Processes

Chapter 3

Processes

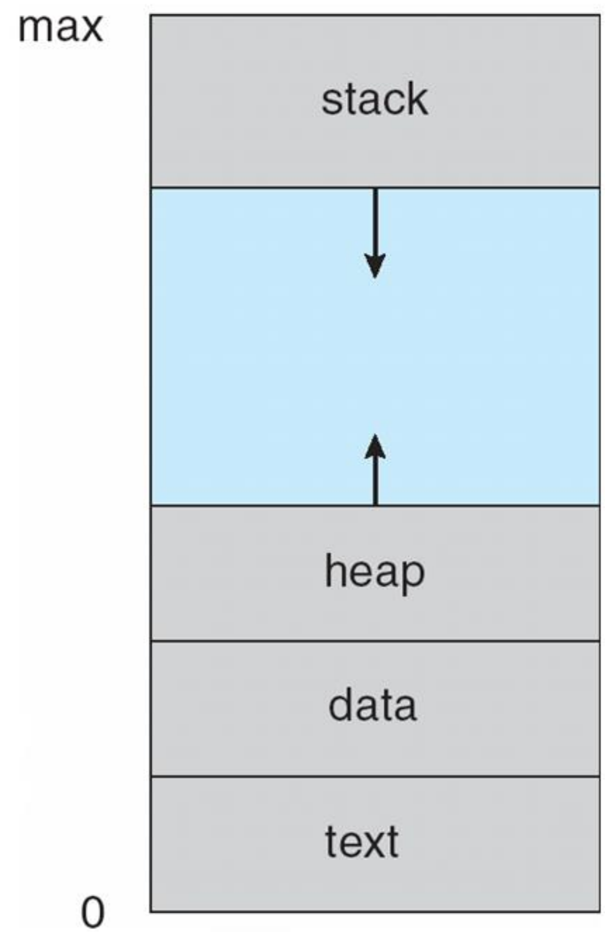
- Process Concept
- Process Scheduling
- Operations on Processes
- Inter-process Communication
- Communication in Client-Server Systems

Process Concept

- **Process** – a program in execution; process execution must progress in sequential fashion
- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

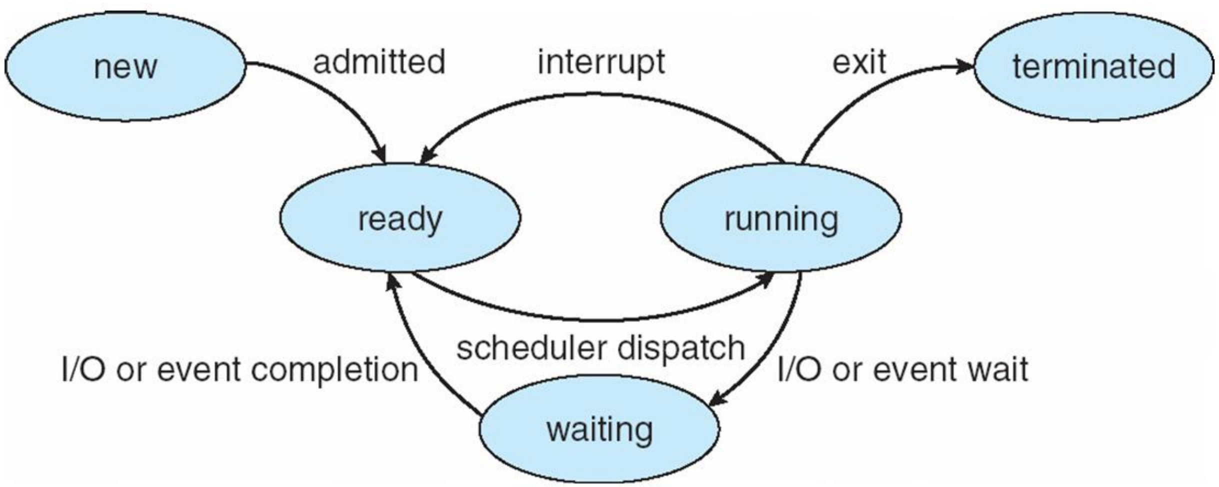
Process in Memory

- Multiple parts of process memory space
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

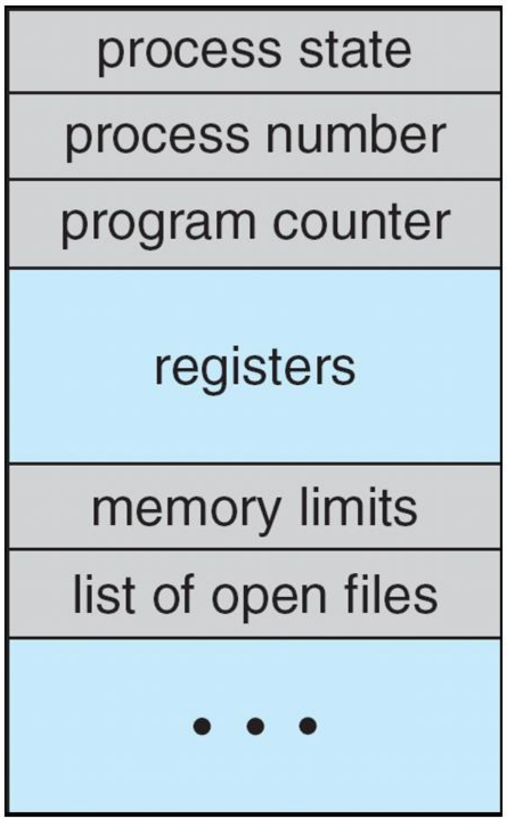


Process State

- 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)



- Information associated with each process
  - Process state – running, waiting, etc
  - Program counter – location of instruction to next execute
  - CPU registers – contents of all process-centric registers
  - CPU scheduling information- priorities, scheduling queue pointers
  - Memory-management information – memory allocated to the process
  - Accounting information – CPU used, clock time elapsed since start, time limits
  - I/O status information – I/O devices allocated to process, list of open files
- Threads

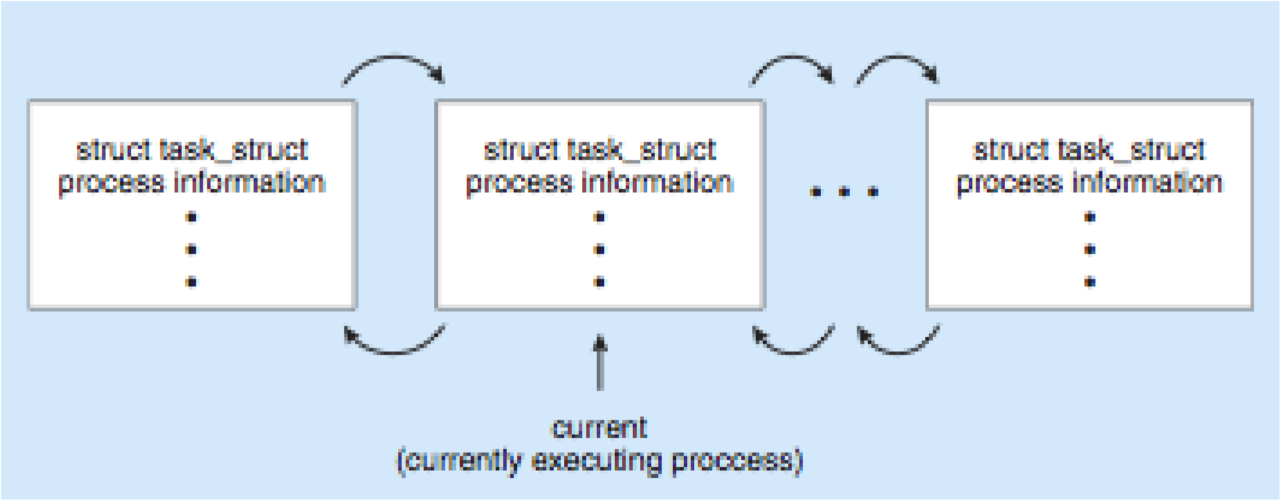
Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB
- More in next chapter

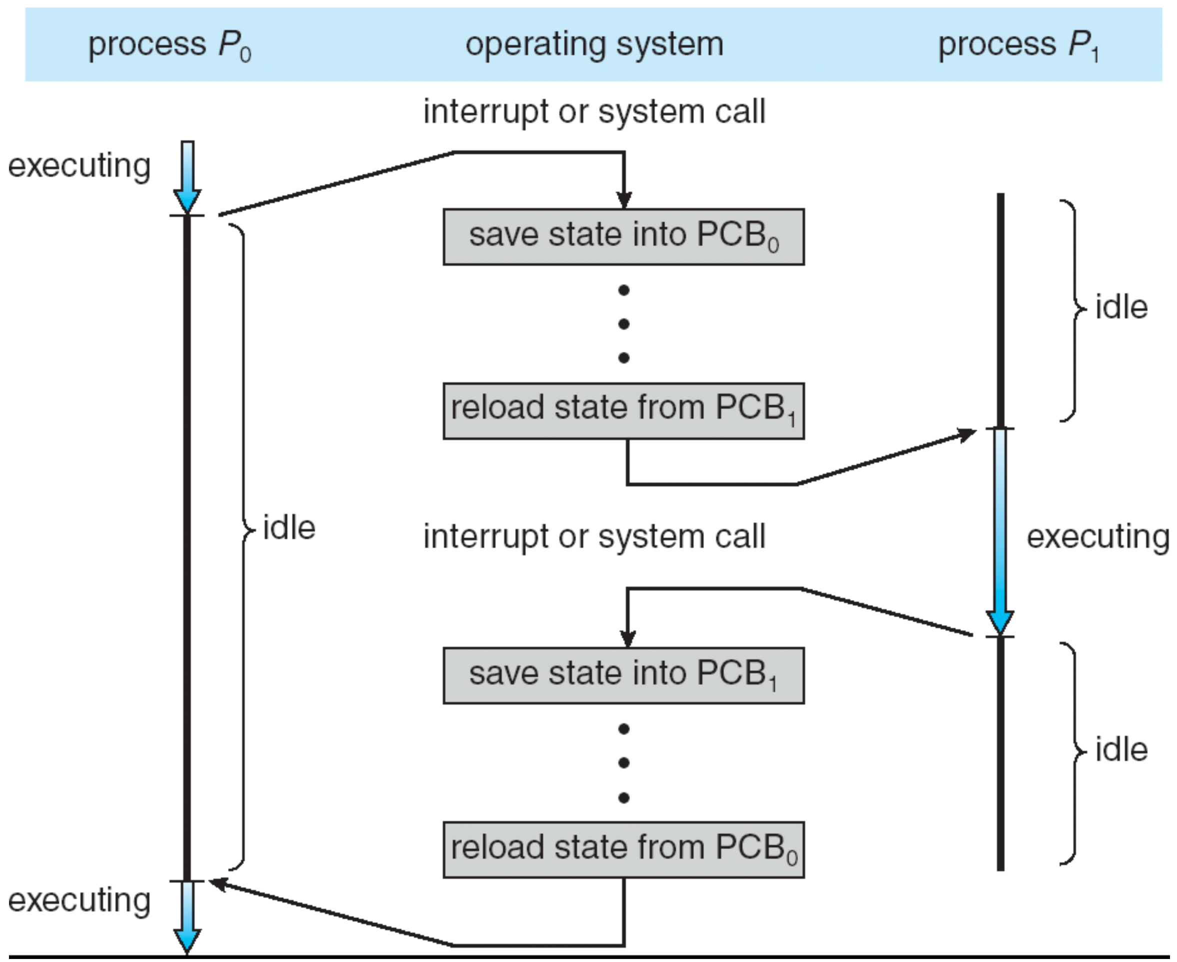
Process Representation in Linux

- 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 */
```



CPU Switch From Process to Process



Context Switch

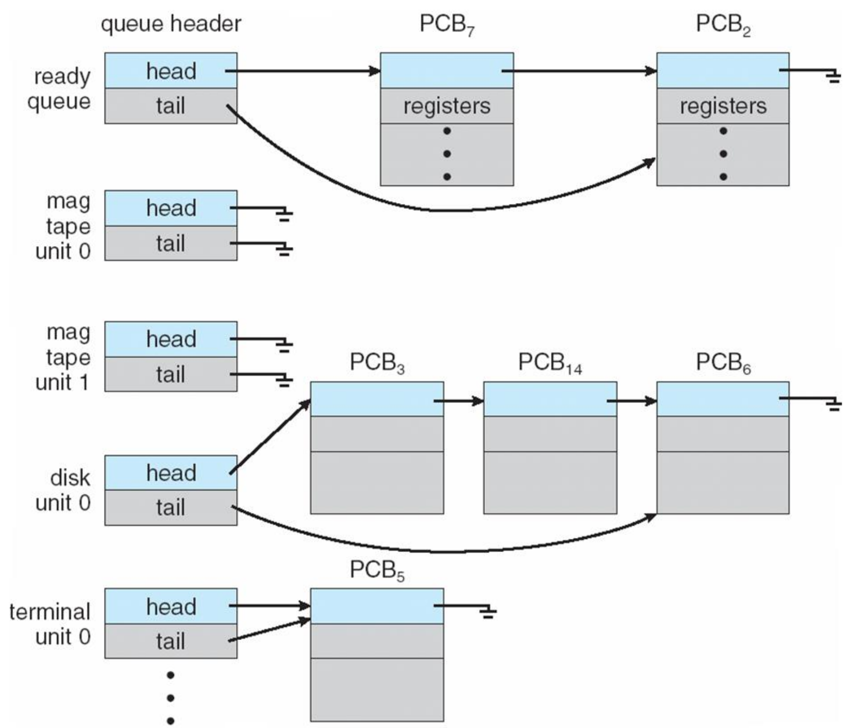
- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB -> longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU -> multiple contexts loaded at once

Process Context

- Which of following is NOT part of process context

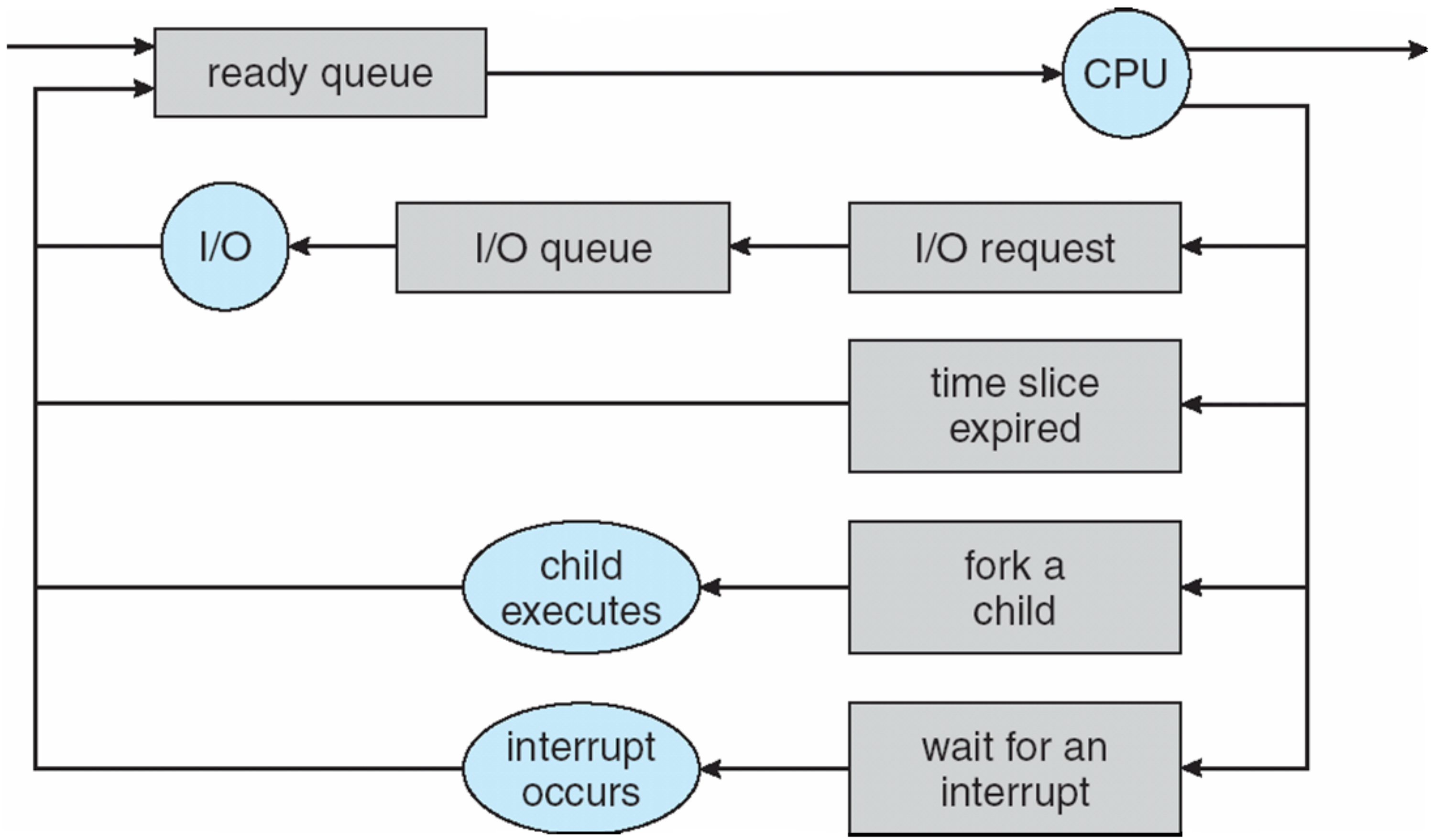
A	Program Counter
B	Register Values
C	Data section
D	Memory-management information
E	All of above are in process context

Process Scheduling Queues



- **Job queue** (process table) – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Representation of Process Scheduling



Schedulers

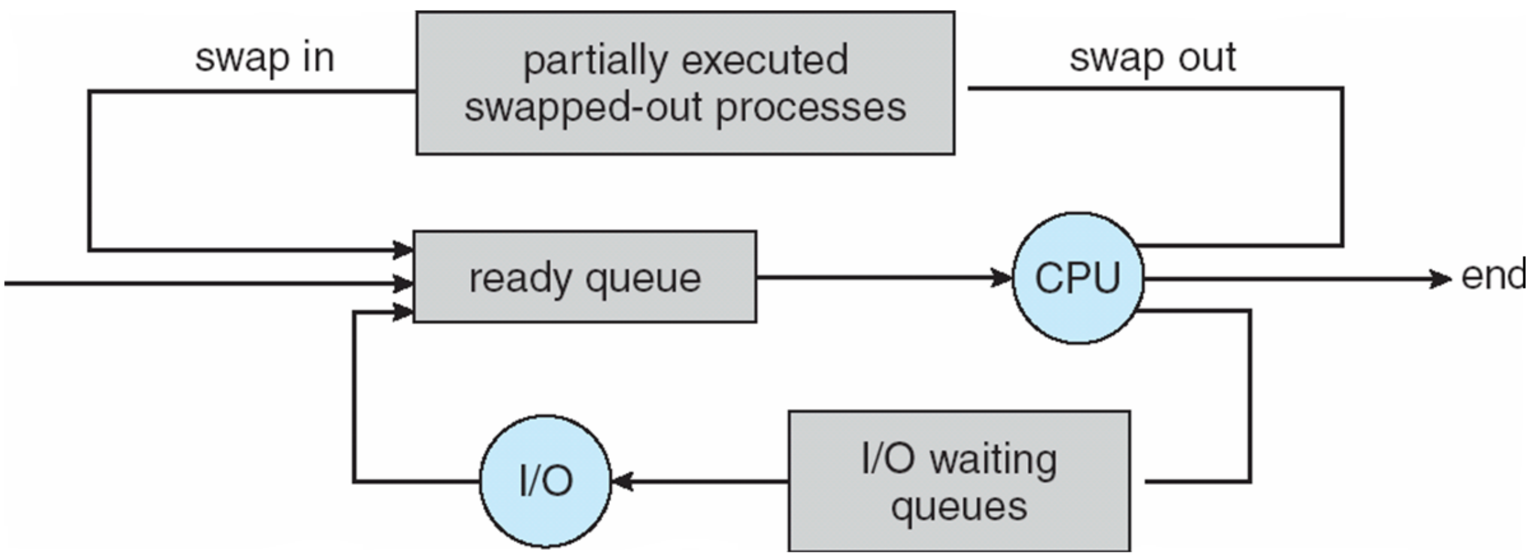
- OS uses schedulers to select processes from various queues.
- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the system
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU

Schedulers (Cont.)

- Short-term scheduler is invoked very frequently (milliseconds) $\Rightarrow$ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
  - Long-term scheduler strives for good process mix

Addition of Medium Term Scheduling

- **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**



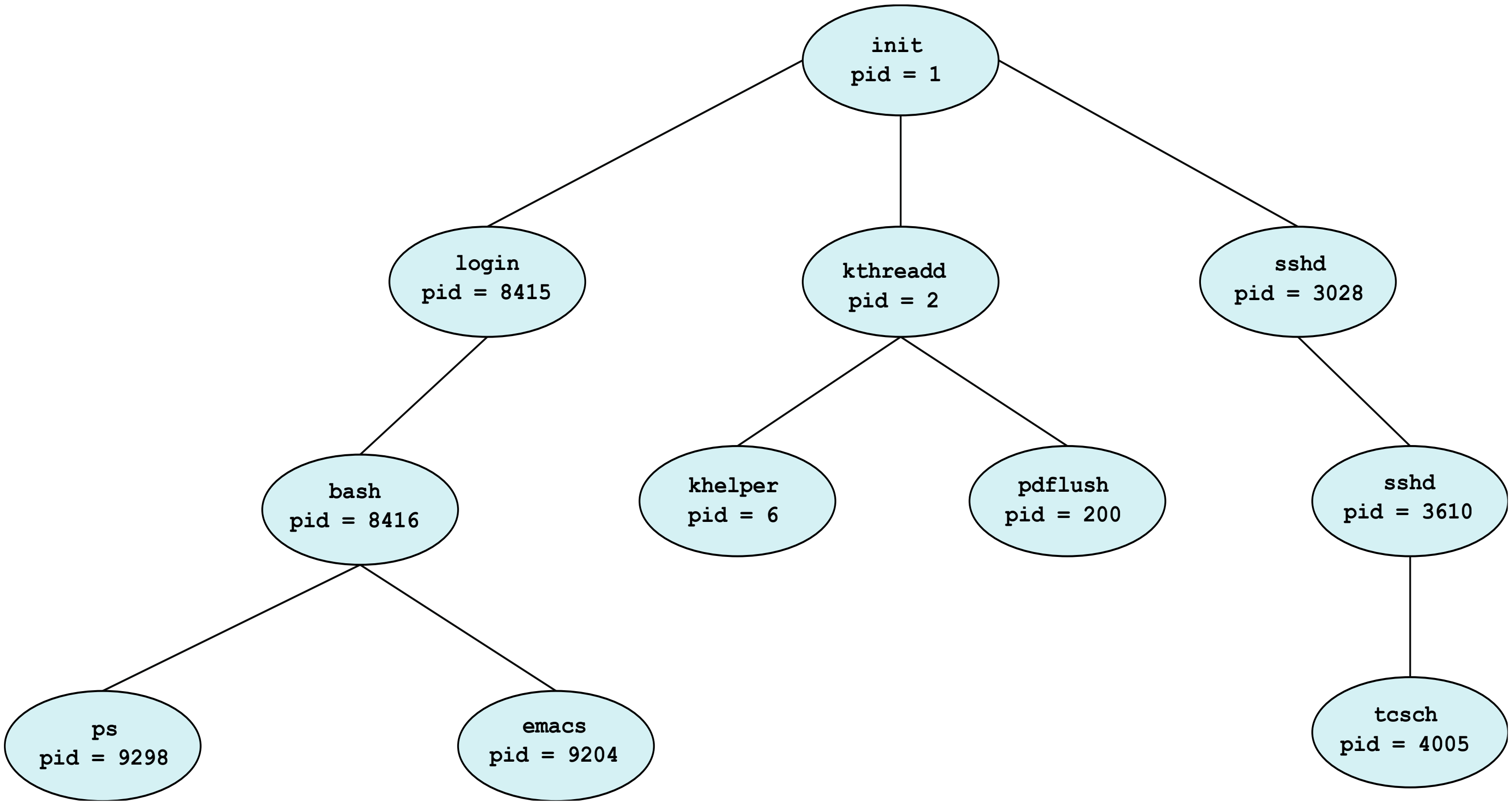
Multitasking in Mobile Systems

- Some systems / early systems allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
  - Single **foreground** process- controlled via user interface
  - Multiple **background** processes– in memory, running, but not on the display, and with limits
    - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
  - Background process uses a **service** to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

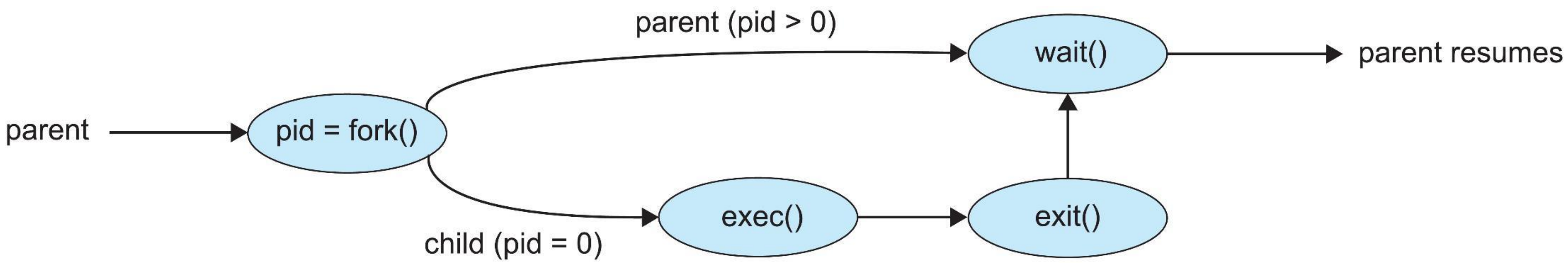
Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- 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 (Cont.)



- 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

Forking Example p1.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
  } else {
   // parent goes down this path (original process)
   printf("hello, I am parent of %d (pid:%d)\n",   rc, (int) getpid());
  }
  return 0;
}
```

Forking Example p2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
   printf("hello world (pid:%d)\n", (int) getpid());
   int rc = fork();
   if (rc < 0) {
      // fork failed; exit
      fprintf(stderr, "fork failed\n");
      exit(1);
   } else if (rc == 0) {
      // child (new process)
      printf("hello, I am child (pid:%d)\n", (int) getpid());
      sleep(1);
   } else {
      // parent goes down this path (original process)
      int wc = wait(NULL);
      printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",  rc, wc, (int) getpid());
   }
   return 0;
}
```

Forking Example p3.c

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

int main(int argc, char *argv[]) {
   printf("hello world (pid:%d)\n", (int) getpid());
   int rc = fork();
   if (rc < 0) {
      // fork failed; exit
      fprintf(stderr, "fork failed\n");
      exit(1);
   } else if (rc == 0) {     // child (new process)
      printf("hello, I am child (pid:%d)\n", (int) getpid());
      char *myargs[3];
      myargs[0] = "wc";   // program: "wc" (word count)
      myargs[1] = "p3.c"; // argument: file to count
      myargs[2] = NULL;          // marks end of array
      execvp(myargs[0], myargs);  // runs word count
      printf("this shouldn't print out\n");
   } else {       // parent goes down this path (original process)
      int wc = wait(NULL);
      printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
                                    rc, wc, (int) getpid());
   }
   return 0;
}
```

Practice Question

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

int value = 5;

int main()
{
pid_t pid;

   pid = fork();

   if (pid == 0) { /* child process */
      value += 15;
      return 0;
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("PARENT: value = %d",value); /* LINE A */
      return 0;
   }
}
```

Figure 3.30 What output will be at Line A?

What output will be at Line A

A	PARENT: value = 0
B	PARENT: value = 5
C	PARENT: value = 15
D	PARENT: value = 20
E	None of Above

Practice Question

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

int main()
{
    /* fork a child process */
    fork();

    /* fork another child process */
    fork();

    /* and fork another */
    fork();

    return 0;
}
```

Figure 3.31 How many processes are created?

How many processes are created, including the parent process?

A	3
B	4
C	8
D	12
E	None of Above

Process Termination

- Process executes last statement and asks the operating system to delete it (**`exit`**), e.g. **`exit(0)`**
  - Process' resources are deallocated by operating system
  - Returns status data from child to parent (via **`wait()`**)
- The parent may wait for a child to terminate using the **`wait()`** system call.

```
int cpid_done, status;
cpid_done = wait(&status);
```

  - The caller sleeps until any child terminates.
  - cpid_done gets the pid of the child that terminated.

Process Termination

- Parent may terminate execution of children processes
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
- If parent is exiting
  - Some operating system do not allow child to continue if its parent terminates
    - All children terminated - *cascading termination*
- If no parent waiting (did not invoke **`wait()`**), a terminated process becomes a **zombie**
- If parent terminated without invoking **`wait`**, a child process is an **orphan**

Basic Shell code

Pseudo-code a basic shell (not real C++ code):

```
while (1) {
   parse_command_line; //get command, args, redirect,
   etc.
   if(cmd == exit) exit();
   p = fork( );
   if (p == 0) {
   execvp (cmd, args)
   } else {
   if (command doesn't end with &)
   wait( );
   }
}
```

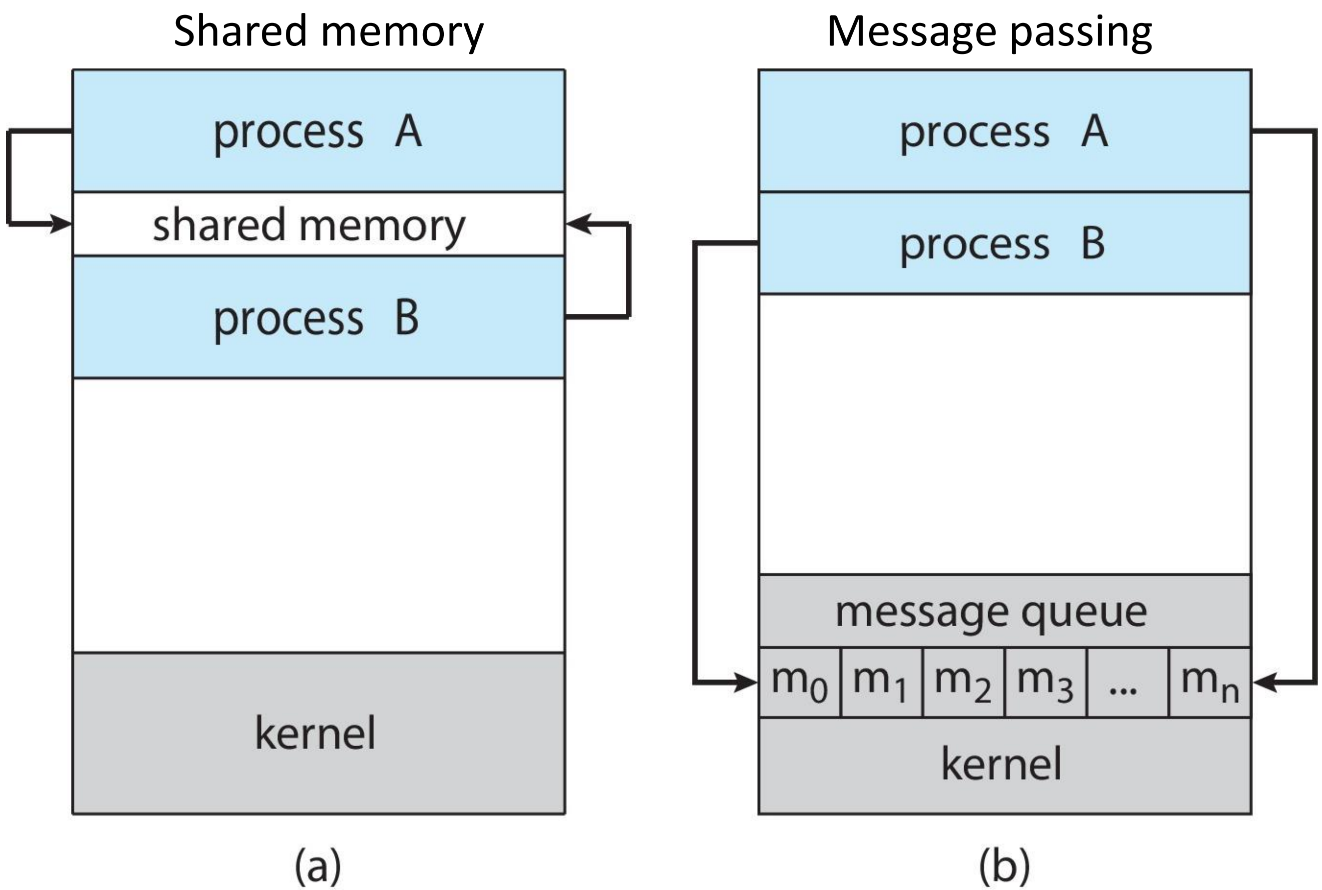
Multi-process Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multi-process with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O. A new browser process is created when Chrome is started
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in

Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models



Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
- A buffer of items that can be filled by the producer and emptied by the consumer
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size

Bounded-Buffer Shared-Memory Solution

- Shared data

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

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

Bounded-Buffer – Producer Process

```
while (true) {
   /* Produce an item */
   while (((in + 1) % BUFFER_SIZE)  == out)
     ; /* do nothing -- no free buffers */
   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
}
```

Bounded Buffer – Consumer Process

```
 while (true) {
   while (in == out)
     ; // do nothing-- nothing to consume

   // remove an item from the buffer
   item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   return item;
}
```

Question

- Which of following is true about the just discussed shared-memory solution for the producer-consumer problem with BUFFER_SIZE =10 ?

A	The solution is incorrect because producer and consumer may modify the same entry at the same time
B	The solution is correct and maximum number of items in the buffer is 10.
C	The solution is correct but maximum number of items in the buffer is 9.
D	The solution is correct only if the producer runs faster than the consumer.
E	None of the above

Examples of IPC Systems – POSIX Shared Memory

- POSIX Shared Memory use memory-mapped files
  - Process first creates shared memory segment

    ```
    shm_fd = shm_open(name, O_CREAT | O_RDRW, 0666);
    ```

  - Also used to open an existing segment to share it
  - Set the size of the object

```
ftruncate(shm_fd, 4096);
```

  - Get pointer to the shared memory object

```
ptr = mmap(0, 4096, PROT_WRITE, MAP_SHARED, shm_fd,0)
```

  - Now the process could write to the shared memory

```
sprintf(ptr, "Writing to shared memory");
```

IPC POSIX Producer

```
#include <stdio.h>
#include <stlib.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_RDRW, 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;
}
```

IPC POSIX Consumer

```
#include <stdio.h>
#include <stlib.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;
}
```

Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details later.

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)

Implementation Questions

- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation options:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Message Passing (Cont.)

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

Direct Communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q
- Properties of direct communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

send(*A, message*) – send a message to mailbox A

receive(*A, message*) – receive a message from mailbox A

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null
- Different combinations of send() and receive() are possible
  - If both send and receive are blocking, we have a **rendezvous**

Producer – Consumer using blocking send & receive

Producer-consumer becomes trivial using blocking send() and receive()

Producer

```
while (true) {
   item = Produce();
   send(item);
}
```

Consumer

```
while (true) {
   receive(&item);

   Consume(item);
}
```

This code is correct and relatively simple. Why don’t we always just use message passing (vs shared memory)

```
/* W/O SHARED MEMORY */

Producer                          Consumer
int item;                         int item;

while (TRUE) {                    while (TRUE) {
    item = Produce ();               receive (Producer, &item);
    send (Consumer, &item);          Consume (item);
}                                 }
```

A. Message passing copies more data.

B. Message passing only works across a network.

C. Message passing is a security risk.

D. We usually do use message passing!

Buffering

- Queue of messages attached to the link;
- Implemented in one of three ways

1. Zero capacity – 0 messages are queued on a link.
   Sender must wait for receiver (rendezvous).
2. Bounded capacity – finite length of *n* messages
   Sender must wait if link full
3. Unbounded capacity – infinite length
   Sender never waits

Unix Pipes

- Unix pipes allow communication between two processes using a buffer.

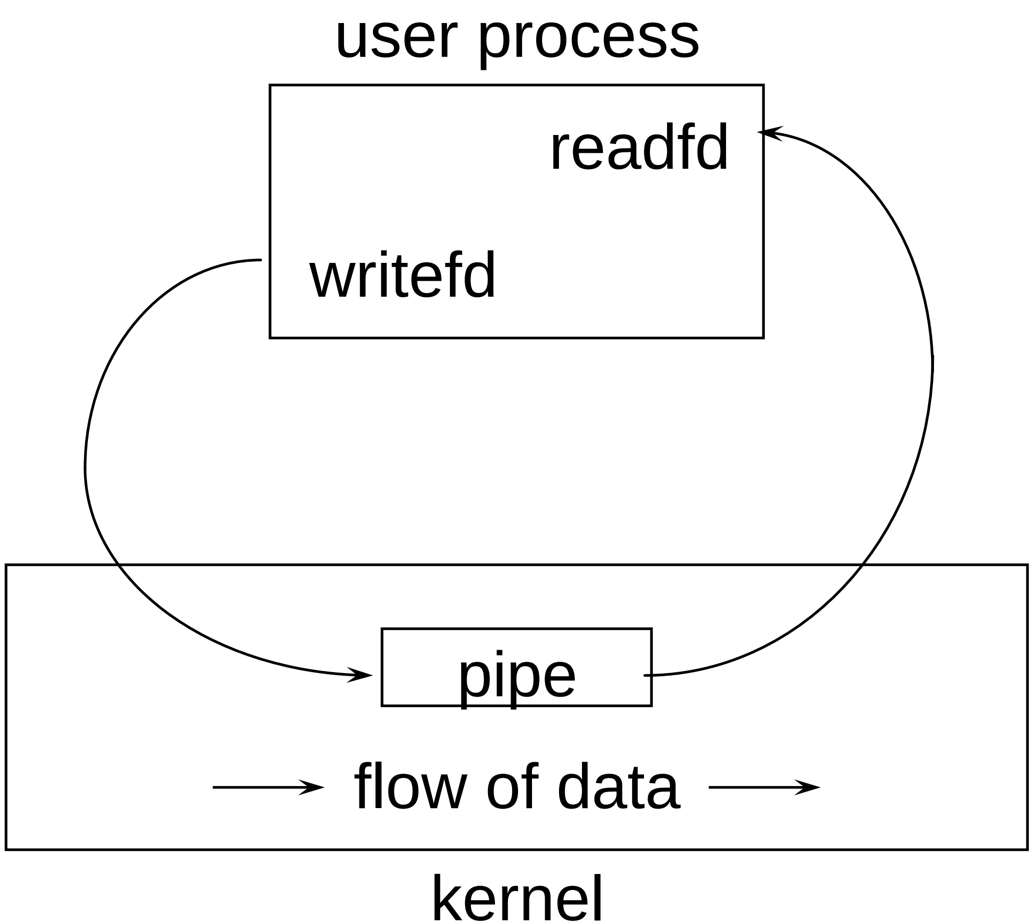
```
ls -l | grep test
```

- One process writes to the buffer and one process reads from the buffer.
- The communication is unidirectional.
- The buffer is accessed using a file descriptor.
- A pipe is created by using the pipe system call

```
int pipe(int* filedes);
```

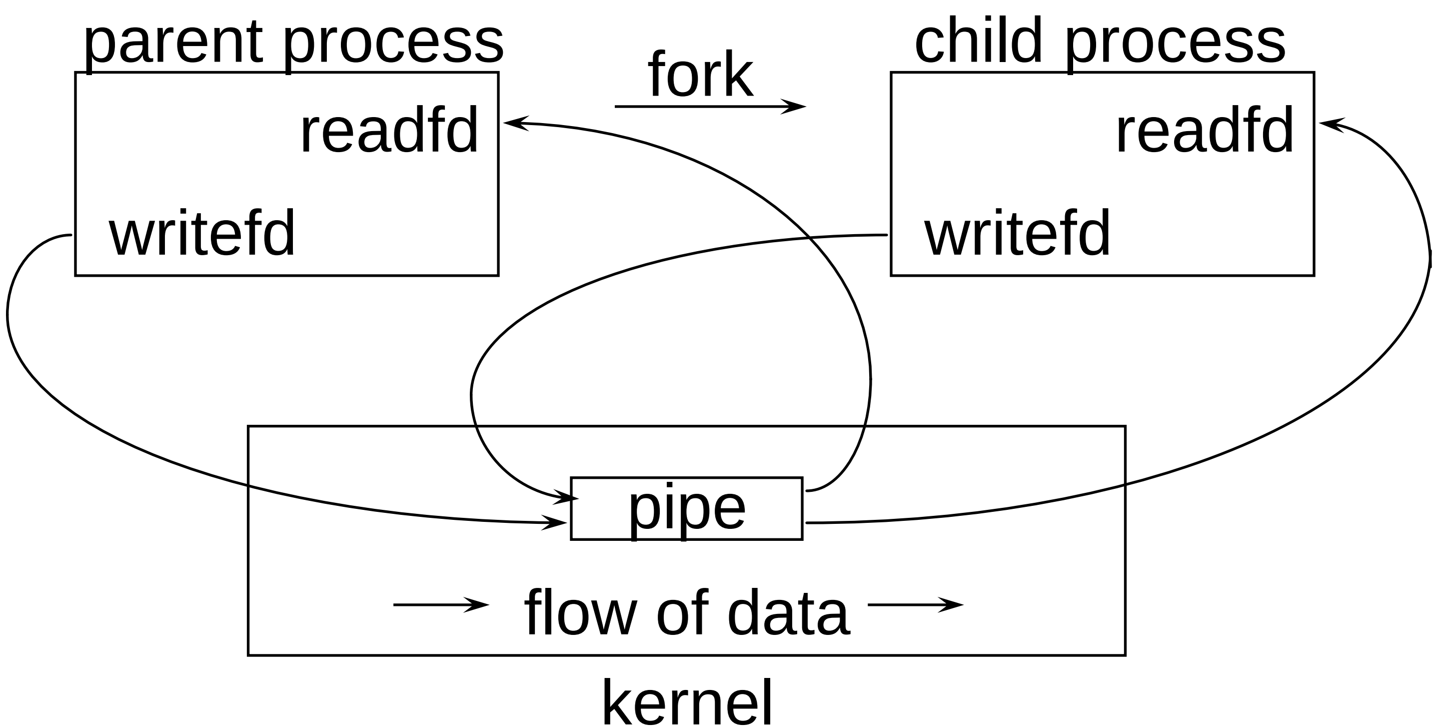
  - Two file descriptors are returned
    - filedes[0] is open for reading
    - filedes[1] is open for writing
- The difference between a file and a pipe: pipe is a data structure in the kernel.
- Typical size is 512 bytes (Minimum limit defined by POSIX)

A Pipe Object



Ordinary Pipe

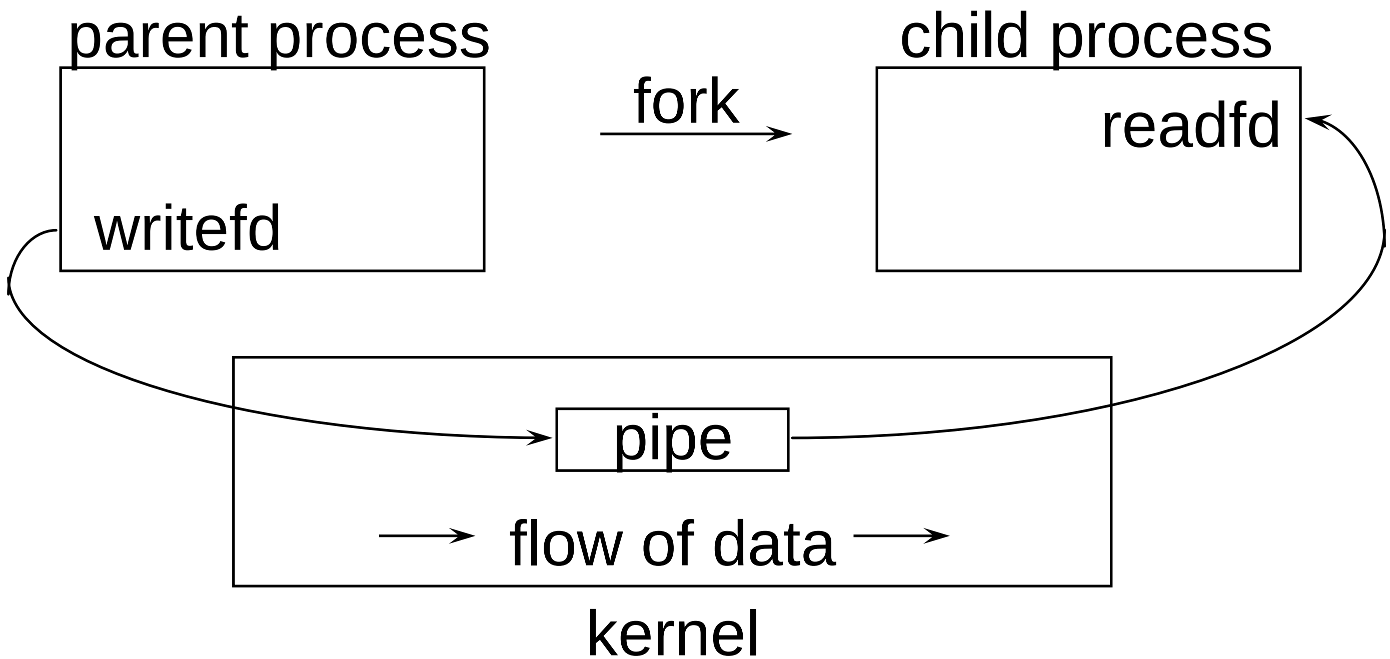
- First, a process creates a pipe, and then forks to create a copy of itself.



Pipe Examples

Parent writes file, child reads file

- parent closes read end of pipe
- child closes write end of pipe



Pipe Example

```
int main(void)
{
        char write_msg[BUFFER_SIZE] = "Greetings";
        char read_msg[BUFFER_SIZE];
        pid_t pid;
        int fd[2];

        /* create the pipe */
        if (pipe(fd) == -1) {
                fprintf(stderr,"Pipe failed");
                return 1;
        }
        printf("readfd = %d, writefd = %d\n", fd[0], fd[1]);

        /* now fork a child process */
        pid = fork();
        if (pid < 0) {
                fprintf(stderr, "Fork failed");
                return 1;
        }
```

Pipe Example (cont.)

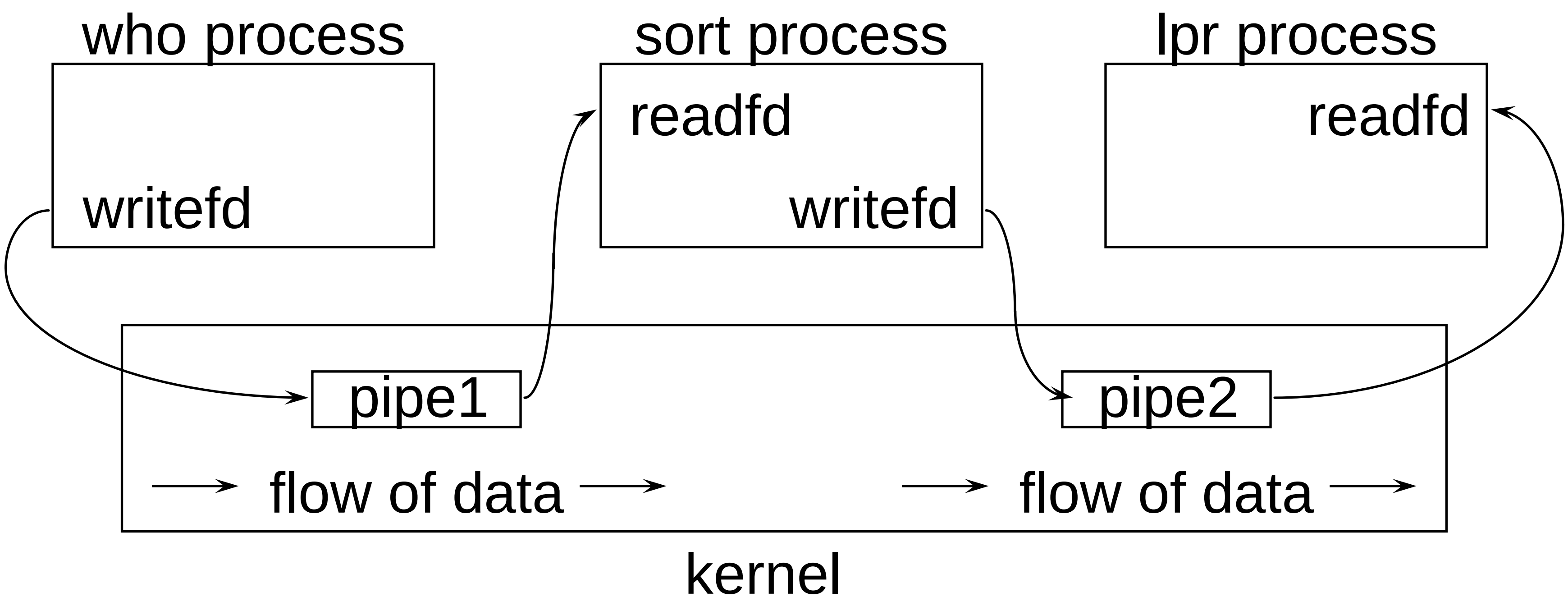
```
if (pid > 0) {  /* parent process */
      /* close the unused end of the pipe */
      close(fd[READ_END]);
      /* write to the pipe */
      write(fd[WRITE_END], write_msg, strlen(write_msg)+1);
      /* close the write end of the pipe */
      close(fd[WRITE_END]);
} else { /* child process */
     /* close the unused end of the pipe */
     close(fd[WRITE_END]);
     /* read from the pipe */
     read(fd[READ_END], read_msg, BUFFER_SIZE);
     printf("child read %s\n",read_msg);

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

If fd[0] = 3 and fd[1] = 4, what is the output of the above code?

Concatenated Pipe Examples

who | sort | lpr

- who process writes to pipe1
- sort process reads from pipe1, writes to pipe2
- lpr process reads from pipe2

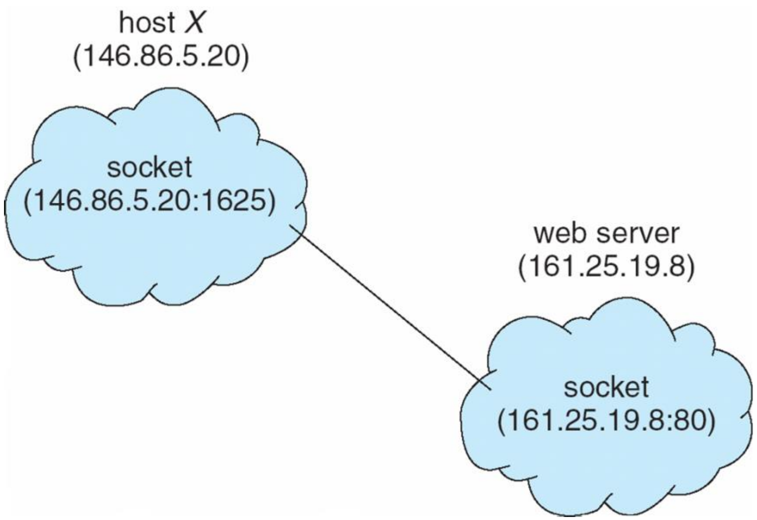


Client-Server Communication

- Sockets
- Remote Procedure Calls

Sockets

- A socket is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- All ports below 1024 are ***well known***, used for standard services
- Communication consists between a pair of sockets



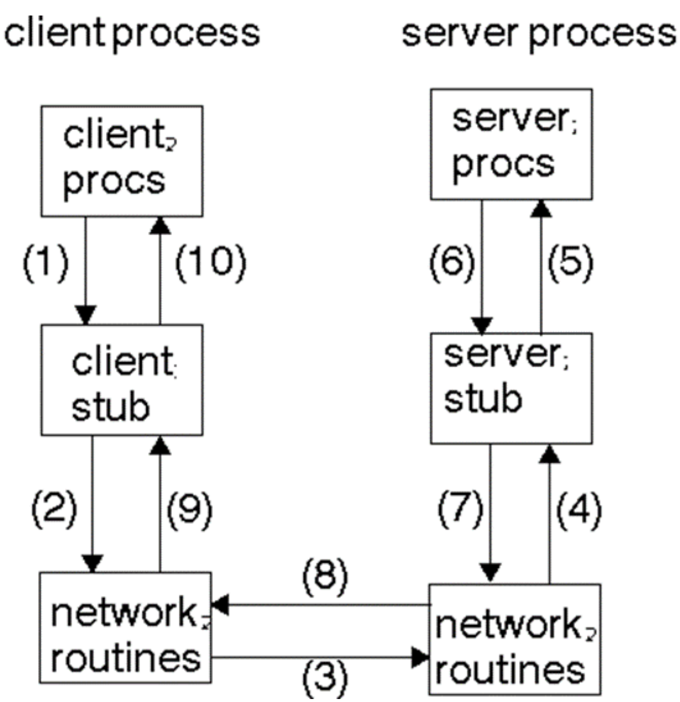
Date Server

```
public static void main(String[] args)  {
   try {
       ServerSocket sock = new ServerSocket(6013);
       // now listen for connections
       while (true) {
         Socket client = sock.accept();
         // we have a connection
         PrintWriter pout = new PrintWriter(client.getOutputStream(),
   true);
         // write the Date to the socket
         pout.println(new java.util.Date().toString());
         // close socket and resume listening for more connections
        client.close();
        }
   } …
```

Date Client

```
public static void main(String[] args)  {
   try {
       // this could be changed to an IP name or address
   other than the localhost
   Socket sock = new Socket("127.0.0.1",6013);
   InputStream in = sock.getInputStream();
   BufferedReader bin = new BufferedReader(new
   InputStreamReader(in));
   // read data from the socket
   String line;
   while( (line = bin.readLine()) != null)
   System.out.println(line);
   sock.close();
   }
```

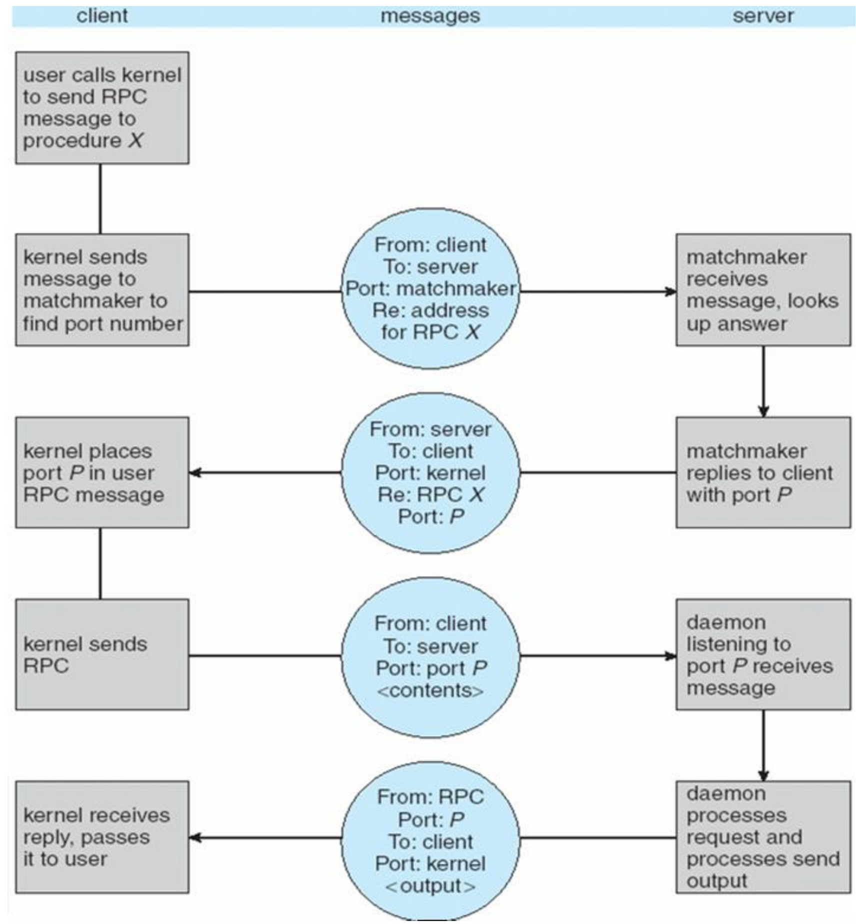
Remote Procedure Calls



- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.
- **Stubs** – client-side proxy for the actual procedure on the server.
- The client-side stub locates the server and *marshalls* the parameters.
  - Data representation handled via **External Data Representation** (**XDR**) format
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server.
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language** (**MIDL**)

Execution of RPC

- Remote communication has more failure scenarios than local
  - Messages can be delivered ***exactly once*** rather than ***at most once***
- OS typically provides a rendezvous (or **matchmaker**) service to connect client and server



Summary

- Process is a program in execution that can be in a number of states
  - New, running, waiting, ready, terminated
- Process creation and termination
- Inter-process communications
  - Shared memory, and message passing
- Client-server communication
  - Socket, RPC, …