Bilkent University
Department of Computer Engineering
CS342 Operating Systems

Processes

Last Update: Feb 21, 2023

Outline

- Process Concept
- Process Scheduling
- Operations on Processes
  - Creating processes
- Inter-process Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

Process Concept and Process Management

Process Concept

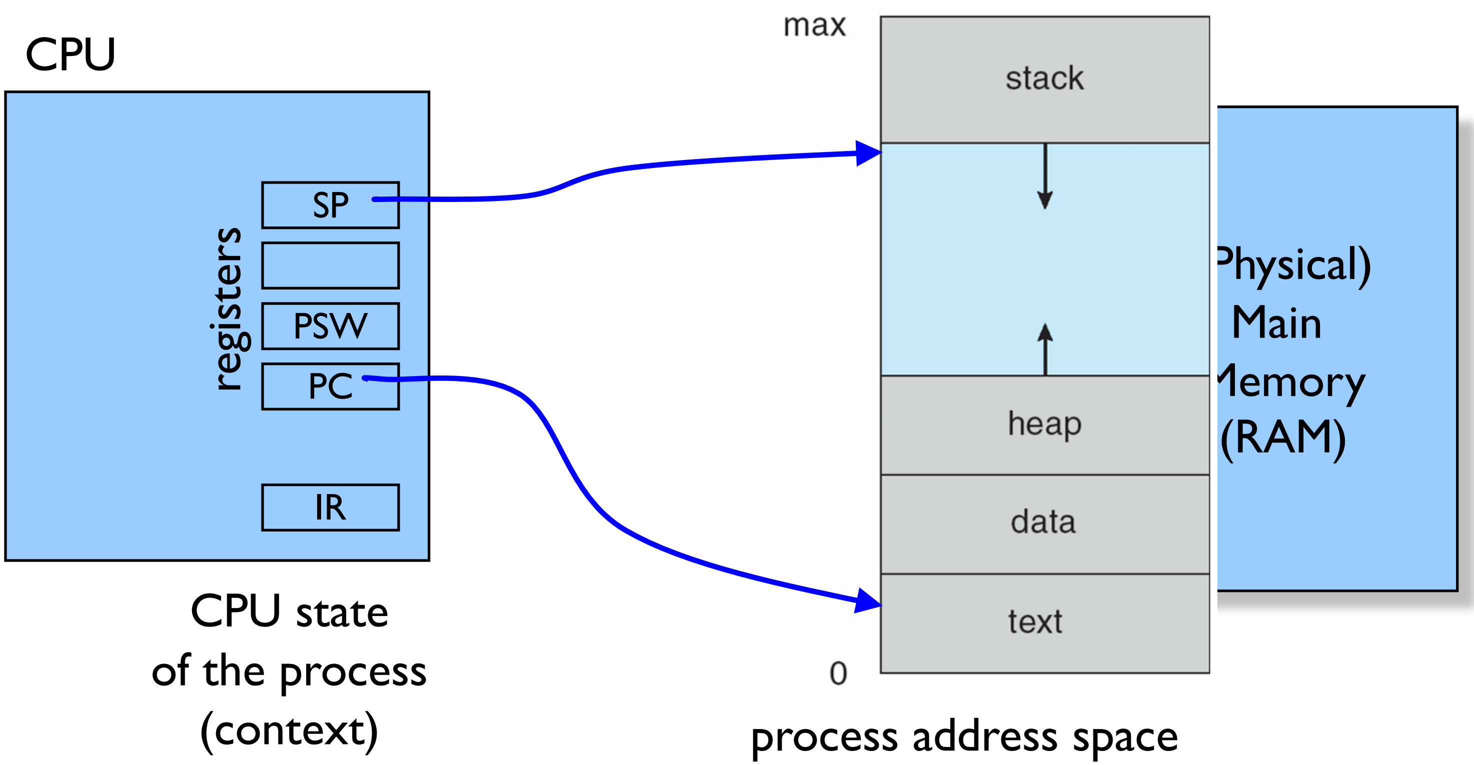
Process: a program in execution; process execution must progress in sequential fashion

- A process includes:
    - text (code) section
      (program counter PC points to next instruction to execute)
    - stack section (stack pointer points to the top of the stack)
    - data section
    - set of open files currently used
    - set of I/O devices currently used
- An operating system executes a variety of programs:
    - Batch systems: jobs
    - Time-shared systems: user programs or tasks
  - We will use the terms *job* and *process* almost interchangeably.

Process: program in execution

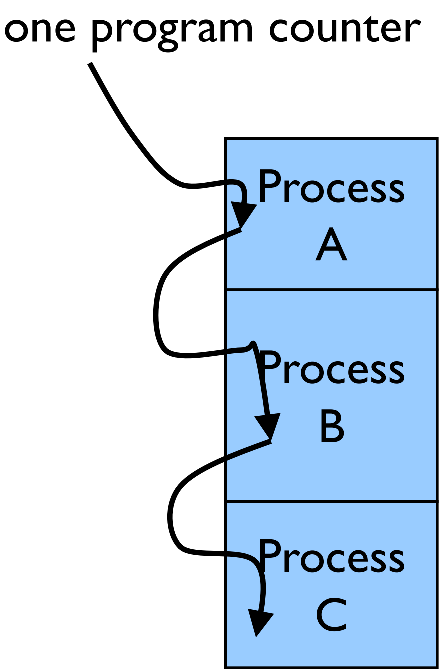
- If we would have a single program running in the system, then the task of OS would be easy:
  - load the program, start it, and program runs in CPU
  - (from time to time it calls OS to get some service done)
- But if we want to start several processes, then the running program in CPU (current process) has to be stopped for a while (suspended) and other program (process) has to run in CPU.
  - Process management becomes an important issue
- To do process switch, we have to save the state/context of the CPU (register values) which belongs to the suspended program, so that later the suspended program can be re-started (resumed) again as if nothing has happened.
  - Keep track of process progress

Process: program in execution

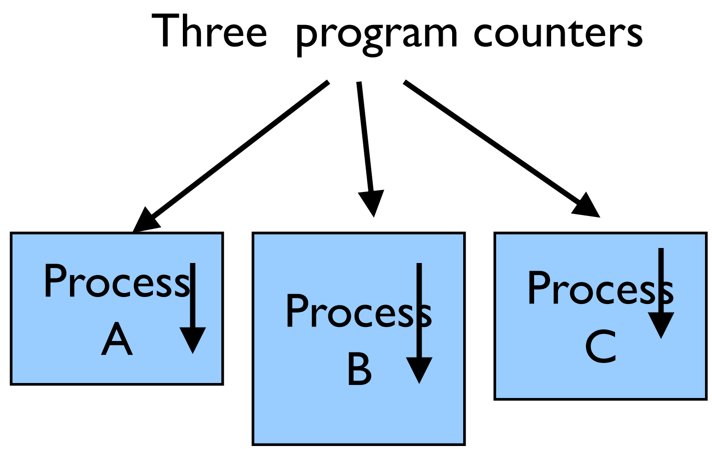


(currently used portion of the address space must be in memory)

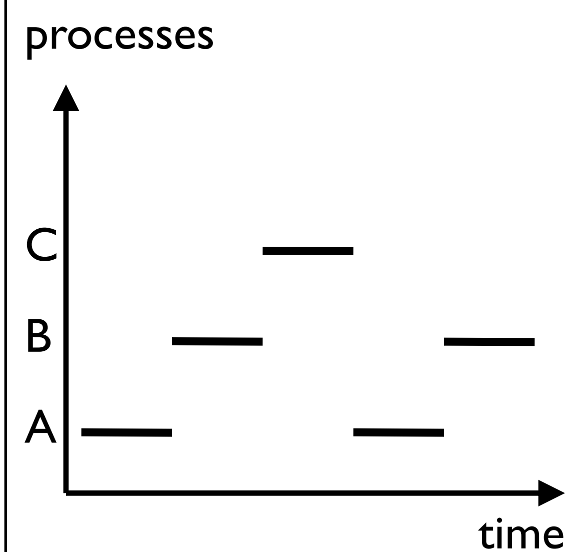
Multiple Processes



what is
happening
physically

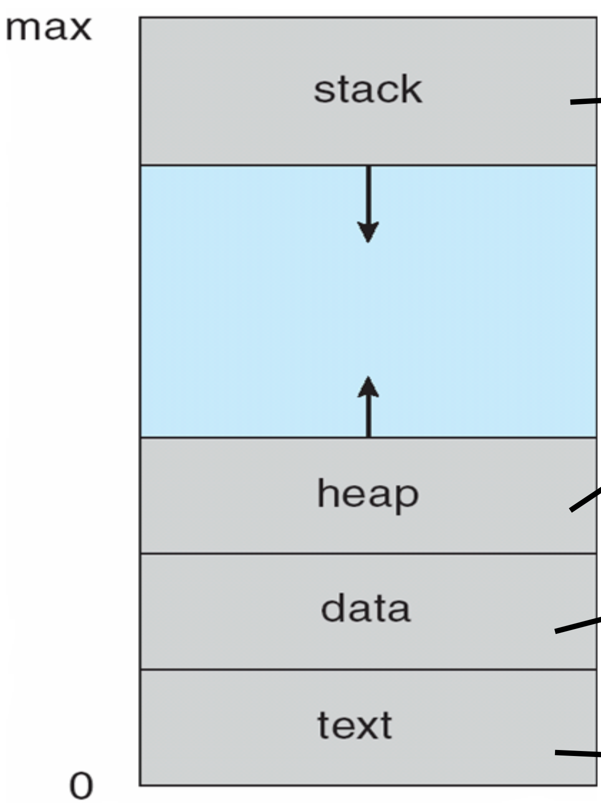


Conceptual model
of three different
processes



one process
executing at
a time

Process in Memory



Stack segment
(holds the called function parameters, local variables, return values)

Storage for dynamically created variables

Data segment
(includes global variables, arrays, etc., you use)

Text segment
(code segment)
(instructions are here)

A process needs this memory content to run
(called address space; memory image)

Process Address Space

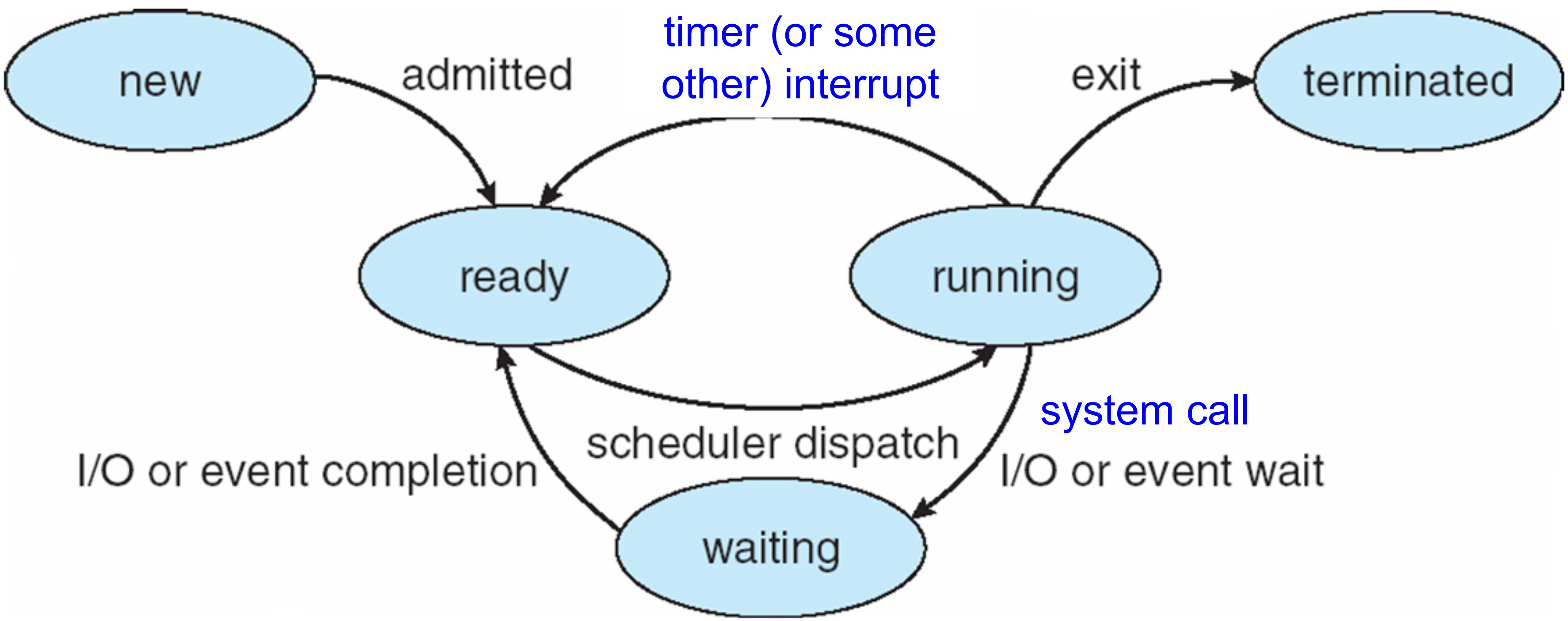
- A process can only access its own address space
- Each process has its own address space
- Kernel can access everything

Process State

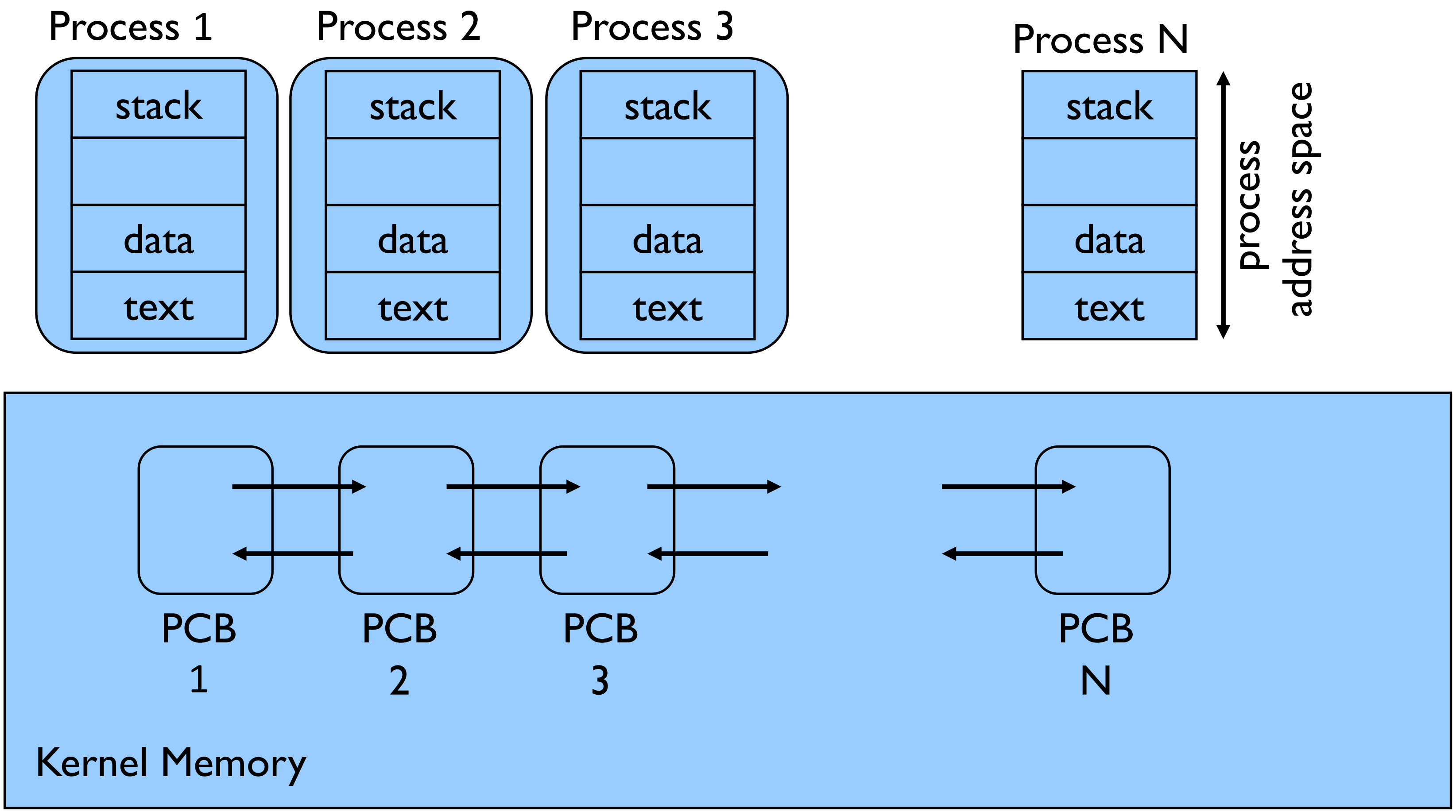
- As a process executes, it changes *state*
  - new: The process is being created
  - running: Instructions are being executed
  - waiting: The process is waiting for some event to occur (i.e., it is blocked, it is sleeping, can not execute)
  - ready: The process is waiting to be assigned to a processor (ready to execute)
  - terminated: The process has finished execution.

  In a single-CPU system, only one process may be in running state; many processes may be in ready or waiting states.

Diagram of Process State



Process Control Block (PCB)



Kernel maintains a PCB for each process. With that kernel keeps track process progress.
PCBs can be linked together in various queues.

Process Control Block

Information associated with each process

- Process state (ready, running, waiting, etc.)
- Program counter (PC)
- CPU registers
- CPU scheduling information
  - Priority of the process, etc.
- Memory-management information
  - text/data/stack section pointers, sizes, etc.
  - pointer to page table, etc.
- Accounting information
  - CPU usage, clock time so far, …
- I/O status information
  - List of I/O devices allocated to the process,
- Process ID (pid)
- Parent process
- Child processes
- File management info
  - Root directory
  - Working directory
  - a list of open files, etc.
  - UID
  - GID
- …

Process Representation in Linux

In Linux kernel source tree, the file include/linux/sched.h contains the definition of the C structure task_struct, which is the PCB for a process.

```
struct task_struct {
        long state;                       /* state of the process */
        ….
        pid_t pid;                        /* identifier of the process */
        …
        unisigned int time_slice;         /* scheduling info */
        …
        struct files_struct *files;       /* info about open files */
        ….
        struct mm_struct *mm;             /* info about the address space of this process */
        …
}
```

Example: Processes in Linux

- Use ps command to see the currently started processes in the system
- Use ps aux to get more detailed information
- See the manual page of the ps to get help about the ps:
  - Type: man ps
- The man command gives info about a command, program, library function, or system call.

- The /proc file system in Linux is the kernel interface to users to look to the kernel state (variables, structures, etc.).
  - Many subfolders
  - One subfolder per process (name of subfolder = pid of process)

Process Queues and Scheduling

Process Scheduling

- In a multiprogramming or time-sharing system, there may be multiple processes ready to execute.
- We need to select one of them and give the CPU to that process (scheduling).
  - There are various criteria that can be used in the scheduling decision.
- Then, the scheduling mechanism (dispatcher) assigns the selected process to the CPU and starts execution of it.

Select (Scheduling Algorithm)	→	Dispatch (mechanism)

Process Behavior

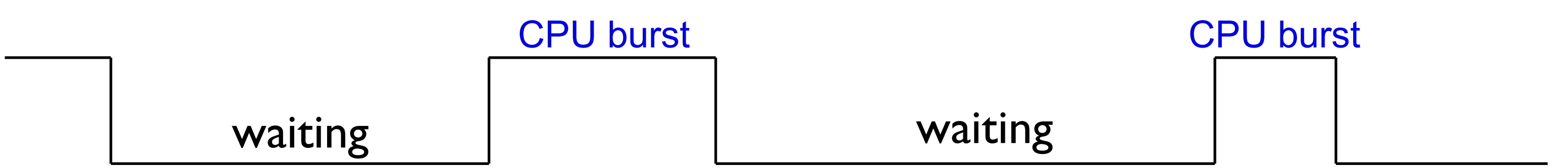
Processes can be described as either:

- 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

CPU burst: the execution of the program in CPU between two I/O requests (i.e., time period during which the process wants to continuously run in the CPU without making I/O)

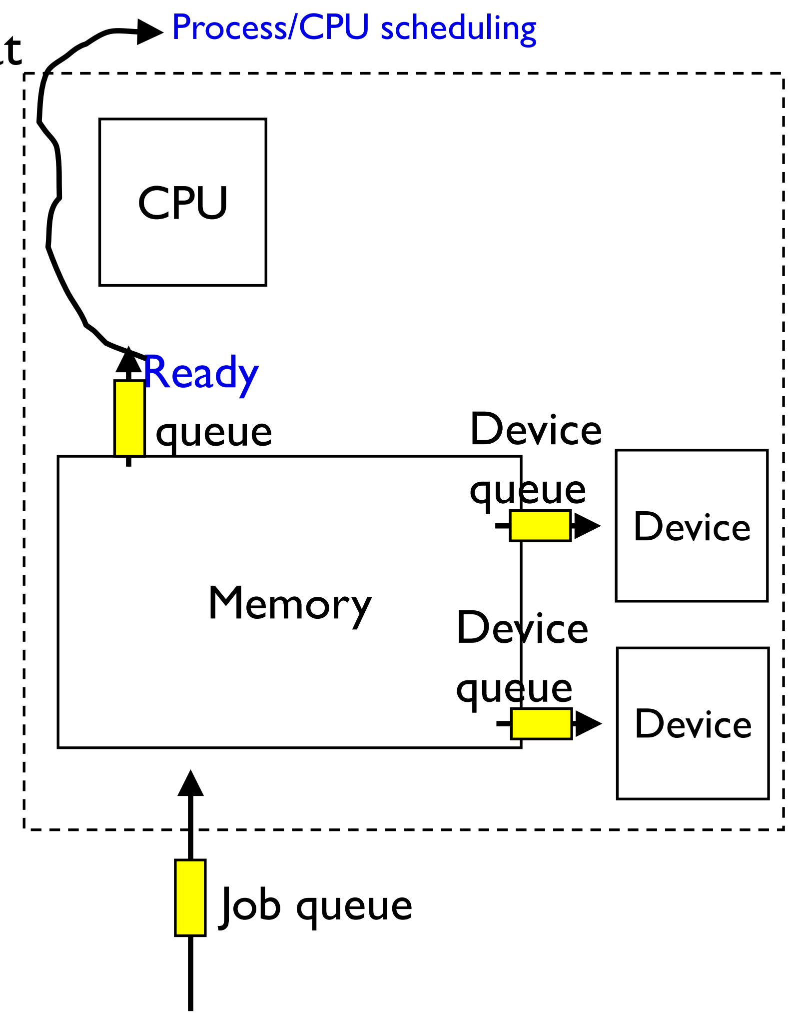
– We may have a short or long CPU burst.

I/O bound CPU bound

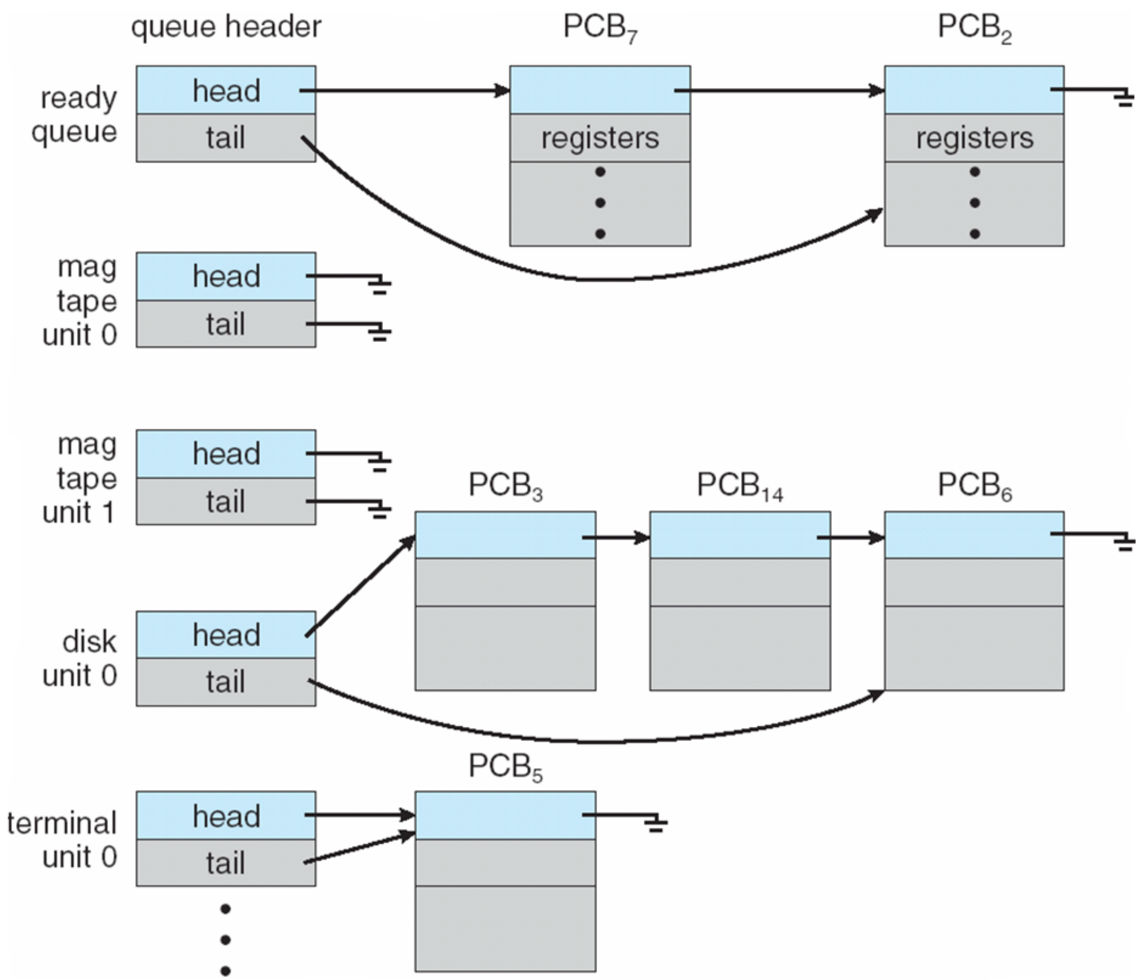


Scheduling Queues

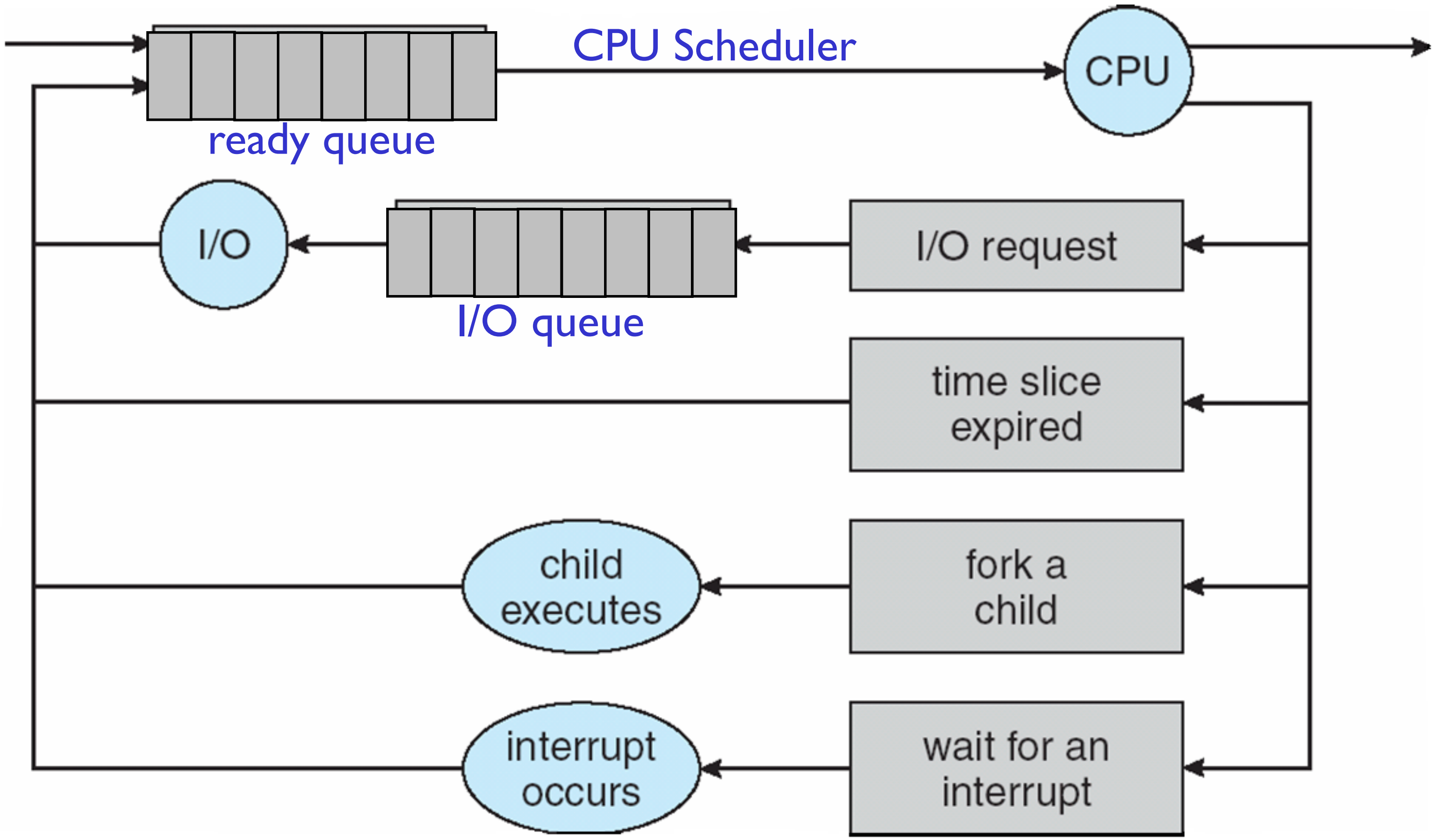
- Ready queue is one of the many queues that a process may be added
  - CPU scheduling is done from ready queue.
- Other queues possible:
  - Job queue – set of all processes started in the system waiting for memory
  - Device queues – set of processes waiting for an I/O device
    - A process will wait in such a queue until I/O is finished or until the waited event happens.
- A process may migrate among the various queues during its lifetime.



Ready Queue and Various I/O Device Queues

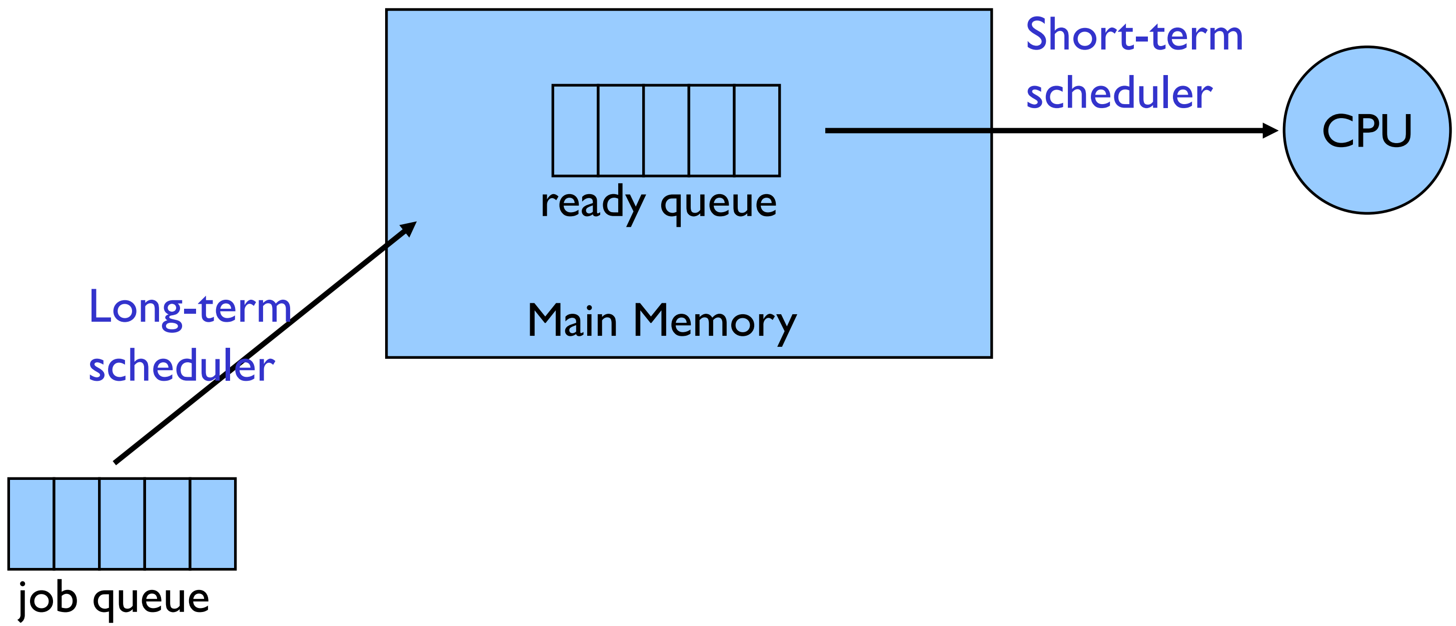


Representation of Process Scheduling



Schedulers

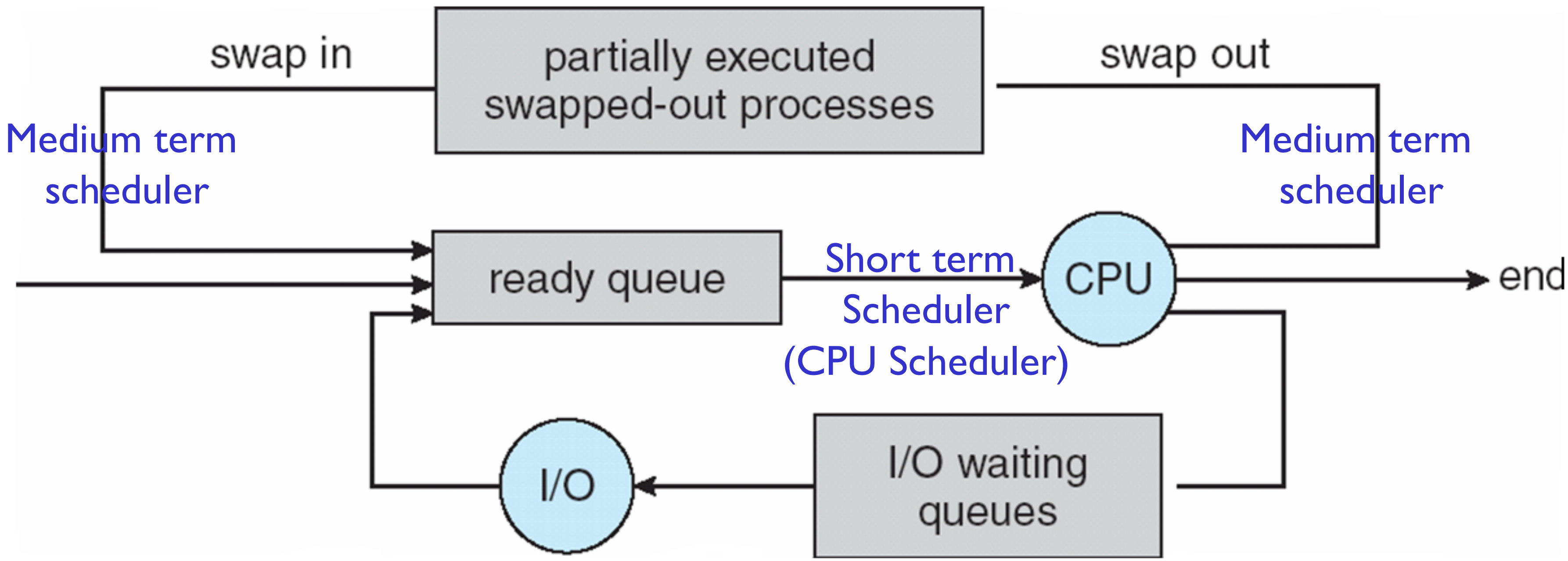
- Long-term scheduler (or job scheduler): selects which processes should be brought into the ready queue
- Short-term scheduler (or CPU scheduler): selects which process should be executed next and allocates CPU



Schedulers

- Short-term scheduler is invoked very frequently (milliseconds) => (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) => (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*
  - i.e., number of processes in memory
  - *Can also control kind of processes in memory!*
    - *Better to have a good mix of I/O bound and CPU bound processes*
- Not all systems have long term scheduler.

Addition of Medium Term Scheduling



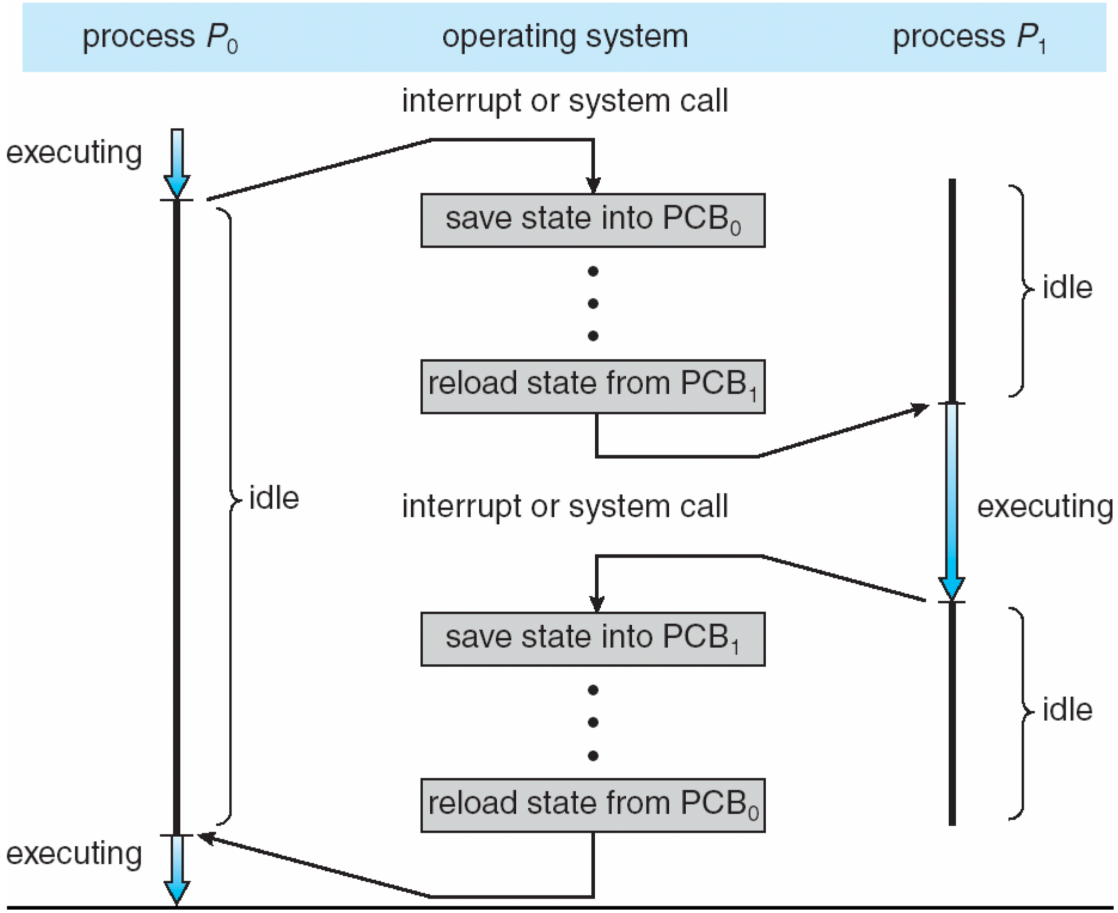
Context Switch

Switching CPU from one process to another process

- 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.
- Time dependent on hardware support.

Context Switch

Switching CPU from one process to another process

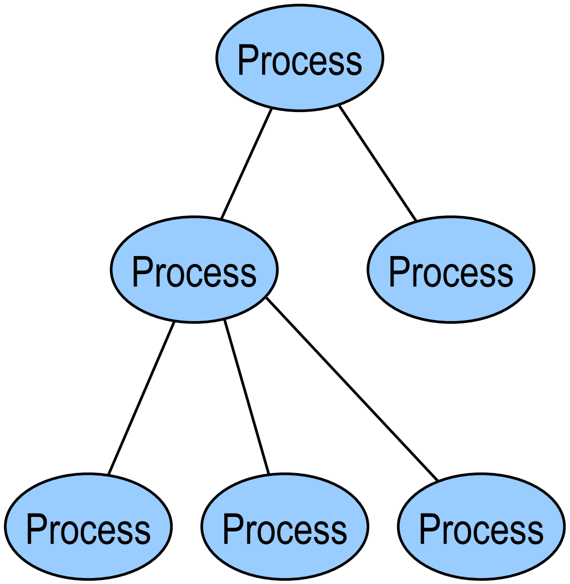


Process Creation and Termination

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)

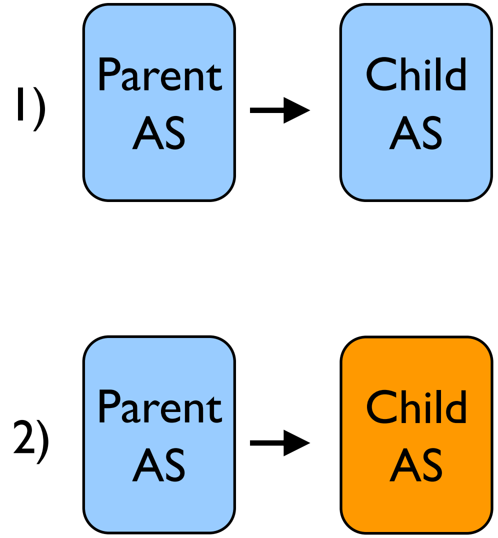
- Resource sharing alternatives:
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution alternatives:
  - Parent and children execute concurrently
  - Parent waits until children terminate



Process Creation (Cont)

- Child's address space?

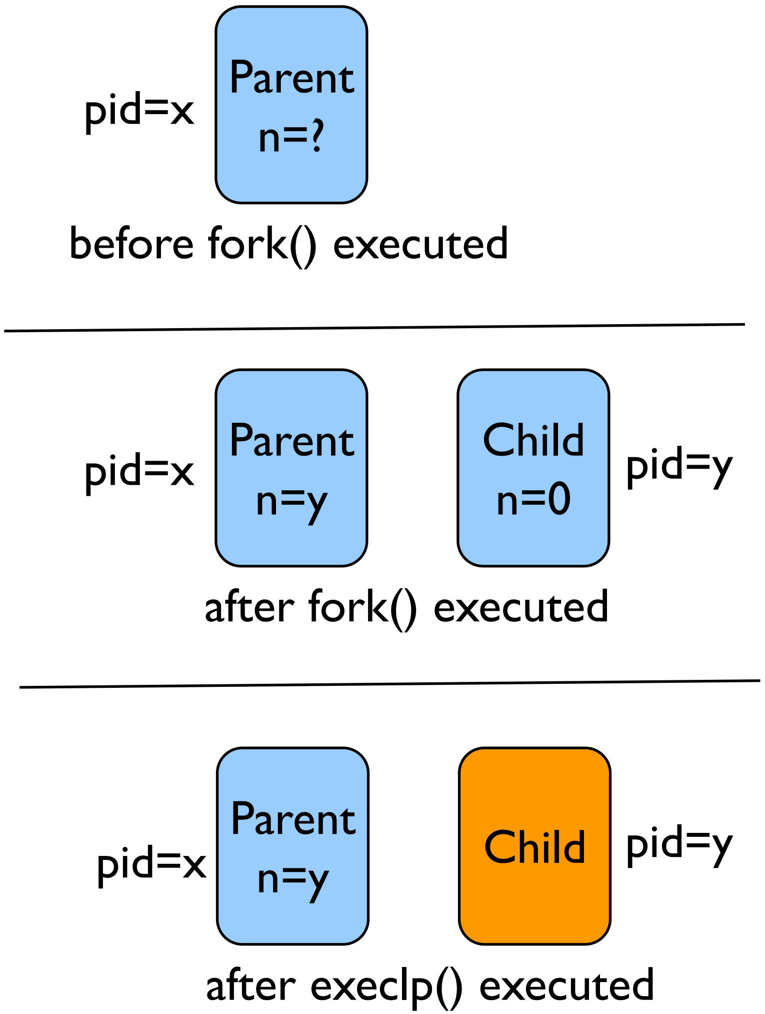
  Child has a new address space.
  Child's address space can contain:
  - 1) the copy of the parent (at creation)
  - 2) has a new program loaded into it



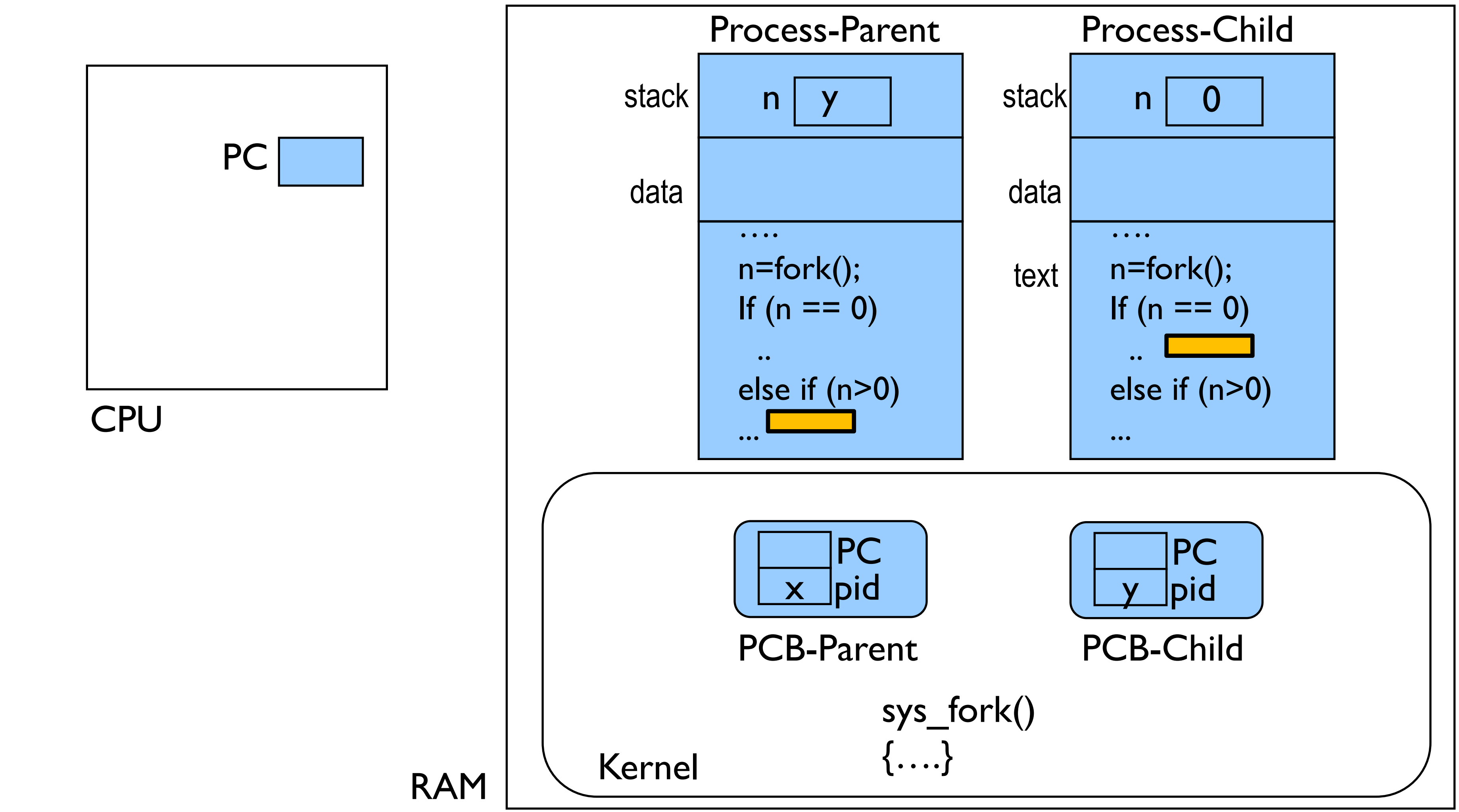
- In Unix-like systems (Linux, etc.)
  - fork system call creates a new process
  - exec system call used after a fork to replace the process' memory space with a new program

C Program Forking Separate Process in Linux

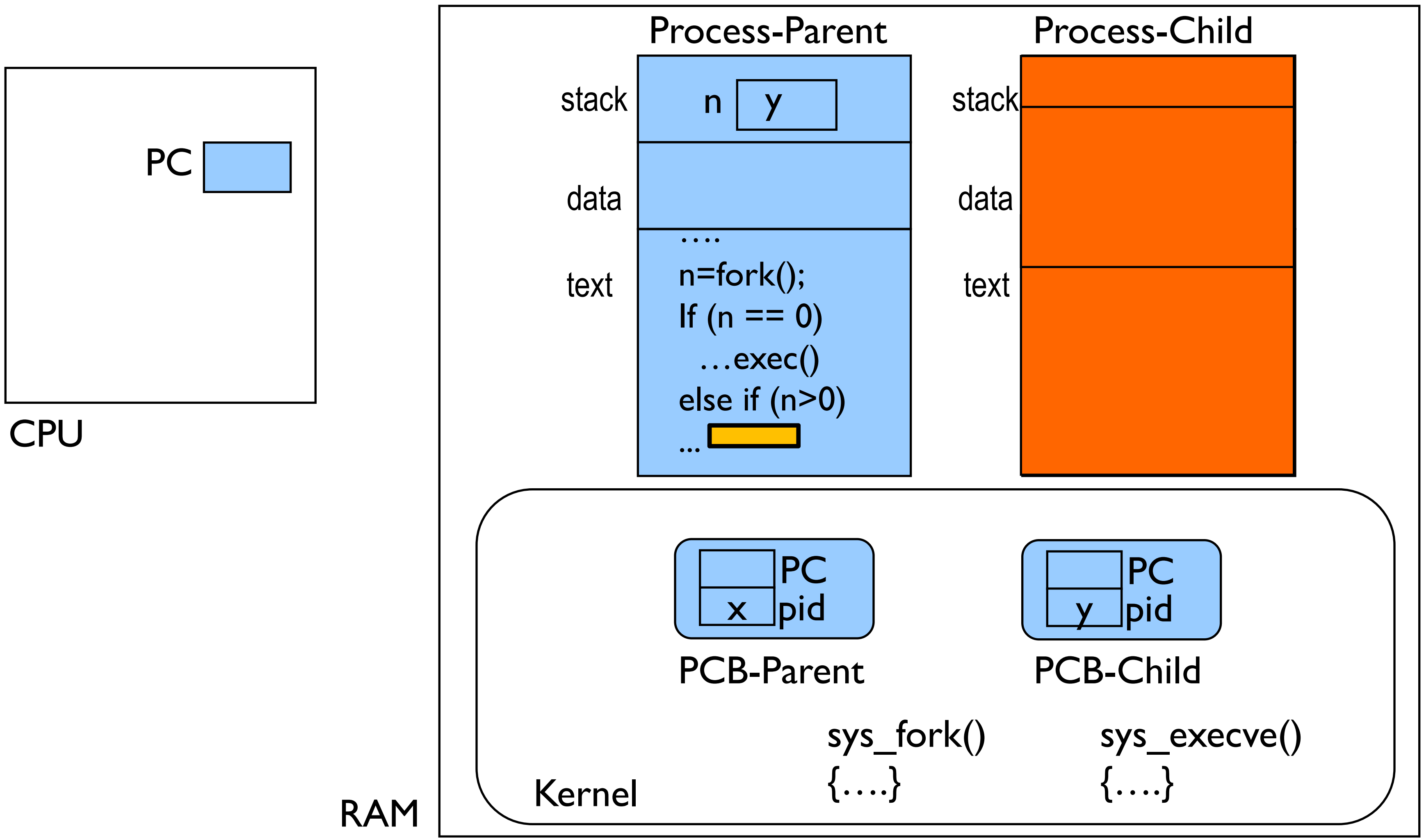
```
int main()
{
   pid_t  n; // stores process id

   n = fork();
   if (n < 0) {
        fprintf(stderr, "fork Failed");
        exit(-1);
   }
   else if (n == 0) {
        /* child process*/
        execlp("/bin/ls", "ls", NULL);
   }
   else {
        /* parent process */
        wait (NULL);
        printf ("child completed");
        exit(0);
   }
}
```



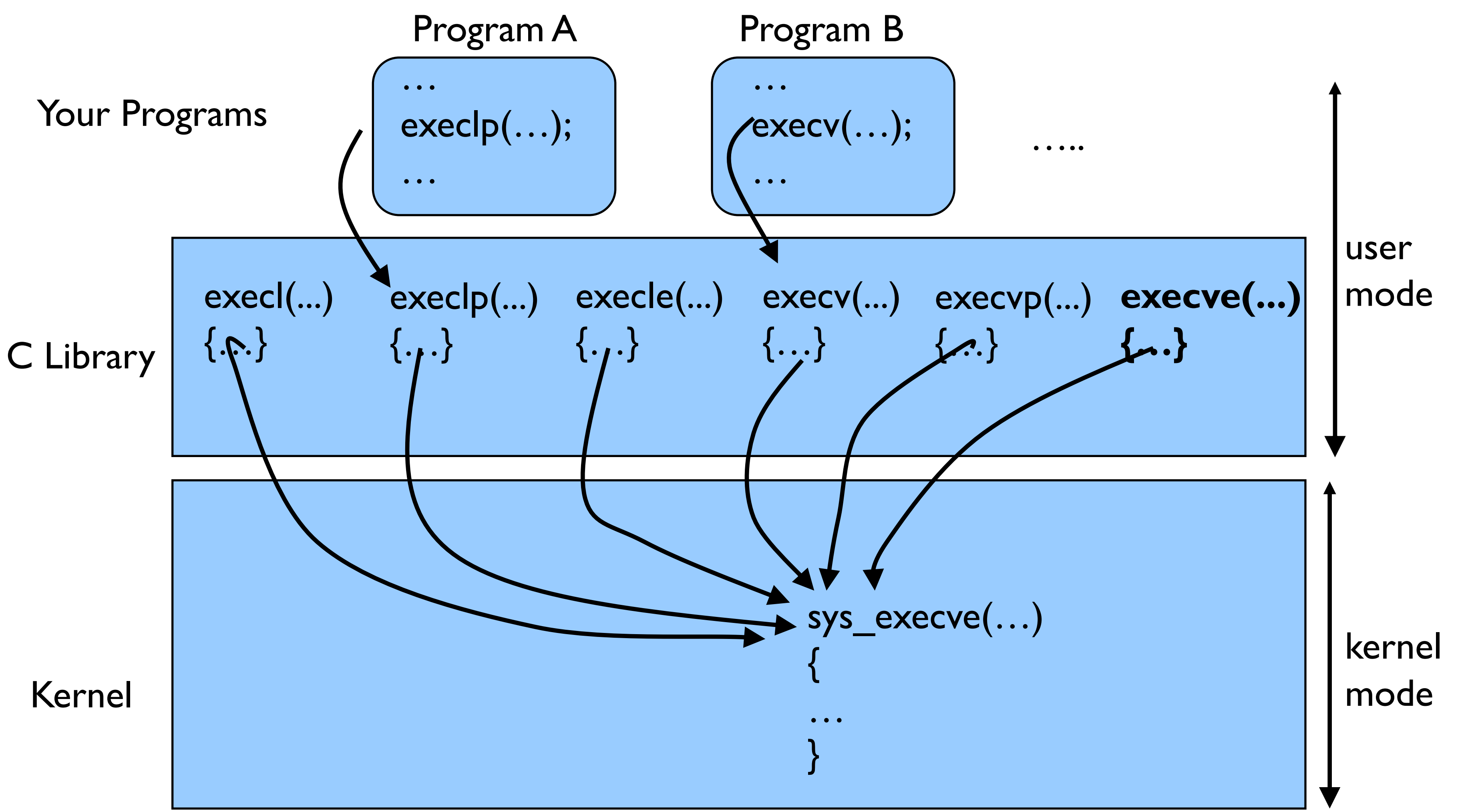
Execution Trace: fork()



Execution Trace: fork() with execlp()



Family of exec() Functions in Unix



Examples

What is the following pseudocode doing?

```
main()
    pid_t  n=0;

    for (i =0; i < 10;  ++i) {
        n = fork();
        if (n==0) {
            print ("hello");
            exit (0);
        }
    }

    for (i=0; i<10; ++i)
         wait();
}
```

What is the following pseudocode doing?

```
main()
    pid_t  n=0;

    for (i = 0;  i < 2;  ++i) {
        print (i);
        n = fork();
        print ("hello");
    }
}
```

Examples

What is the following pseudocode doing?

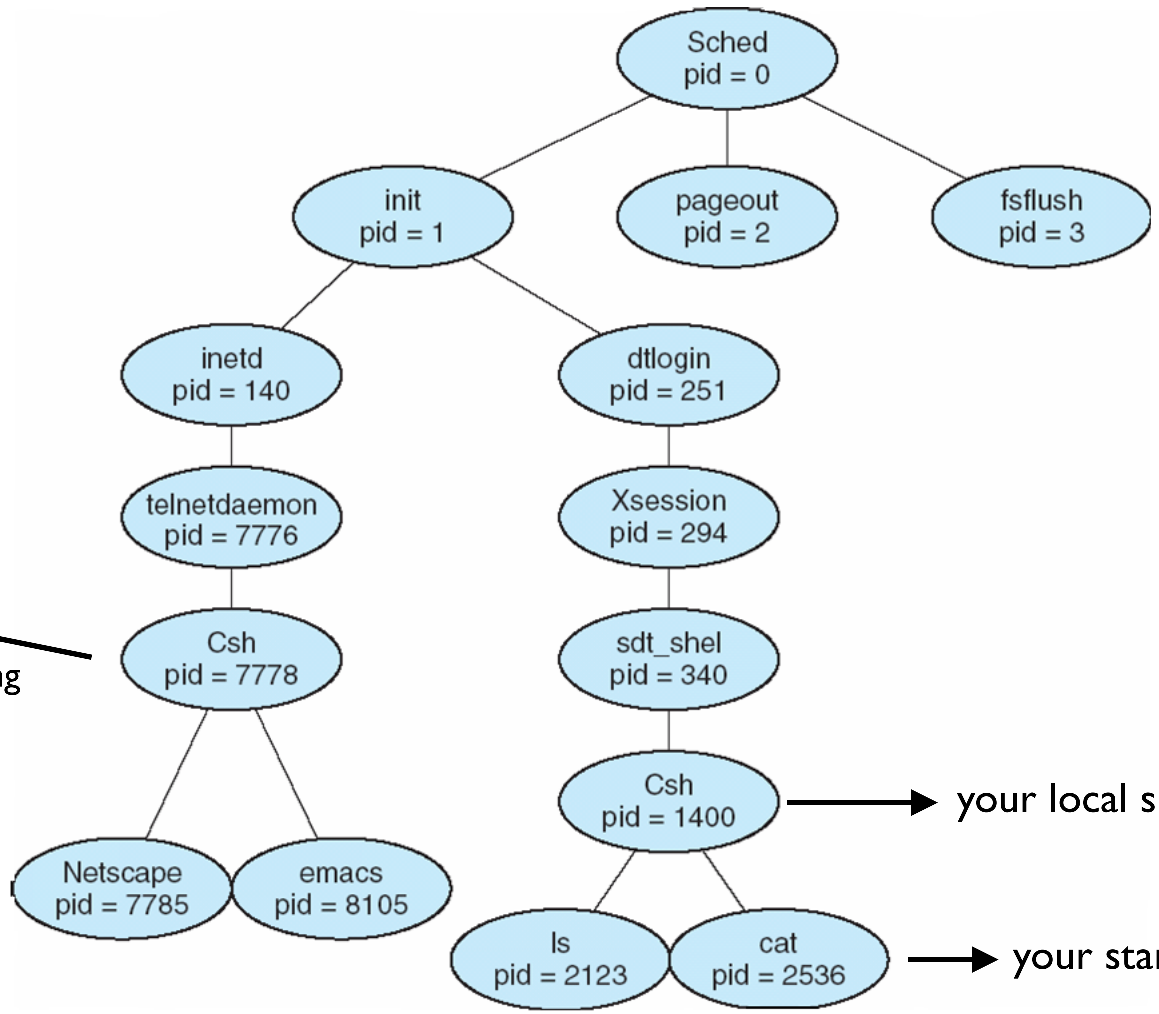
```
main()
    pid_t  x,y;

    x = fork();
    if (x == 0) {
        y = fork();
        if (y == 0) {
            print ("hello");
        }
    }
}
```

Examples

What is the following pseudocode doing?

```
main() {
    int x,y;

    x = fork();
    if (x == 0) {
        y = fork();
        if (y == 0) {
            print ("hello");
            exit(0);
        }
        exit(0);
    }
    waitpid (x);
}
```

A tree of processes on a typical Solaris



the shell that
a remote user is using

your local shell

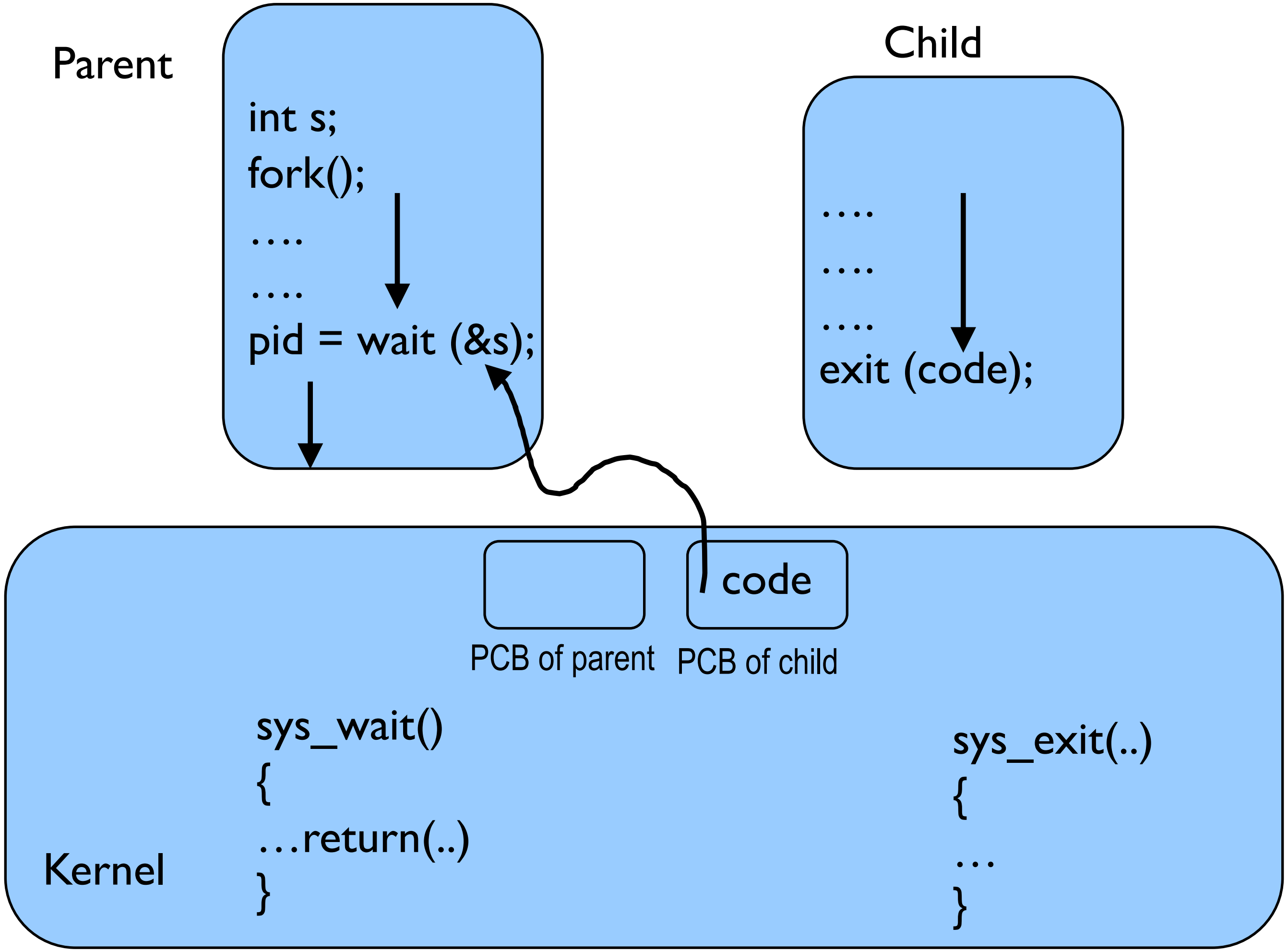
your started programs

Process Termination

- Process executes last statement and asks the operating system to delete it (can use exit system call)
  - Output data from child to parent (via wait)
  - Process' resources are deallocated by operating system

- Parent may terminate execution of children processes (abort)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
- When parent is exiting
  - Some operating systems do not allow child to continue if its parent terminates
  - All children terminated - cascading termination

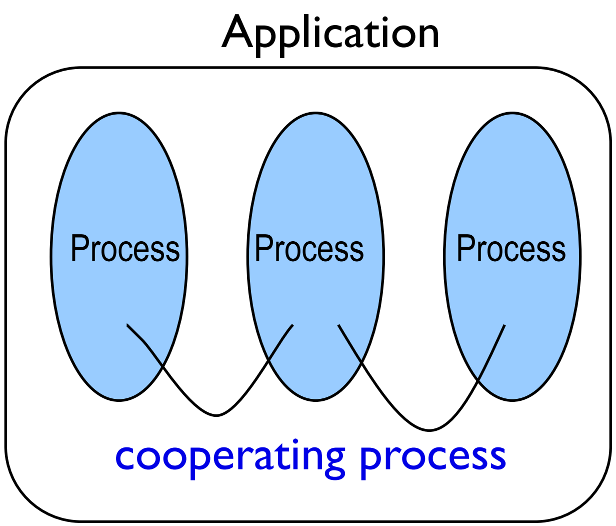
Process Termination



Inter-process Communication (IPC)

Cooperating Processes and the need for Interprocess Communication

- Processes within a system may be independent or cooperating
  - Independent processes cannot affect each other.
  - Cooperating processes can affect each other.
- Reasons for using multiple processes that cooperate:
  - Computation speed-up
  - Modularity (application will be divided into sub-tasks)
  - Convenience (may be better to work with multiple processes)
- Need for communication (IPC)



The overall application is designed to consist of cooperating processes

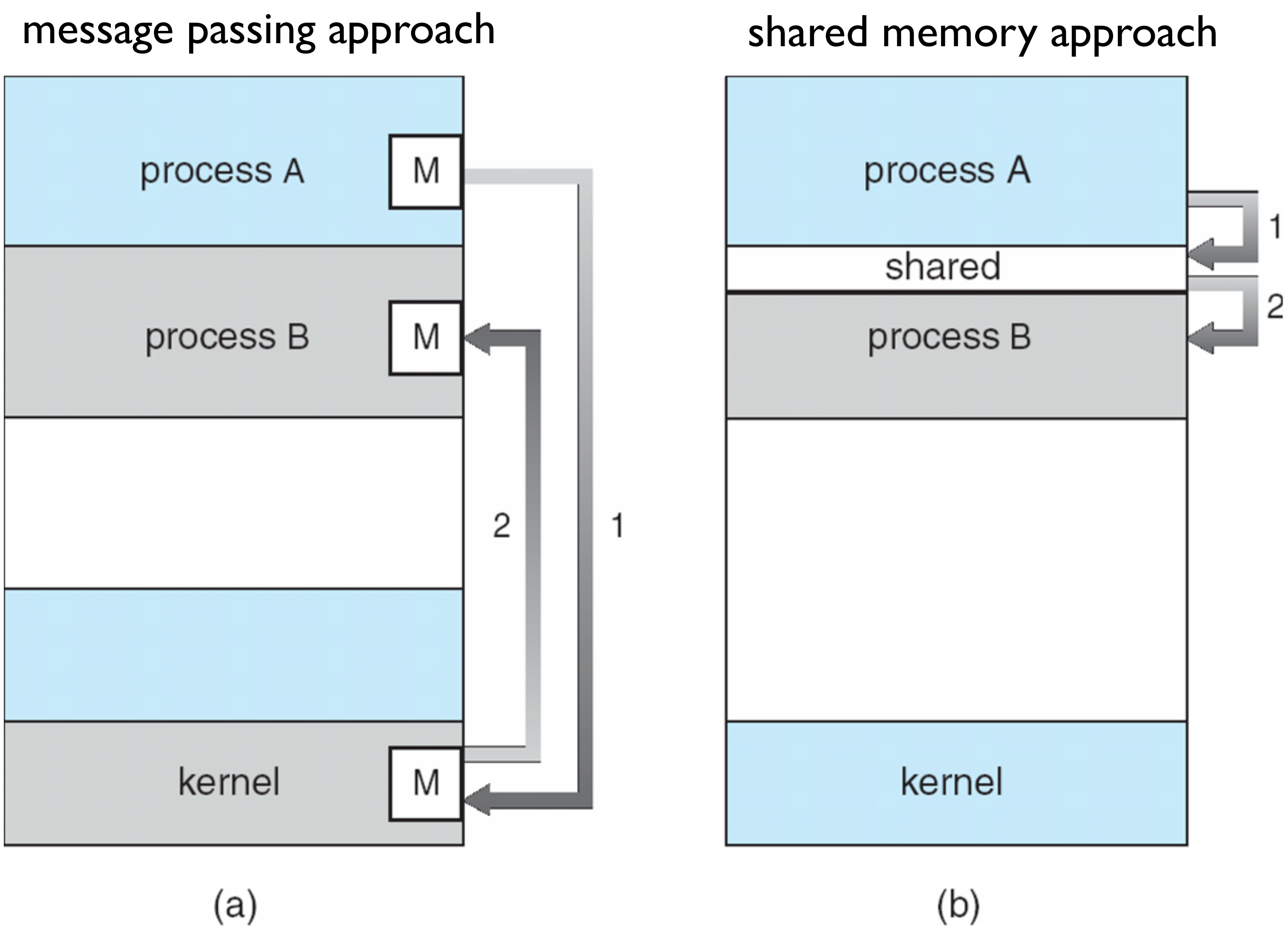
IPC Mechanisms

- Cooperating processes require a facility/mechanism for inter-process communication (IPC)
- There are two basic IPC models provided by most systems:
  - Shared memory model

    processes use a shared memory to exchange data
  - Message passing model

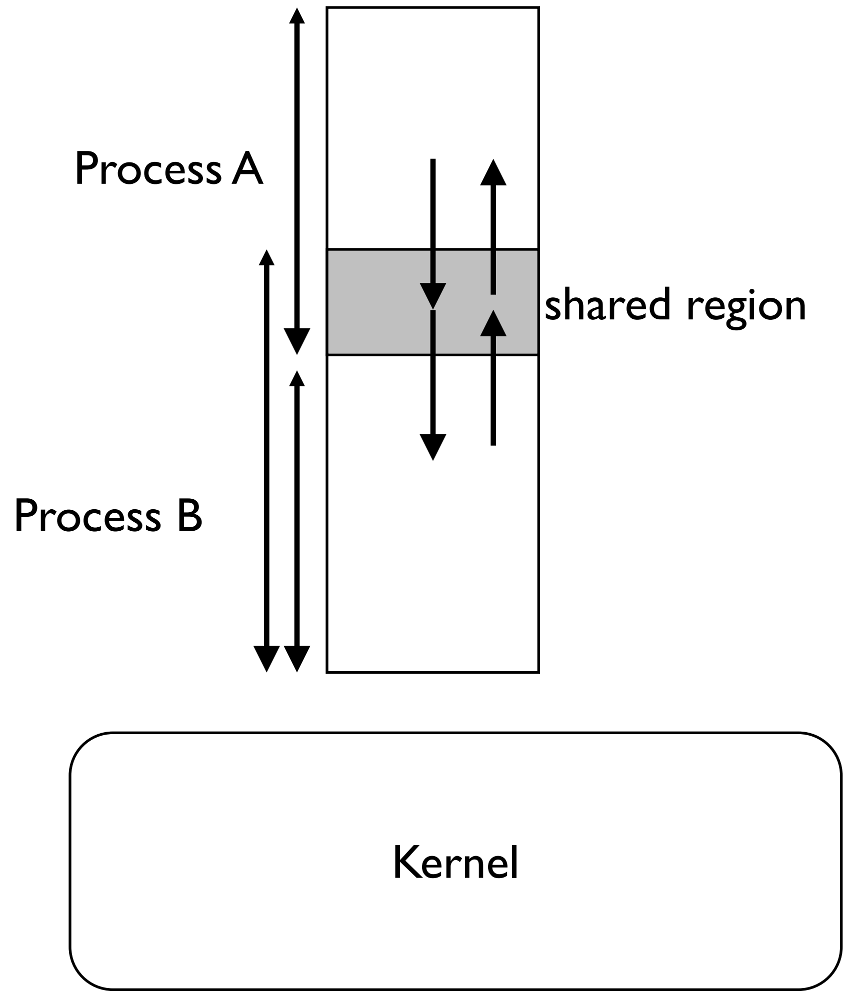
    processes send messages to each other through the kernel

Communication Models



Shared Memory IPC Mechanism

- A region of shared memory is established among two or more processes.
  - via the help of the operating system kernel (i.e., a system call).
- Processes can read and write the shared memory region (segment) directly as ordinary memory accesses (using pointers)
  - kernel not involved in data exchange
  - fast

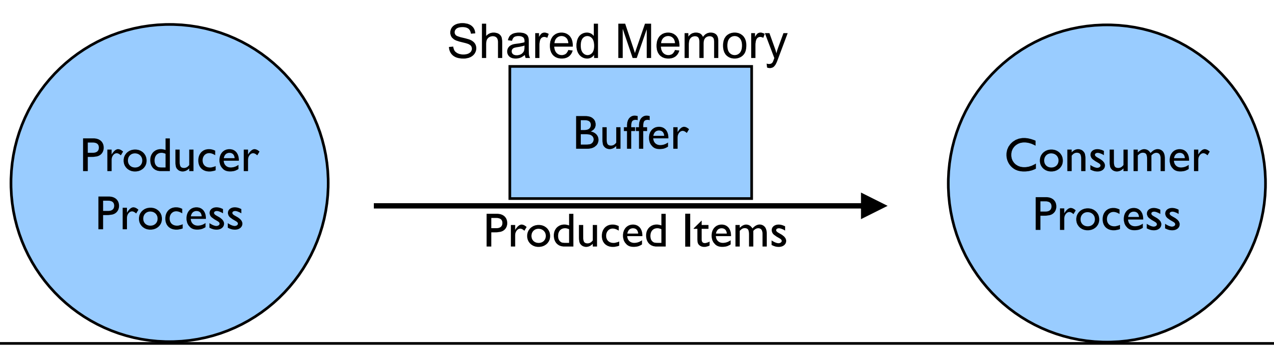


Access via pointers

```
char *cptr;
char c;

cptr = open (attach to) shared memory region

/*
   cptr points to the beginning (start) of the shared memory.
   shared memory is a sequence of bytes/characters
*/

cptr[0] = 'A';    // write a value into byte 0 of shared memory
cptr[1] = 'B';    // write a value into byte 1 …

c = cptr[5];      // read value at byte 5 of shared memory

// we can also read / write integers and structures
// all via pointers
```

Shared Memory IPC Mechanism

- To illustrate the use of an IPC mechanism, a general model problem, called producer-consumer problem, can be used. A lot of problems look like this.
- We have a producer process, a consumer process, and data is sent from producer to consumer.
- Buffering options:
  - Unbounded buffer: unlimited buffer (not very practical)
  - Bounded buffer: fixed buffer size
- We will see how shared memory can be used to pass data from producer to consumer.



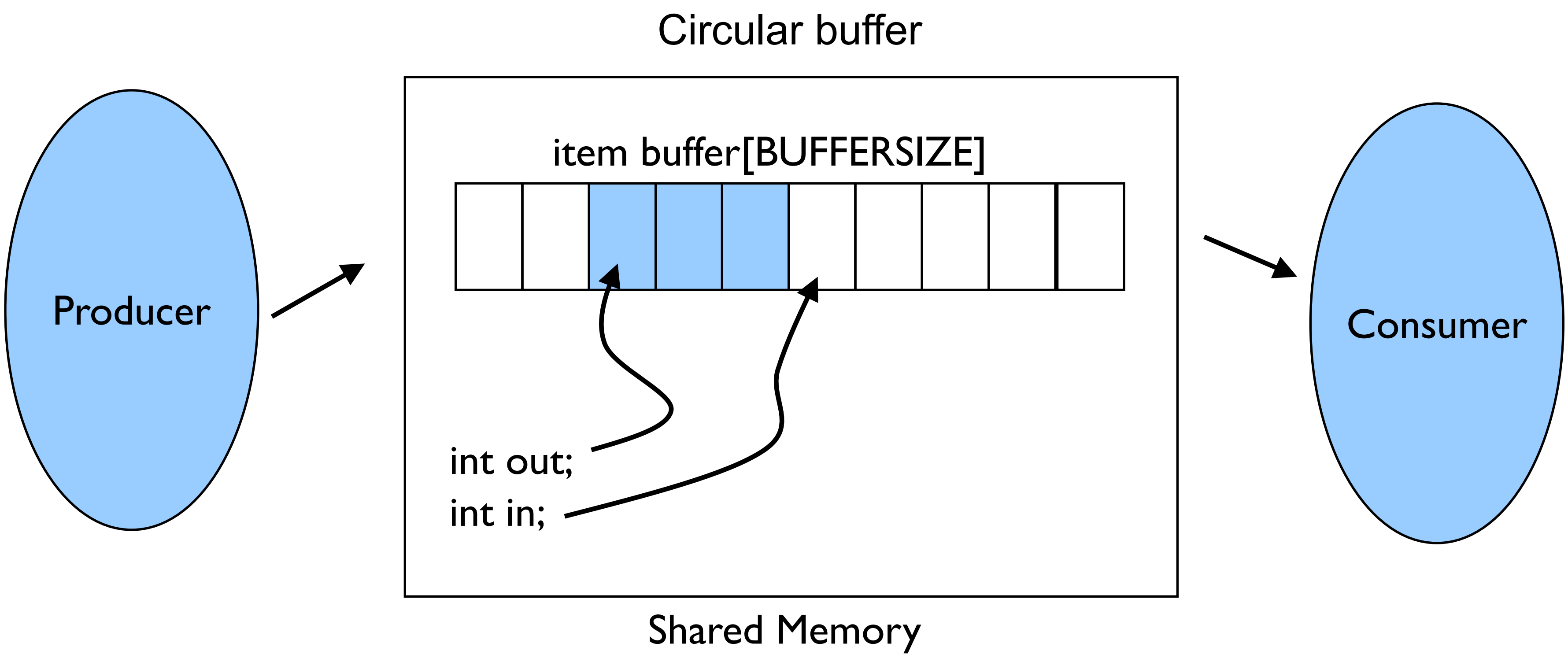
Bounded-Buffer – Shared-Memory Solution

```
// Shared data
#define BUFFERSIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFERSIZE];
int in = 0;  // next free position
int out = 0; // first full position
```

Solution is correct, but can only use BUFFERSIZE-1 elements

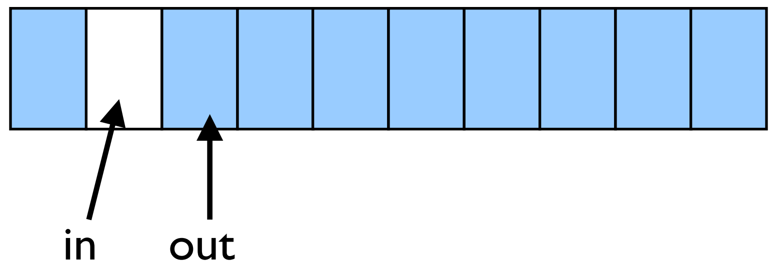
We will implement a circular buffer.

Buffer State in Shared Memory



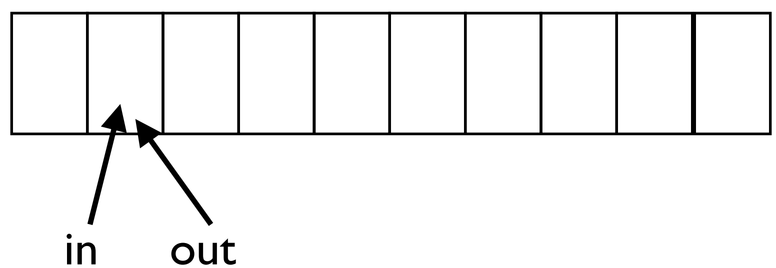
Buffer State in Shared Memory

Buffer Full



((in+1) % BUFFERSIZE == out) : considered full buffer

Buffer Empty



(in == out) : empty buffer

Bounded-Buffer – Producer and Consumer Code

Producer

```
while (true) {
        // produce an item
        while ( ((in + 1) % BUFFERSIZE)  == out)
                ;   // do nothing -- no free buffers
        buffer[in] = item; // put item into buffer
        in = (in + 1) % BUFFER SIZE;
}
```

buffer (an array)
in,out integer variables

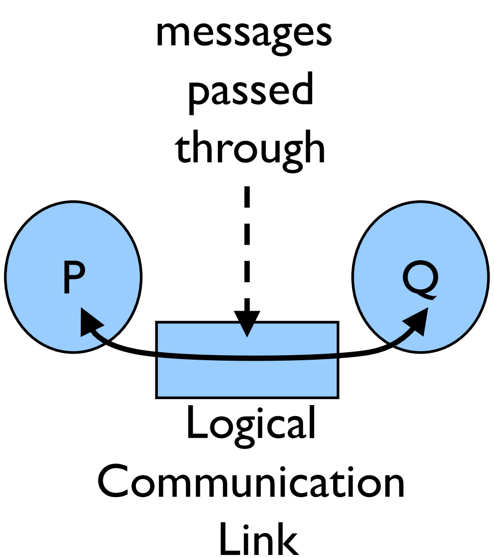
Shared Memory

Consumer

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

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

Message Passing IPC Mechanism

- Another mechanism for processes to communicate and to synchronize their actions.
- Communication without needing shared variables.
- Provides two operations:
  - send (*message*) - message size can be fixed or variable
  - receive (*message*)
- If processes *P* and *Q* wish to communicate, they need first to establish a (logical) *communication link (channel)* between them.
  - can be realized with a message queue – mailbox implemented in the kernel.
- Then, they exchange messages by sending to or receiving from the messages queue (through channel).



Implementation in a system

A messaging passing facility can be implemented in various ways.

- Will have different features depending on the system.

Design alternatives:

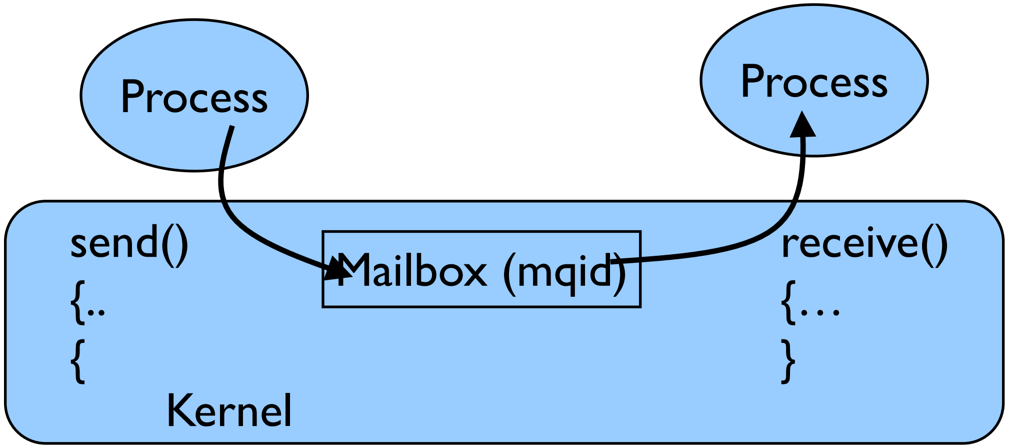
- How are links established?
  - Explicitly by a process? Or implicitly by the kernel?
- Can a link be associated with more than two processes?
- How many links can be there between a pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Naming: identifying the receiver

Naming (how do we identify the receiver)

- Direct naming and communication: receiver and sender process explicitly specified.
  - send (*P*, *message*): send a message to process P
  - receive(*Q*, *message*): receive a message from process Q
- Indirect naming and communication: messages are directed and received from mailboxes (also referred to as ports or message queues)
  - send (mqid, message)
  - receive (mqid, message)

Linux uses message queues, hence indirect naming.



indirect naming

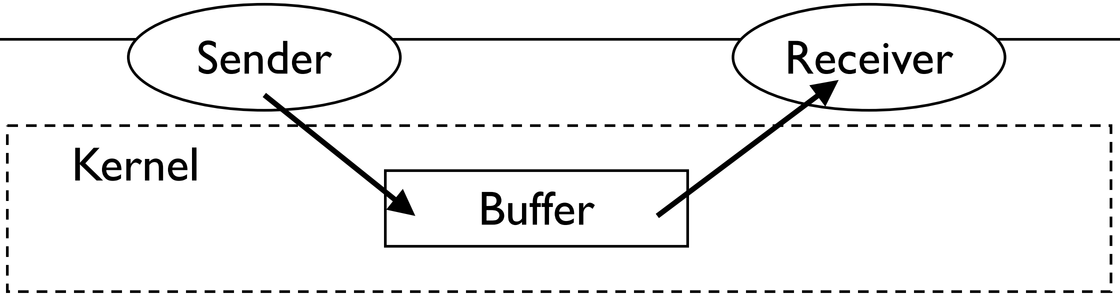
Synchronization

- How does a sender or receiver behave if it can not send or receive a message immediately
  - blocks (synchronous communication)
  - does not block (asynchronous communication)
- Blocking is considered synchronous
  - sender blocks until receiver or kernel receives the message.
  - receiver blocks until it can get a message
- non-blocking is considered asynchronous
  - send call tries to send the message and returns, without blocking. The message is either sent or could not be sent. May try later.
  - receive call tries to receive a message and returns, without blocking. Either a valid message is received or nothing is received. May try later.

Buffering

- Exact behavior depends also on space availability in the buffer.
- Buffer: queue of messages sent to the link. Buffer options:
  - Zero capacity buffer
    Sender must wait for receiver to receive (rendezvous)
  - Bounded capacity – at most *n* messages can be buffered. Sender must wait if link full.
  - Unbounded capacity: infinite length buffer. Sender never blocks.

Synchronization



	Zero Buffer	Some Buffer
Blocking Send	Wait until receiver receives	Wait until kernel receives (if buffer has space no wait)
Blocking Receive	Wait until sender has a message	Wait until kernel has a message (if buffer has message no wait)
Nonblocking Send	Return with receiver received the message or error	Return with kernel received the message or error
Nonblocking Receive	Return with a message or none	Return with a message or none

Example IPC: POSIX message queues

POSIX (Portable Operating System Interface) is the standard API for Unix-like systems.

shareddefs.h

```
struct item {
		int id;
		char astr[64];
};

#define MQNAME   "/just_a_filename"
```

- We have a producer and a consumer process.
- Producer sends messages to the consumer.
- In this example, we need to run the consumer first.
- Consumer creates the message queue.

producer.c

```
#include <stdlib.h>
#include <mqueue.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

#include "shareddefs.h"

int main()
{
        mqd_t mq;  struct item item; int n;

        mq = mq_open(MQNAME, O_RDWR);
        if (mq == -1) { perror("mq_open failed\n"); exit(1); }
        int i = 0;
        while (1) {
                item.id = i; // prepare an item
                strcpy(item.astr, ”cs342 operating systems\n");
                n = mq_send(mq, (char *) &item, sizeof(struct item), 0);
                if (n == -1) {perror("mq_send failed\n"); exit(1); }
                i++;
                sleep(1); // sleep for 1 second
        }
        mq_close(mq);
        return 0;
}
```

consumer.c

```
#include <stdlib.h>
#include <mqueue.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include "shareddefs.h"
int main()
{       mqd_t mq;   struct mq_attr mq_attr;
        struct item *itemptr; int n, buflen; char *bufptr;

        mq = mq_open(MQNAME, O_RDWR | O_CREAT, 0666, NULL);
        if (mq == -1) { perror("can not create msg queue\n"); exit(1); }
        mq_getattr(mq, &mq_attr);
        printf("mq maximum msgsize = %d\n", (int) mq_attr.mq_msgsize);
        buflen = mq_attr.mq_msgsize; // max message size supported
        bufptr = (char *) malloc(buflen); // allocate large enough space
        while (1) {
                n = mq_receive(mq, (char *) bufptr, buflen, NULL);
                if (n == -1) { perror("mq_receive failed\n"); exit(1); }
                printf("mq_receive success, message size=%d\n", n);
                itemptr = (struct item *) bufptr; // for easy access to item
                printf("item-id   = %d\n", itemptr->id);
                printf("item-astr = %s\n", itemptr->astr); printf("\n");
        }
        free(bufptr);
        mq_close(mq);
        return 0;
}
```

Compiling and running consumer and producer

Makefile

```
all: producer consumer

consumer: consumer.c
	gcc -Wall -o consumer consumer.c -lrt

producer: producer.c
	gcc -Wall -o producer producer.c -lrt

clean:
	rm -fr *~ producer consumer
```

```
$make
$ ./consumer
```

```
$./producer
```

Compile. Run consumer in one terminal.

Run producer in another terminal.

Linux message queue

- Provides message oriented communication.
- Message boundaries preserved.
  - If sender sends a message of *N* bytes, the receiver receives the complete message of *N* bytes (not a portion of it).
- A message queue has kernel persistence (can exist in kernel if not explicitly removed by the program).
- A message queue is bidirectional.
- Usually, max message size is 8192 bytes.
- More than one process can write into a message queue. More than one process can receive from a message queue (but a message goes to only one process – not duplicated).
- A process can create more than one message queue.

Example IPC: POSIX shared memory

- The following functions are defined to create and manage shared memory in POSIX API
    - shm_open():
        - *create or open a shared memory region/segment (also called shared memory object)*
    - shm_unlink():
        - *remove the shared memory object*
    - ftruncate():
        - *set the size of shared memory region*
    - mmap():
        - *map the shared memory into the address space of the process. With this, a process gets a pointer to the shared memory region and can use that pointer to access the shared memory.*

Interprocess Communication: Examples: POSIX shared memory

producer.c

In this example, run the producer first.

```
…
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/mman.h>
#include <sys/types.h>
#define SNAME “shmname”
int main()
{
        const int SIZE = 4096;
        const char *message0= "Studying ";
        const char *message1= "Operating Systems ";
        const char *message2= "Is Fun!\n";
        int shm_fd; void *ptr;

        shm_fd = shm_open(SNAME, O_CREAT | O_RDWR, 0666);
        ftruncate(shm_fd,SIZE); // set size of shared memory

        ptr = mmap(0,SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
        if (ptr == MAP_FAILED) { printf("Map failed\n"); return -1; }

        sprintf(ptr,"%s",message0);
        ptr += strlen(message0);
        sprintf(ptr,"%s",message1);
        ptr += strlen(message1);
        sprintf(ptr,"%s",message2);
        ptr += strlen(message2);
        printf (“%done\n”,
        return 0;
}
```

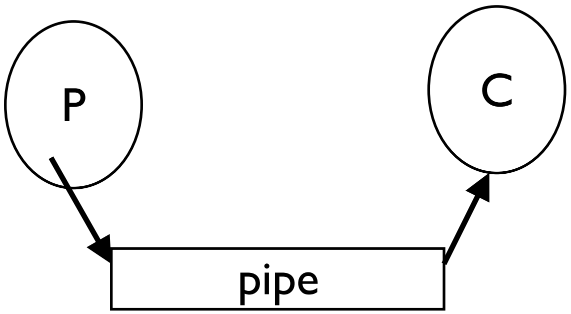
Interprocess Communication: Examples: POSIX message queues

consumer.c

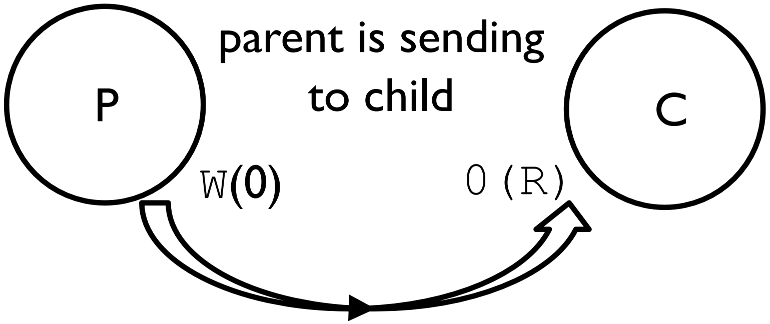
```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>
#include <sys/types.h>


#define SNAME “shmname”

int main()
{
        const int SIZE = 4096;
        int shm_fd;
        void *ptr;  int

        shm_fd = shm_open(SNAME, O_RDONLY, 0666);  // open shared memory
        if (shm_fd == -1) { printf("shared memory failed\n"); exit(-1); }

        ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);  // get pointer
        if (ptr == MAP_FAILED) {printf("Map failed\n"); exit(-1); }

        printf("%s",ptr);

        if (shm_unlink(SNAME) == -1)
                {printf("Error removing %s\n",SNAME); exit(-1);}
}
```

Other IPC methods: pipes

- Unix Pipes:
  - A pipe enables one-way communication (unidirectional) between a parent and child and vice versa.
  - It is easy to use.
  - When process terminates, pipe is removed automatically
  - Byte stream oriented communication. No message boundaries.
  - pipe() system call used. It puts two descriptors into an array argument.



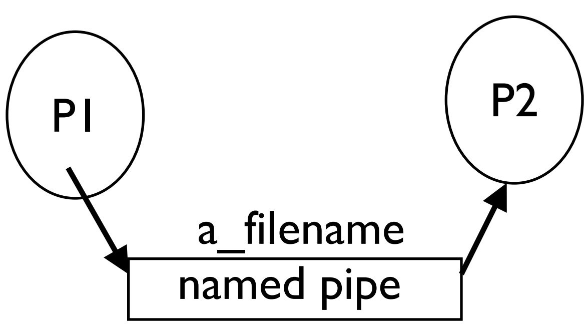
Interprocess Communication: Pipes




```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#define BUFFER_SIZE 32
#define R 0     // Read end of pipe
#define W 1     // Write end of pipe
int main(void)
{
        char write_msg[BUFFER_SIZE] = "Greetings";
        char read_msg[BUFFER_SIZE]; pid_t pid;
        int fd[2];     // an array of 2 integers: fd[0] and fd[1]
                        // fd[0] is read end, always, and fd[1] write end
        if (pipe(fd) == -1) { fprintf(stderr,"Pipe failed"); return 1;}
        pid = fork();
        if (pid < 0) { fprintf(stderr, "Fork failed"); return 1; }
        if (pid > 0) {  // this is parent
                close(fd[R]); //close read end, since not used here
                write(fd[W], write_msg, strlen(write_msg)+1);
                close(fd[W]);
        }
        else { // this is child
                close(fd[W]); //close write end, since not used here.
                read(fd[R], read_msg, BUFFER_SIZE);
                printf("child read %s\n",read_msg);
                close(fd[R]);
        }
        return 0;
}
```

Other IPC methods: named-pipes (FIFOs)

- A named-pipe is called FIFO.
  - It has a name (a filename – a string)
  - Call mkfifo() to create
- When processes terminate, it is not removed automatically.
- No need for parent-child relationship
- Bidirectional.
- Any two processes can create and use named pipes.



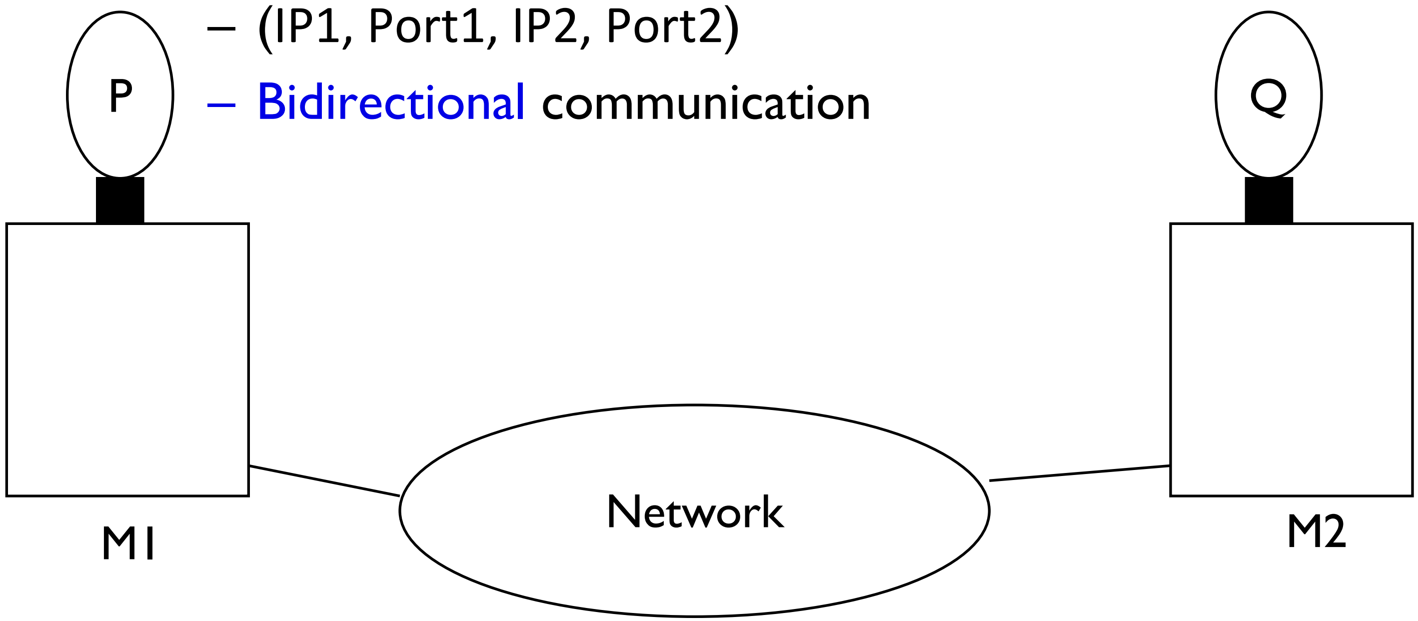
Communication Through Network: Client-Server Communication

Communications in Client-Server Systems

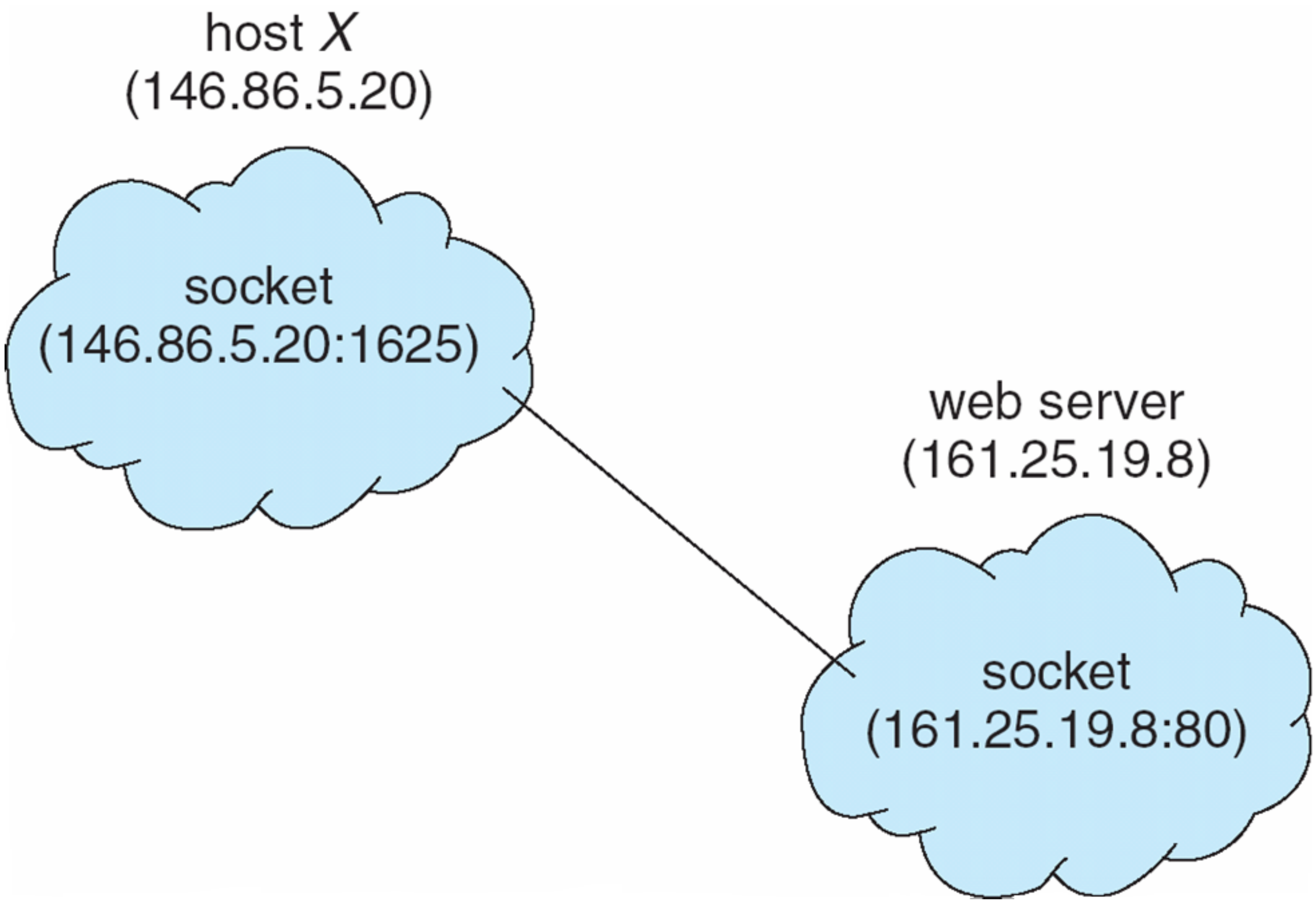
- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

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
- Communication happens between a pair of sockets
- A communication can be identified as a pair of sockets
  - (IP1, Port1, IP2, Port2)
  - Bidirectional communication



Socket Communication



Sockets

- Two types
  - TCP (STREAM): reliable connection oriented transport service
  - UDP: unreliable connectionless transport service
- A socket is bound to an address and port.

- A network application
  - Two parts
    - Usually a server and a client

TCP Server and Client

- Create a socket s and put to listening mode
- c = Accept (s)
- Read(c) and Write(c)
- Close(c)

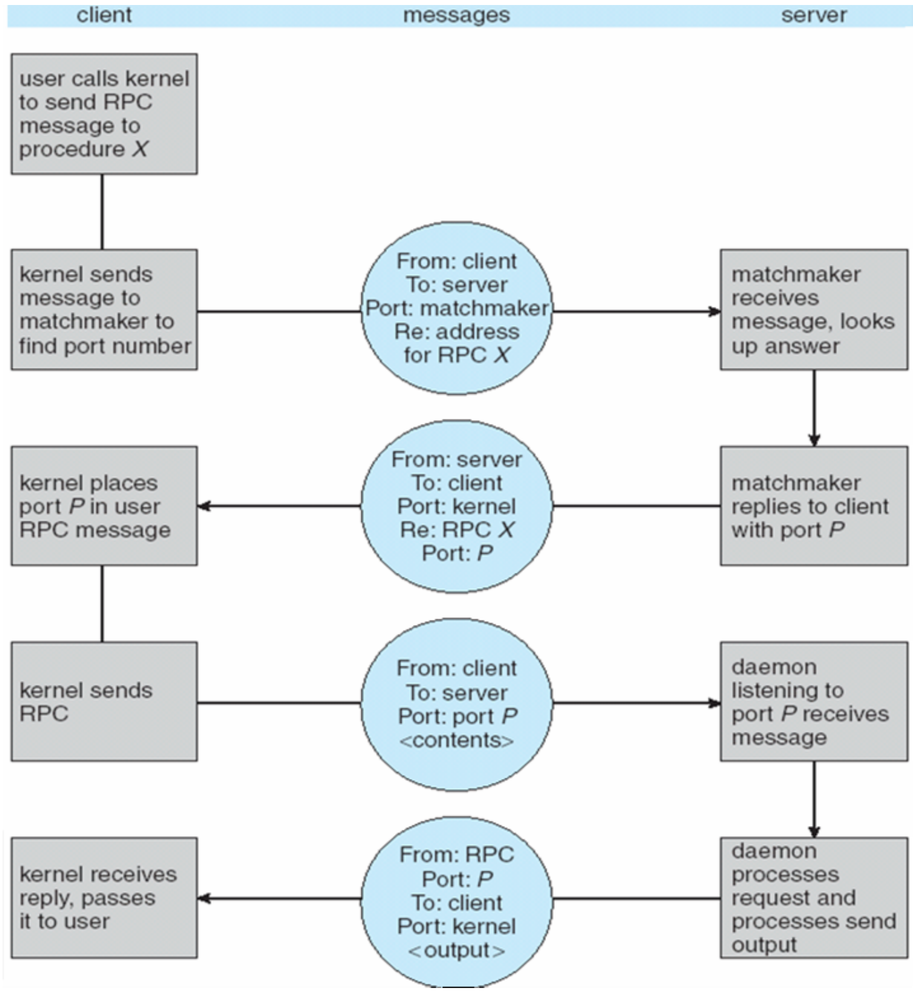
- Create a socket c
- Connect (c, s_address_port)
- Read (c) and Write (c)
- Close (c)

You will learn Socket programming in detail in Computer Networks course.

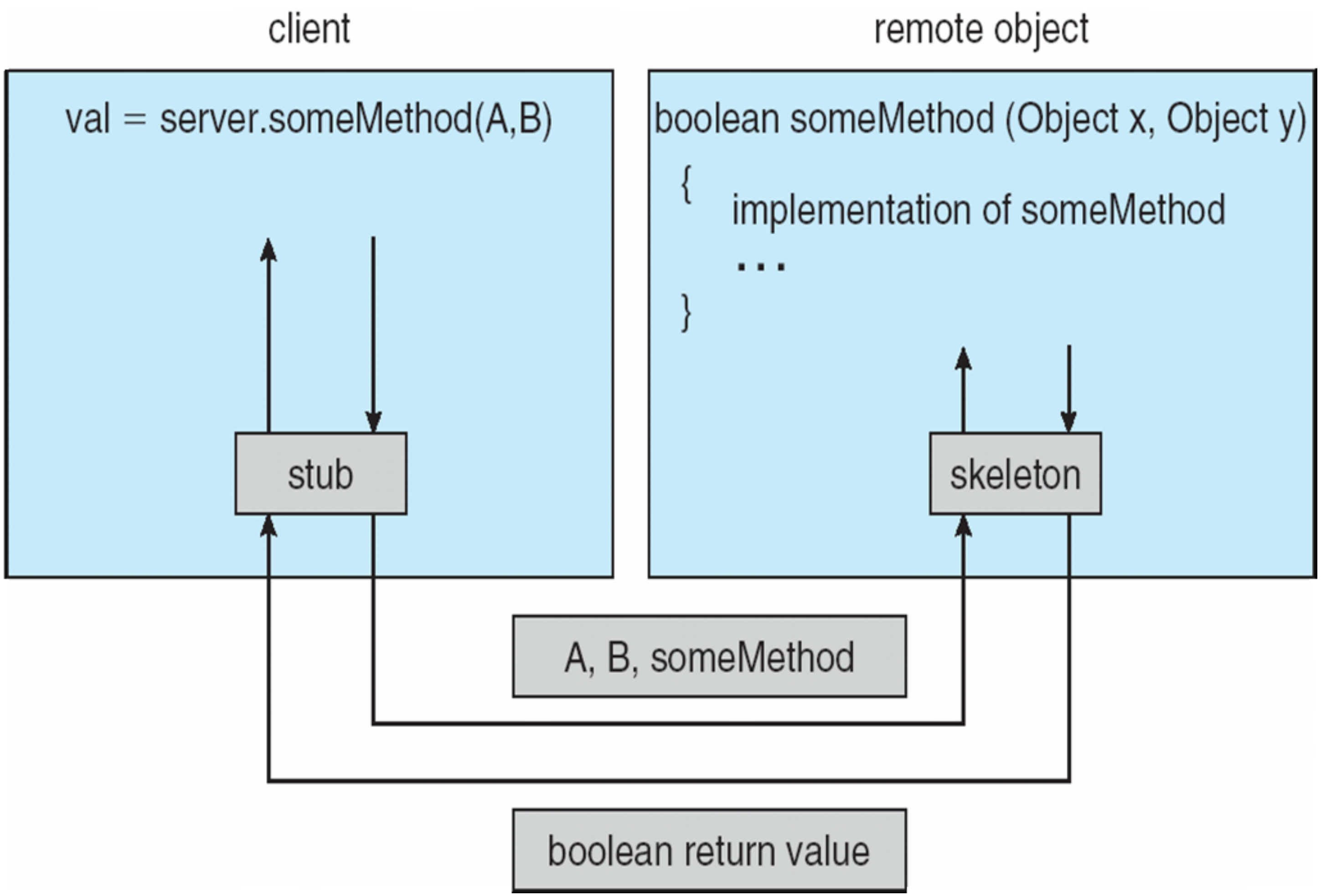
Remote Procedure Calls

- 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 into a message and sends a message to the server
- The server-side stub receives this message, unpacks the marshalled parameters, and performs (executes) the procedure on the server
- The return value is put into a reply message and sent back to the client-side

Execution of RPC

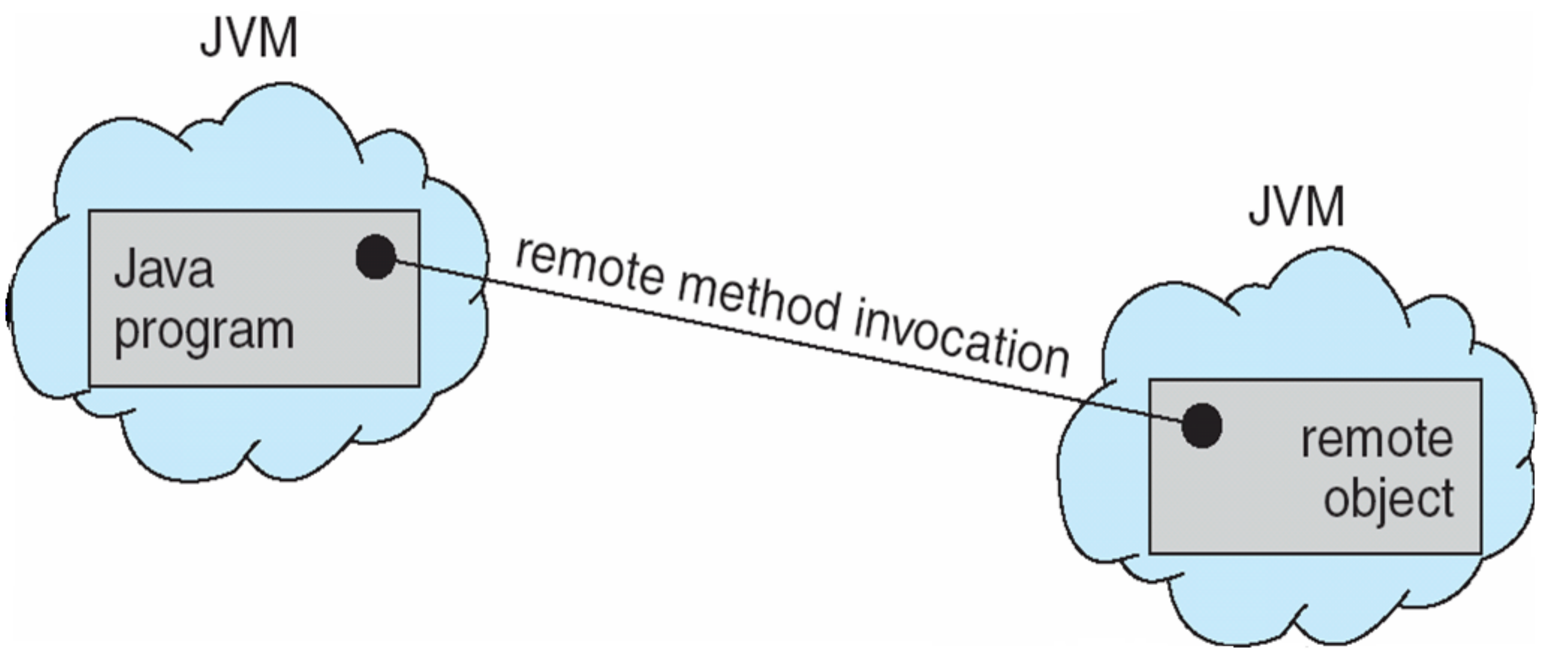


Marshalling Parameters



Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs
- RMI allows a Java program on one machine to invoke a method on a remote object



References

- Operating System Concepts, Silberschatz et al. Wiley.
- Modern Operating Systems, Andrew S. Tanenbaum et al.