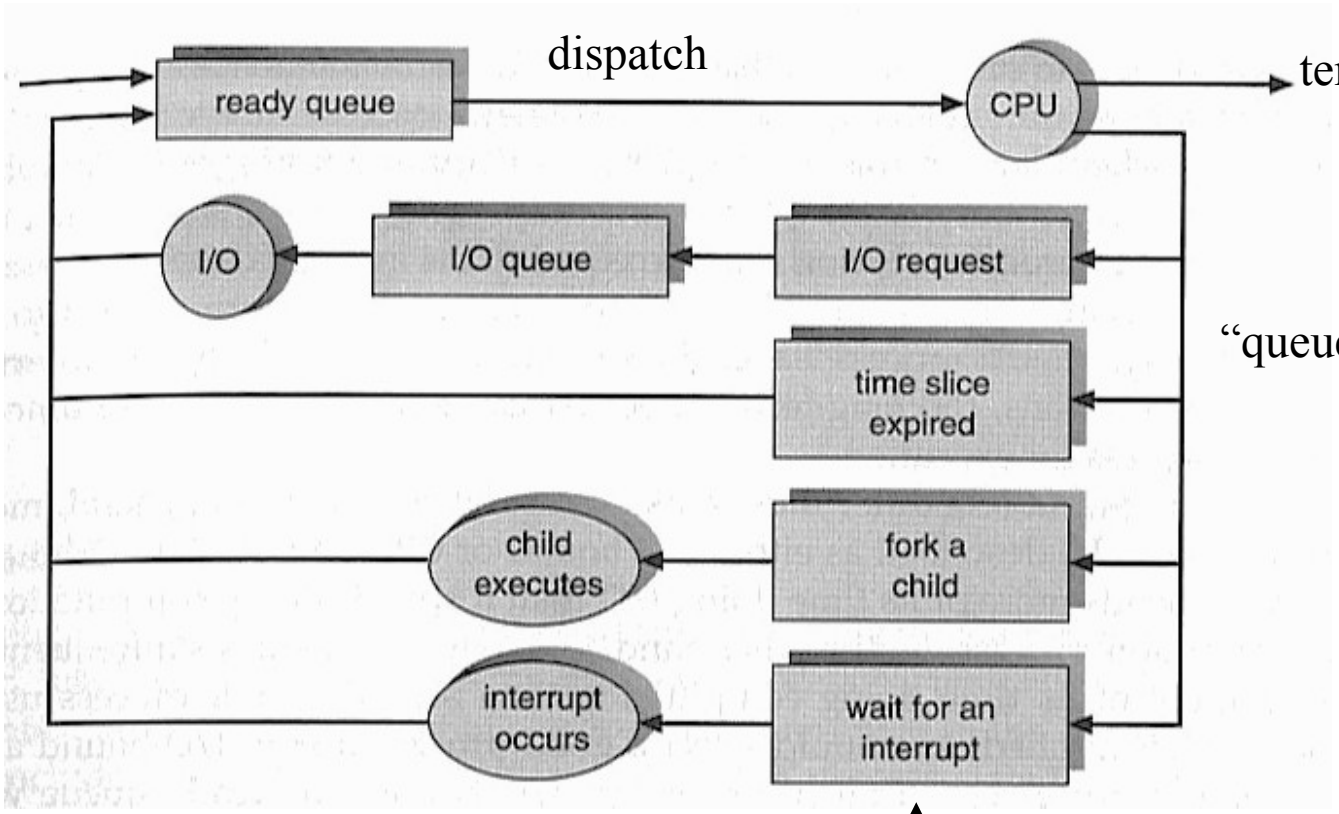

# Schedulers

## (Mid-level Scheduler)



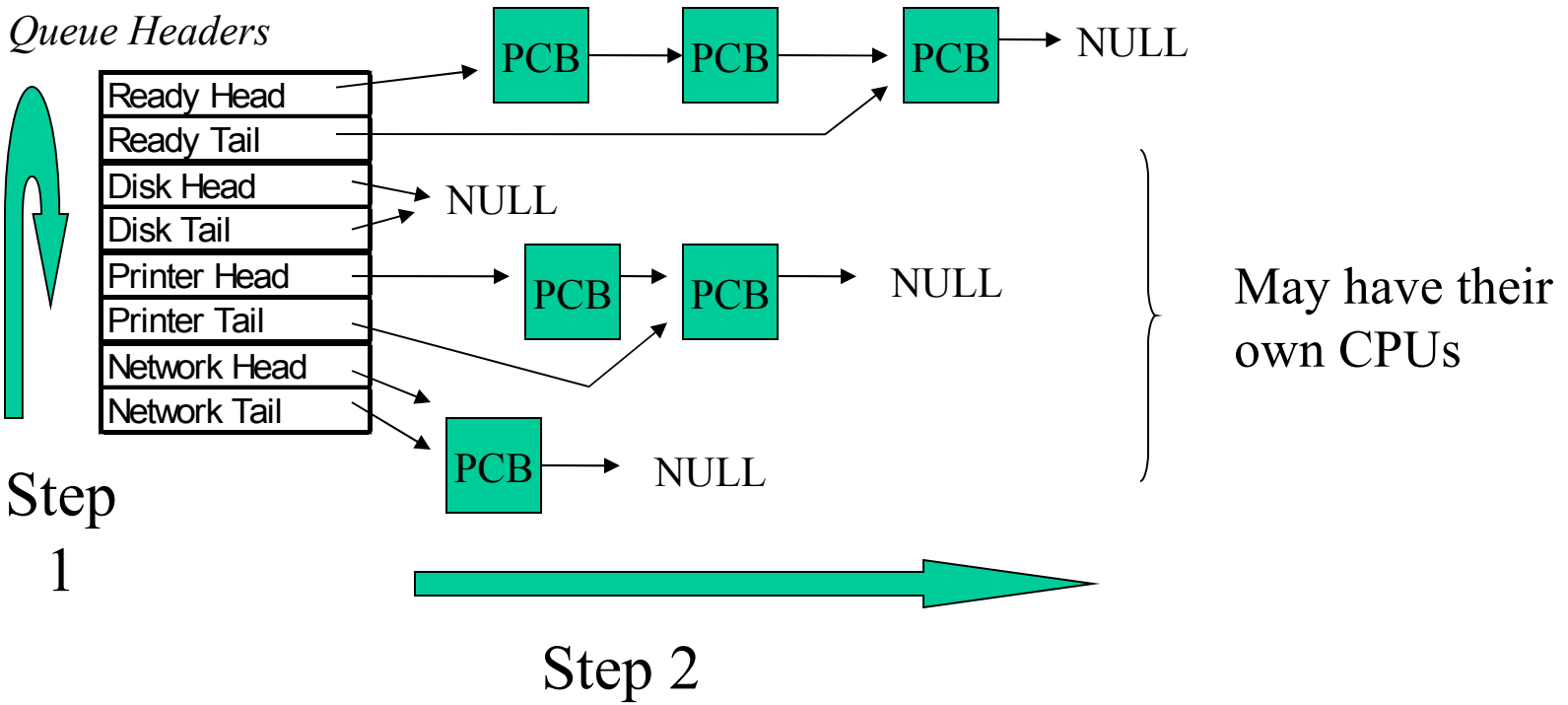

Did not have time to execute… Or system load too heavy…
Therefore, swap out of ready queue, put on Overload Queue.

# Process Scheduling




Pseudo-code and assembler discussion...

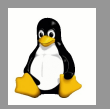
# Multiple OS Run-Time Queues




This is a double nested loop:

- For each queue
- Execute next PCB

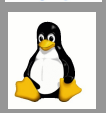





# Part 2

## Mechanics of a Process






# Process Creation

- Two ways to create a new process:
  - Build one from scratch:
    - Load code and data into memory
    - Create (empty) a dynamic memory workspace (heap)
    - Create and initialize the PCB
    - Make process known to the process scheduler(dispatcher)
  - Clone an exiting one:
    - Stop current process and save its state
    - Make a copy of code, data, heap and PCB
    - Make process known to process scheduler (dispatcher)




# Process Creation

- The OS
- User from command-line
- User's program

Process 1 → Syscall new-process

*Child* Process 2

- Child overlay of parent, or
- Child is a copy, or
- Child is a different program

Syscall new-process

End process 2

No effect on parent

Process 1 *Original Parent*

- Execute concurrently with child, or
- Sleep until child terminates

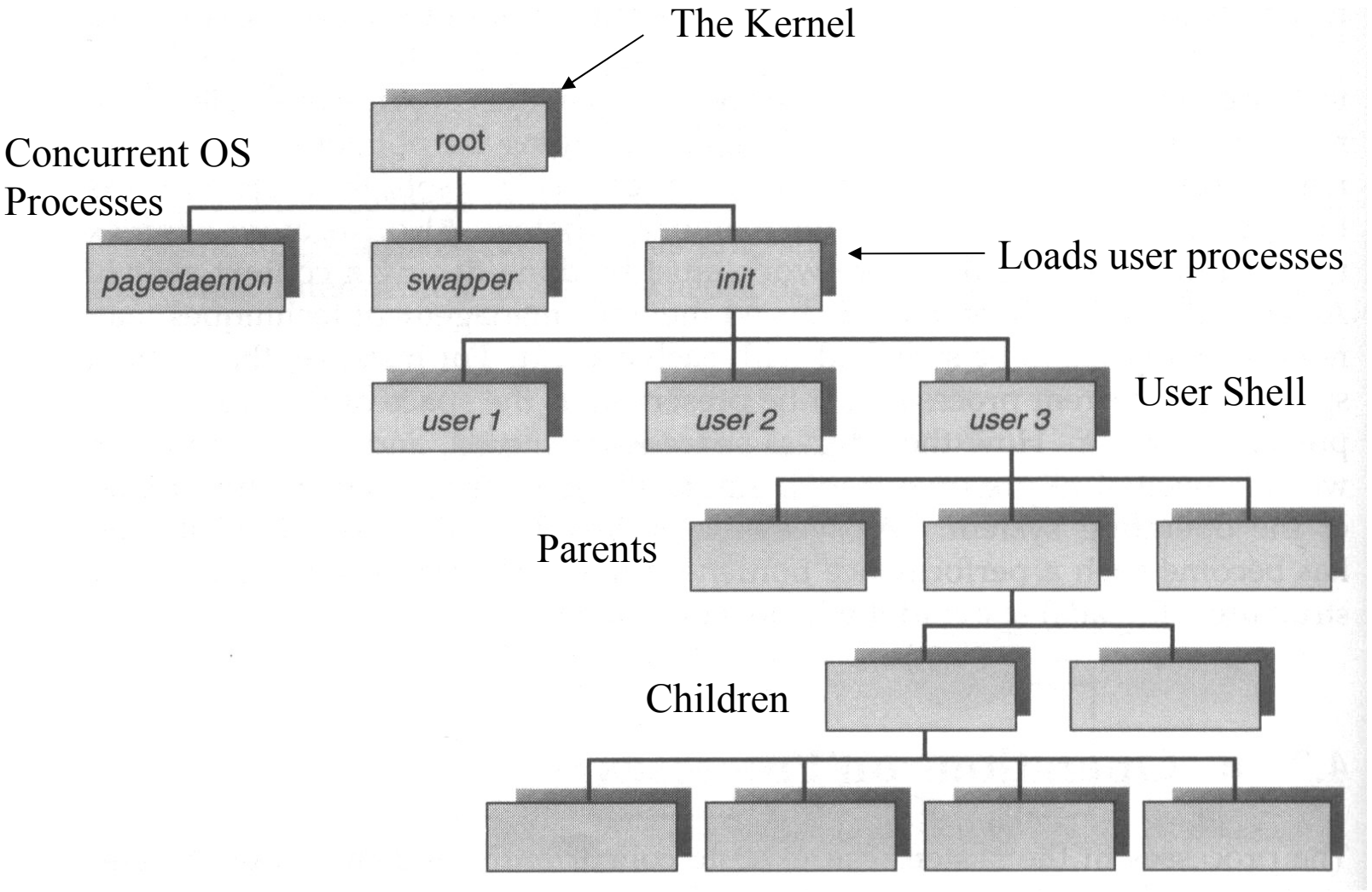
Syscall terminate-child
(P1 still executes)

End process 1

- OS auto terminates all children "*cascading termination*"




# The OS is Made of Processes



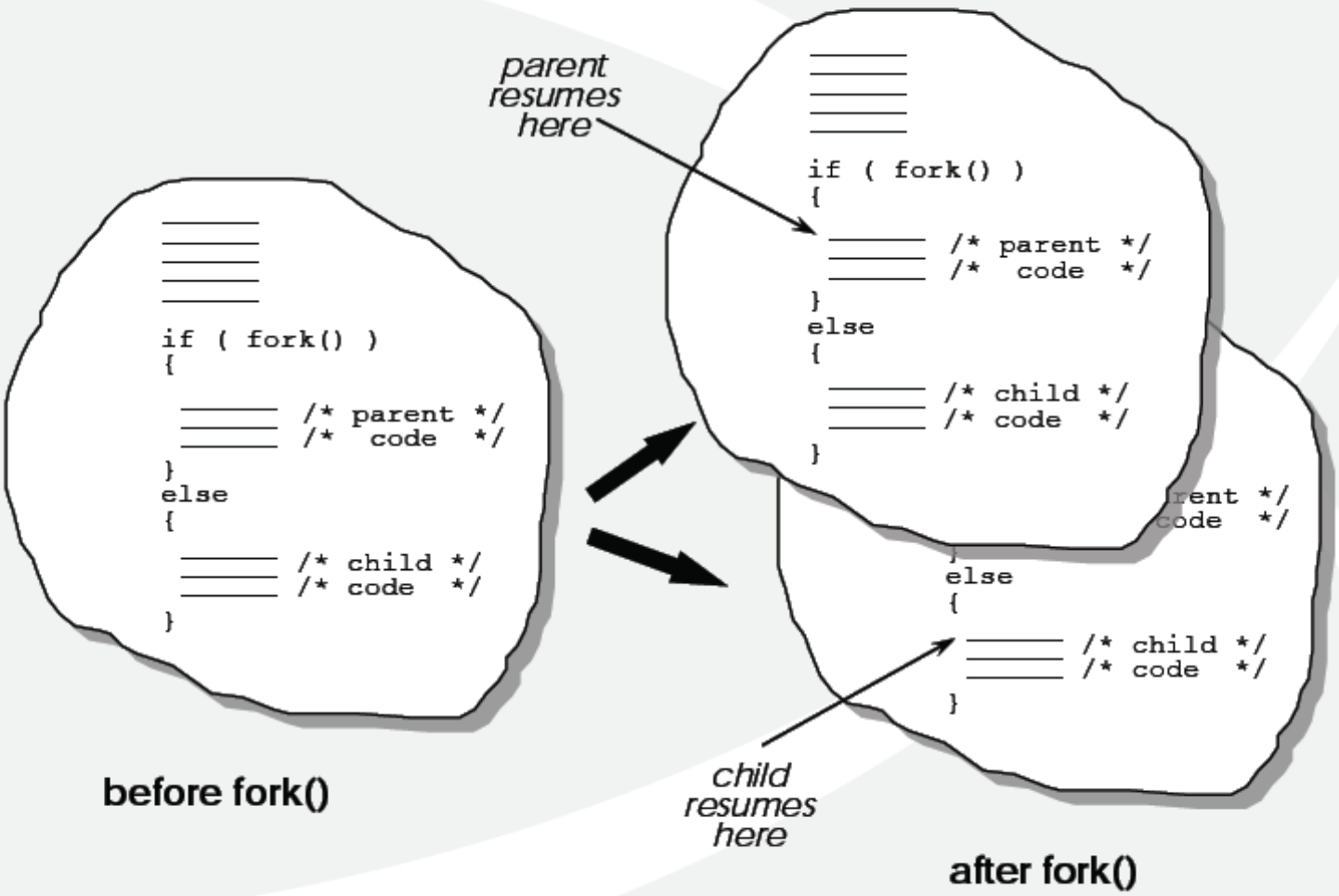



# Unix Process Creation

- In Unix, the fork() system call is used to create processes
  - fork() creates an identical copy of the calling process
  - After the fork(), the parent continues running concurrently with the child competing equally for the CPU.

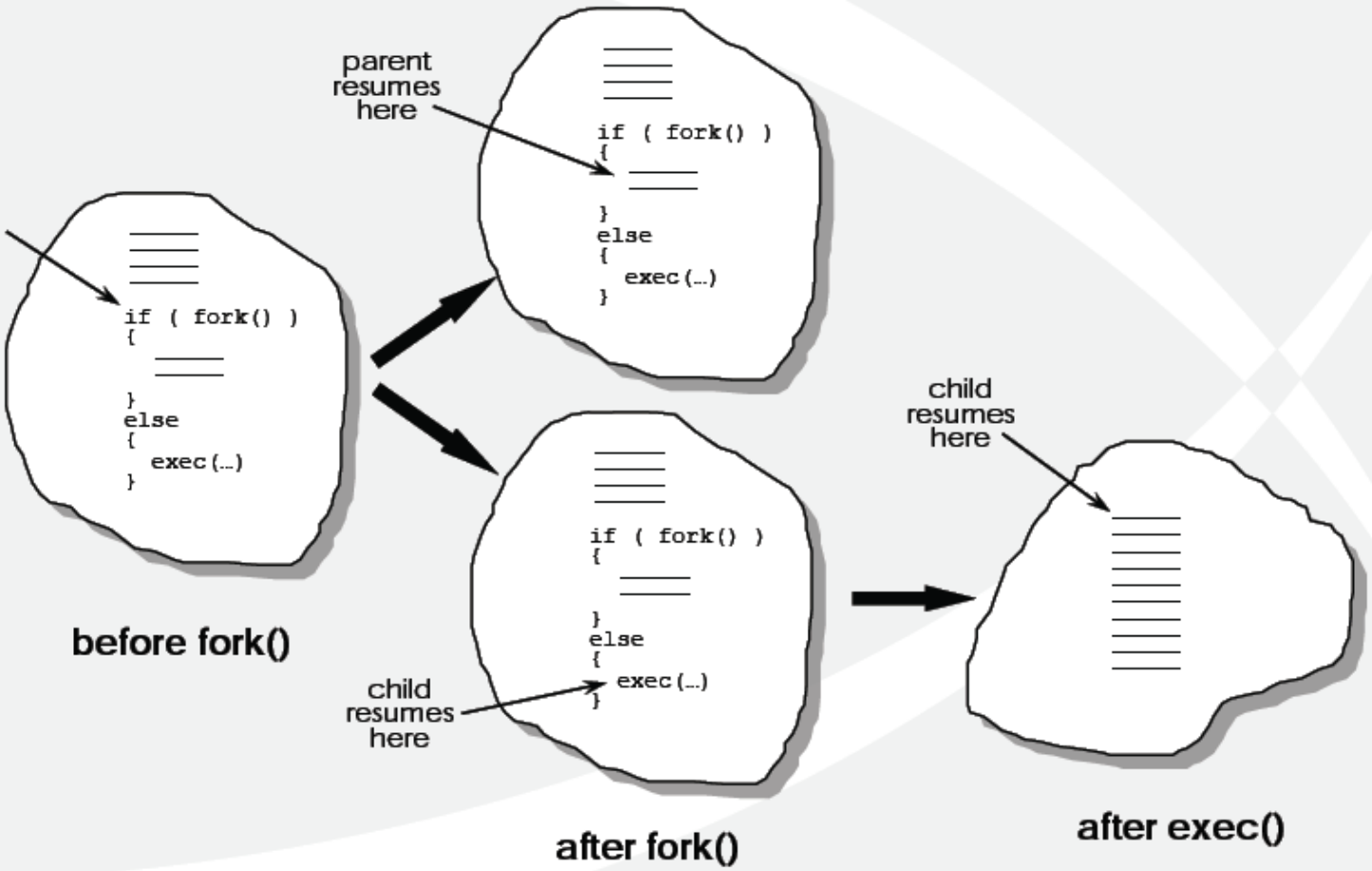



# UNIX process creation...

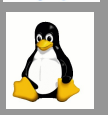





# A typical use of `fork()`








# Example

- What are the outputs of the following simple C programs?

```
main() {
  int i;
  i = 10;
  if (fork() == 0) i += 20;
  printf(" %d ", i);
}
```




# C Programming Example
## (Forking a Process)

```
#include <stdio.h>

void main(int argc, char *argv[])
{
int pid;

    /* fork another process */
    pid = fork();  -> same memory space

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       exit(-1);
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);  -> Independant space
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);  -> sleep
       printf("Child Complete");
       exit(0);  -> kill process & children.
    }
}
```

Parent could have executed concurrently




# C Programming Example
## (Created a Process with system)

```
#include <string.h>
#include <stdio.h>

int main(void)
{
        char name[100];
        char command[300];

        printf("File Name:");
        scanf("%s",name);

        strcpy(command,"del ");
        strcat(command, name);

        system(command);

        printf("Execution completed");
}
```

Child process created & parent process sleeps until child is complete.




# Part 3

## Inter-process Communication



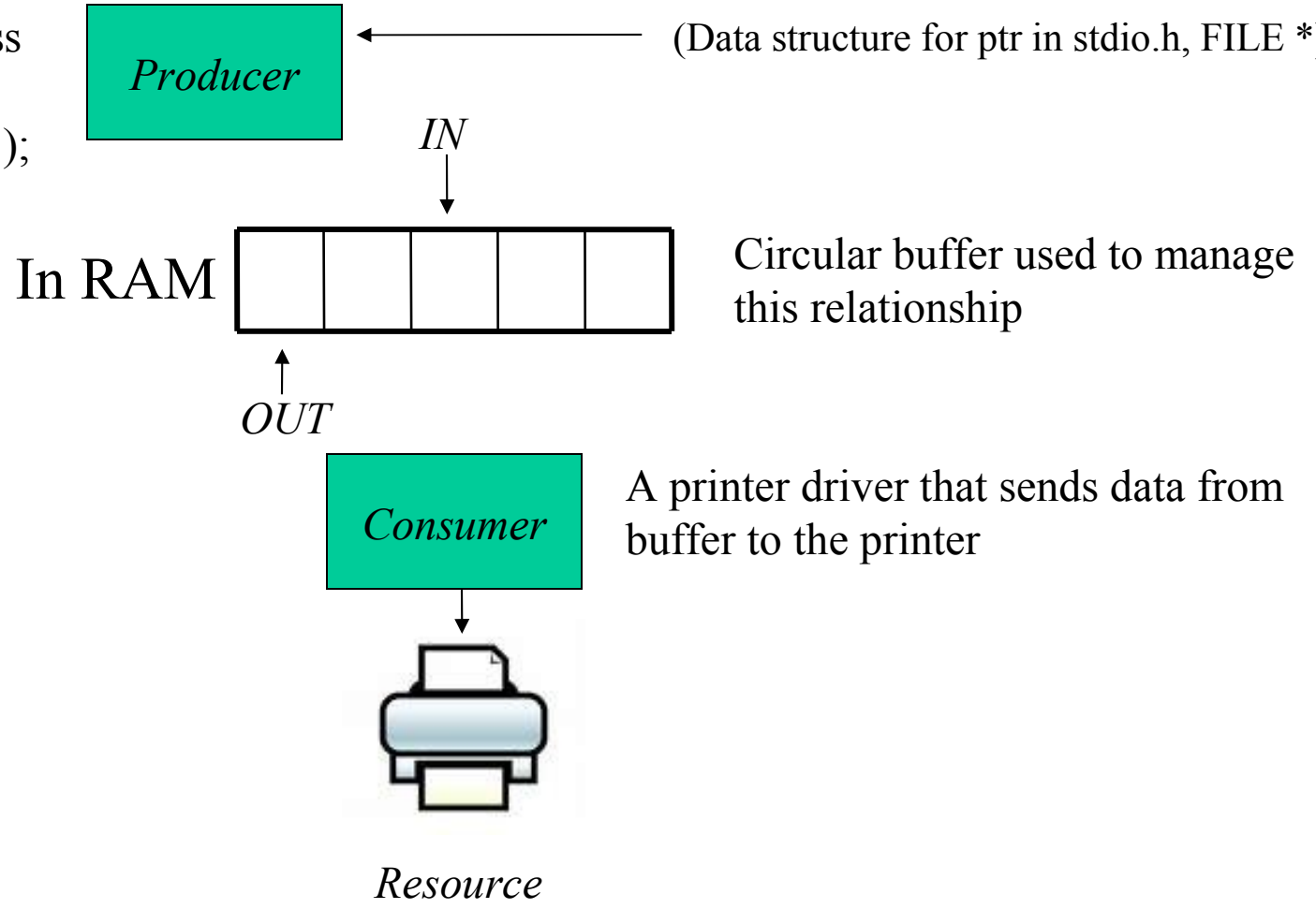

# Why Process Cooperation?

- Information Sharing
  - Shared resources: files, variable, buffer, …
- Computation Speedup
  - SETI computation (need multi-CPU/data channels)
- Modularity
  - Programming requirements need concurrency
- Convenience
  - Multi-windows to work concurrently

# Classic Example

## (Producer / Consumer Problem)

A user process printing with fprintf(prn,…);




Note: a PRN runs much slower than a process in RAM.




# fprintf(prn,…); Internal Code:

```
while(1) {
        // Code to produce an item in nextProduced HERE
        while (((in + 1) % BUFFER_SIZE) == out); // nothing
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Printer driver's code

```
while(1) {
        while (in == out) ; // do nothing
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        // Code the consume the item HERE
}
```

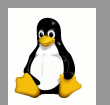
Unbounded-buffer or bounded-buffer?




# Message Passing Systems


- Direct or indirect communication
  - Share RAM? Use OS?
- Symmetric or asymmetric communication
  - Take turns, Scheduled? No management?
- Automatic or explicit buffering
- Send by copy or by reference
- Fixed-size or variable sized messages

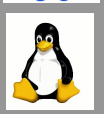





# Question

- How can we implement message passing using a simple text file?
  - One message at a time?
  - Infinite messages?

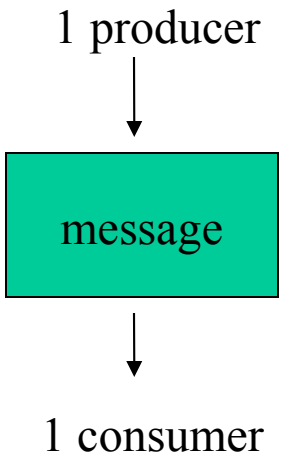





# Direct Communication

(Controlled Variable Sharing)

- Syntax:
  - `Send(pid, message);`
  - `Receive(pid, buffer);`
- Send: needs to know pid
- Receive: gives permission to receive
- Rules:
  - Only 1 link can exist at any time
  - This must be a binary link






# Message Passing with Pipes

```
#include <stdio.h>    /* standard I/O routines.                */
#include <unistd.h>   /* defines pipe(), amongst other things. */
```

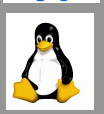
```
int main(int argc, char* argv[])
{
   int data_pipe[2]; /* an array to store the file descriptors of the pipe. */
   int pid;       /* pid of child process, or 0, as returned via fork.    */
   int rc;        /* stores return values of various routines.            */

   /* first, create a pipe. */
   rc = pipe(data_pipe);
   if (rc == -1) {
              perror("pipe");
              exit(1);
   }

   /* now fork off a child process, and set their handling routines. */
   pid = fork();

   switch (pid) {
              case -1:       /* fork failed. */
                 perror("fork");
                 exit(1);
              case 0:                        /* inside child process.  */
                 do_child(data_pipe);
                 /* NOT REACHED */
              default:       /* inside parent process. */
                 do_parent(data_pipe);
                 /* NOT REACHED */
   }

   return 0;   /* NOT REACHED */
}
```




# Message Passing with Pipes

```
void do_parent(int data_pipe[])
{
   int c;      /* data received from the user. */
   int rc;     /* return status of getchar().  */

   /* first, close the un-needed read-part of the pipe. */
   close(data_pipe[0]);

   /* now enter a loop of read user input, and writing it to the pipe. */
   while ((c = getchar()) > 0) {
            /* write the character to the pipe. */
      rc = write(data_pipe[1], &c, 1);
            if (rc == -1) { /* write failed - notify the user and exit */
               perror("Parent: write");
               close(data_pipe[1]);
               exit(1);
      }
   }

   /* probably got EOF from the user. */
   close(data_pipe[1]); /* close the pipe, to let the child know we're done. */
   exit(0);
}
```






# Message Passing with Pipes

```
void do_child(int data_pipe[]) {
    int c;      /* data received from the parent. */
    int rc;     /* return status of read().       */

    /* first, close the un-needed write-part of the pipe. */
    close(data_pipe[1]);

    /* now enter a loop of reading data from the pipe, and printing it */
    while ((rc = read(data_pipe[0], &c, 1)) > 0) {
            putchar(c);
    }

    /* probably pipe was broken, or got EOF via the pipe. */
    exit(0);
}
```




# Direct Communication

(Copied Memory)

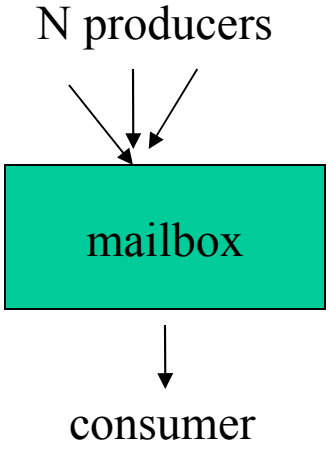
```
int msg;

while (!done)
{
        scanf("%d",&msg);
        x = fork();
        if (x == 0)
        {…}
        else
        {…}
}
```

Child process has msg also, but is a copy.




# Indirect Communication

- Use of third party:
  - Mailbox file: append to end of file messages
  - Port queue: insert into queue messages
- Syntax:
  - `Send(portid, message);`
  - `Receive(portid, buffer);`
- Rules:
  - Must know port or mailbox ID
  - Not a binary link, any process can send
  - Only one process can receive
  - Each mailbox/port has its rules of communication
    - Queue technique (priority rules, regular)
    - ASCII or UNICODE or Binary




Implemented as a text file or a database.




# Synchronization Types

- Blocking send (synchronization)
  - Sending process is put to sleep until received
- Non-blocking send (asynchronous)
  - Sender sends and continues execution
- Blocking receive (synchronization)
  - Receiver waits in busy-loop/ sleep-interrupt until message
- Non-blocking receive
  - Look at mailbox, if (message) good else NULL

*"Rendezvous"*




# Buffered Communication

- Zero Capacity Buffer
  - Buffer does not exist, must use synchronization
  - OS or Programmer Problem? Implementation?
- Bounded Capacity Buffer
  - If space, add message and continue execution, otherwise busy-loop until space
  - OS or Programmer Problem? Implementation?
- Unbounded Capacity Buffer
  - Send message regardless of space (assume infinite space) and continue execution
  - OS or Programmer Problem? Implementation?




# Part 4

## Examples




# Windows




# Windows 2000 Technology

(2000, Millennium, XP)

- "Subsystems"
  - Multiple operating environments
  - Message-passing control mechanism
  - Processes are clients of a particular subsystem
  - Binary communication is provided by an object called a *port*. The port has a send, receive, buffer and an optional queue.
- Port Management
  - OS initializes public Port Manager objects, one for each subsystem.




# Windows 2000 Technology

(2000, Millennium, XP)

- Port Management (continued)
  - Client gets *handle* (ptr) of port manager
  - Client requests for communication
  - Port Manager creates a port object with a TOOS and FROMOS port ID numbers. Returns handle of port object to client.
  - Client and OS use handle to communicate

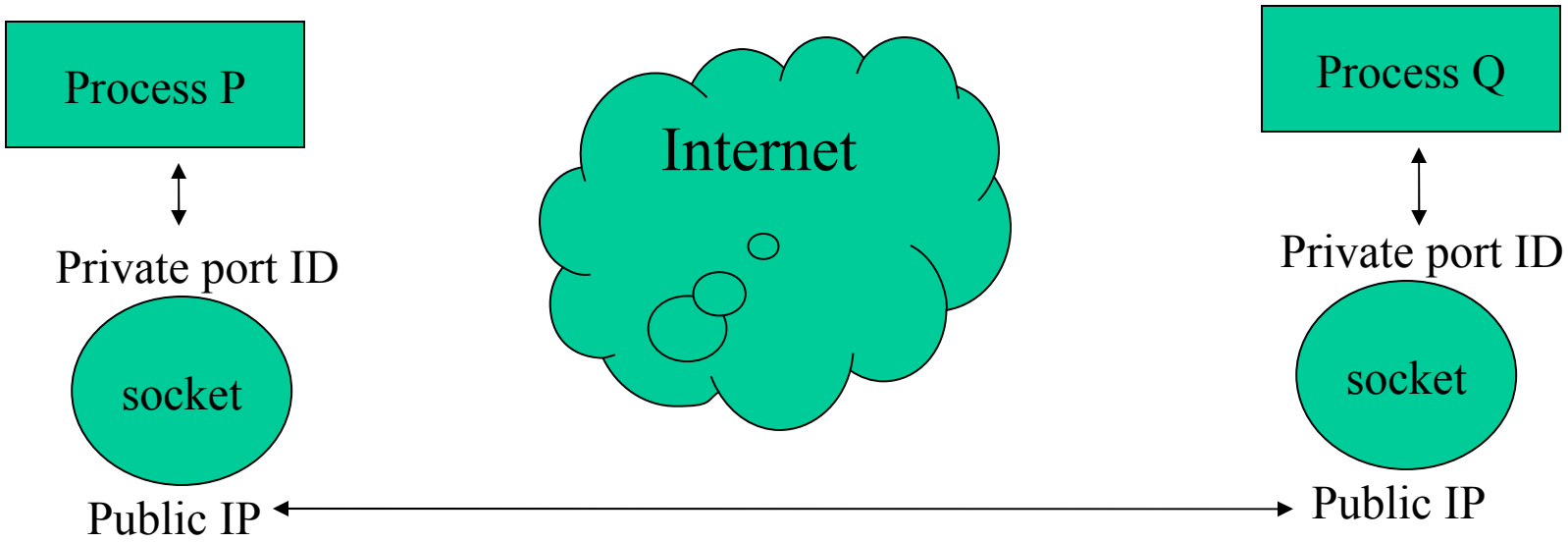



# Internet




# Internet Technology

(Java Sockets)




- `Send(ip:port, message);`
- Similar to Windows 2000




```
import java.net.*;
import java.io.*;

public class Server
{
   public static void main(String[] args) throws IOException {
      Socket client = null;
      PrintWriter pout = null;
      ServerSocket sock = null;

      try {
         sock = new ServerSocket(5155);
         // now listen for connections

         while (true) {
            client = sock.accept();

            // we have a connection
            pout = new PrintWriter(client.getOutputStream(), true);

            // write the Date to the socket
            pout.println(new java.util.Date().toString());

            pout.close();
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
      finally {
         if (client != null)
            client.close();
         if (sock != null)
            sock.close();
      }
   }
}
```

Create a socket

Process to use socket

Terminate communication

**Figure 4.10** Time-of-day server.




```
import java.net.*;
import java.io.*;

public class Client
{
   public static void main(String[] args) throws IOException {
      InputStream in = null;
      BufferedReader bin = null;
      Socket sock = null;

      try {
         //make connection to socket
         sock = new Socket("127.0.0.1",5155);

         in = sock.getInputStream();
         bin = new BufferedReader(new InputStreamReader(in));

         String line;
         while ( (line = bin.readLine()) != null)
            System.out.println(line);
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
      finally {
         if (sock != null)
            sock.close();
      }
   }
}
```

Socket object

Ask for connection

Read from socket

Terminate connection

**Figure 4.11** The client.




# Standard Connections

- TELNET is port 23
- FTP is port 21
- HTTP is port 80
- < 1024 are all pre-defined and known
- >= 1024 are private and local to computer
  - This is not a rule, just an agreement




# At Home




# Things to try out

1. Download a program called ETHEREAL and listen to the network and port communication on your computer.
   - The McGill labs lock you out
   - So try this at home - you'll need to be on a network. If you do not have a network then the program has sample files to study.
   - Google Ethereal
- Internet resources:
   - http://users.actcom.co.il/~choo/lupg/tutorials/multi-process/multi-process.html