Operating System

Chapter 3: Processes

Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

Objectives

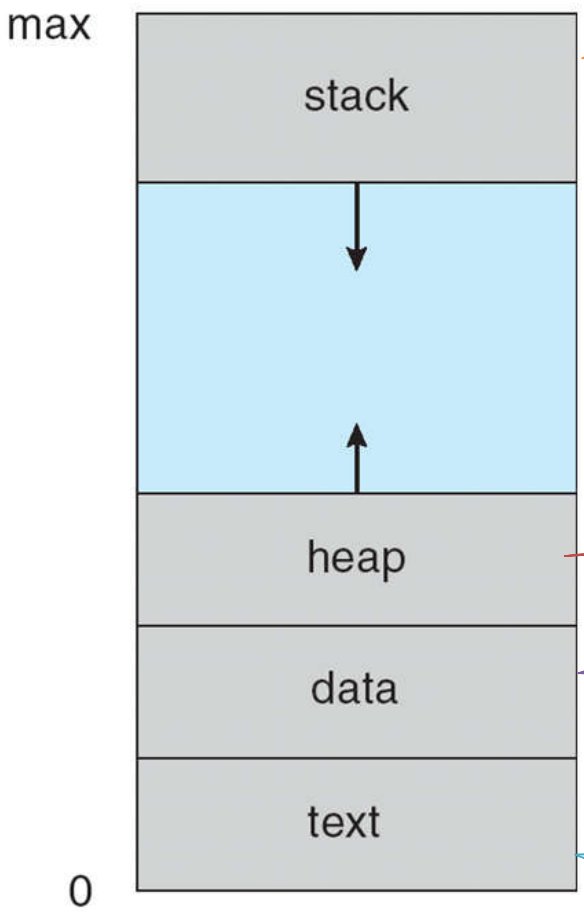
- To introduce the notion of a process
- To describe the various features of processes
- To explore inter-process communication

Process Concept

Process 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



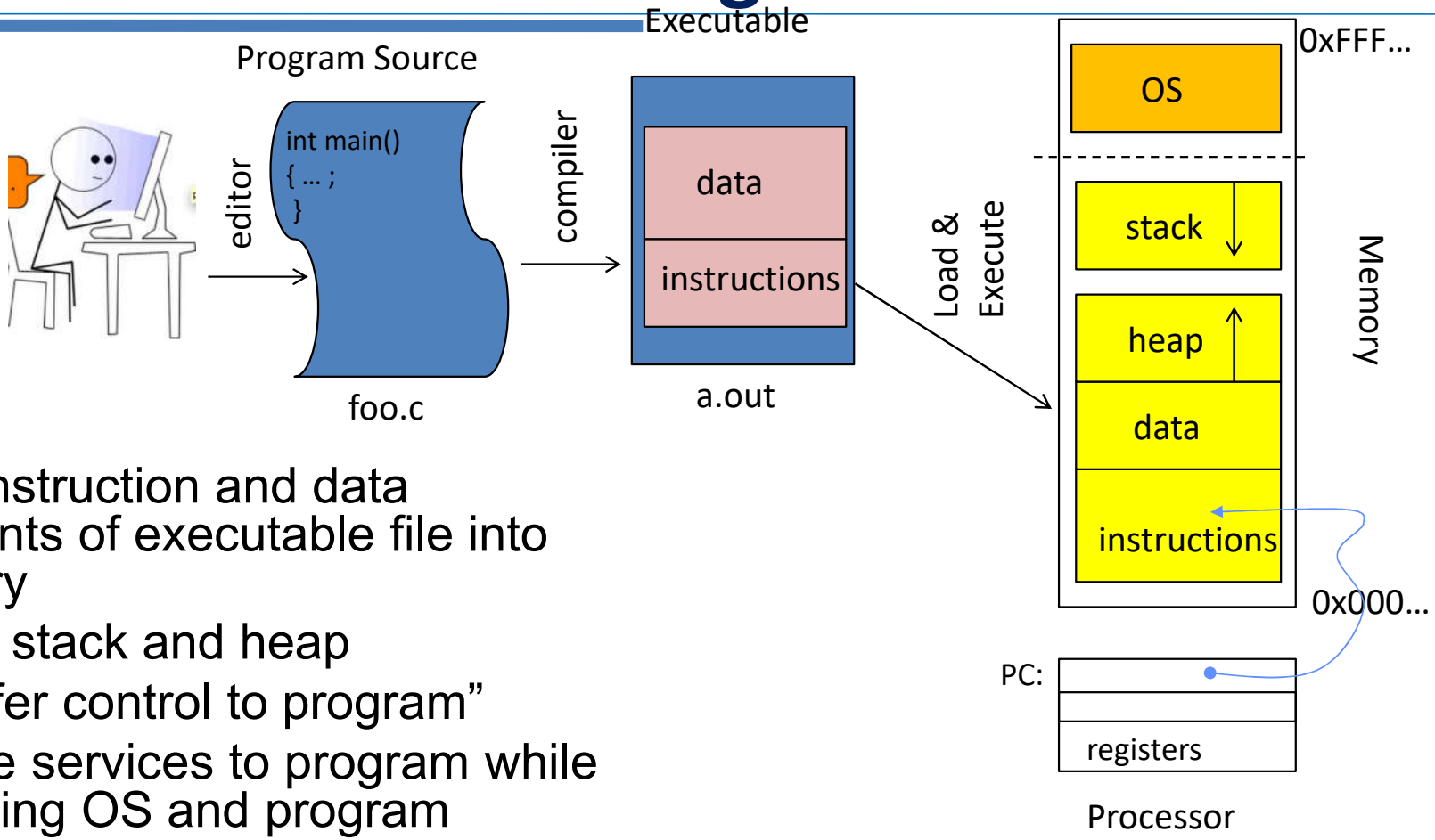
contains temporary data, like Function parameters, return addresses, local variables

memory dynamically allocated during run time

contains global variables

The program code, current activity including **program counter**, processor registers

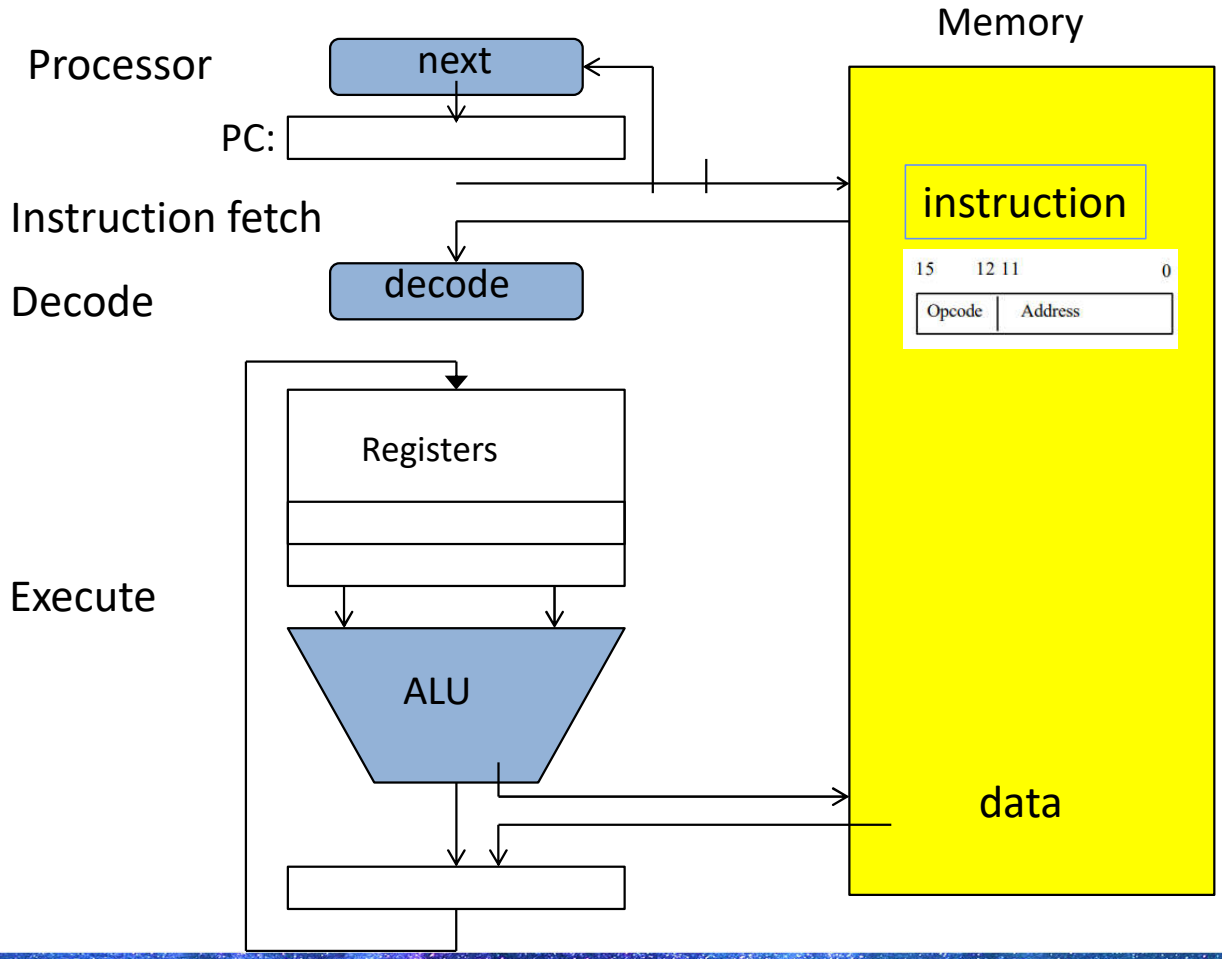
OS Bottom Line: Run Programs



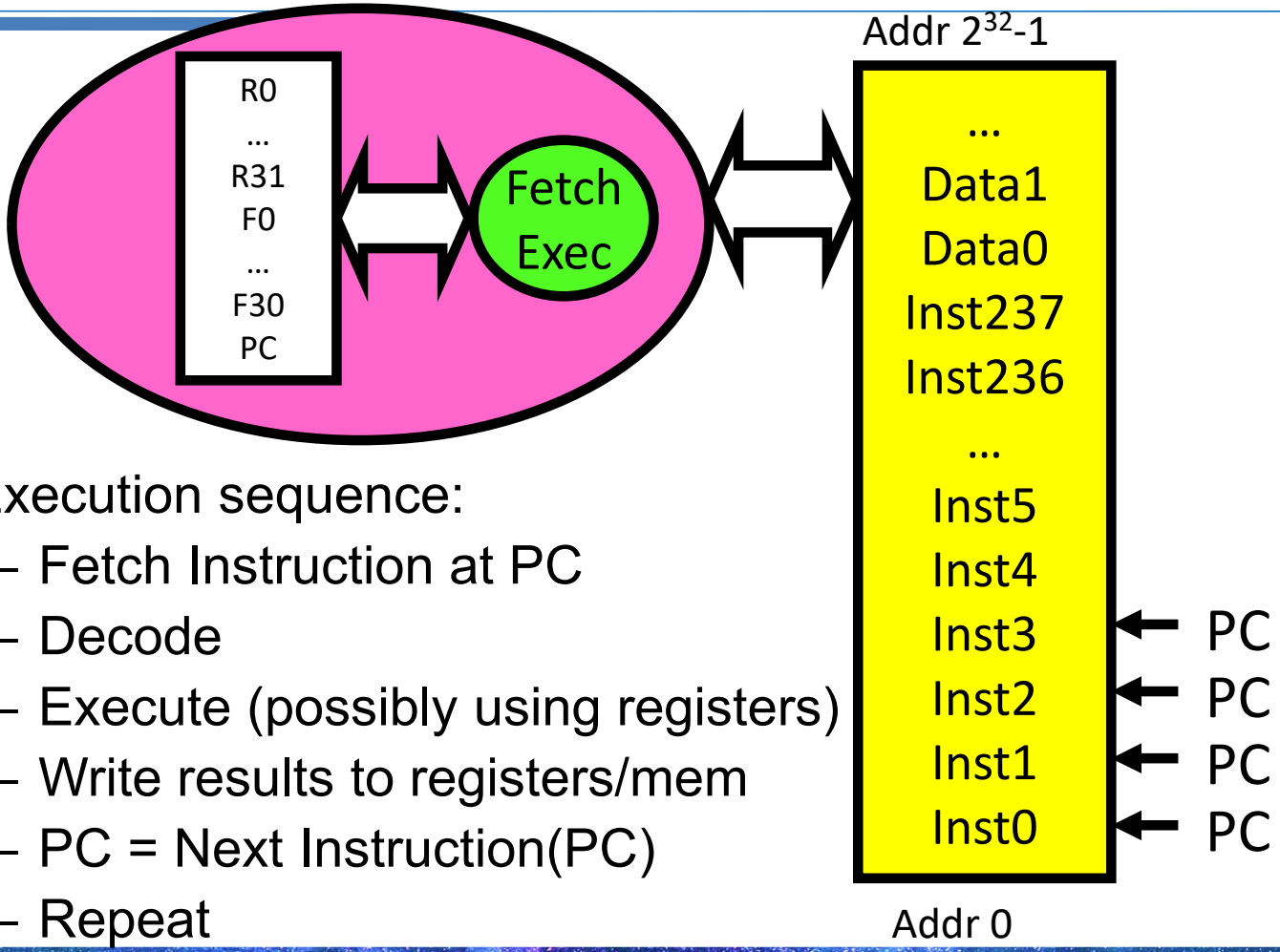
- Load instruction and data segments of executable file into memory
- Create stack and heap
- “Transfer control to program”
- Provide services to program while protecting OS and program

Instruction Fetch/Decode/Execute

The instruction cycle



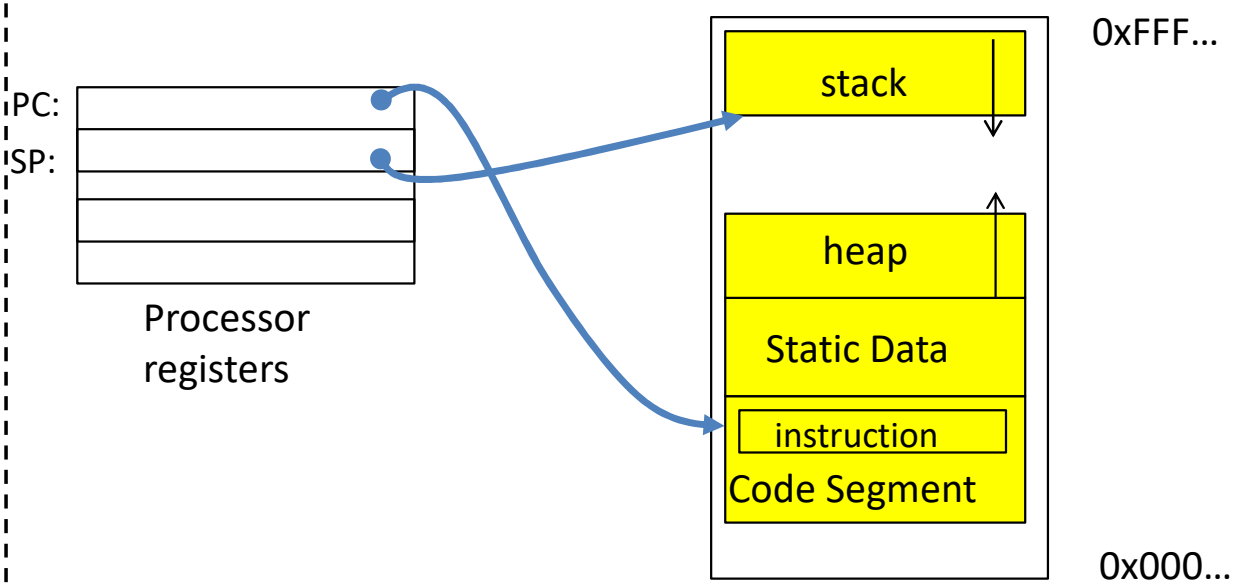
What happens during program execution?



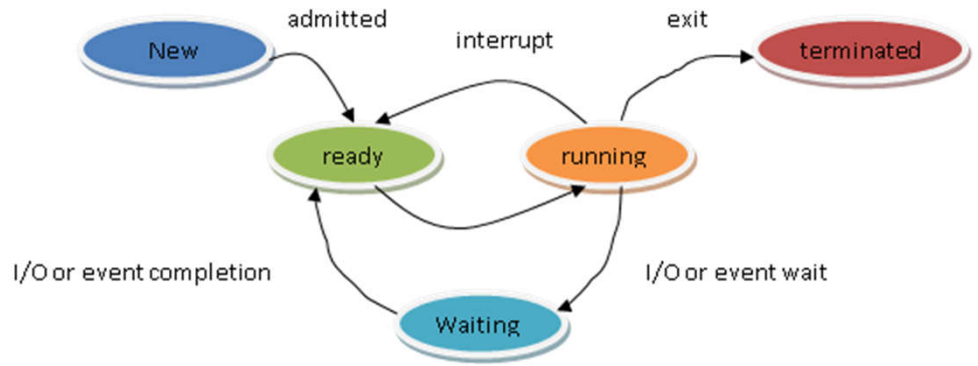
- Execution sequence:
  - Fetch Instruction at PC
  - Decode
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

Address Space: In a Picture

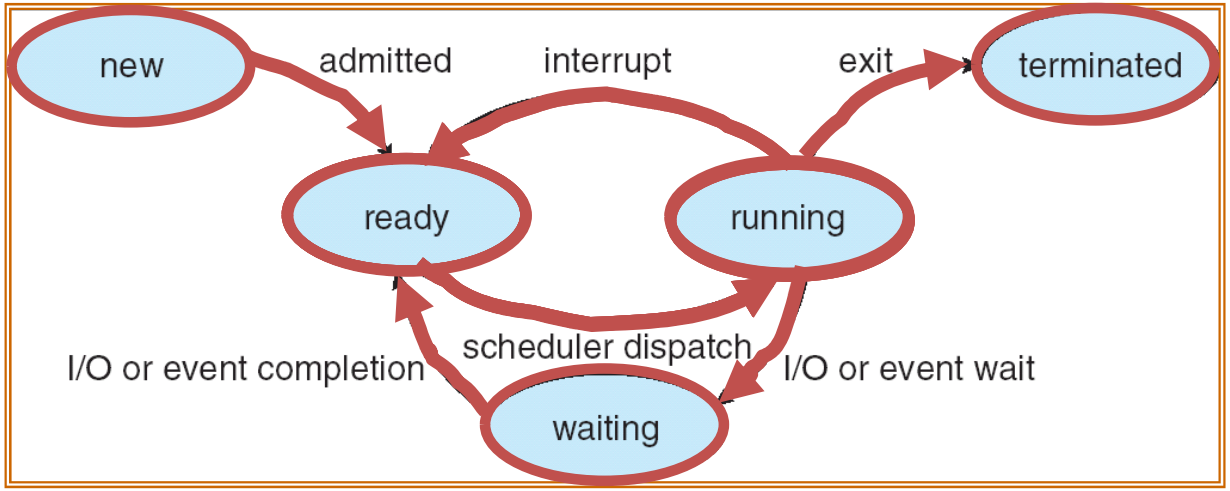
- What's in the code segment? Static data segment?
- What's in the Stack Segment?
  - How is it allocated? How big is it?
- What's in the Heap Segment?
  - How is it allocated? How big?



Process State



Lifecycle of a Process

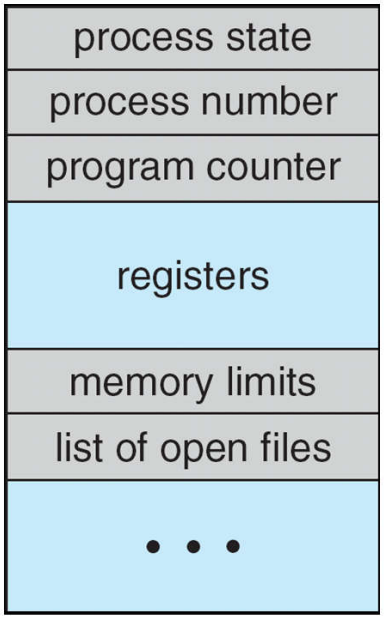


- As a process executes, it changes state:
  - new: The process is being created
  - ready: The process is waiting to run
  - running: Instructions are being executed
  - waiting: Process waiting for some event to occur
  - terminated: The process has finished execution

Process Control Block (PCB)

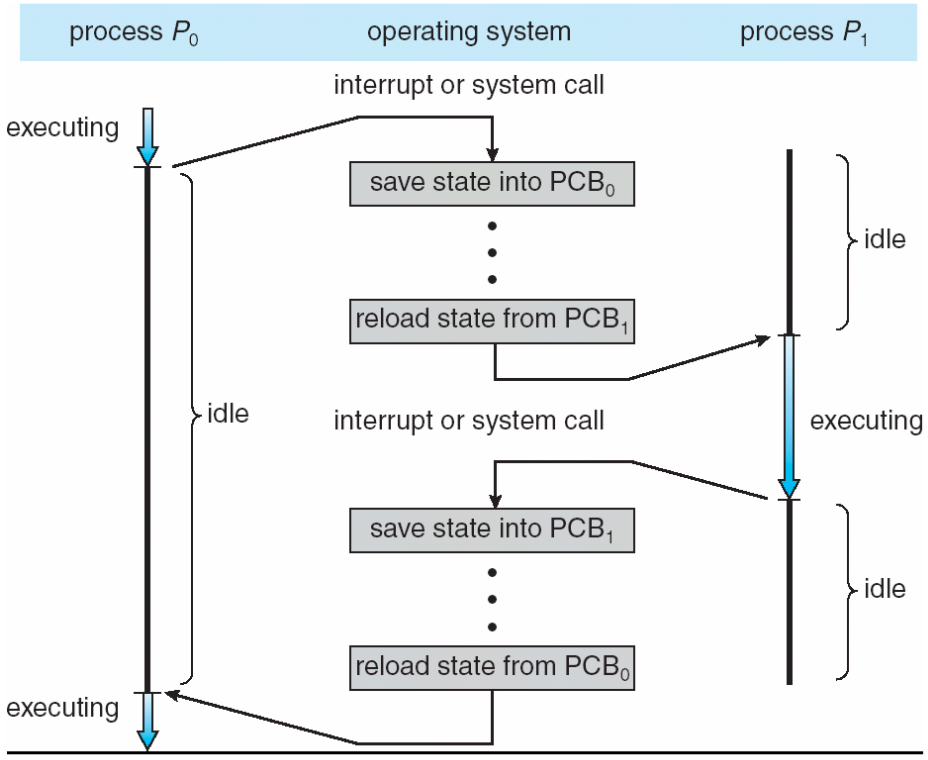
- How to manage processes?
- Information associated with each process

  (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



Context Switch

- Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process.



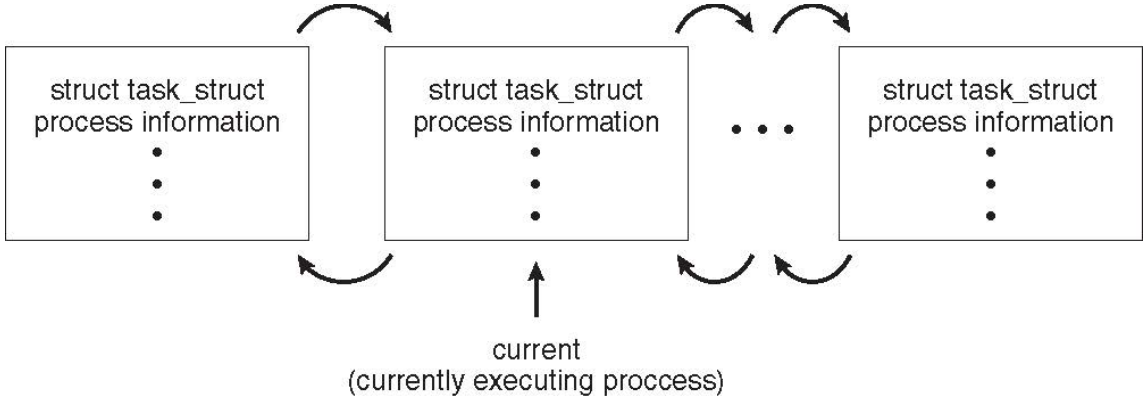
CPU Switch From Process to Process

- The context is represented in the PCB of the process
- Context-switch time is pure overhead

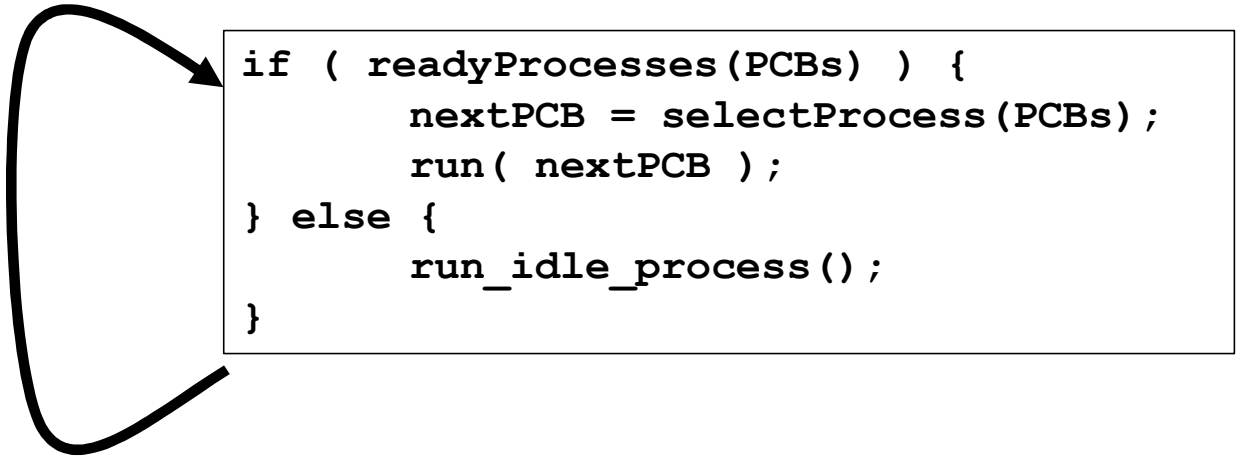
Process Representation in Linux

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 */
```



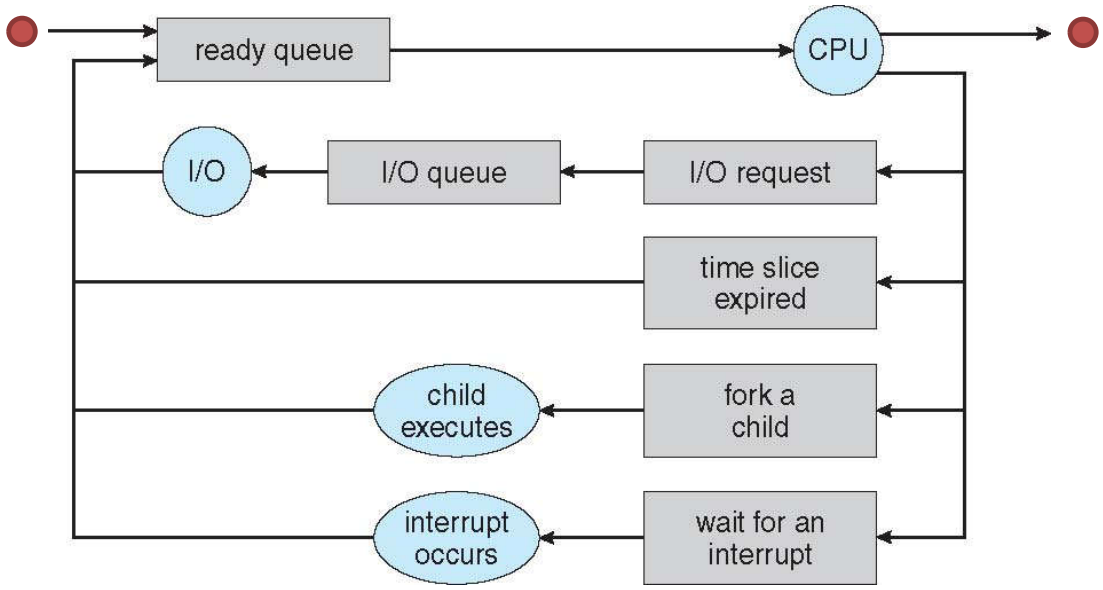
Scheduler



- Scheduling: Mechanism for deciding which processes receive the CPU
- Lots of different scheduling policies provide …
  - Fairness or
  - Realtime guarantees or
  - Latency optimization or ..

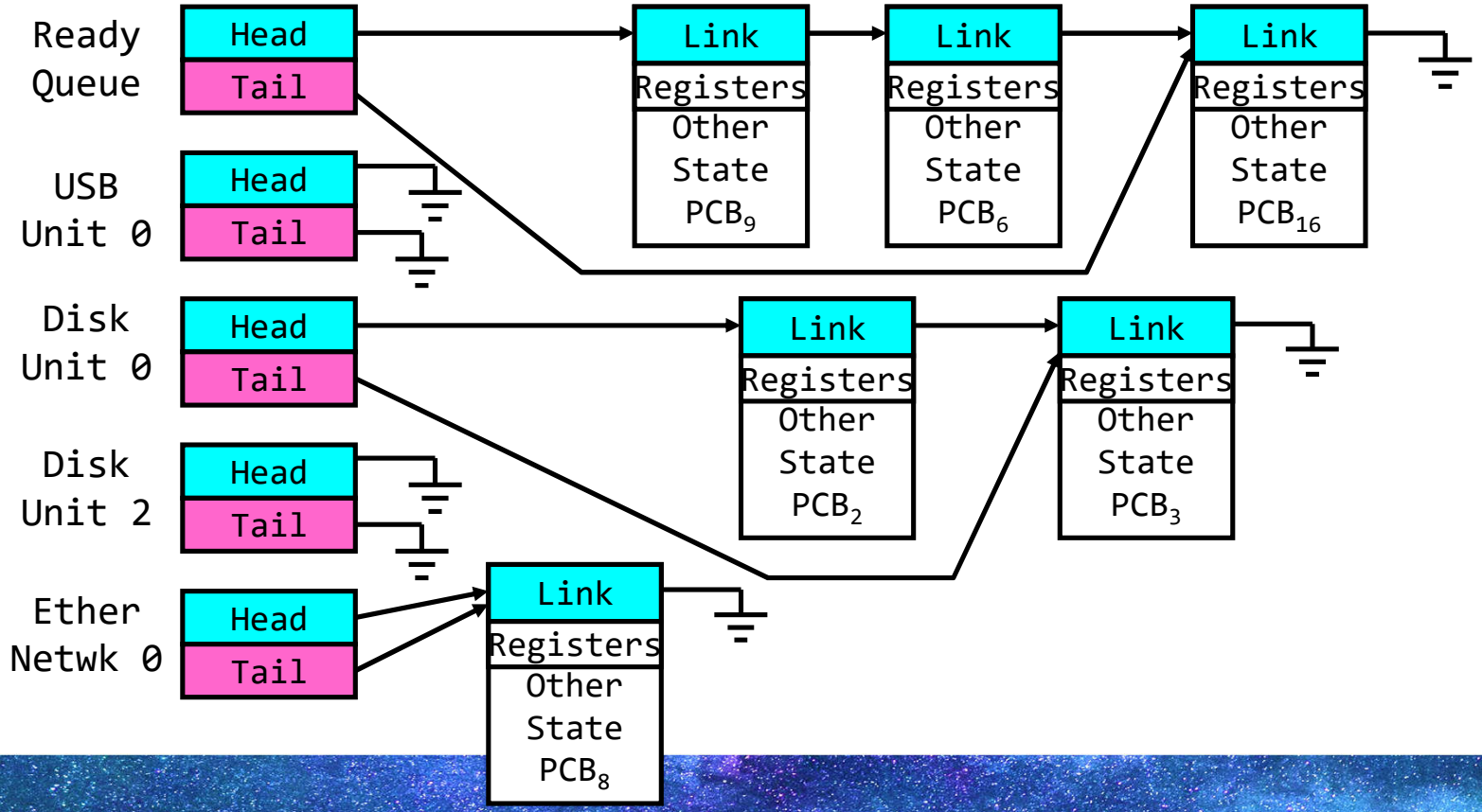
Diagram representation of process scheduling

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



Ready Queue And Various I/O Device Queues

- Process not running ⇒ PCB is in some scheduler queue
  - Separate queue for each device/signal/condition
  - Each queue can have a different scheduler policy

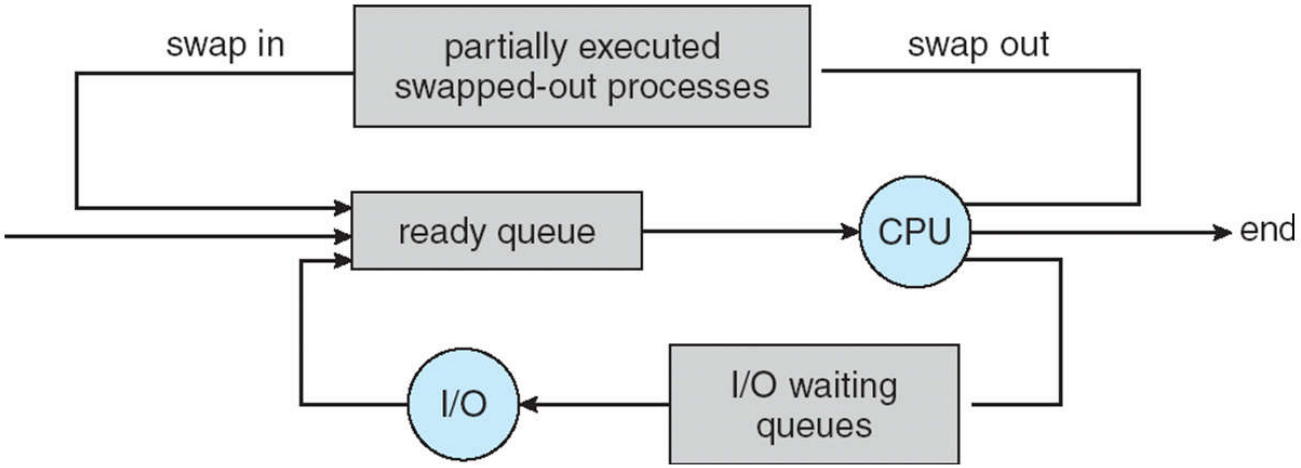


Schedulers

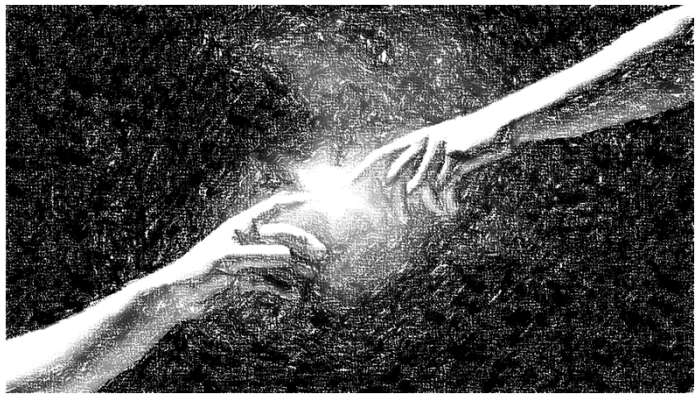
- **Short-term scheduler** (or **CPU scheduler**)
  - selects which process should be executed next and allocates CPU
    - Sometimes the only scheduler in a system
    - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**)
  - selects which processes should be brought into the ready queue
    - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (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**



Process Creation



Can a process create a process ?

- Yes! Unique identity of process is the "process ID" (or PID)
- **fork()** system call creates a *copy* of current process with a new PID
- Return value from **fork()**: integer
  - When > 0:
    - Running in (original) Parent process
    - return value is pid of new child
  - When = 0:
    - Running in new Child process
  - When < 0:
    - Error! Must handle somehow
    - Running in original process

Process creation

- Parent vs. Child
- Generally, process identified and managed via a **process identifier** (**pid**)

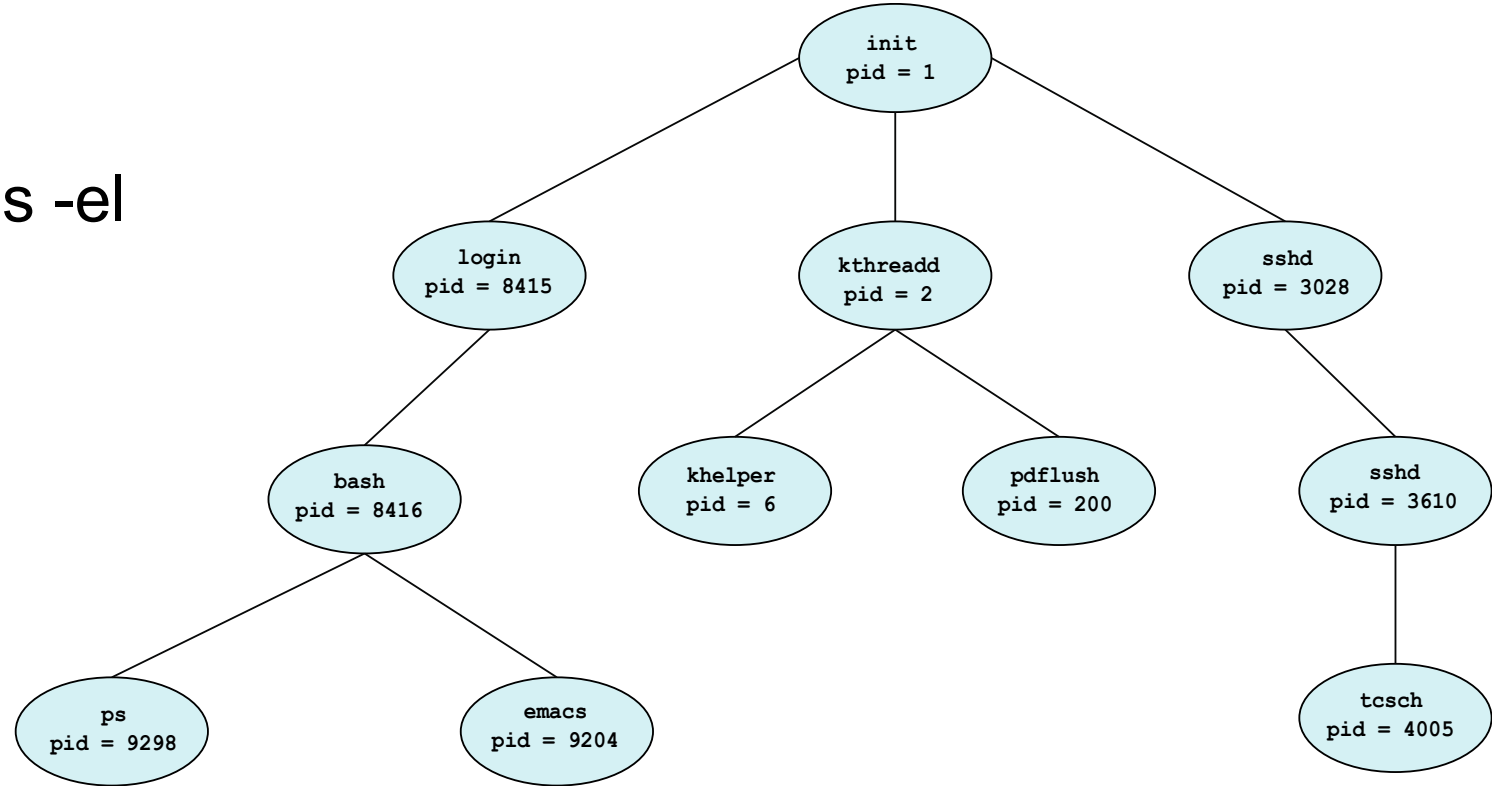
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources

- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

State of original process duplicated in *both* Parent and Child! Memory, File Descriptors (next topic), etc…

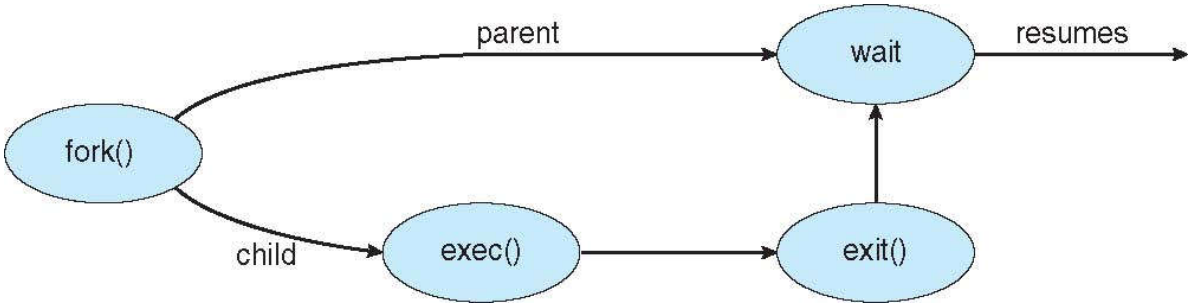
A Tree of Processes in Linux

ps -el



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



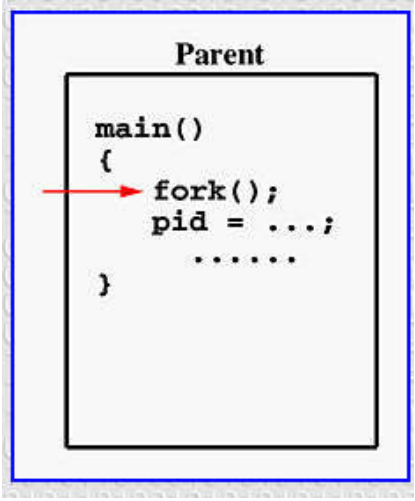
Process creation with C

```
#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;
}
```

POSIX

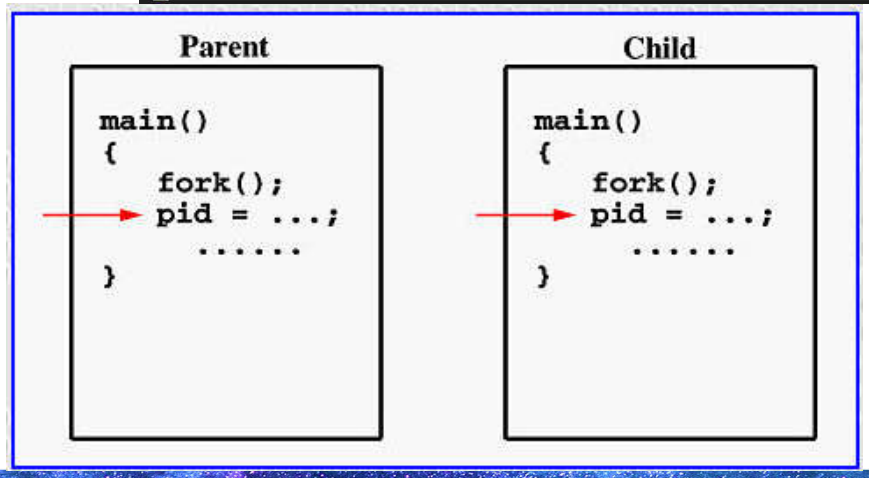
Why pid has two values when fork() return ?

EX 1



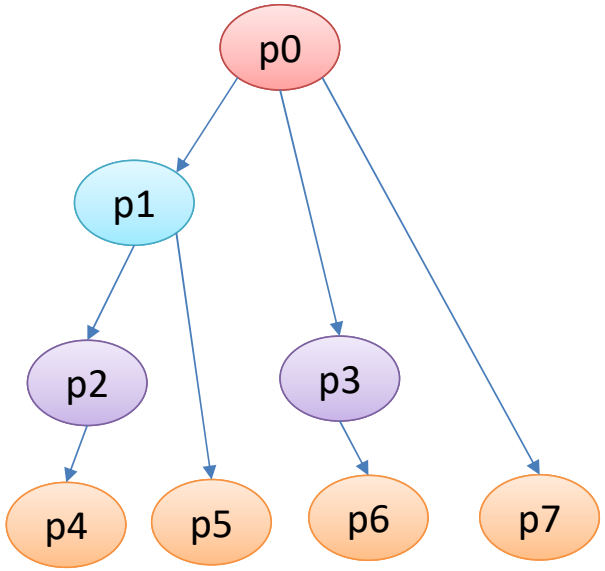

```c
#include <stdio.h>
#include <string.h>
#include <sys/types.h>

#define MAX_COUNT 200
#define BUF_SIZE 50

void main(void){

    pid_t pid;
    int i;
    char buf[BUF_SIZE];

    fork();
    pid = getpid();
    for(i=1;i<=MAX_COUNT;i++){
        sprintf(buf, "This line is from pid %d, value = %d\n", pid, i);
        write(1, buf, strlen(buf));
    }
}
```

```
This line is from pid 95978, value = 1
This line is from pid 95978, value = 2
This line is from pid 95978, value = 3
This line is from pid 95979, value = 1
This line is from pid 95978, value = 4
This line is from pid 95979, value = 2
This line is from pid 95978, value = 5
This line is from pid 95979, value = 3
This line is from pid 95978, value = 6
This line is from pid 95979, value = 4
This line is from pid 95978, value = 7
This line is from pid 95979, value = 5
```



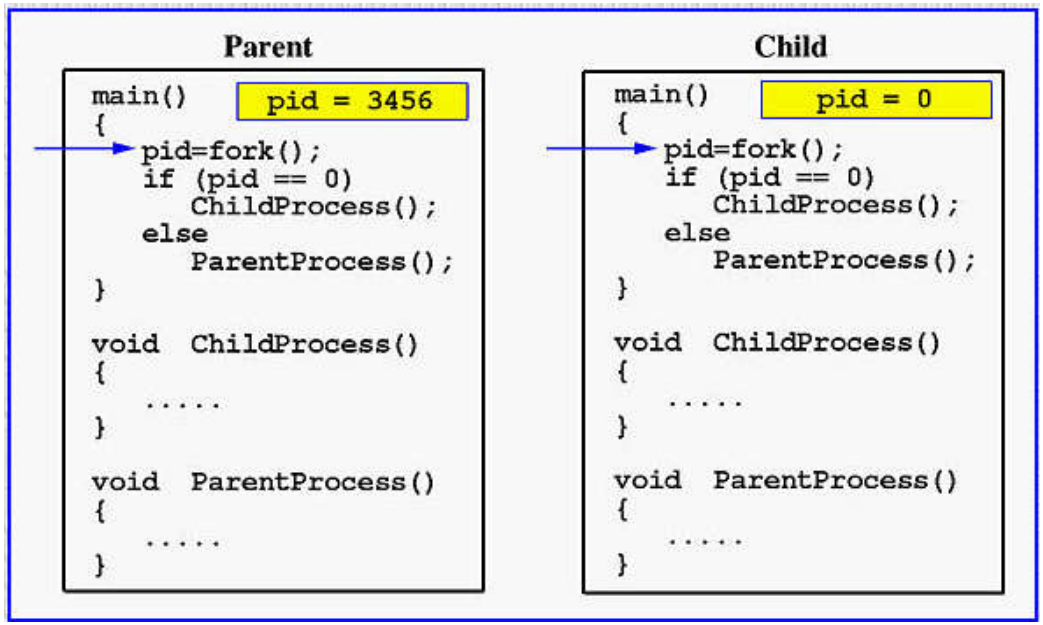
Question ?

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

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

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

    /* and fork another */
    fork();

    return 0;
}
```



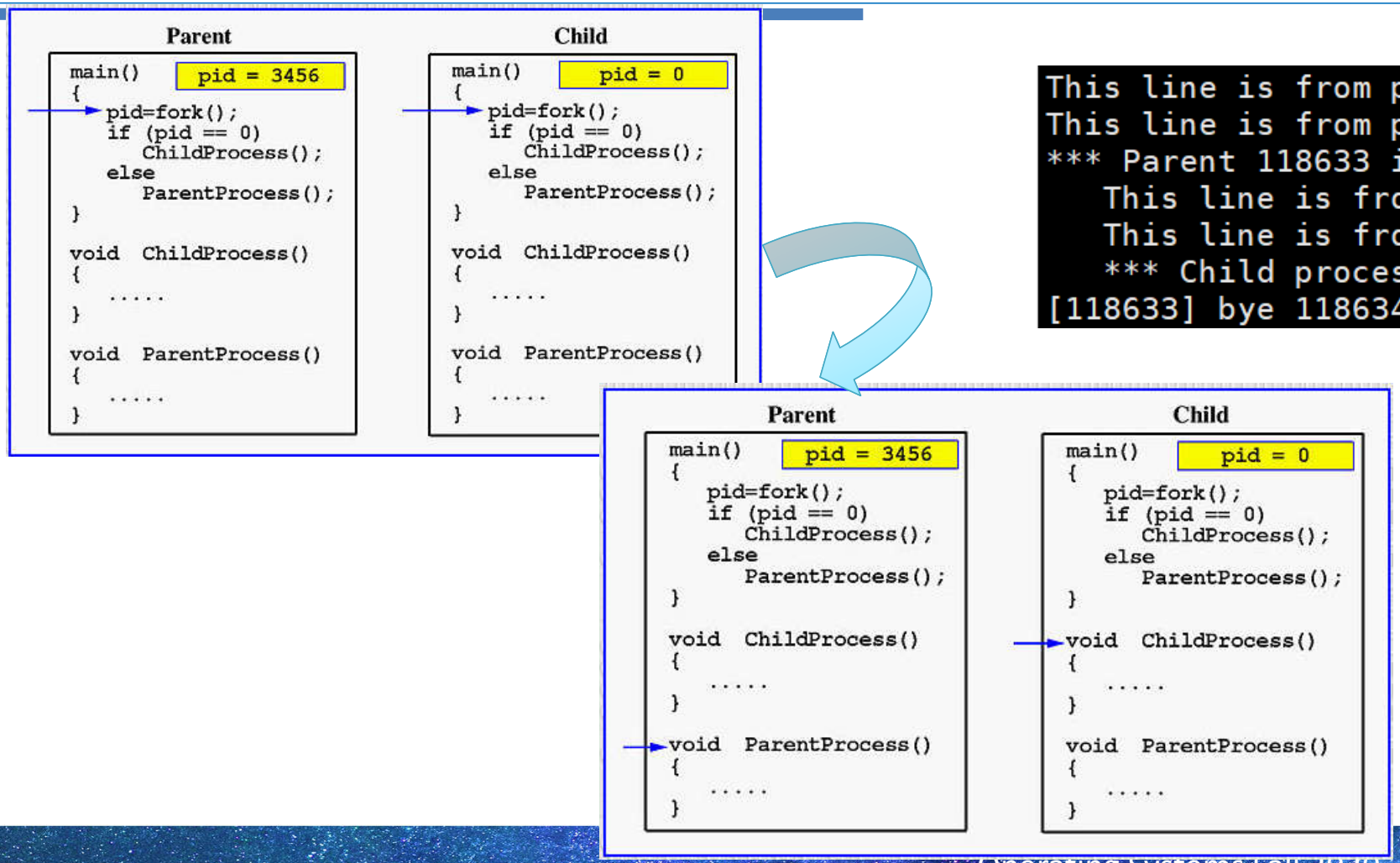
How many processes are created?

Ex 2




```c
void  main(void)
{
     pid_t  pid;
     pid = fork();
     if (pid == 0)
          ChildProcess();
     else
          ParentProcess();
}
void  ChildProcess(void)
{
     int   i;
     int   mypid;
     mypid = getpid();
     for (i = 1; i <= MAX_COUNT; i++)
          printf("   This line is from child, value = %d\n", i);
     printf("   *** Child process %d is done ***\n",mypid);
     exit(0);
}
void  ParentProcess(void)
{
     int   i,status;
     int got_pid,mypid;
     mypid = getpid();
     for (i = 1; i <= MAX_COUNT; i++)
          printf("This line is from parent, value = %d\n", i);
     printf("*** Parent %d is done ***\n",mypid);
     got_pid = wait(&status);
     printf("[%d] bye %d (%d)\n", mypid, got_pid, status);

}
```

EX2 (cont.)




```
This line is from parent, value = 1
This line is from parent, value = 2
*** Parent 118633 is done ***
   This line is from child, value = 1
   This line is from child, value = 2
   *** Child process 118634 is done ***
[118633] bye 118634 (0)
```

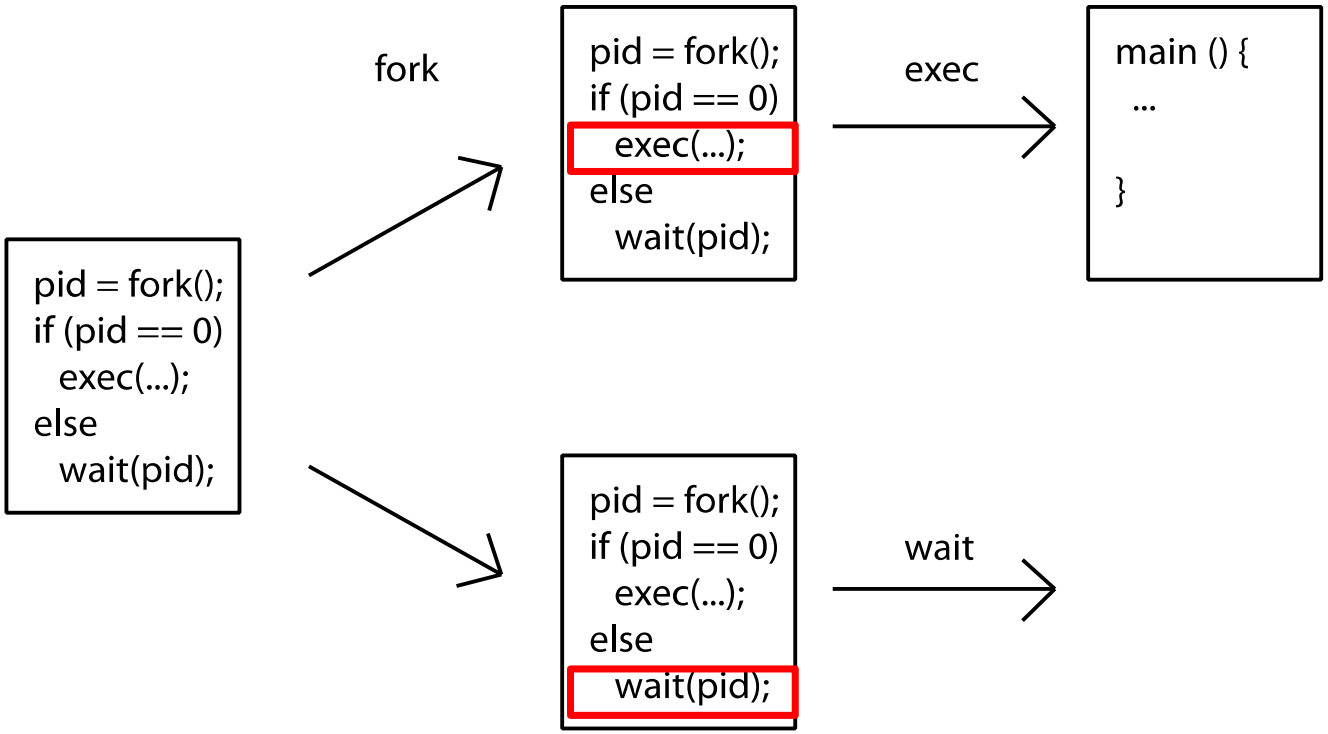
Process termination

- Child → Parent
  - Process' resources are deallocated when:
    - *exit(n)*
    - *return()* in *main()*
  - Catch exit status → *wait()*
    - `pid = wait(&status);`
- Parent → Child
  - *terminate child*
  - 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

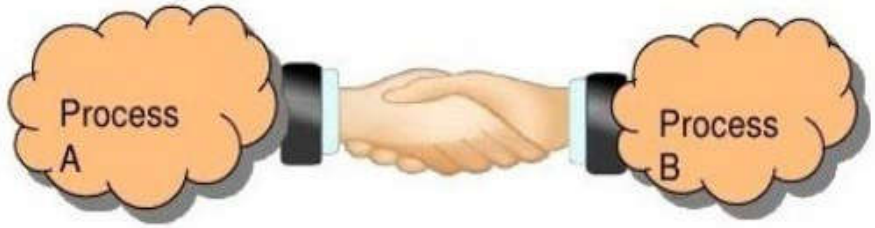
Problems of process termination

- **zombie process**
  - **No parent waiting**
- **orphan process**
  - **Parent termination without wait**
- Multi process example: Chrome Browser
  - Browser, Renderer, Plugins, etc

UNIX Process Management



Interprocess communication (IPC)



Interprocess communication (*IPC*)

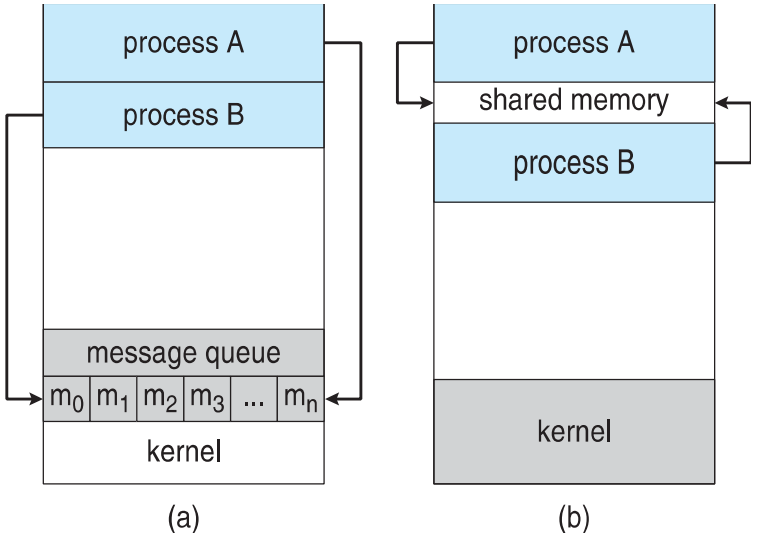
Process:

independent vs. cooperating

- Cooperating process:
  - Shared memory
  - Message passing

Reasons for process cooperation:

- Information sharing
- Computation speedup
- Modularity
- convenience



(a) Message passing (b) Shared memory.

Circular buffer & producer-consumer problem

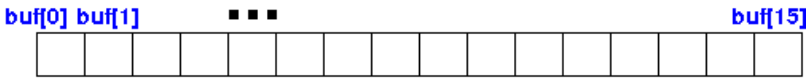
```
#define BUFFER_SIZE 16
typedef struct {
   . . .
} item;

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;
}
```

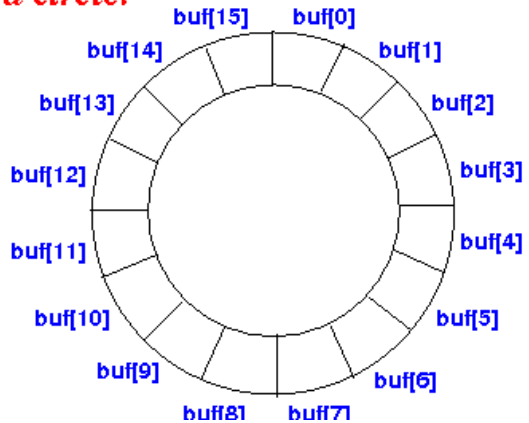
```
item next_consumed;
while (true) {
        while (in == out) ; /* do
nothing */

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

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

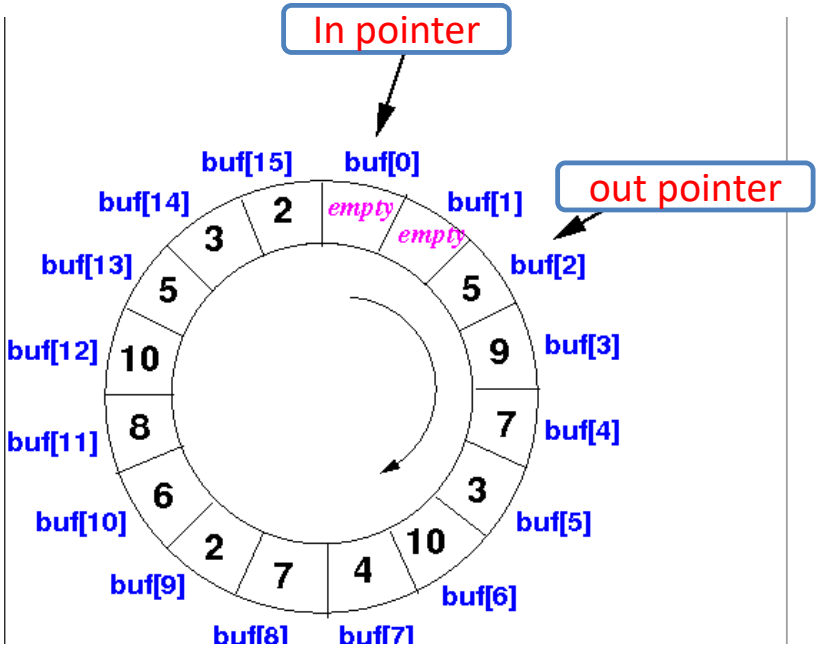
circular array

Array:



Pretend array is a circle:





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

Producer

```c
#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;
}
```

Consumer

```c
#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;
}
```

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- IPC facility provides two operations:
  - **`send`**(*message*) – message size fixed or variable
  - **`receive`**(*message*)
- If P and Q wish to communicate, they need to:
  - establish a ***communication link*** between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., direct or indirect, synchronous or asynchronous, automatic or explicit buffering)

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - 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

- 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

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 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

- 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**

Synchronization (Cont.)

Producer-consumer becomes trivial

```
message next_produced;
while (true) {
   /* produce an item in next produced */
   send(next_produced);
}
```

```
message next_consumed;
while (true) {
   receive(next_consumed);

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

Buffering

- Queue of messages attached to the link.
- implemented in one of three ways
  1. Zero capacity – no 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

Examples of IPC Systems - Mach

- Mach communication is message based
  - Even system calls are messages
  - Each task gets two mailboxes at creation- Kernel and Notify
  - Only three system calls needed for message transfer
    `msg_send(), msg_receive(), msg_rpc()`
  - Mailboxes needed for commuication, created via
    `port_allocate()`
  - Send and receive are flexible, for example four options if mailbox full:
    - Wait indefinitely
    - Wait at most n milliseconds
    - Return immediately
    - Temporarily cache a message

Pipes

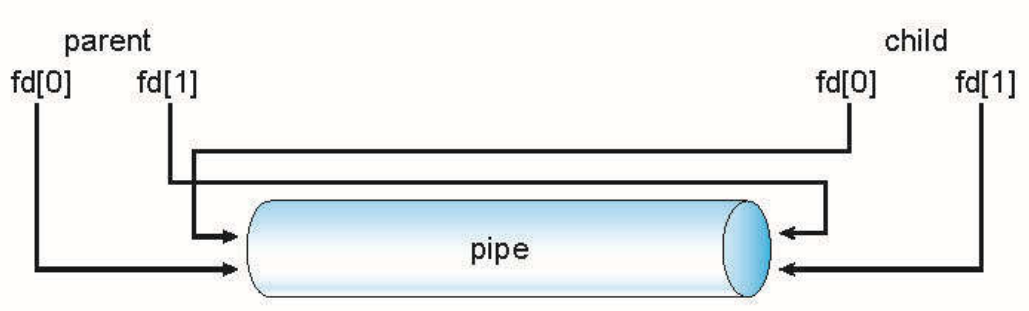
- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the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

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;
}
```

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

- **Creating a FIFO file:**
  - In order to create a FIFO file, a function calls i.e. mkfifo is used.

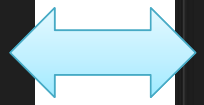
```
int mkfifo(const char *pathname, mode_t mode);
```

mkfifo() makes a FIFO special file with name ***pathname***. Here ***mode*** specifies the FIFO's permissions.

Using FIFO: As named pipe(FIFO) is a kind of file, we can use all the system calls associated with it i.e. *open*, *read*, *write*, *close*.

EX： Named pipe

```c
int main()
{
    int fd;
    char * myfifo = "/tmp/myfifo";  // FIFO file path
    // Creating the named file(FIFO)
    // mkfifo(<pathname>, <permission>)
    mkfifo(myfifo, 0666);

    char arr1[80], arr2[80];
    while (1)
    {
        // Open FIFO for write only
        fd = open(myfifo, O_WRONLY);
        // Take an input arr2ing from user.
        // 80 is maximum length
        fgets(arr2, 80, stdin);
        // Write the input arr2ing on FIFO
        // and close it
        write(fd, arr2, strlen(arr2)+1);
        close(fd);
        // Open FIFO for Read only
        fd = open(myfifo, O_RDONLY);
        // Read from FIFO
        read(fd, arr1, sizeof(arr1));
        // Print the read message
        printf("User2: %s\n", arr1);
        close(fd);
    }
    return 0;
}
```



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

int main()
{
    int fd1;
    char * myfifo = "/tmp/myfifo"; // FIFO file path
    // Creating the named file(FIFO)
    // mkfifo(<pathname>,<permission>)
    mkfifo(myfifo, 0666);
    char str1[80], str2[80];
    while (1)
    {
        // First open in read only and read
        fd1 = open(myfifo,O_RDONLY);
        read(fd1, str1, 80);
        // Print the read string and close
        printf("User1: %s\n", str1);
        close(fd1);
        // Now open in write mode and write
        // string taken from user.
        fd1 = open(myfifo,O_WRONLY);
        fgets(str2, 80, stdin);
        write(fd1, str2, strlen(str2)+1);
        close(fd1);
    }
    return 0;
}
```

Communications in client-server systems

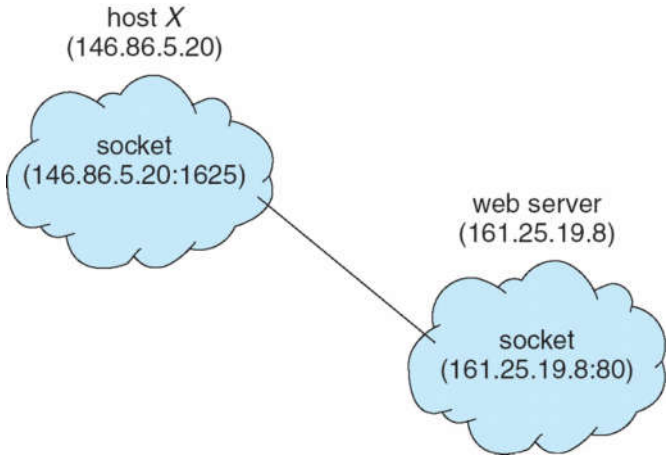
- Sockets
- Remote Procedure Calls (windows)
- Pipes
- Remote Method Invocation (Java)

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

- 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 DateClient
{
   public static void main(String[] args) {
      try {
         /* make connection to server socket */
         Socket sock = new Socket("127.0.0.1",6013);

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

         /* read the date from the socket */
         String line;
         while ( (line = bin.readLine()) != null)
            System.out.println(line);

         /* close the socket connection*/
         sock.close();
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

```
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);
      }
   }
}
```

End of Chapter 3