Amirkabir University of Technology (Tehran Polytechnic)
Department of Computer Engineering

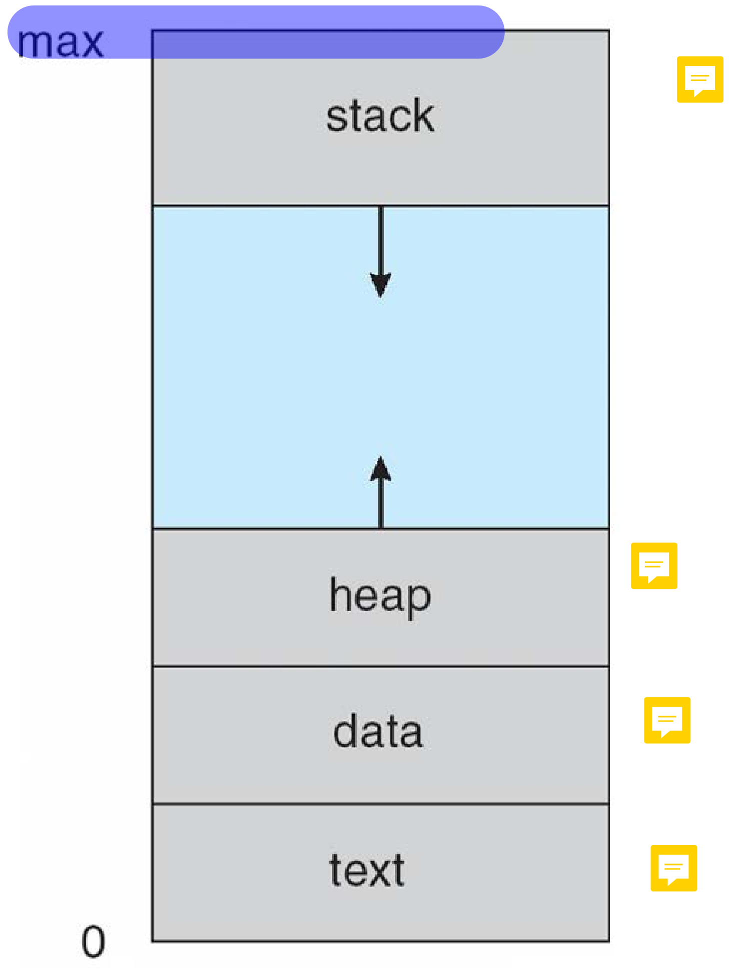
Processes (فرآیندها)

Hamid R. Zarandi
h_zarandi@aut.ac.ir

Definition

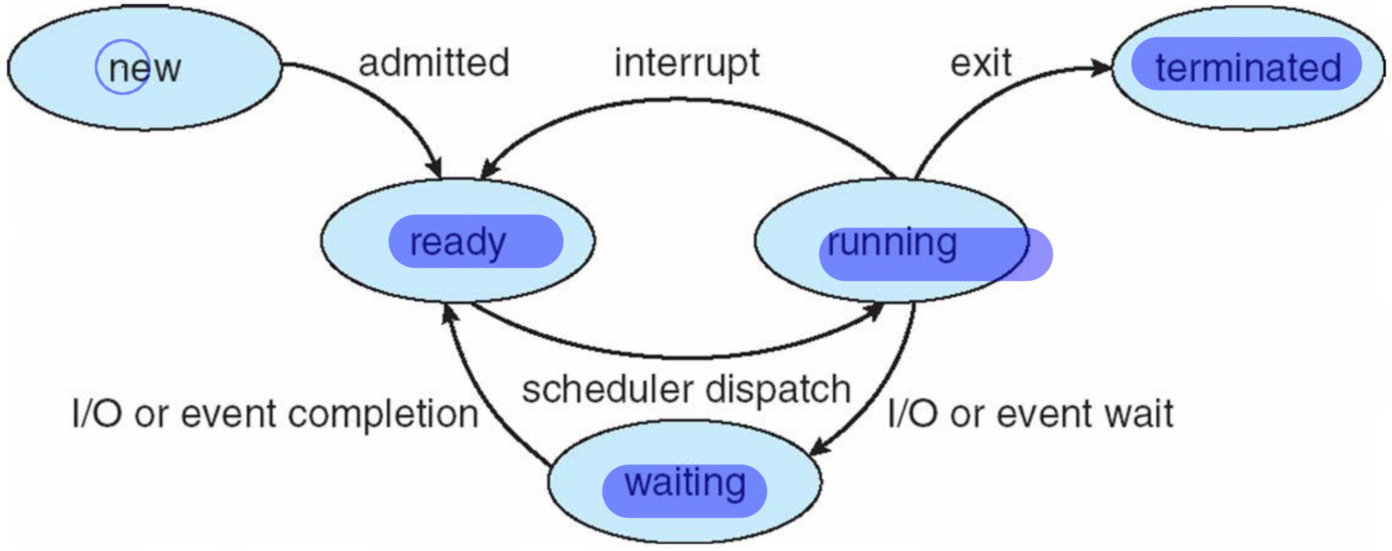
- Process
  - A program in execution; process execution must progress in sequential fashion
    - In time-sharing sys: unit of work
  - All processes are executed concurrently
- Process vs. Job?
  - Passive: program
  - Active: process
    - Program becomes process when executable file loaded into memory
    - One program can be several processes
  - Question?
    - java program

Process in memory



Process state

- As a process executes, it changes **state**
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

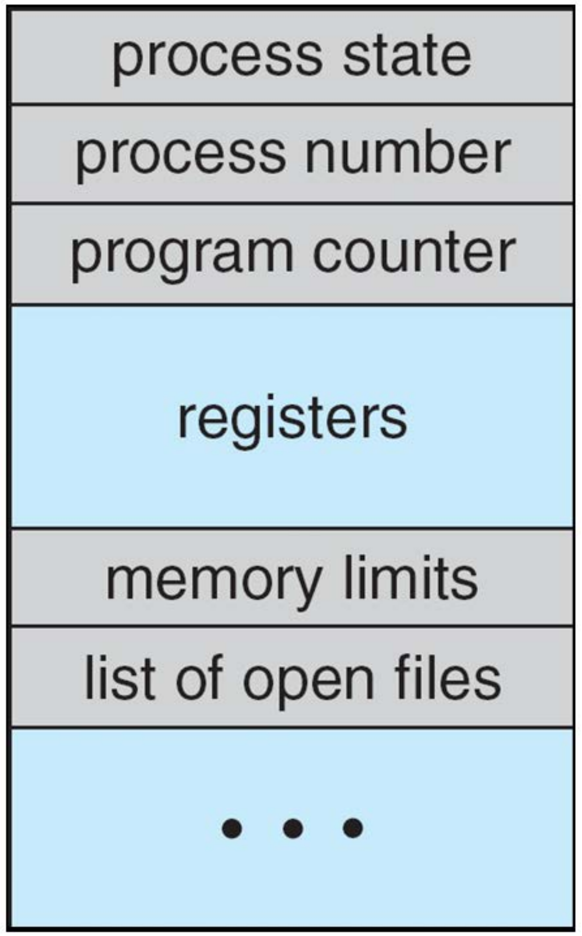


Process Control Block (PCB)

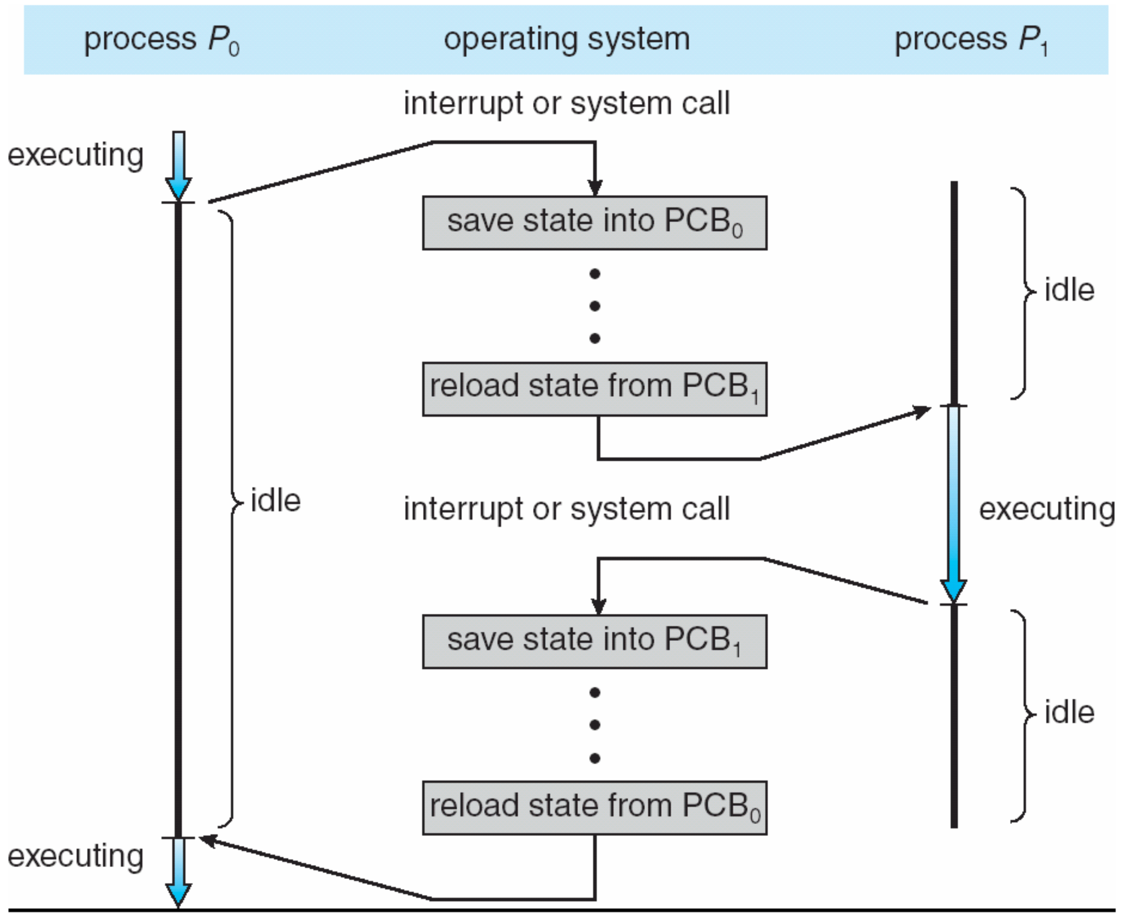
- How to manage processes?
- Information associated with each process

(also task control block)

- Process state
- Program counter
- CPU registers – contents of all process-centric registers
- CPU scheduling info. – priorities, scheduling queue pointers
- Memory-management info. – memory allocated to the process
- Accounting info. – CPU used, clock time elapsed since start, time limits
- I/O status info. – I/O devices allocated to process, list of open files



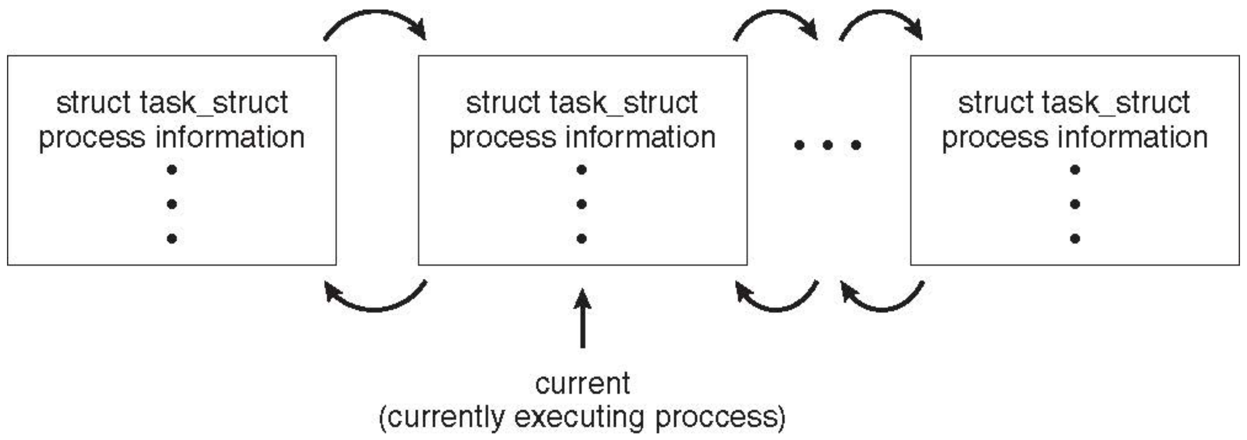
CPU switch from process to process



Process representation in Linux

Represented by the C structure `task_struct`

```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process’ s parent */
struct list_head children; /* this process’ s children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



Process scheduling

- Process scheduler selects among available processes for next execution on CPU

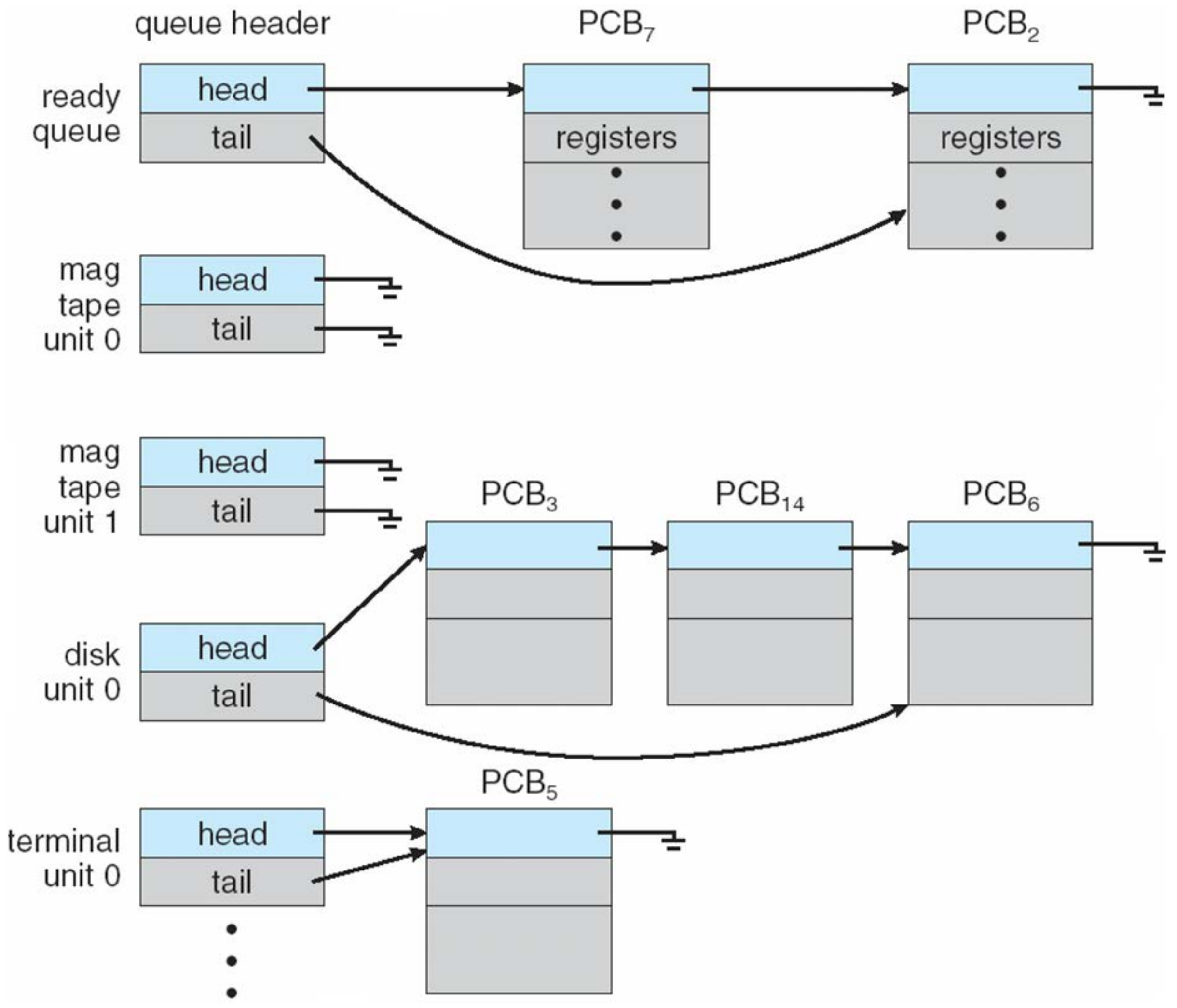
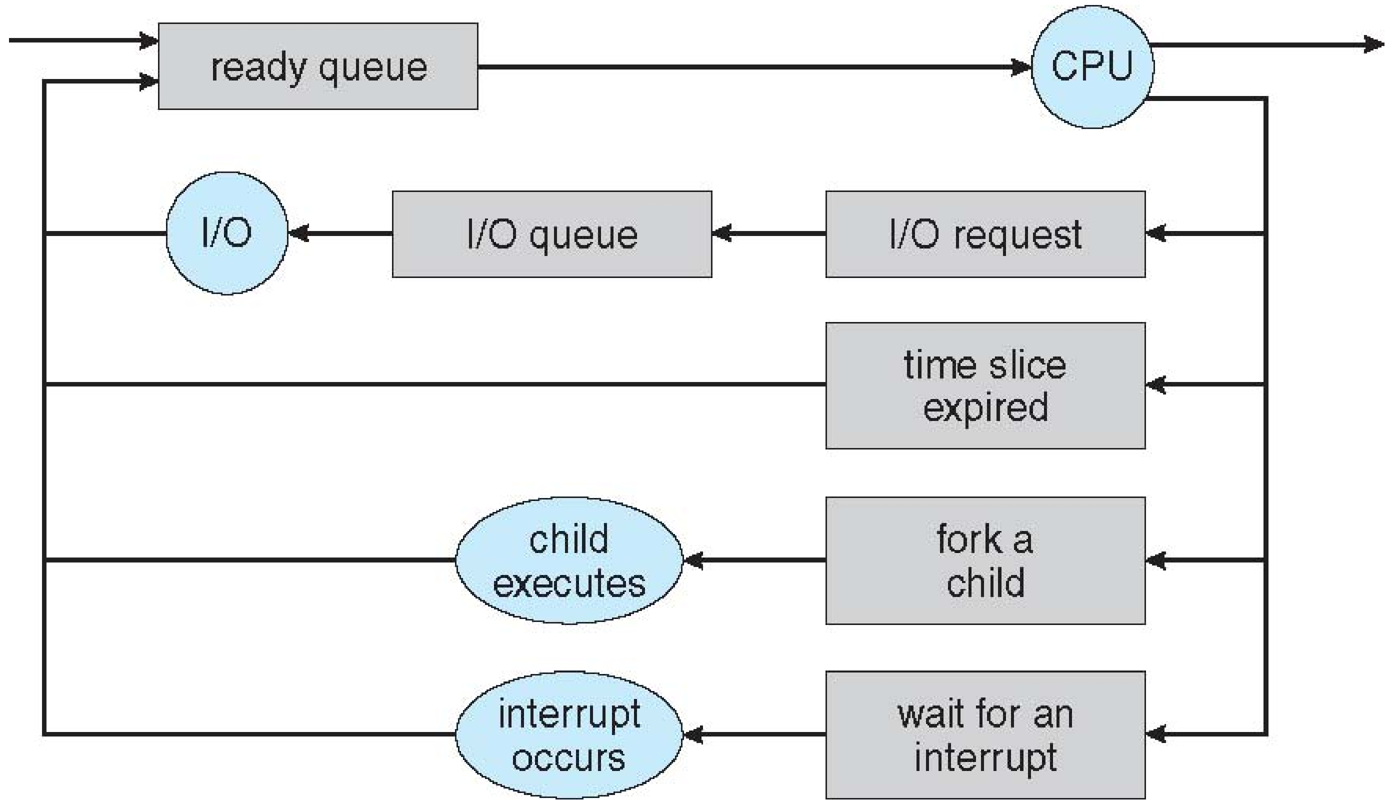


Diagram representation of process scheduling

- **Queueing diagram** represents queues, resources, flows

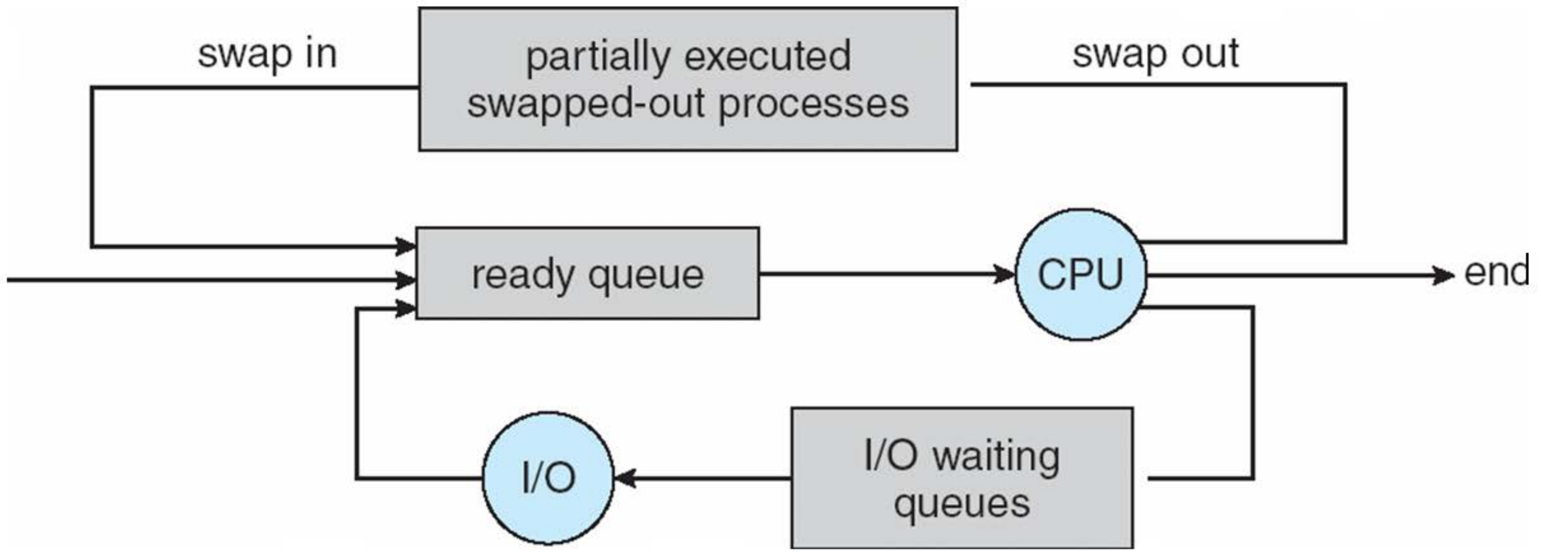


Schedulers

- Short-term scheduler (or CPU scheduler)
  - selects which process should be executed next and allocates CPU
    - Sometimes the only scheduler in a system
    - Short-term scheduler is invoked frequently (milliseconds) $\Rightarrow$ (must be fast)
- Long-term scheduler (or job scheduler)
  - selects which processes should be brought into the ready queue
    - Long-term scheduler is invoked infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
    - The long-term scheduler controls the degree of multiprogramming
- Processes:
  - I/O-bound
    - spends more time doing I/O than computations, many short CPU bursts
  - CPU-bound
    - spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good *process mix*

Example of standard API

- **Medium-term scheduler**
  - Can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



Context switch (تعویض متن)

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch

- Context of a process represented in the PCB

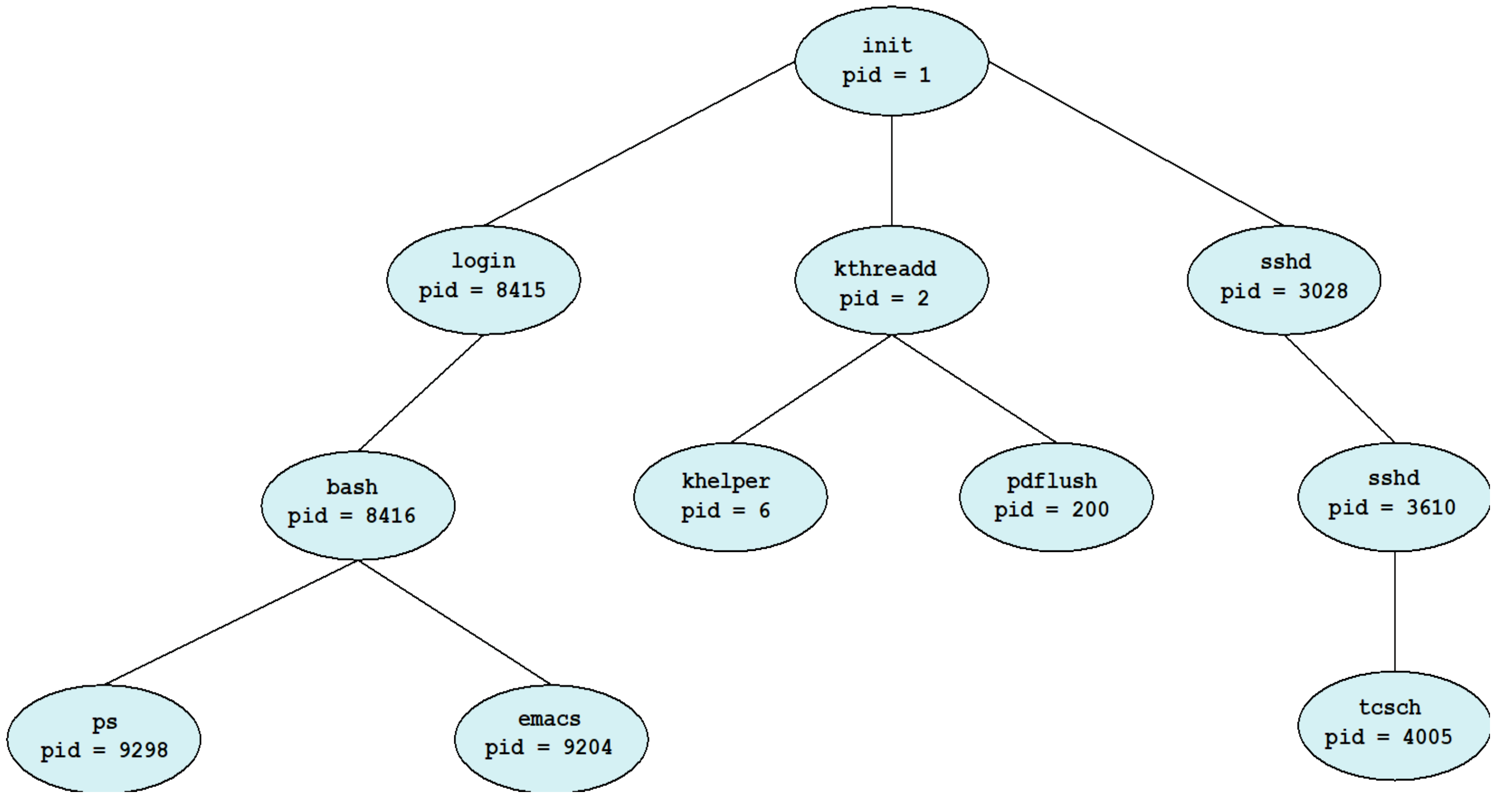
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch

- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once

Process creation

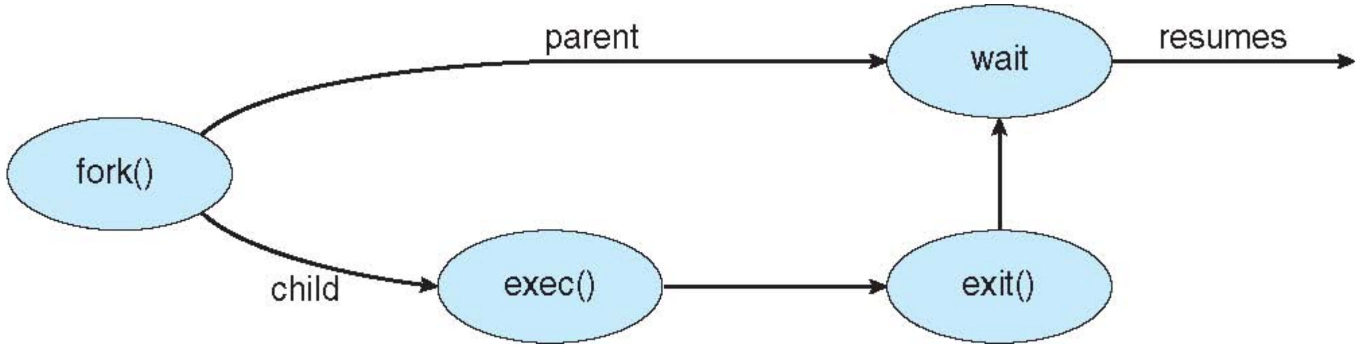
- Parent vs. Child
- Generally, process identified and managed via a process identifier (pid)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

A tree of processes in Linux



Process creation

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - `fork()` system call creates new process
  - `exec()` system call used after a `fork()` to replace the process' memory space with a new program



Process creation with C

POSIX

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

   /* fork a child process */
   pid = fork();

   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }
   else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
   }
   else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
   }

   return 0;
}
```

Windows

```c
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

   /* allocate memory */
   ZeroMemory(&si, sizeof(si));
   si.cb = sizeof(si);
   ZeroMemory(&pi, sizeof(pi));

   /* create child process */
   if (!CreateProcess(NULL, /* use command line */
    "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
    NULL, /* don't inherit process handle */
    NULL, /* don't inherit thread handle */
    FALSE, /* disable handle inheritance */
    0, /* no creation flags */
    NULL, /* use parent's environment block */
    NULL, /* use parent's existing directory */
    &si,
    &pi))
   {
     fprintf(stderr, "Create Process Failed");
     return -1;
   }
   /* parent will wait for the child to complete */
   WaitForSingleObject(pi.hProcess, INFINITE);
   printf("Child Complete");

   /* close handles */
   CloseHandle(pi.hProcess);
   CloseHandle(pi.hThread);
}
```

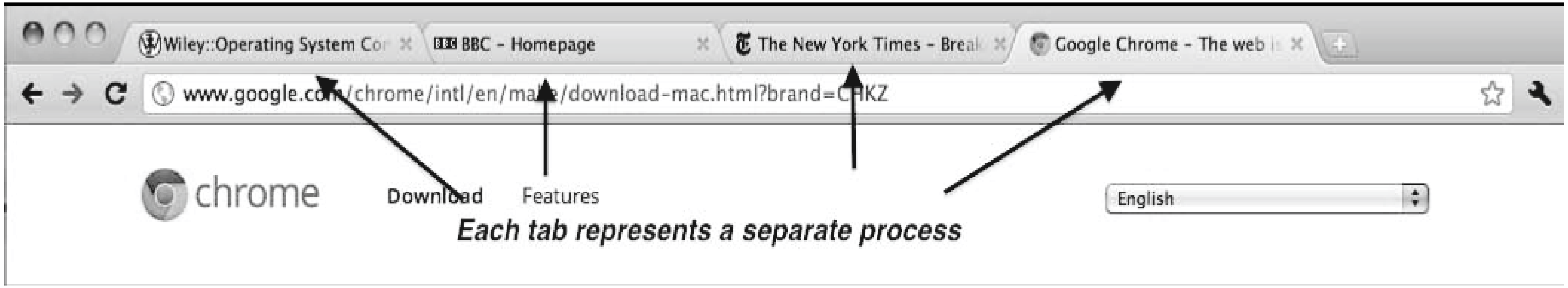
Process termination

- Child → Parent
  - Process' resources are deallocated when:
    - *exit(n)*
    - *return()* in *main()*
  - Catch exit status → *wait()*
    - `pid = wait(&status);`
- Parent → Child
  - *abort()*
  - Why?
    - Child has exceeded allocated resources
    - Task assigned to child is no longer required
    - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Problems of process termination

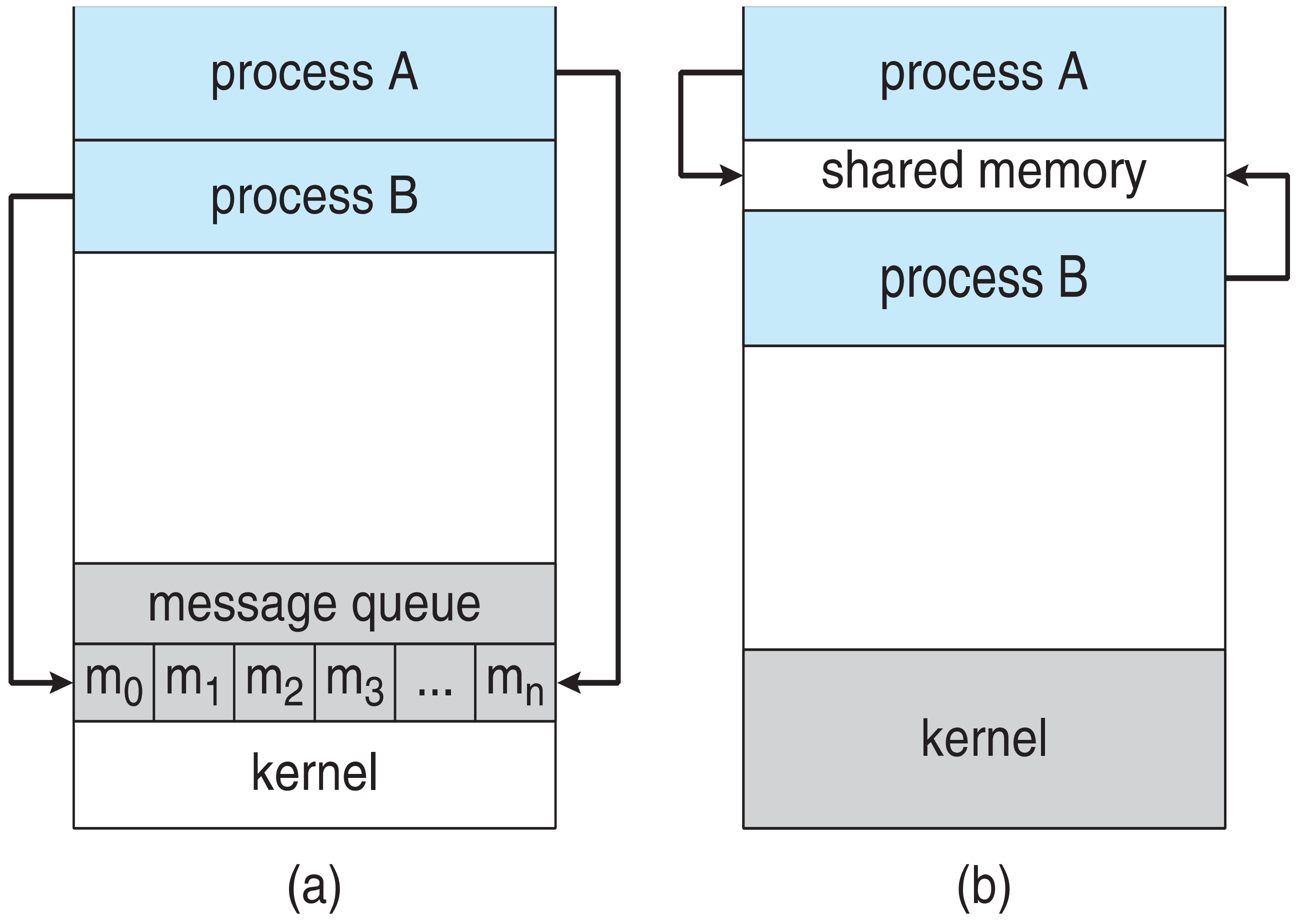
- zombie process
  - No parent waiting
- orphan process
  - Parent termination without wait

- Multi process example: Chrome Browser
  - Browser, Renderer, Plugins, etc


Wiley::Operating System Co
BBC – Homepage
The New York Times – Break
Google Chrome – The web i
www.google.com/chrome/intl/en/make/download-mac.html?brand=CHKZ
chrome
Download
Features
English
Each tab represents a separate process

Interprocess communication (*IPC*)

- Process:
  - independent vs. cooperating
- Cooperating process:
  - Shared memory
  - Message passing



Circular buffer & producer-consumer problem

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

```
item next_produced;
while (true) {
        /* produce an item in next produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
       while (in == out) ; /* do nothing */

       next_consumed = buffer[out];
       out = (out + 1) % BUFFER_SIZE;

       /* consume the item in next consumed */
}
```

Message passing

- **Direct communication (unidirectional)**
  - **send** (*P, message*) – send a message to process P
  - **`receive`**(*Q, message*) – receive a message from process Q

- **Indirect communication (uni & bidirectional)**
  - Messages are directed and received from mailboxes (or ports)
  - Can be used by multiple processes
  - Primitives are defined as:
    - **send**(*A, message*) – send a message to mailbox A
    - **`receive`**(*A, message*) – receive a message from mailbox A

Synchronization

- Blocking vs. non-blocking

- Blocking is considered synchronous
  - Blocking send
  - Blocking receive

- Non-blocking is considered asynchronous
  - Non-blocking send
  - Non-blocking receive
    - The receiver receives
      - A valid message
      - Null message

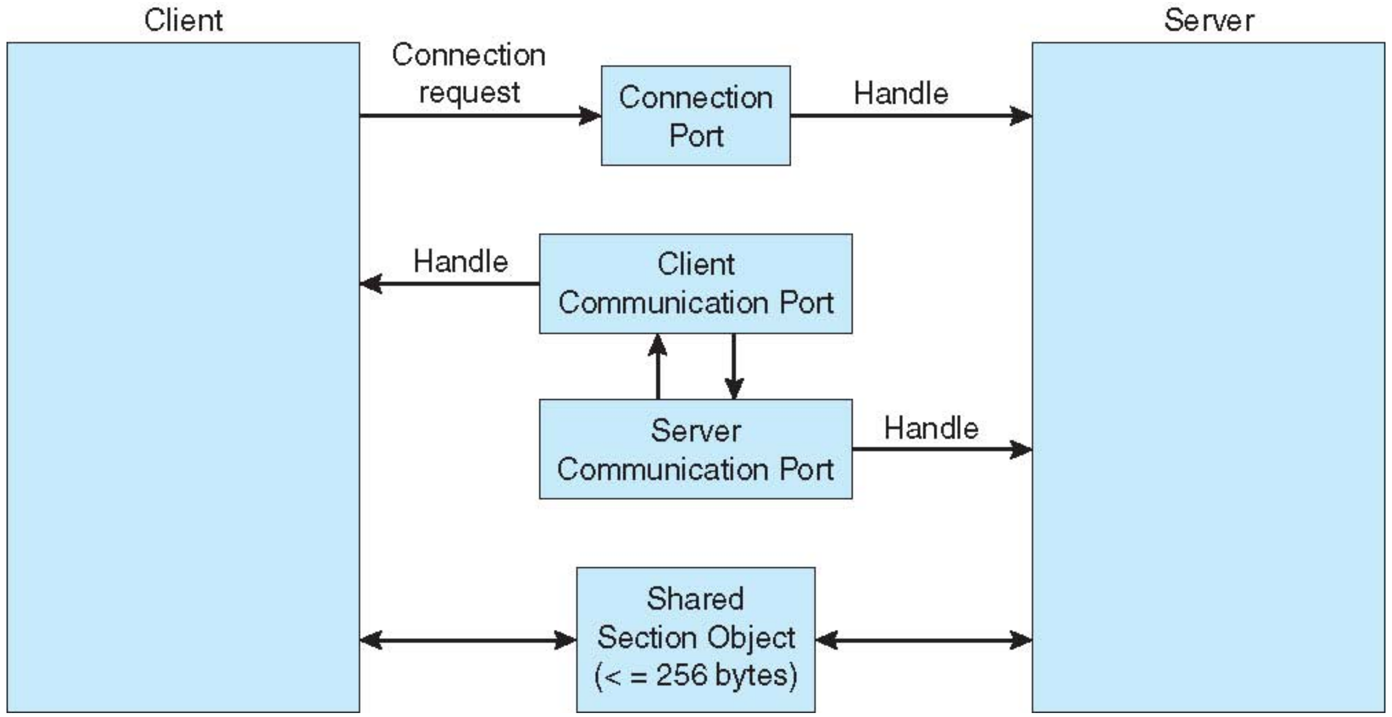
- Different combinations possible
  - If both send and receive are blocking, we have a rendezvous

POSIX examples of shared memory: (sender->receiver)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* create the shared memory object */
   shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

   /* configure the size of the shared memory object */
   ftruncate(shm_fd, SIZE);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* open the shared memory object */
   shm_fd = shm_open(name, O_RDONLY, 0666);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

   /* read from the shared memory object */
   printf("%s",(char *)ptr);

   /* remove the shared memory object */
   shm_unlink(name);

   return 0;
}
```

Local procedure calls in Windows


Client
Server
Connection request
Connection Port
Handle
Handle
Client Communication Port
Server Communication Port
Handle
Shared Section Object (< = 256 bytes)

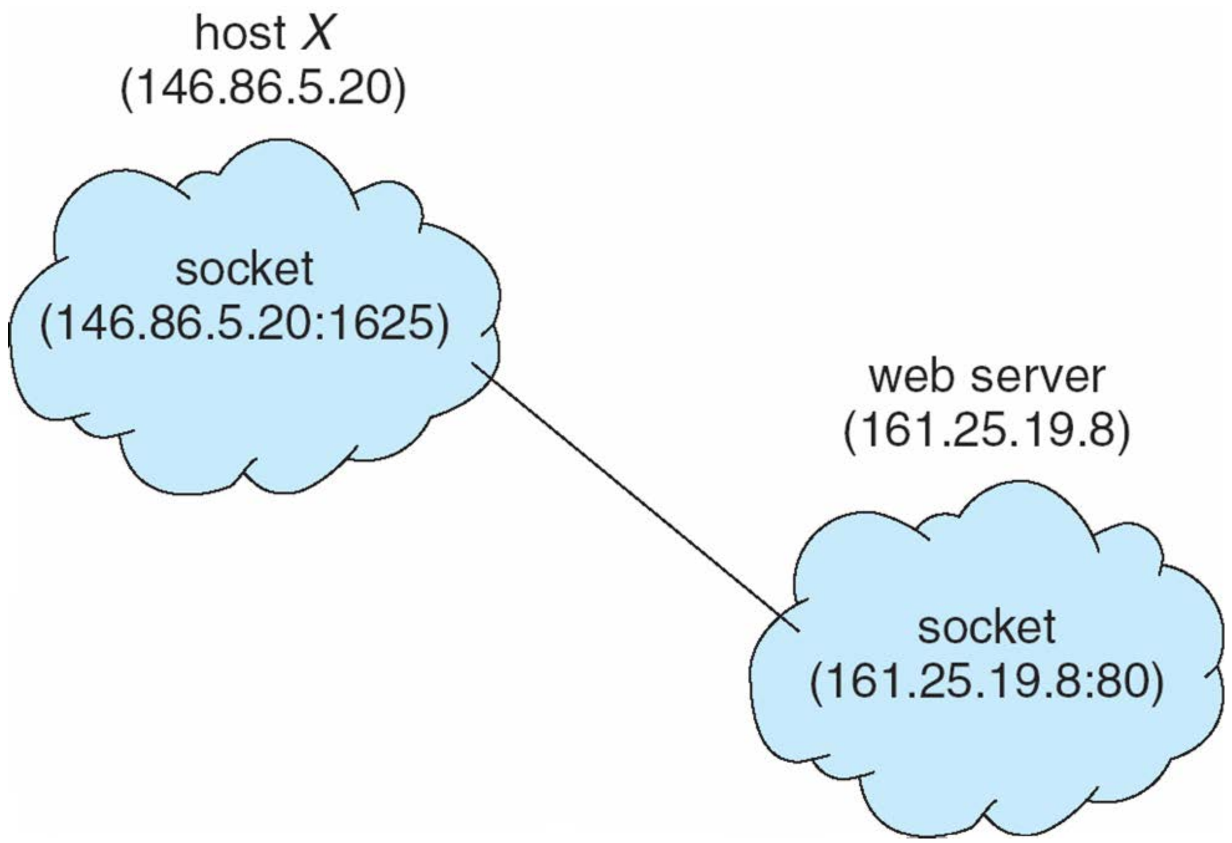
Communications in client-server systems

- Sockets
- Remote Procedure Calls (windows)
- Pipes
- Remote Method Invocation (Java)

Sockets

- A socket is defined as an endpoint for communication
- Concatenation of IP address and port – a number included at start of message packet to differentiate network services on a host
- The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- Communication consists between a pair of sockets
- All ports below 1024 are *well known*, used for standard services
- Special IP address 127.0.0.1 (loopback) to refer to system on which process is running

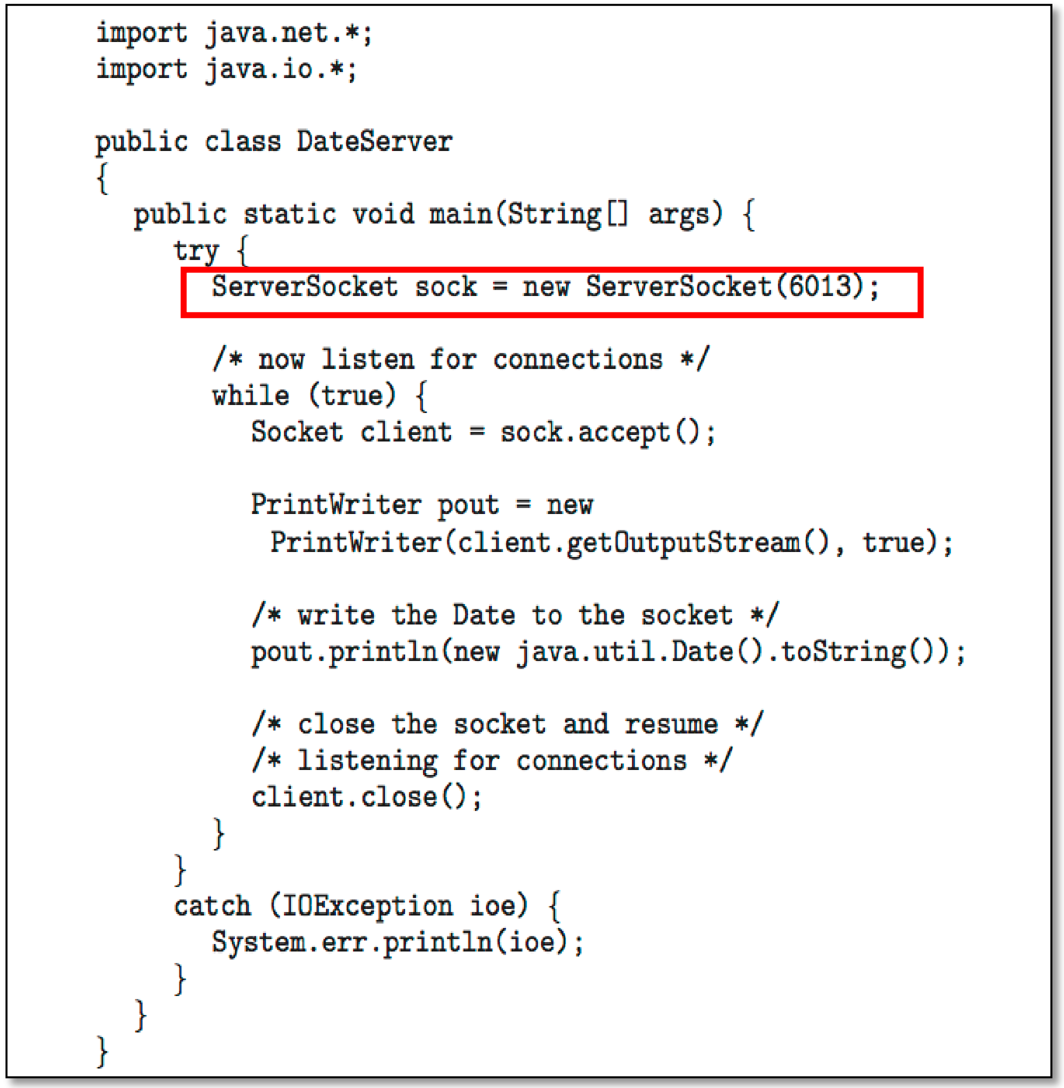
Socket communication



Sockets in Java

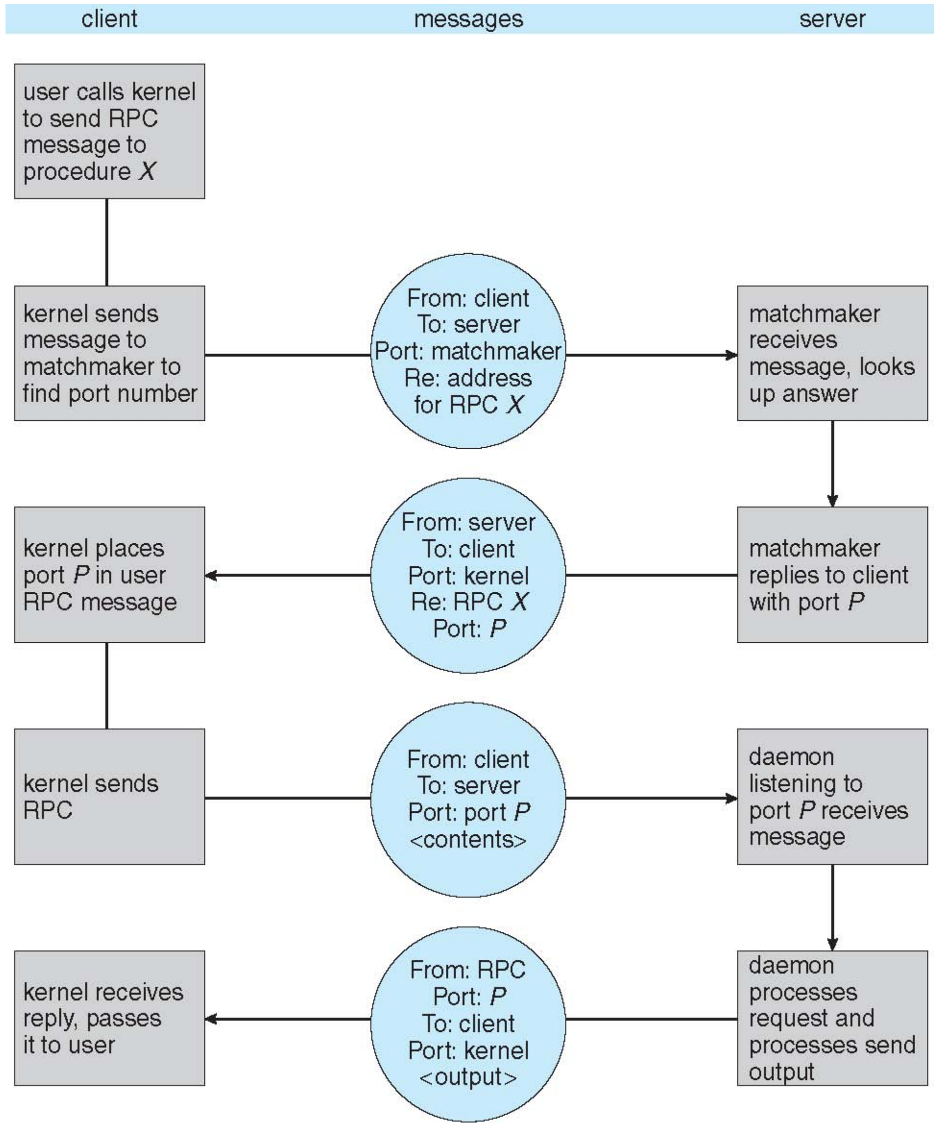
- Three types of sockets
  - Connection-oriented (TCP)
  - Connectionless (UDP)
  - `MulticastSocket` class– data can be sent to multiple recipients

- Consider this "Date" server:



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

public class DateServer
{
  public static void main(String[] args) {
    try {
      ServerSocket sock = new ServerSocket(6013);

      /* now listen for connections */
      while (true) {
        Socket client = sock.accept();

        PrintWriter pout = new
         PrintWriter(client.getOutputStream(), true);

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

        /* close the socket and resume */
        /* listening for connections */
        client.close();
      }
    }
    catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

Execution of RPC (Remote Procedure Call)


client
messages
server
user calls kernel to send RPC message to procedure *X*
kernel sends message to matchmaker to find port number
From: client To: server Port: matchmaker Re: address for RPC *X*
matchmaker receives message, looks up answer
matchmaker replies to client with port *P*
From: server To: client Port: kernel Re: RPC *X* Port: *P*
kernel places port *P* in user RPC message
kernel sends RPC
From: client To: server Port: port *P* <contents>
daemon listening to port *P* receives message
daemon processes request and processes send output
From: RPC Port: *P* To: client Port: kernel <output>
kernel receives reply, passes it to user

Pipes

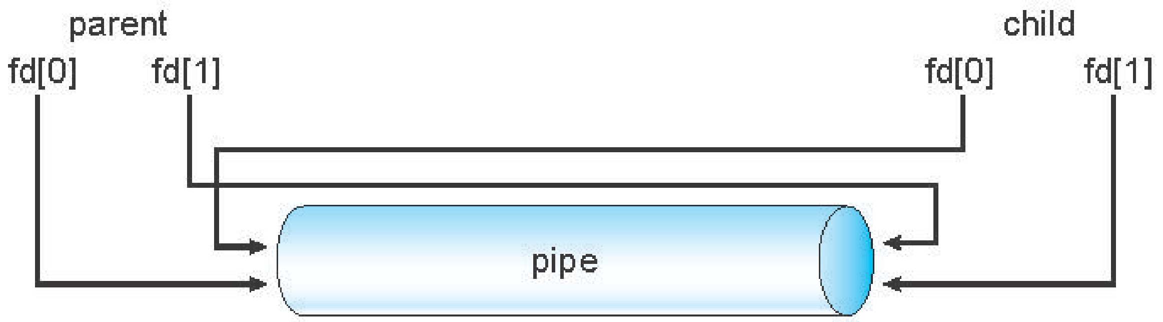
- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., *parent-child*) between the communicating processes?
  - Can the pipes be used over a network?

- Ordinary pipes
  - cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.

- Named pipes
  - can be accessed without a parent-child relationship.

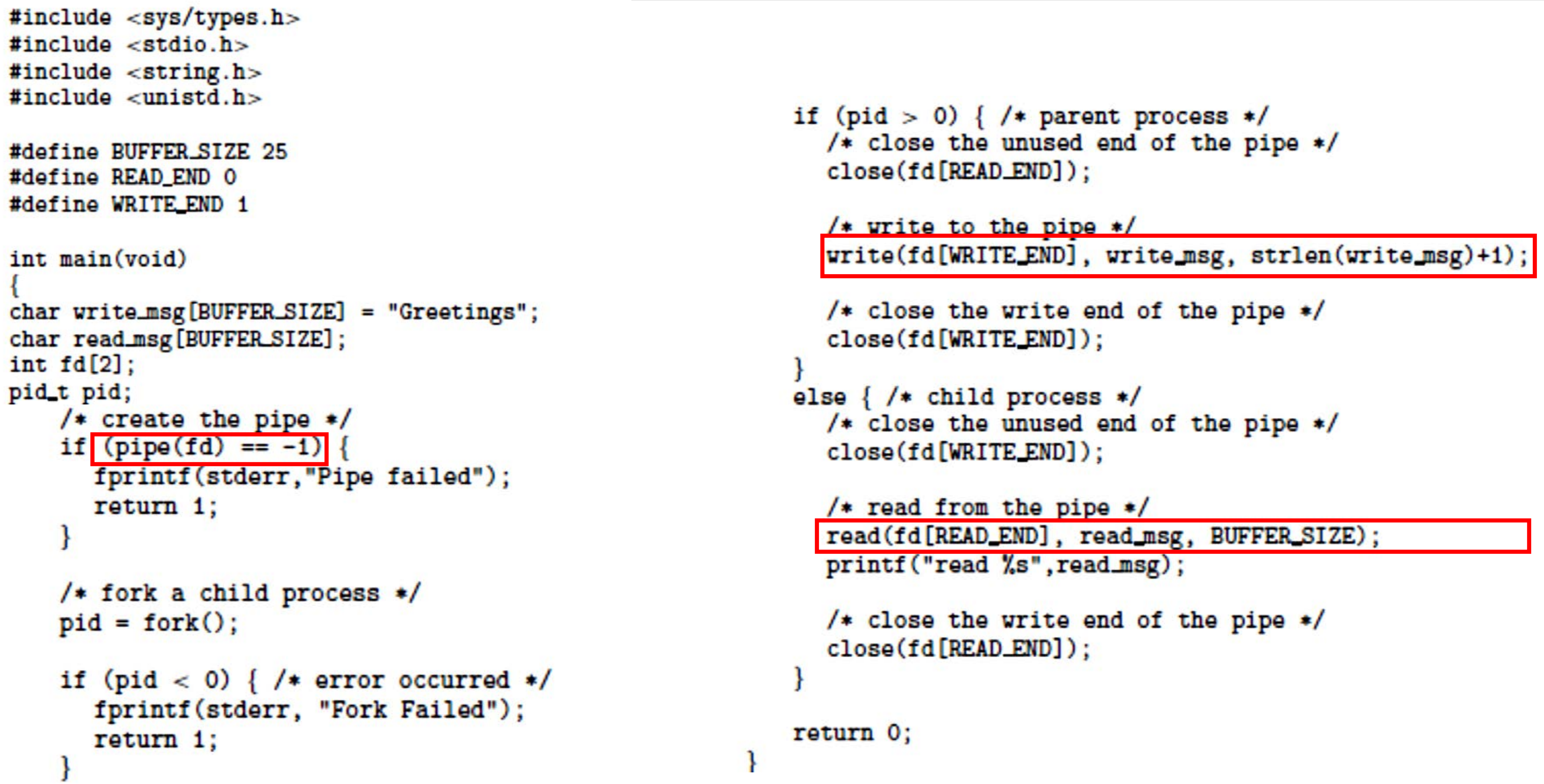
Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the write-end of the pipe)
- Consumer reads from the other end (the read-end of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes



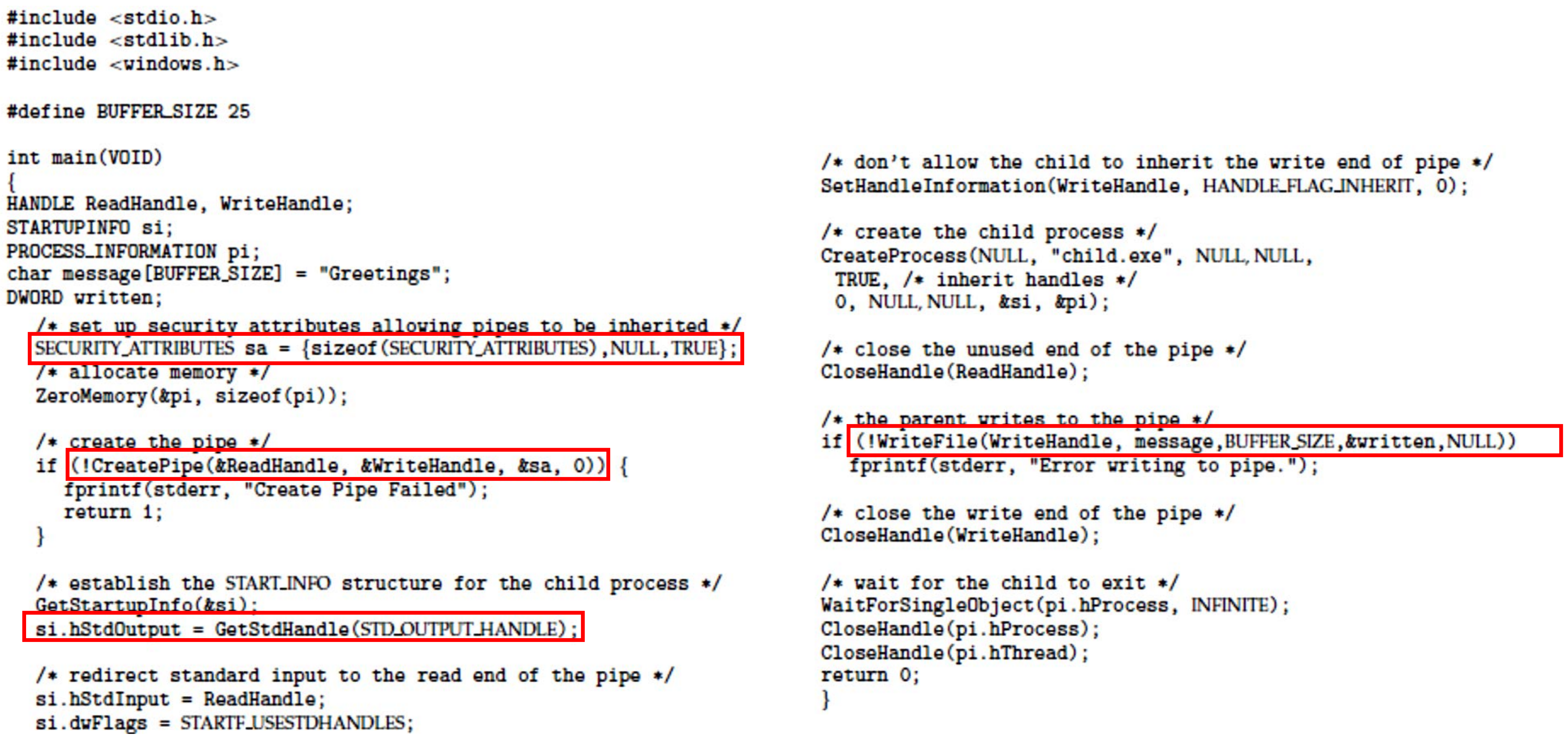
- Windows calls these anonymous pipes
- See Unix and Windows code samples in textbook

Ordinary pipe (POSIX), parent-child



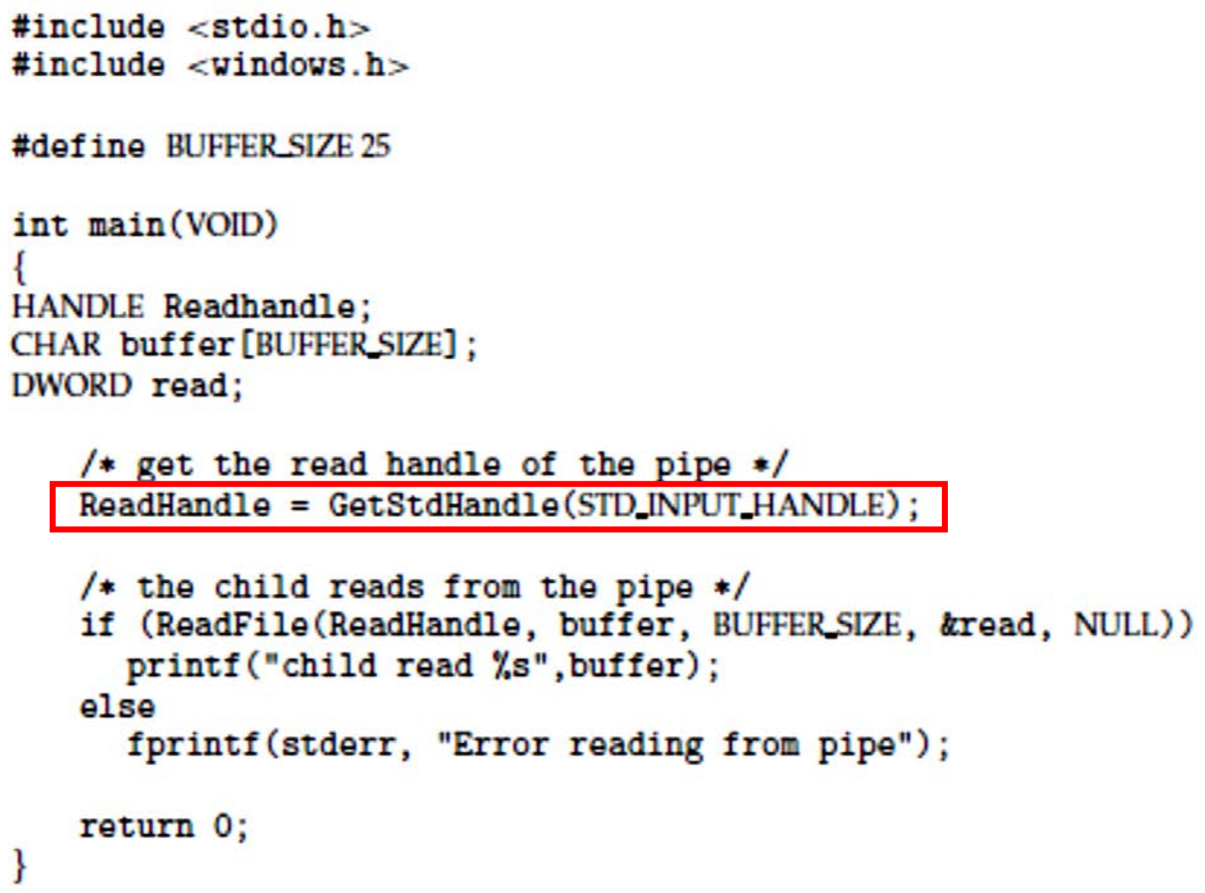
```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
char write_msg[BUFFER_SIZE] = "Greetings";
char read_msg[BUFFER_SIZE];
int fd[2];
pid_t pid;
    /* create the pipe */
    if (pipe(fd) == -1) {
      fprintf(stderr,"Pipe failed");
      return 1;
    }

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
    }

    if (pid > 0) { /* parent process */
      /* close the unused end of the pipe */
      close(fd[READ_END]);

      /* write to the pipe */
      write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

      /* close the write end of the pipe */
      close(fd[WRITE_END]);
    }
    else { /* child process */
      /* close the unused end of the pipe */
      close(fd[WRITE_END]);

      /* read from the pipe */
      read(fd[READ_END], read_msg, BUFFER_SIZE);
      printf("read %s",read_msg);

      /* close the write end of the pipe */
      close(fd[READ_END]);
    }

    return 0;
}
```

Ordinary pipe (windows), parent



```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE ReadHandle, WriteHandle;
STARTUPINFO si;
PROCESS_INFORMATION pi;
char message[BUFFER_SIZE] = "Greetings";
DWORD written;
  /* set up security attributes allowing pipes to be inherited */
  SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES),NULL,TRUE};
  /* allocate memory */
  ZeroMemory(&pi, sizeof(pi));

  /* create the pipe */
  if (!CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)) {
     fprintf(stderr, "Create Pipe Failed");
     return 1;
  }

  /* establish the START_INFO structure for the child process */
  GetStartupInfo(&si);
  si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

  /* redirect standard input to the read end of the pipe */
  si.hStdInput = ReadHandle;
  si.dwFlags = STARTF_USESTDHANDLES;

/* don't allow the child to inherit the write end of pipe */
SetHandleInformation(WriteHandle, HANDLE_FLAG_INHERIT, 0);

/* create the child process */
CreateProcess(NULL, "child.exe", NULL, NULL,
 TRUE, /* inherit handles */
 0, NULL, NULL, &si, &pi);

/* close the unused end of the pipe */
CloseHandle(ReadHandle);

/* the parent writes to the pipe */
if (!WriteFile(WriteHandle, message,BUFFER_SIZE,&written,NULL))
  fprintf(stderr, "Error writing to pipe.");

/* close the write end of the pipe */
CloseHandle(WriteHandle);

/* wait for the child to exit */
WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
return 0;
}
```

Ordinary pipe (windows), child



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

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE Readhandle;
CHAR buffer[BUFFER_SIZE];
DWORD read;

   /* get the read handle of the pipe */
   ReadHandle = GetStdHandle(STD_INPUT_HANDLE);

   /* the child reads from the pipe */
   if (ReadFile(ReadHandle, buffer, BUFFER_SIZE, &read, NULL))
     printf("child read %s",buffer);
   else
     fprintf(stderr, "Error reading from pipe");

   return 0;
}
```

Named pipes

- Named Pipes are more powerful than ordinary pipes (?)
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

Questions?