Chapter 3: Processes

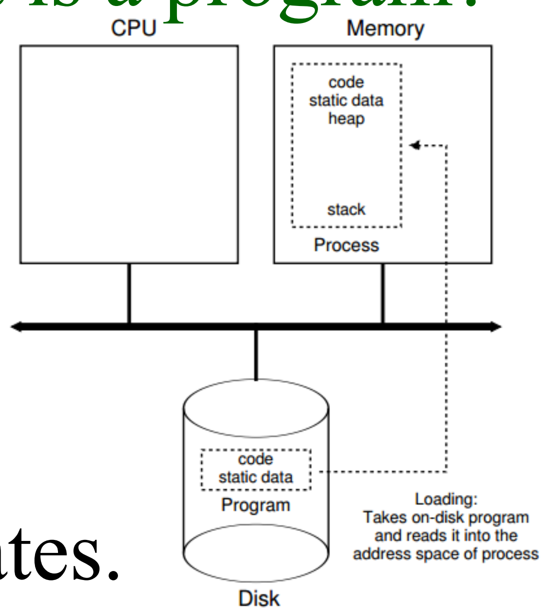
肖 卿 俊
办公室：江宁区无线谷6号楼226办公室
电邮：csqjxiao@seu.edu.cn
主页：https://csqjxiao.github.io/PersonalPage
电话：025-52091022

Chapter 3: Processes

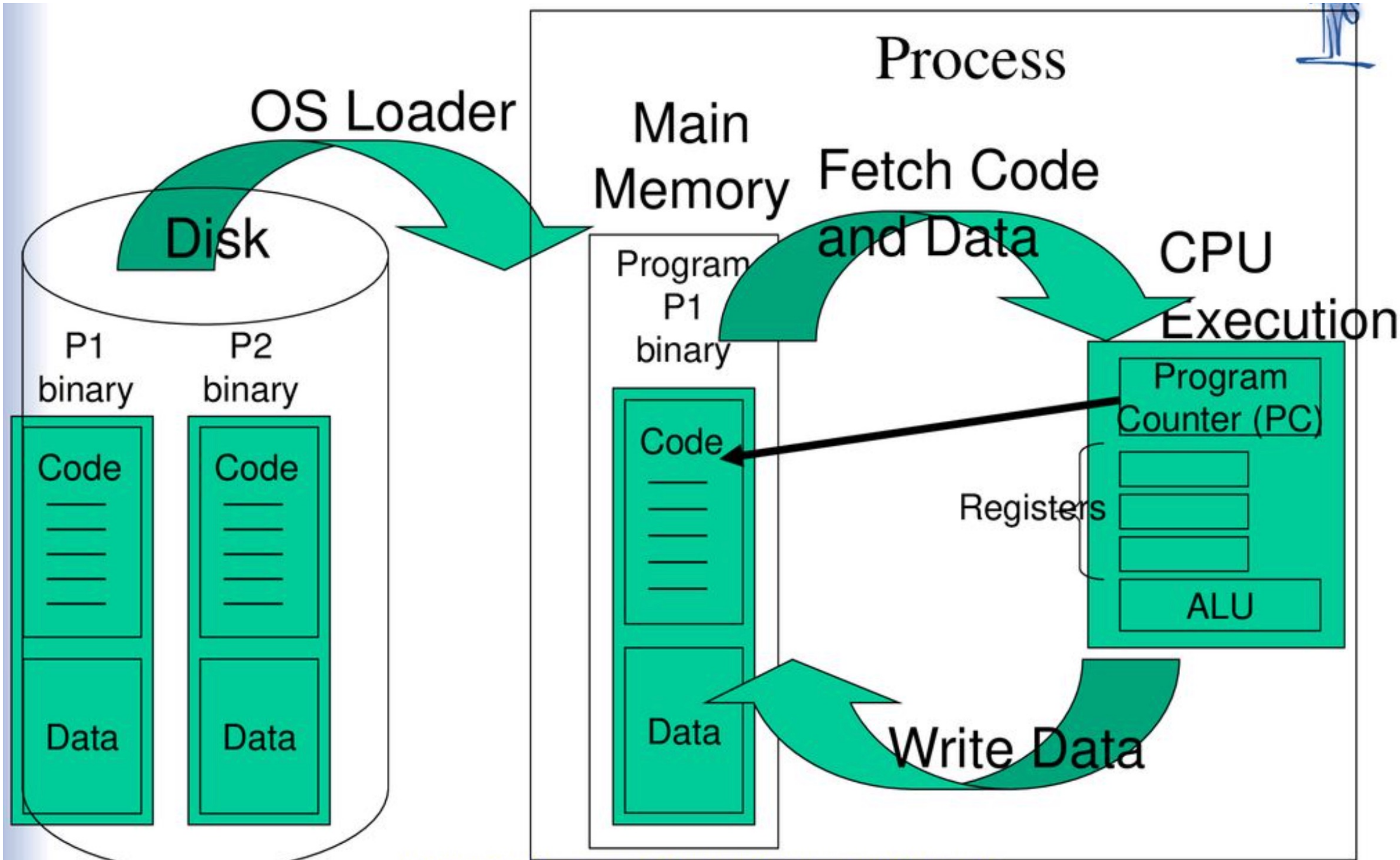
- Process Concept
- Operations and APIs on Processes
- Process Scheduling
- Cooperating Processes
- Inter-process Communication
- Communication in Client-Server Systems

Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably.
- Q: Why process, not program? What is a program?
- Process: running program
  - A program is lifeless, the OS makes it running (as a process).
  - A process can be viewed as a running program with machine states.



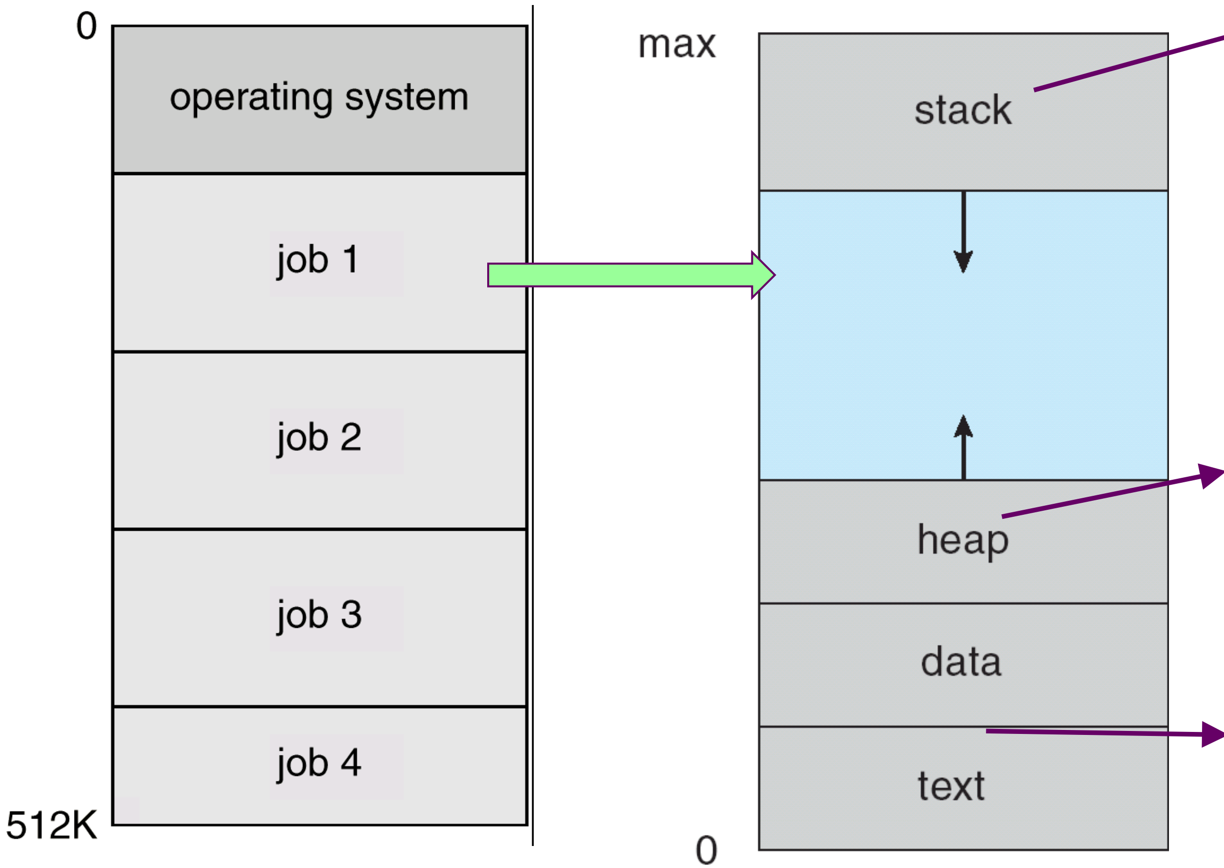
Loading into Memory: From Program To Process



Multiple processes are loaded into main memory

Virtualizing the Memory:
Each process has its own address space

Runtime memory image of a process

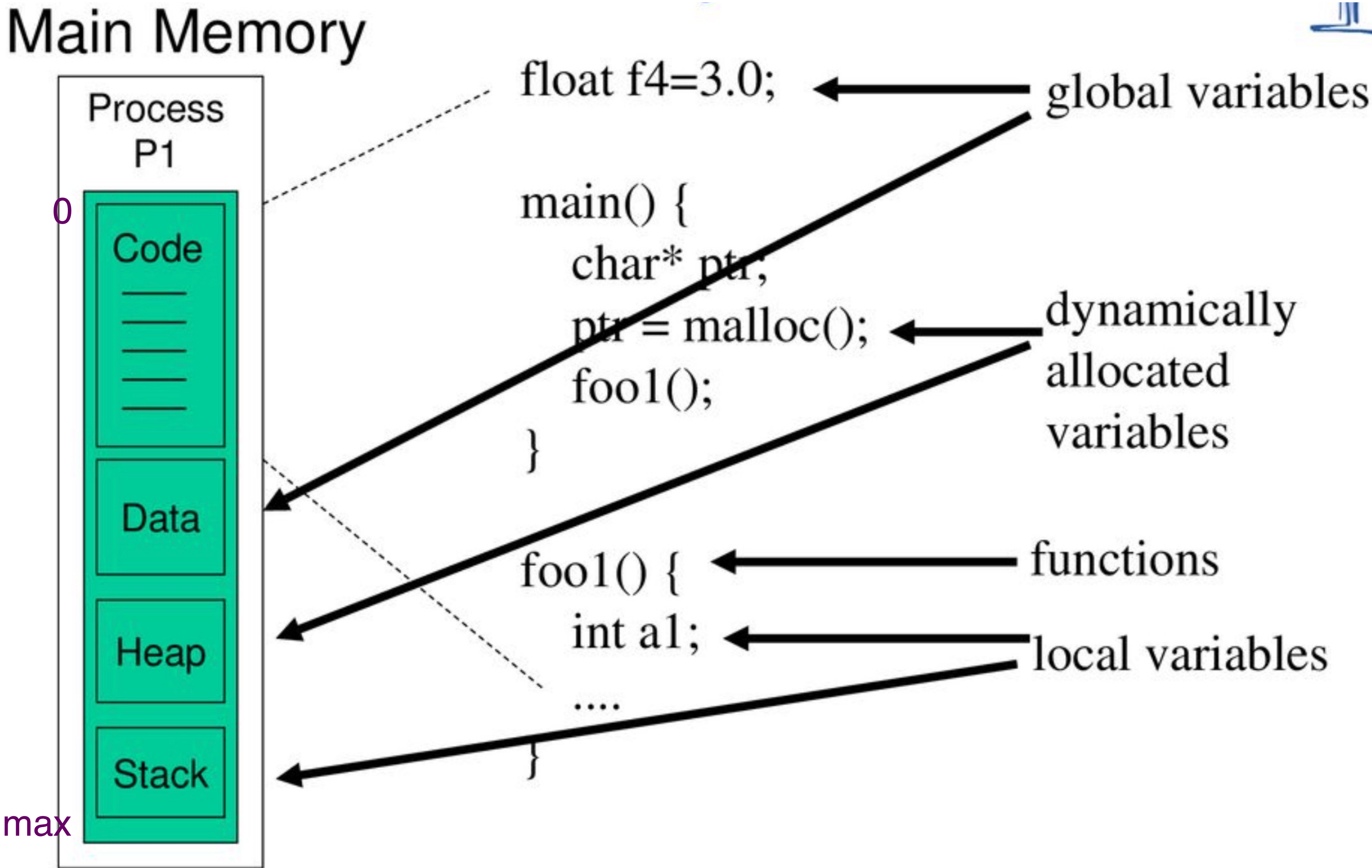


Stack is memory that pushes and pops temporal data during function call

Heap is memory that is dynamically allocated during process run time.

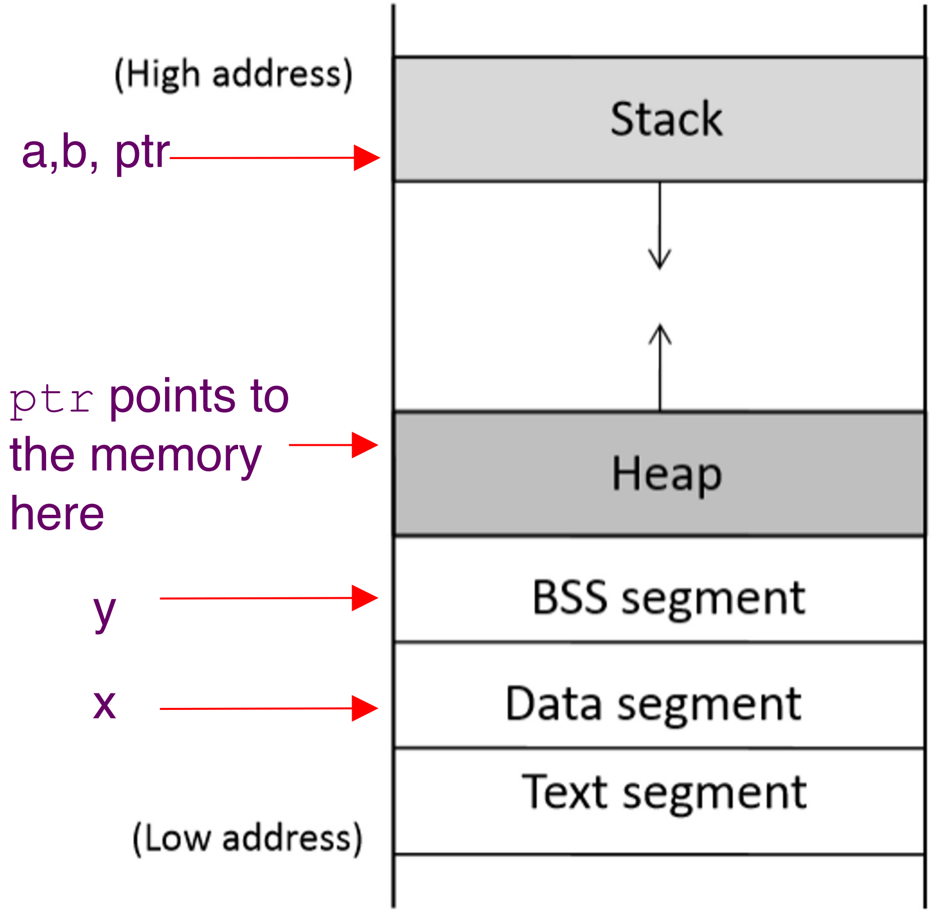
Code and data loaded from executable file

How is a process structured in memory?



Process Memory Layout

```
int x = 100;
int main()
{
   // data stored on stack
   int   a=2;
   float b=2.5;
   static int y;

   // allocate memory on heap
   int *ptr = (int *) malloc(2*sizeof(int));

   // values 5 and 6 stored on heap
   ptr[0]=5;
   ptr[1]=6;

   // deallocate memory on heap
   free(ptr);

   return 1;
}
```



The *data* segment contains any global or static variables which have a pre-defined value and can be modified.

The BSS segment contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

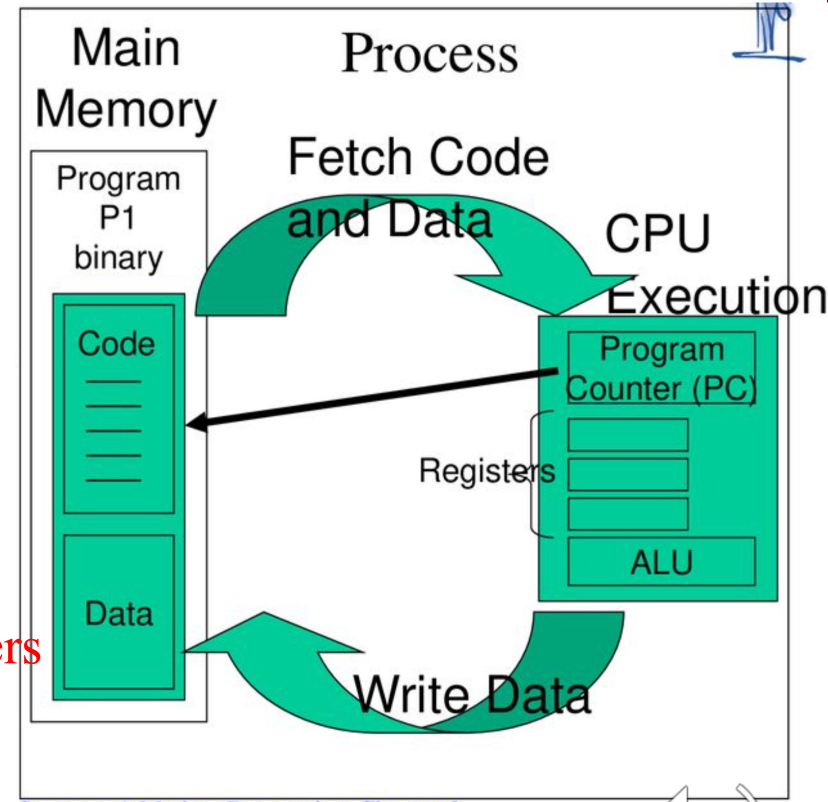
https://en.wikipedia.org/wiki/Data_segment

A Quiz on Process Memory Layout

```
//main.cpp
int a = 1;                      ← 数据段
char *p1=&a;                    ← 数据段
main()
{   int b;                      ← 栈段
    char s[] = "abc";           ← 栈段
    char *p2;                   ← 栈段
    char *p3 = "123456";        ← 栈段
    p1 = (char *)malloc(10);    ← 堆段
    p2 = (char *)malloc(20);    ← 堆段
}
```

Process Concept (Cont.)

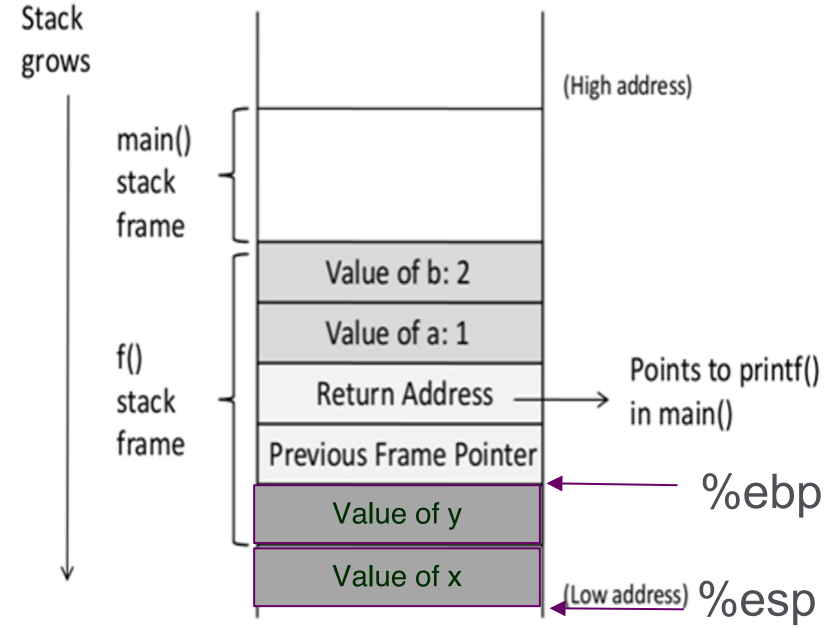
- Process – a program in execution; process execution must progress in sequential fashion.
- The running state of a process includes:
  - Memory
    - ✓ Address space: Instructions and data.
  - Registers
    - ✓ Program counter (PC) / instruction pointer (IP): current instruction.
    - ✓ Stack pointer, frame pointer: management of stack for parameters, local variables and return addresses.
    - ✓ Contents of the processor's other registers
  - I/O information
    - ✓ A list of the files the process currently has open.



Order of the function arguments in stack

```
void f(int a, int b)
{
  int x, y;
  x = a + b;
  y = a - b;
}

void main()
{
  f(1,2);
  printf("hello world");
}
```

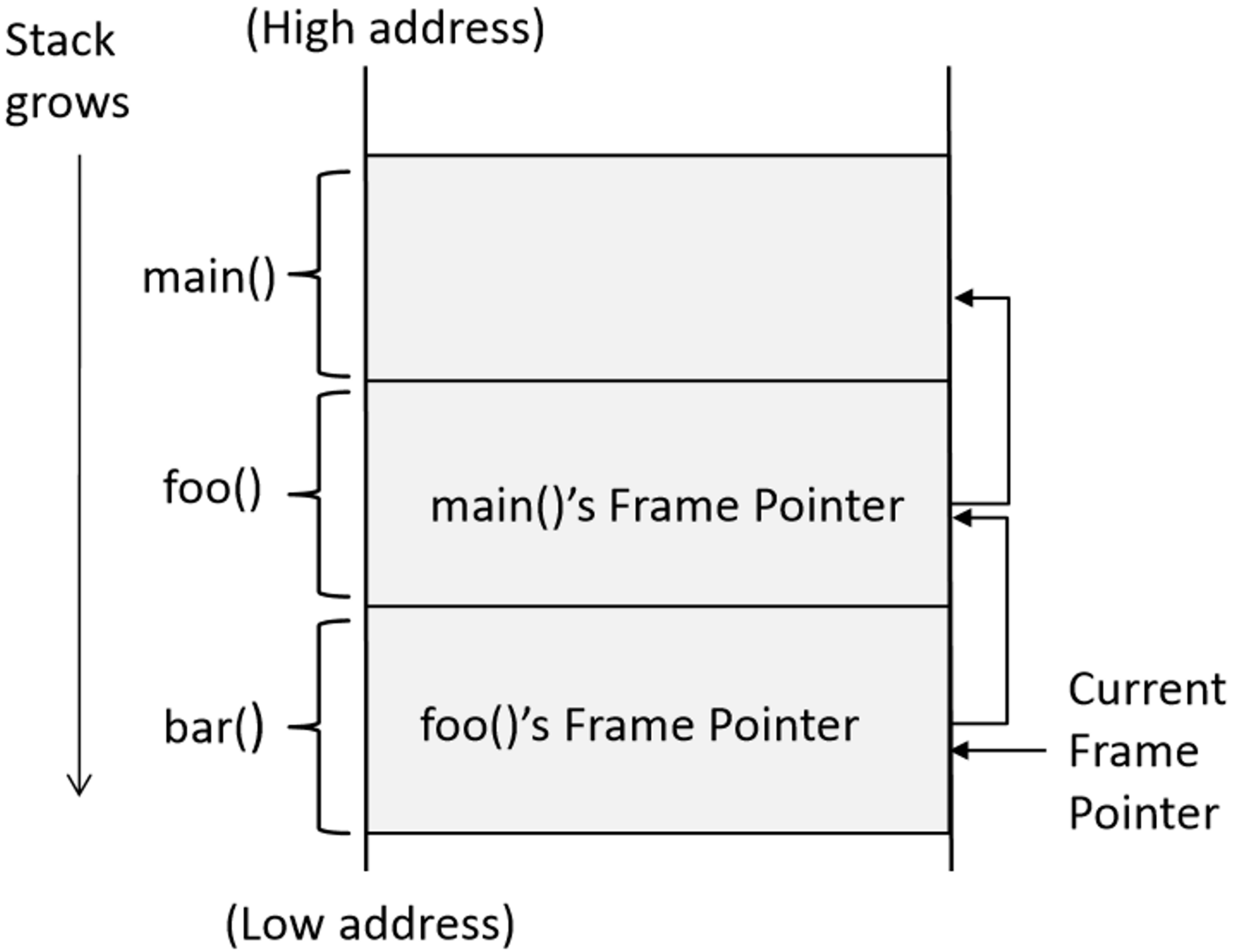
扩展基址指针寄存器(extended base pointer) %ebp：存放一个指针，该指针指向系统栈最上面一个栈帧的底部



void f(int a, int b) 的汇编语言代码

```
movl    12(%ebp), %eax     ; b is stored in %ebp + 12
movl    8(%ebp), %edx      ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)     ; x is stored in %ebp - 8
```

Stack Layout for Function Call Chain



main()

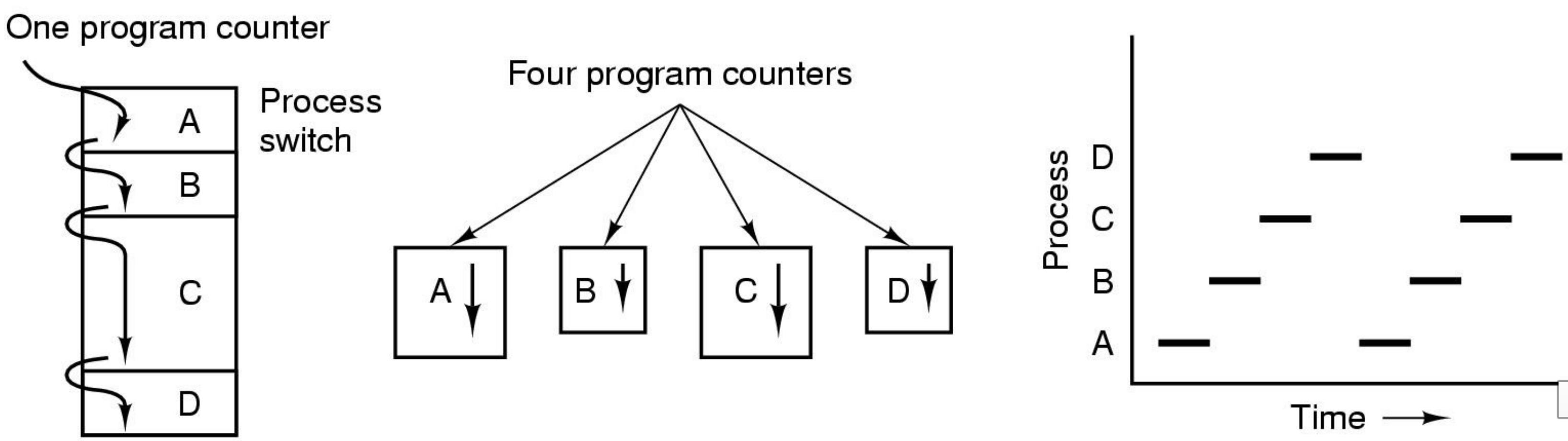
foo()

bar()

Processes

The Process Model

- Virtualizing the CPU:
  - By running one process, then stopping it and running another, and so forth.
- An Example: Multiprogramming of four programs
  - Conceptual model of four independent, sequential processes that can be run in parallel, i.e., figure (b)
  - Only one program active at any instant, i.e., figures (a) and (c)



(a) (b) (c)

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 CPU.
  - **terminated**: The process has finished execution.

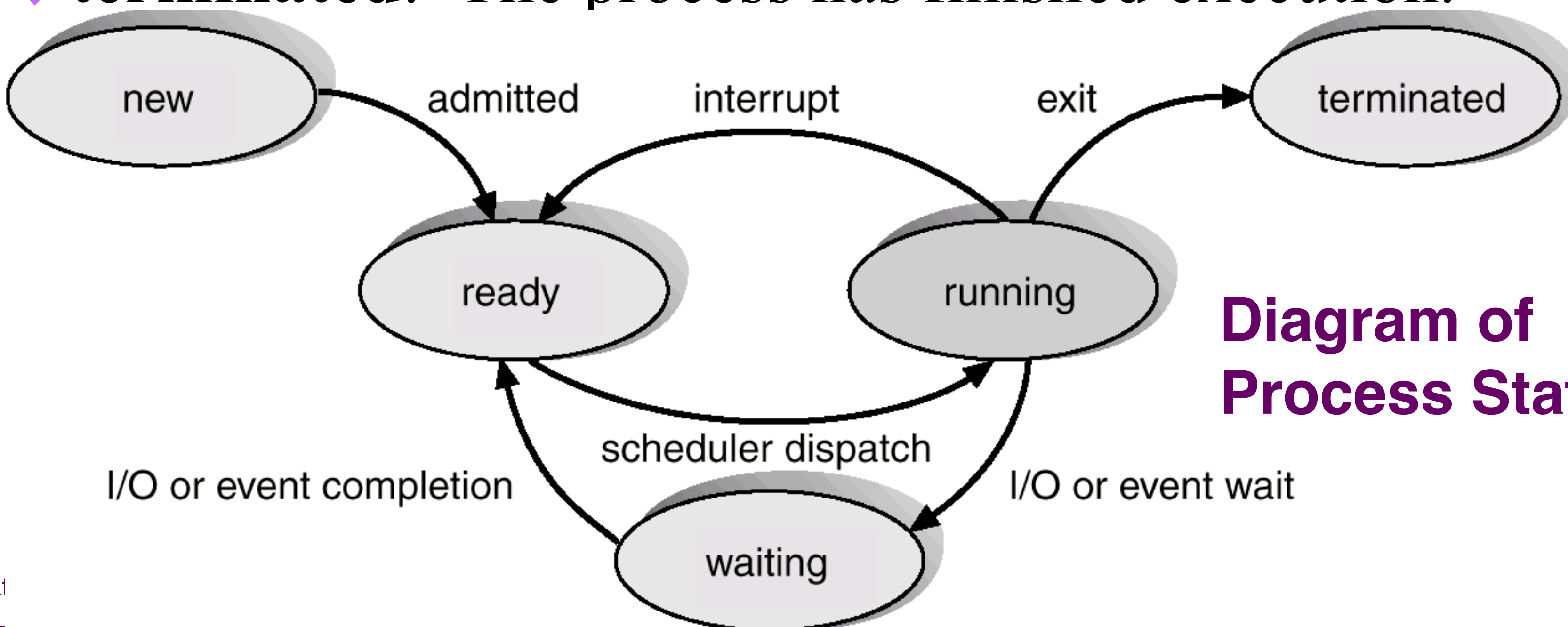
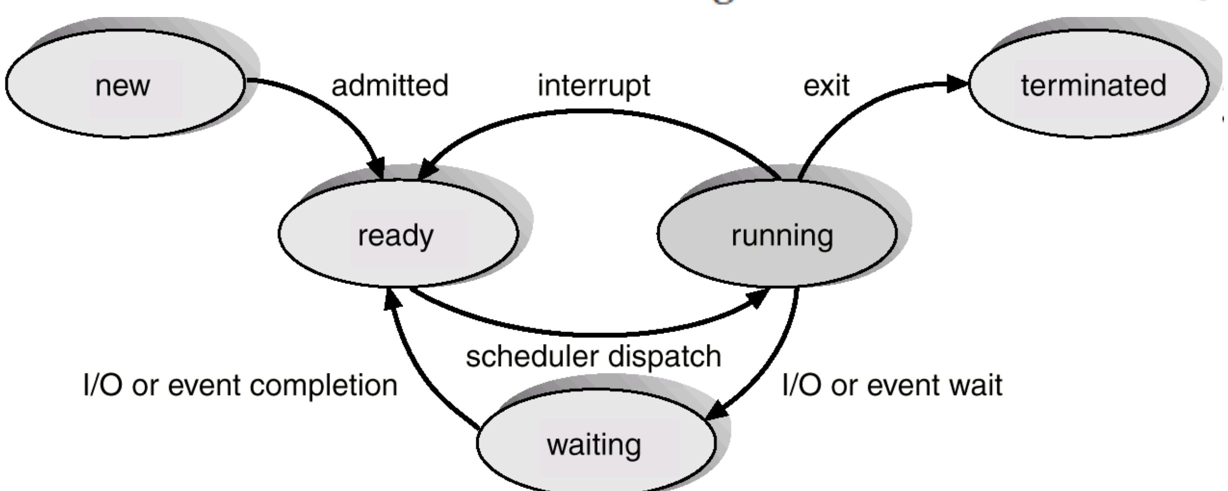


Diagram of Process State

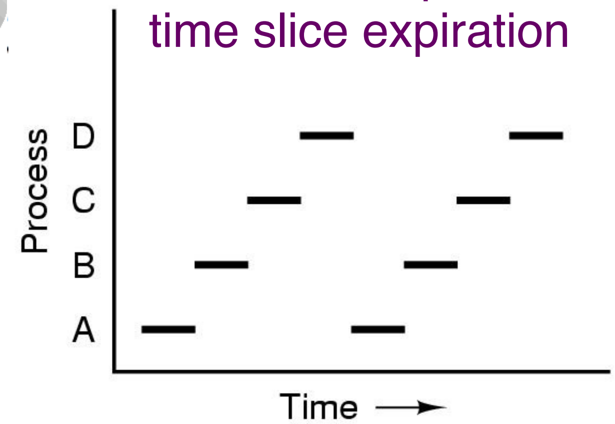
Tracing Process State

- CPU switches from running $Process_0$ to $Process_1$, and then return back to $Process_0$

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done



Timer interrupts due to time slice expiration



Discussion

- Q1: Draw on the blackboard the Diagram of Process State

- Q2: 下列哪一种情况不会引起进程之间的切换？
  A. 进程调用本程序中定义的函数进行计算
  B. 进程处理I/O请求
  C. 进程创建子进程并等待子进程结束
  D. 产生中断

Data Structure

- OS is a software program, so it has some key data structures that track the state of each process.
  - Process lists for all ready / running / waiting processes

- An example: xv6 kernel
  - types of information an OS needs to track processes

```c
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;                 // If non-zero, sleeping on chan
  int killed;                 // If non-zero, have been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;          // Current directory
  struct context context;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```

All registers

Memory: address space

Stack

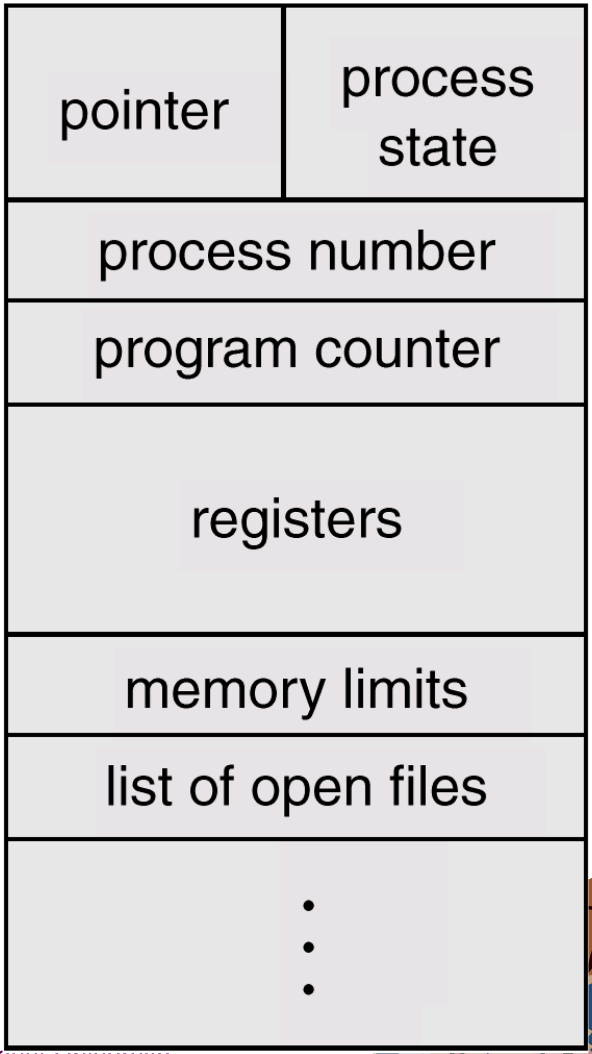
Process state

Process ID

I/O information

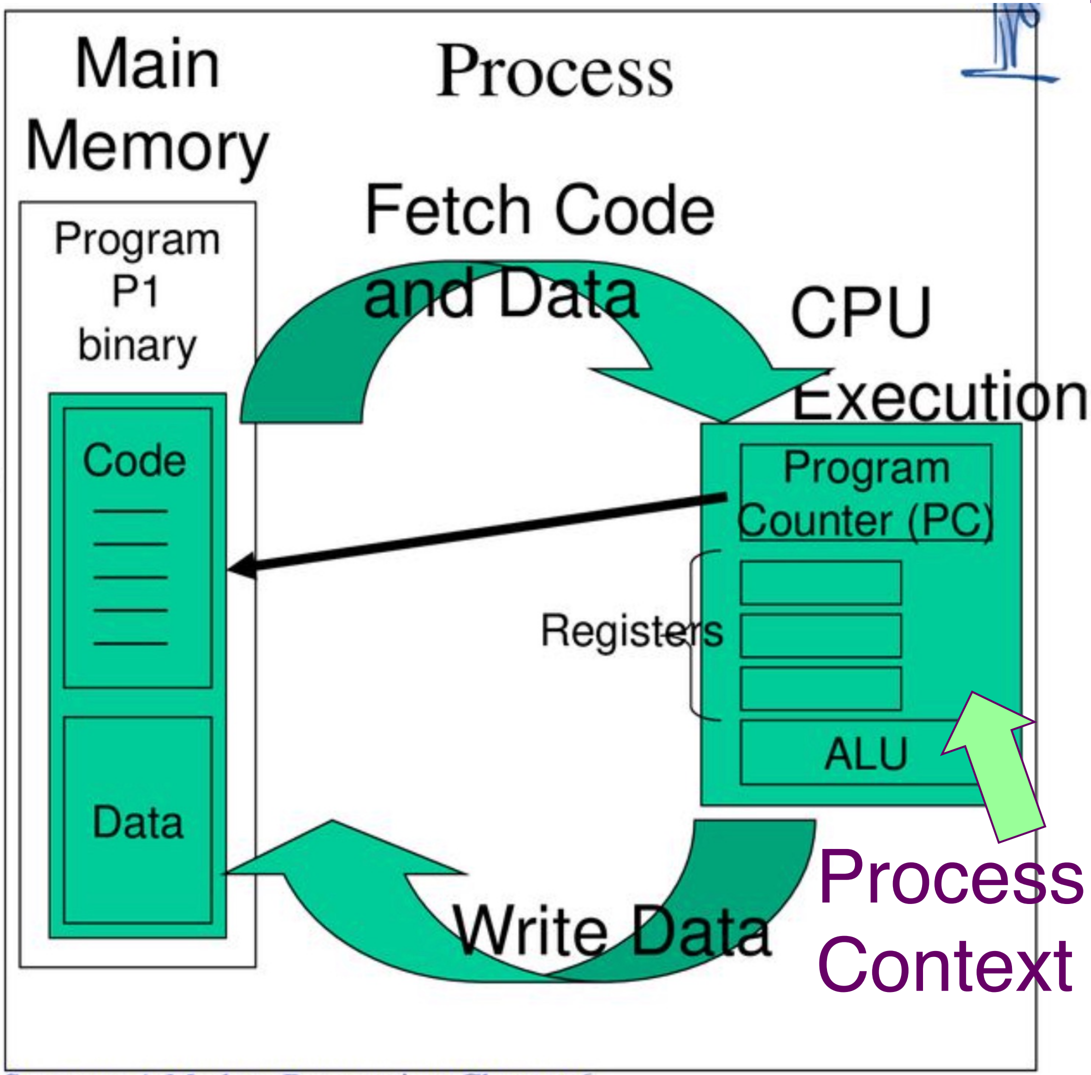
Process Control Block (PCB)

- Information associated with each process.
  - Process state
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - File usage and I/O status information



Context Switch

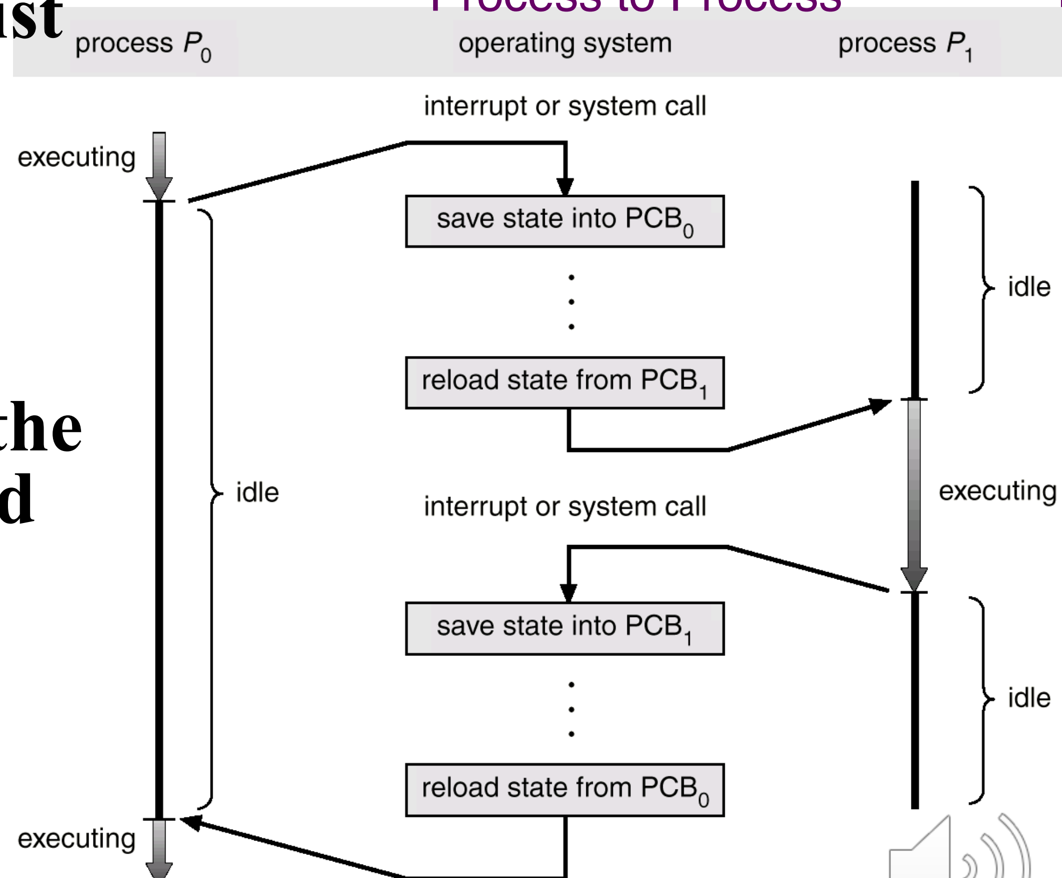
- **What is a process context?**
  - **The *context* of a process includes the values of CPU registers, the process state, the program counter, and other memory/file management information.**



Context Switch (cont.)

- **What is a context switch?**
  - **After the CPU scheduler selects a process (from the *ready queue*) and before allocates CPU to it, the CPU scheduler must**
    - ✓ **save the *context* of the currently running process,**
    - ✓ **put it into a queue,**
    - ✓ **load the *context* of the selected process, and**
    - ✓ **let it run.**

CPU Switch From Process to Process



Context Switch (Cont.)

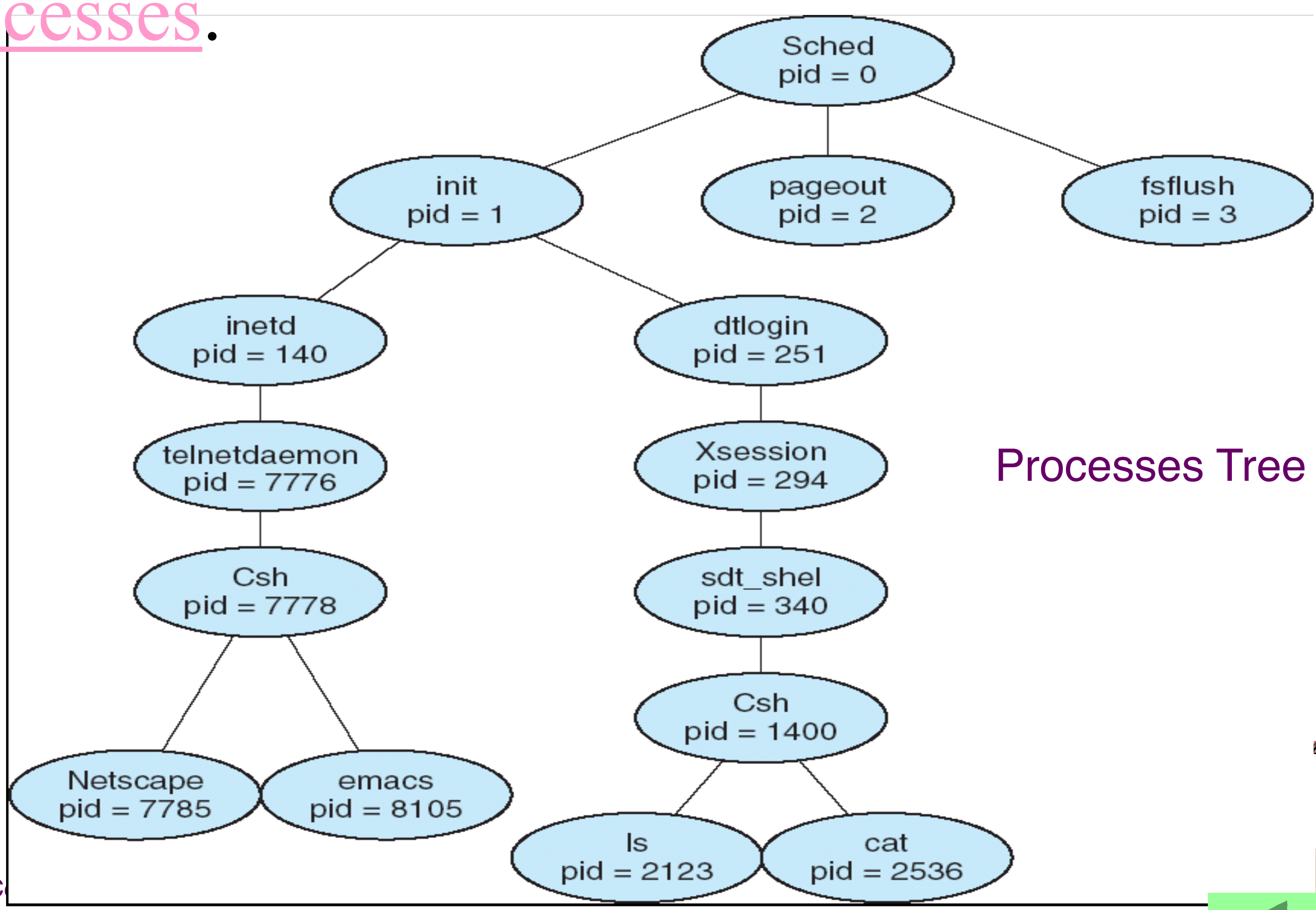
- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process.
- Context-switch time is overhead; the system does no useful work while switching.
- Time dependent on hardware support.

Chapter 3: Processes

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

Process Creation

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



Processes Tree on Solaris

PID and PPID

- Every process except the root process has a parent process ID (PPID), PID of process that spawned it
- An example *ps* dump for a macOS system is shown here (formatted to fit the page):

```
$ ps -faxcel | head -10
  UID   PID  PPID   C STIME   TTY           TIME CMD              F PRI NI      SZ   RSS W
    0     1     0   0 16Feb17 ??      46:46.47 launchd         4004  37  0 2537628 13372 -
    0    51     1   0 16Feb17 ??       1:01.35 syslogd         4004   4  0 2517212  1232 -
    0    52     1   0 16Feb17 ??       2:18.75 UserEventAgent  4004  37  0 2547704 40188 -
    0    54     1   0 16Feb17 ??       0:56.90 uninstalld      4004  20  0 2506256  5256 -
    0    55     1   0 16Feb17 ??       0:08.61 kextd           4004  37  0 2546132 13244 -
    0    56     1   0 16Feb17 ??       2:04.08 fseventsd    1004004  50  0 2520544  6244 -
   55    61     1   0 16Feb17 ??       0:03.61 appleeventsd    4004   4  0 2542188 11320 -
    0    62     1   0 16Feb17 ??       0:07.72 configd         400c  37  0 2545392 13288 -
    0    63     1   0 16Feb17 ??       0:17.60 powerd          4004  37  0 2540644  8016 -
```

Explanations of Useful Process Fields

- The *ps* command receives different options

Name	Type	ps options	Notes
PID	Integer		The process ID for a process
PPID	Integer	`-f`	The parent process ID of a process; i.e., the PID of the process that spawned it
UID	Integer	`-f`	The ID of the user who spawned the process
Command	String		The name of the process
Path	String	`-E`	The path of the process's executable
Memory	Integer(s)	`-l`	The memory used by the application
CPU	Numeric	`-O cpu`	The amount of CPU consumed
Terminal	String	`-f`	The ID of the terminal the process is attached to
Start Time	Date	`-f`	The time the process was invoked

The pstree command

```
% pstree -p 1 | head -15
-+= 00001 root /sbin/launchd
 |--= 00057 root /usr/sbin/syslogd
 |--= 00058 root /usr/libexec/UserEventAgent (System)
 |--= 00061 root
/System/Library/PrivateFrameworks/Uninstall.framework/Resources/uninstalld
 |--= 00062 root
/System/Library/Frameworks/CoreServices.framework/Versions/A/Frameworks/FSEve
nts.framework/Versions/A/Support/fseventsd
 |--= 00063 root
/Library/Frameworks/OVPNHelper.framework/Versions/Current/usr/sbin/ovpnhelper
 |--= 00064 root
/System/Library/PrivateFrameworks/MediaRemote.framework/Support/mediaremoted
 |--= 00066 root /Library/Application Support/CCB_HDZB_UKEY/moniter_CCB_HDZB
 |--= 00068 root
/Library/Frameworks/OpenVPNConnect.framework/Versions/Current/usr/sbin/ovpnag
ent
 |-+= 00069 root /usr/sbin/systemstats --daemon
 | \--- 00883 root /usr/sbin/systemstats --logger-helper
/private/var/db/systemstats
 |--= 00071 root /usr/libexec/configd
 |--= 00072 root endpointsecurityd
 |--= 00073 root /System/Library/CoreServices/powerd.bundle/powerd
 |--= 00076 root /usr/libexec/remoted
```

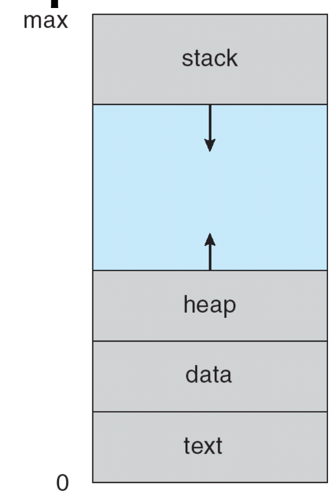
Process Creation (cont.)

- Parent and child may have different styles of sharing resources (e.g. memory address space, open file table)
  1. Parent and children share all resources.
  2. Children share subset of parent's resources.
  3. Parent and child share no resources.

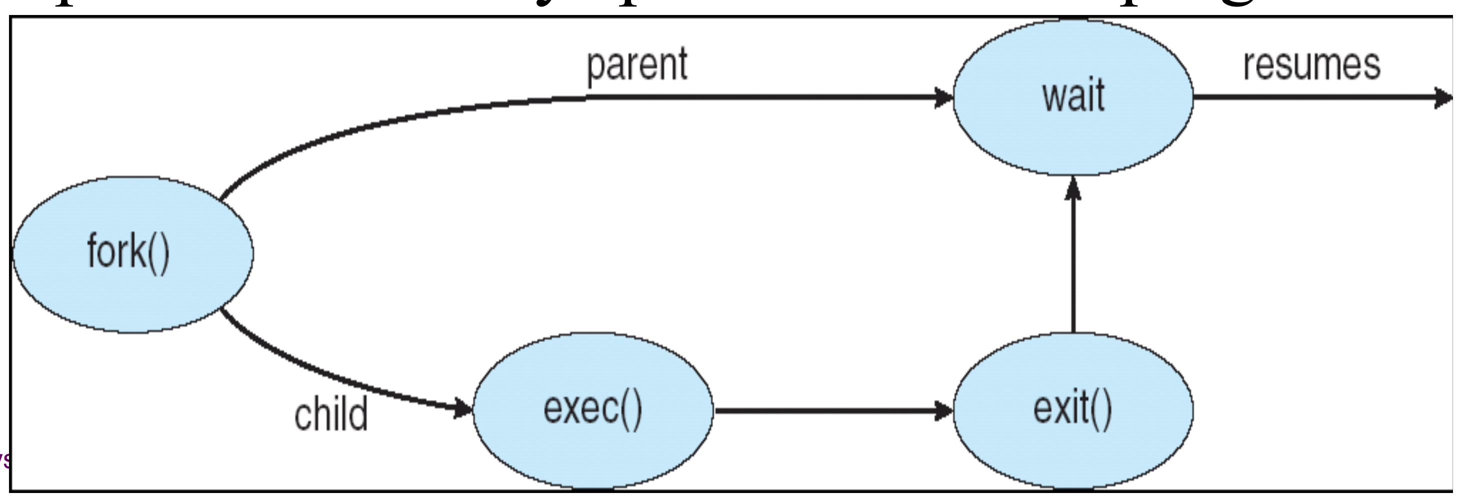
- Execution
  1. Parent and children execute concurrently.
  2. Parent waits until children terminate.

A Typical Way of Process Creation

- Parent and child have separated address spaces
  - 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.



The fork() System Call

- The process that is created by using the fork() system call is an (almost) exact **copy** of the calling process.
  - For parent, fork() returns the process ID of child
  - For child, fork() returns zero

```
int rc = fork();
if (rc < 0) {
    printf(“A");
    exit(1);
} else if (rc == 0) {
    printf(“B");
} else {
    printf(“C");
}
return 0;
```

- Discussion:

what is the output?

The fork() System Call

```
#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 (main)
        printf("hello, I am parent of %d (pid:%d)\n",
                rc, (int) getpid());
    }
    return 0;
}
```

Guess what is the output of the above program?

The fork() System Call

```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 (main)
        printf("hello, I am parent of %d (pid:%d)\n",
                rc, (int) getpid());
    }
    return 0;
}
```

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

ODD?

The fork() System Call

- Discussion: What is the output if we add a loop command before the screen print command?

```
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)
        int sum = 0;
        for (int i = 0; i < 100000000; i ++)
            sum += i;
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {
        // parent goes down this path (original process)
        int sum = 0;
        for (int i = 0; i < 100000000; i ++)
            sum += i;
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
    return 0;
}
```

The fork() System Call

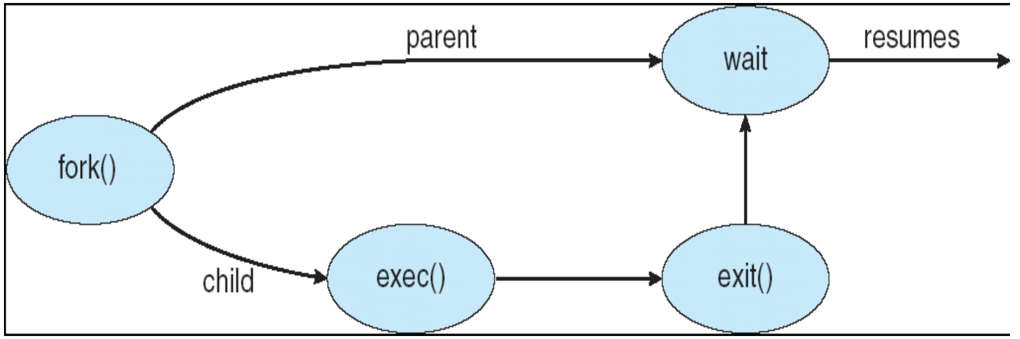
```
Qingjuns-MacBook-Pro:OSC3_code_cpu-api csqjxiao$ ./p1-2
hello world (pid:43349)
hello, I am parent of 43350 (pid:43349)
hello, I am child (pid:43350)
Qingjuns-MacBook-Pro:OSC3_code_cpu-api csqjxiao$ ./p1-2
hello world (pid:43352)
hello, I am child (pid:43353)
hello, I am parent of 43353 (pid:43352)
Qingjuns-MacBook-Pro:OSC3_code_cpu-api csqjxiao$
```

- Discussion: why not deterministic?

Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**).
  - Output data from child to parent (via **wait**).
  - Process' resources are deallocated by OS.
- Parent may terminate execution of children processes (**abort**).
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting.
    - ✓ Operating system does not allow child to continue if its parent terminates.
    - ✓ Cascading termination.

The wait() System Call




```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());
    } else {               // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
                rc, wc, (int) getpid());
    }
    return 0;
}
```

parent waits for child process to finish

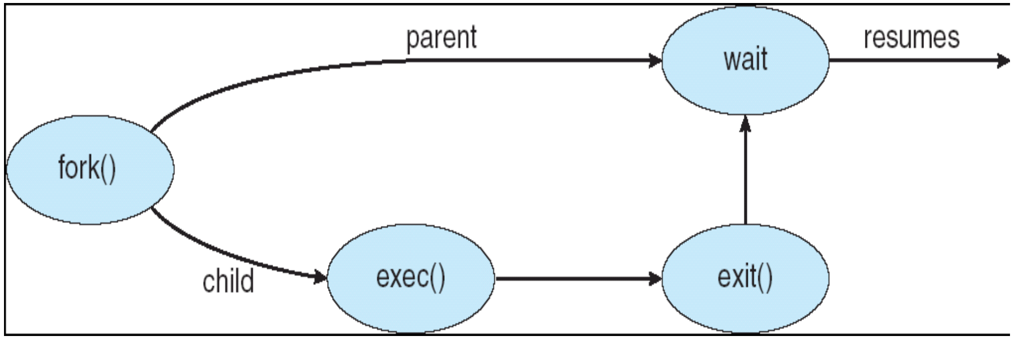
```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```

The exec() System Call

- The process that is created by using the exec() system call can be a different program.
- Some details in exec()
  - It does not create a new process; rather, it transforms the currently running program into a different running program.

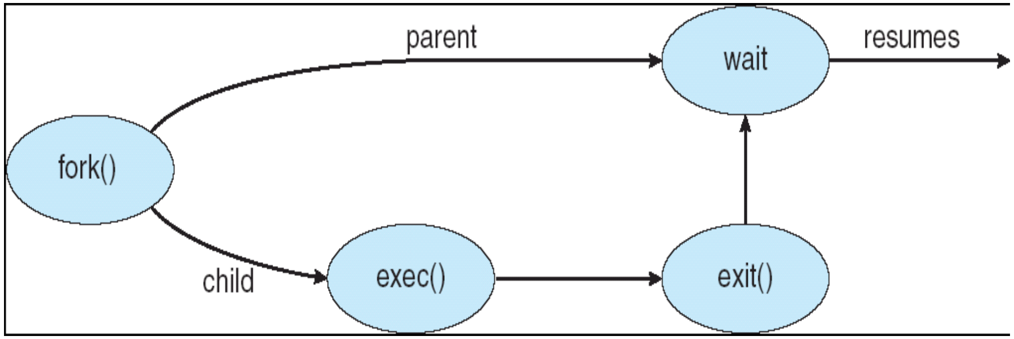
The exec() System Call

```
#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] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("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");
    } else {               // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
                rc, wc, (int) getpid());
    }
    return 0;
}
```



Guess what is the output of the above program?

The exec() System Call




```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] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("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");
    } else {               // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
                rc, wc, (int) getpid());
    }
    return 0;
}
```

```
hello world (pid:29383)
hello, I am child (pid:29384)
       29      107     1030 p3.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
```

Review

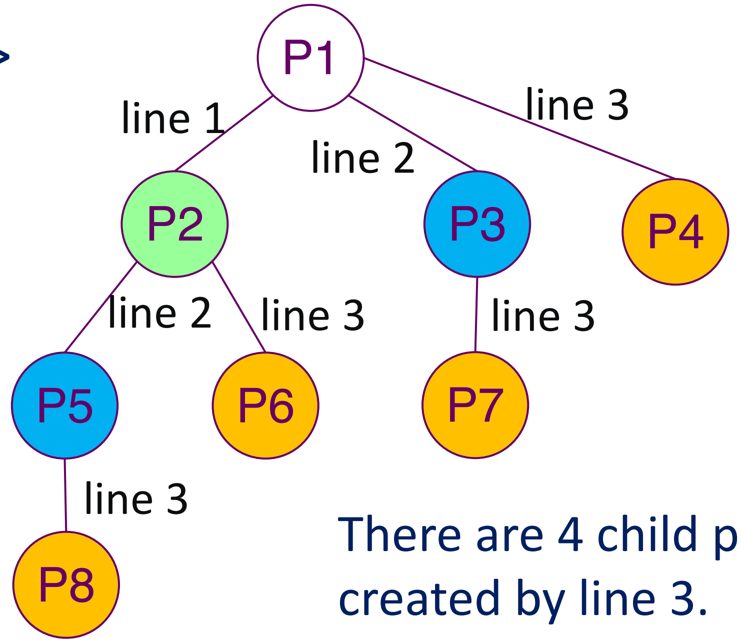
- Process creation APIs
  - fork()
  - wait()
  - exec()

  - What are the differences?

An initial example for fork() problem

- Calculate number of times hello is printed.

```
#include  <stdio.h>
#include  <sys/types.h>
int main()
{
   fork();   // line 1
   fork();   // line 2
   fork();   // line 3
   printf("hello\n");
   return 0;
}
```



There is 1 child process created by line 1.

There are 2 child processes created by line 2.

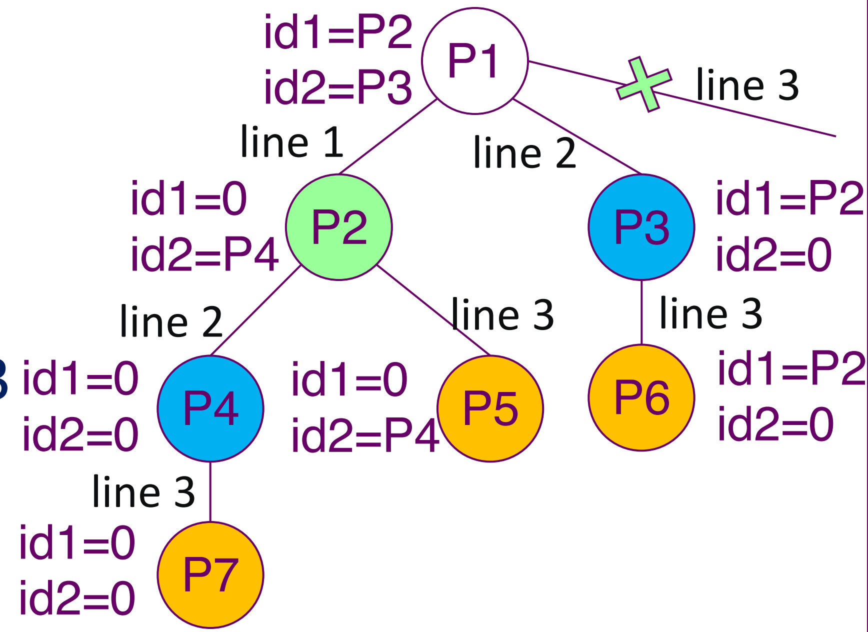
There are 4 child processes created by line 3.

- Number of times hello printed is equal to number of process created.
- Total **Number of Processes = 2^n** where n is number of fork system calls. Here $n = 3$, $2^3 = 8$.

Quiz about the fork() problem

- Consider the following C program. Guess how many lines of output will be printed.

```
int main(int argc, char * argv[])
{
   int id1, id2;
   id1 = fork(); // line 1
   id2 = fork(); // line 2
   if (id1 == 0 || id2 == 0) fork(); // line 3
   printf("I am %d\n", getpid());
}
```

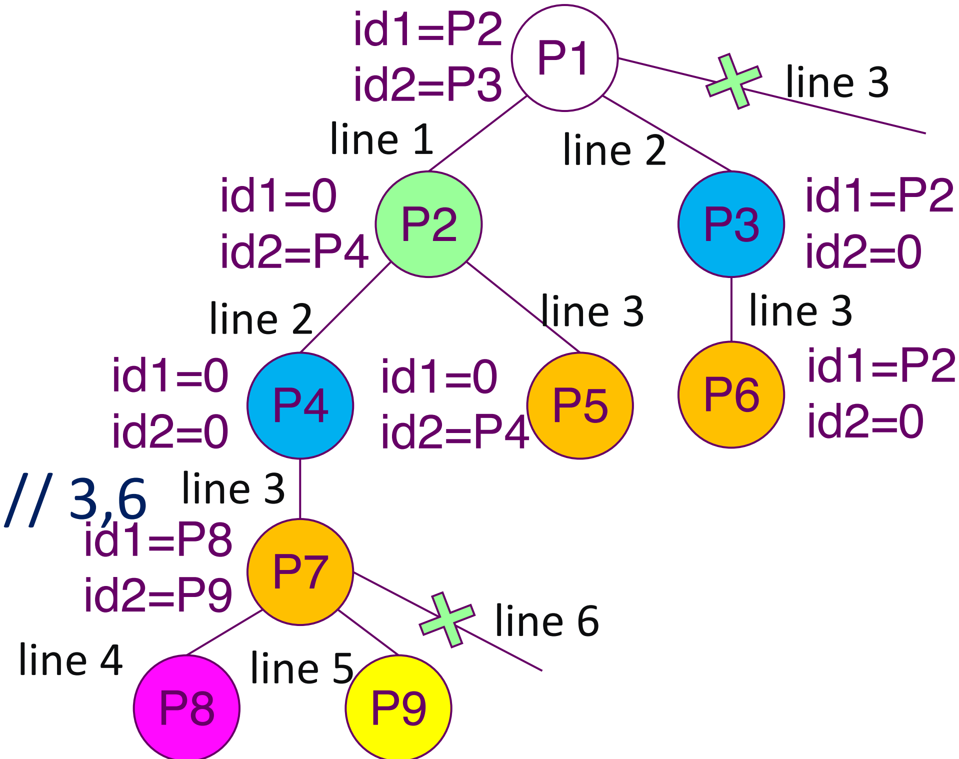


- There are $n = 3$ forks. The \`line 3\` branch of P1 is trimmed. So $2^3 - 1 = 7$ processes are created.

An Extended Quiz

- What if we use a loop with two iterations?

```
int main(int argc, char * argv[])
{
   int id1, id2;
   for (int i = 0; i < 2; i++) {
      id1 = fork(); // line 1,4
      id2 = fork(); // line 2,5
      if (id1 == 0 || id2 == 0) fork(); // 3,6
   }
   printf("I am %d\n", getpid());
}
```



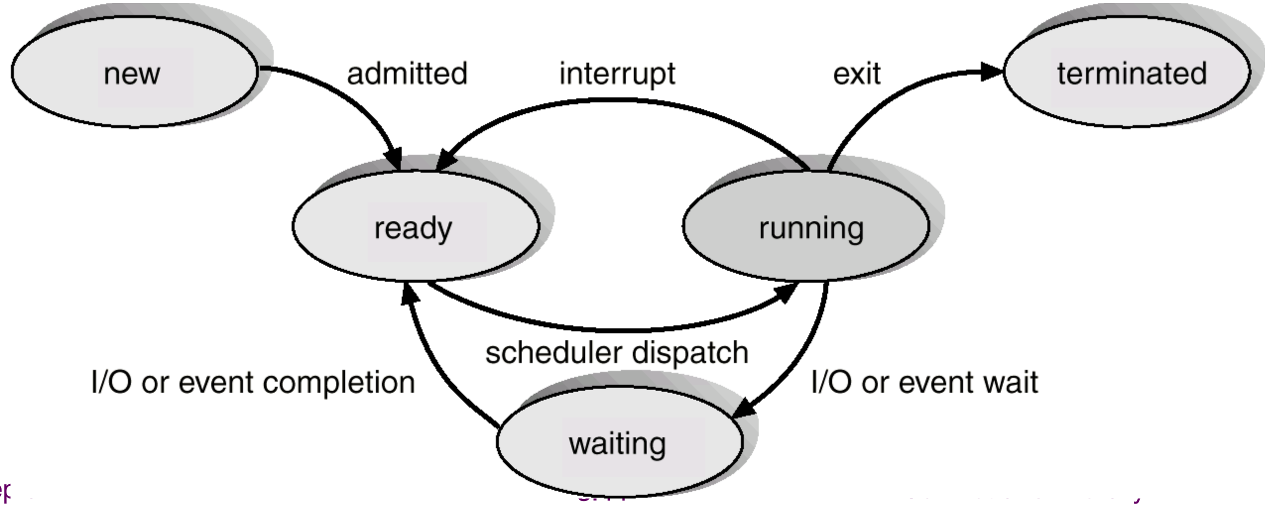
- There are $n = 6$ forks. The `line 3` branch of P1 is trimmed. The `line 6` branches of 7 processes are trimmed. Each of P1-P7 spawns a 7-process tree. So $(2^3-1)^2 = 49$ processes are created.

Chapter 3: Processes

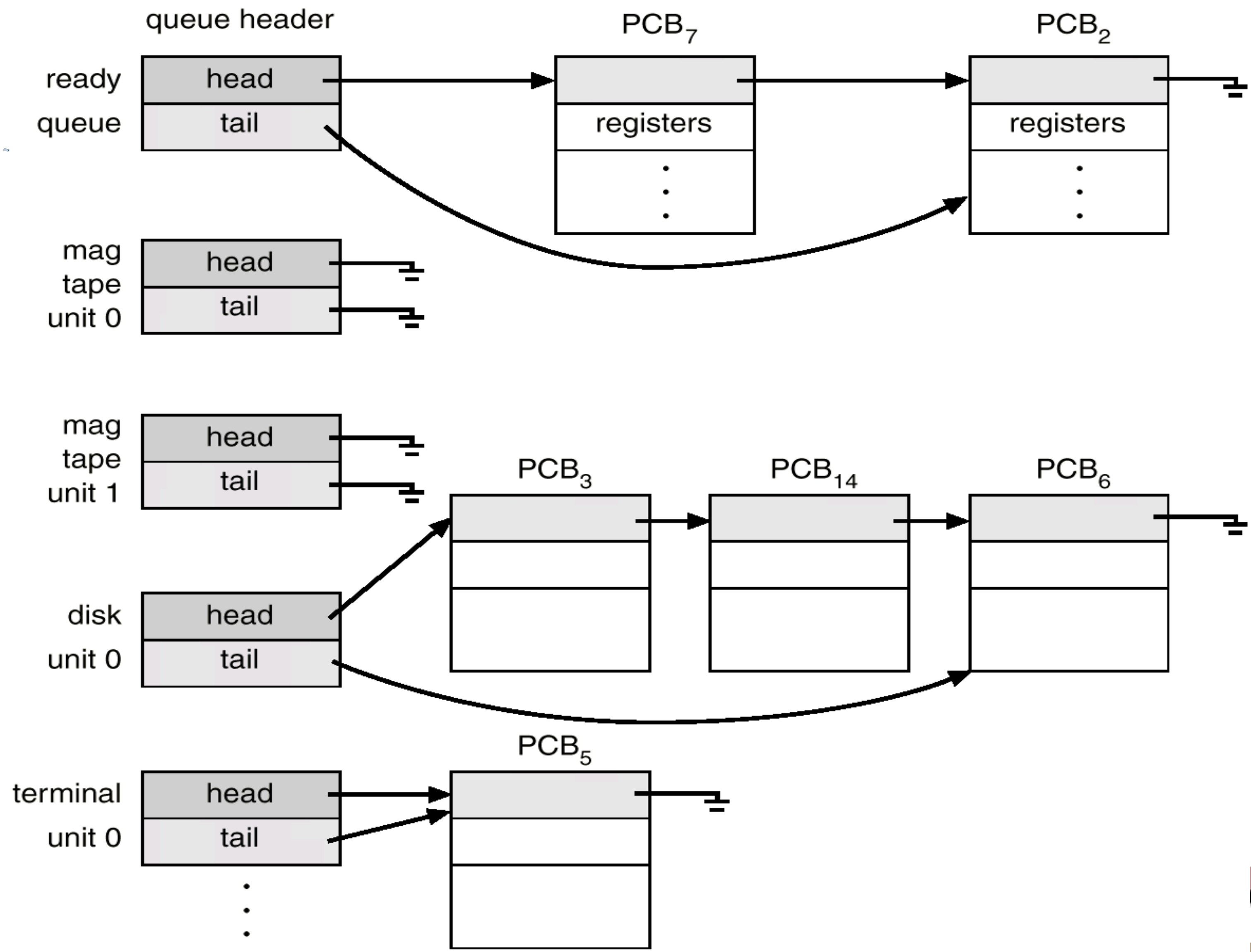
- Process Concept
- Operations and APIs on Processes
- Process Scheduling
- Cooperating Processes
- Interprocess Communication
- Communication in Client-Server Systems

Process Scheduling Queues

- Job queue – 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.
- Process migration between the various queues.

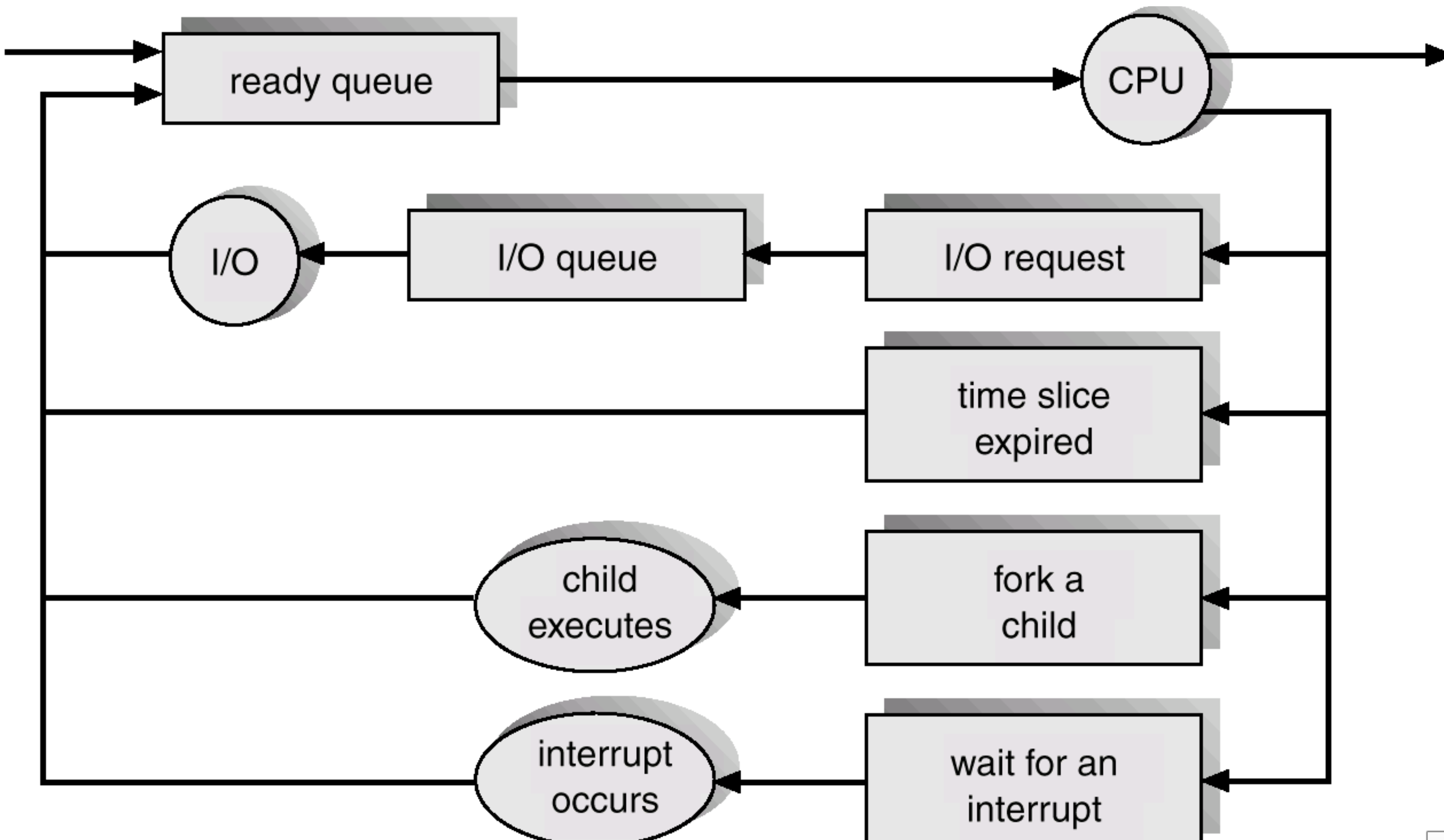


Ready Queue And Various I/O Device Queues



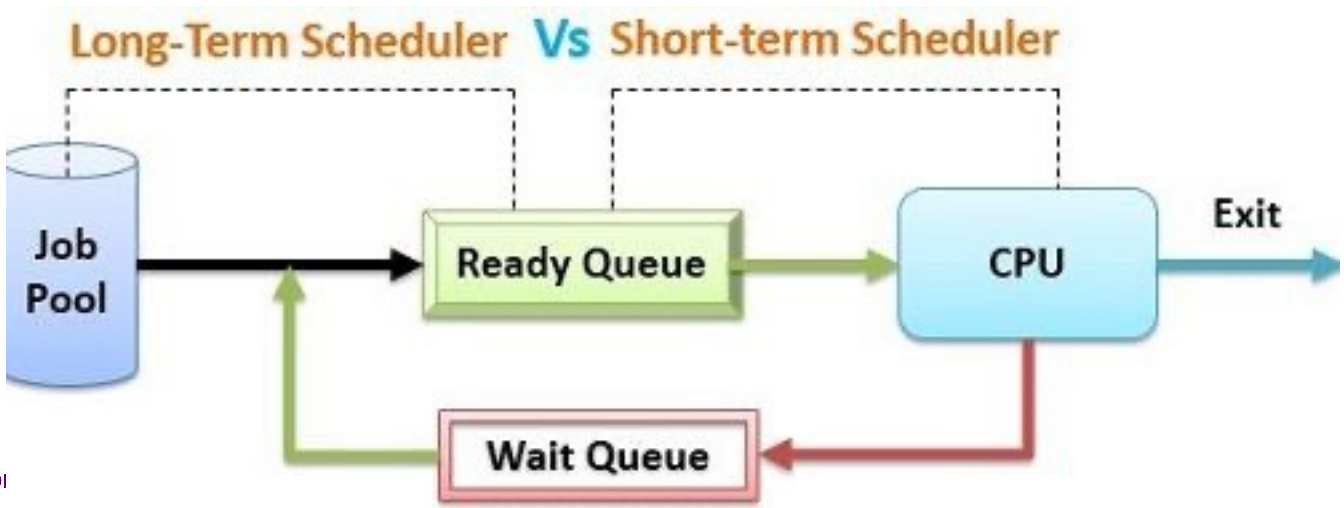
Representation of Process Scheduling

Resource



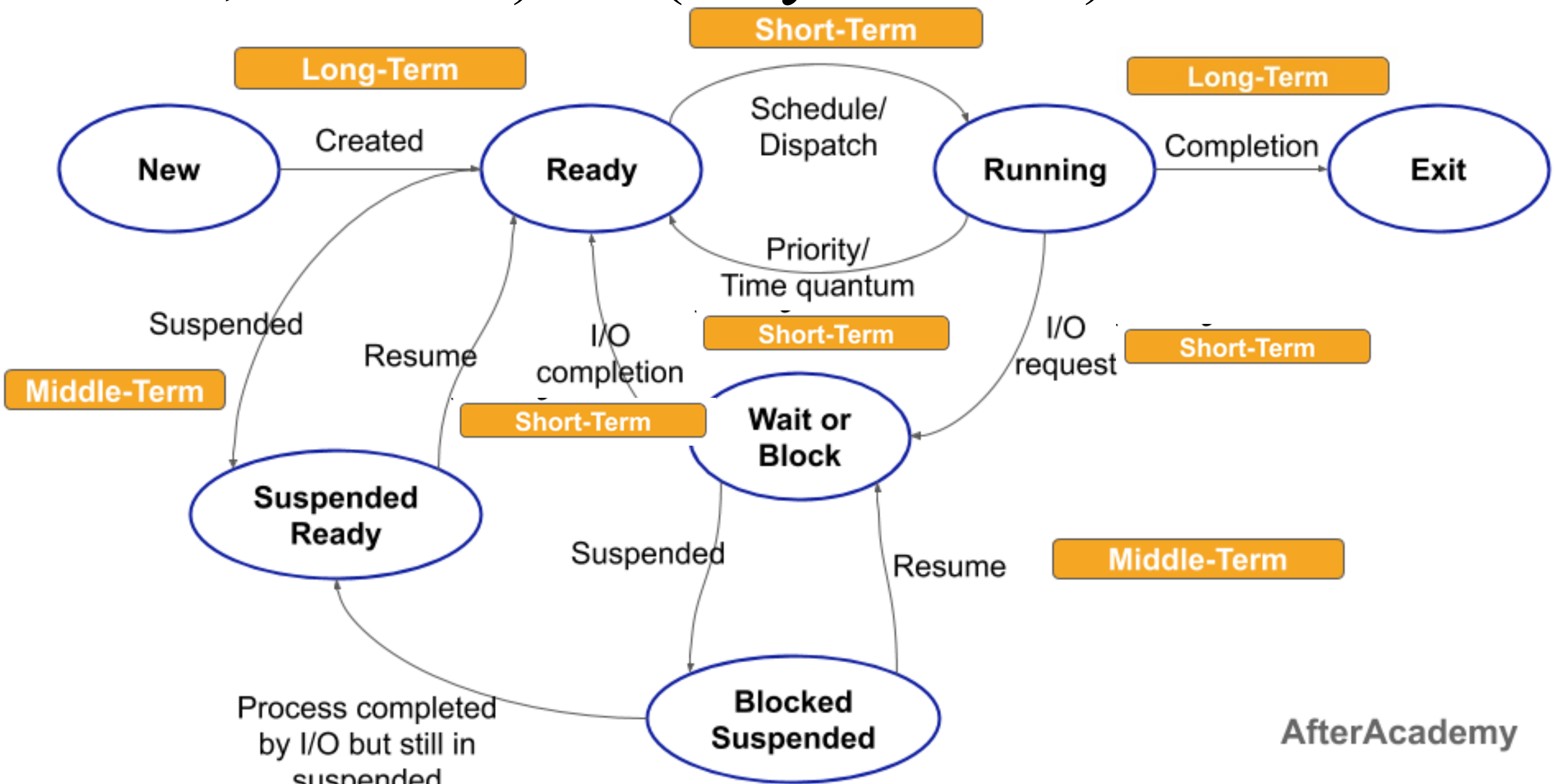
Schedulers

- Long-term scheduler (or job scheduler) – selects which processes should be loaded into memory for execution.
- 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) ⇒ (must be fast).
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow).



Schedulers (Cont.)

- The long-term scheduling performs a **gatekeeping function**. It decides whether there's enough memory, or room, to allow new programs into the system.

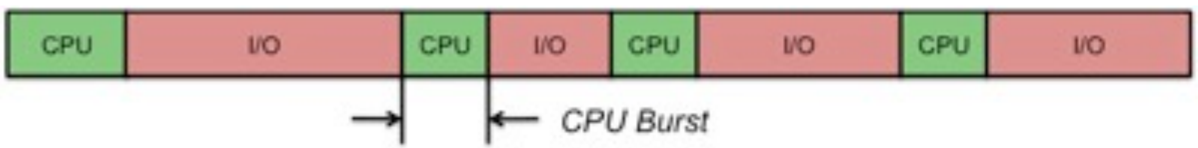
- The long-term scheduler controls the *degree of multiprogramming*.

- Short-term scheduler is affected by _____ processes
  - running;
  - ready;
  - blocked;

- Long-term scheduler is affected by __ processes
  - new;
  - exited;

I/O-bound vs. CPU-bound Processes

- The period of computation between I/O requests is called the **CPU burst**.

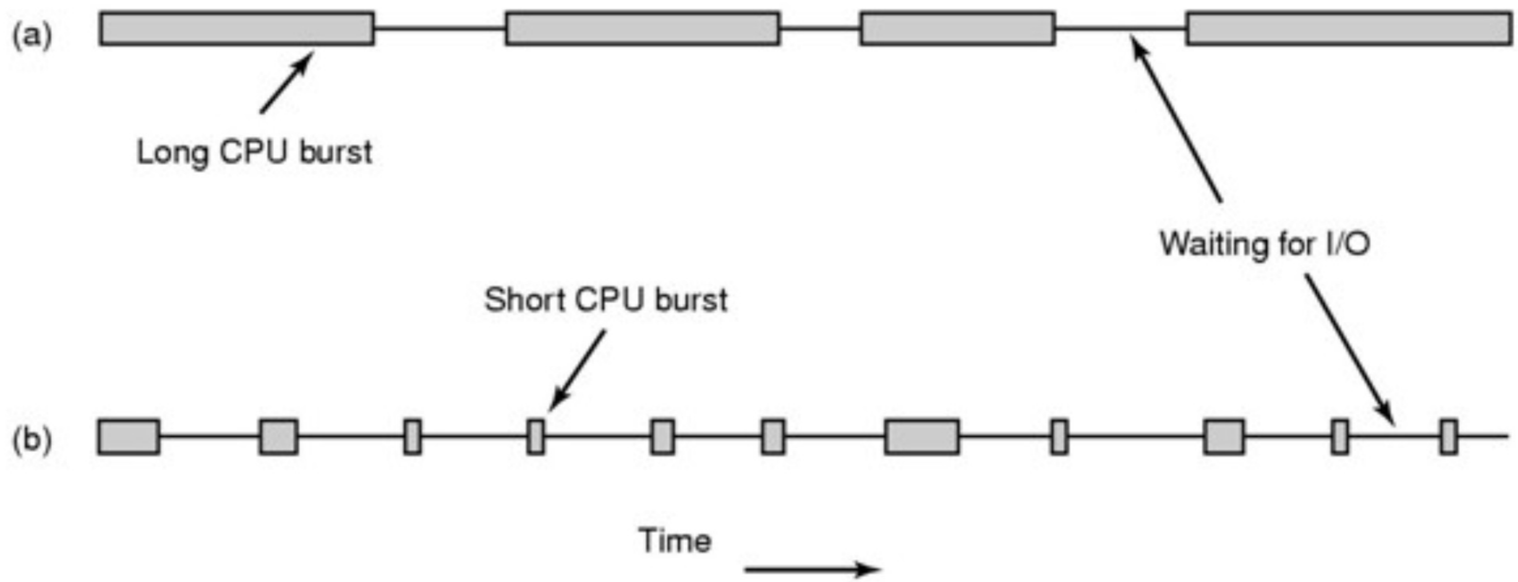


- Processes can be described as either:
  - *I/O-bound process* – spends more time doing I/O than computations, many short CPU bursts.

  - *CPU-bound process* – spends more time doing computations; few very long CPU bursts.

I/O-bound vs. CPU-bound Processes

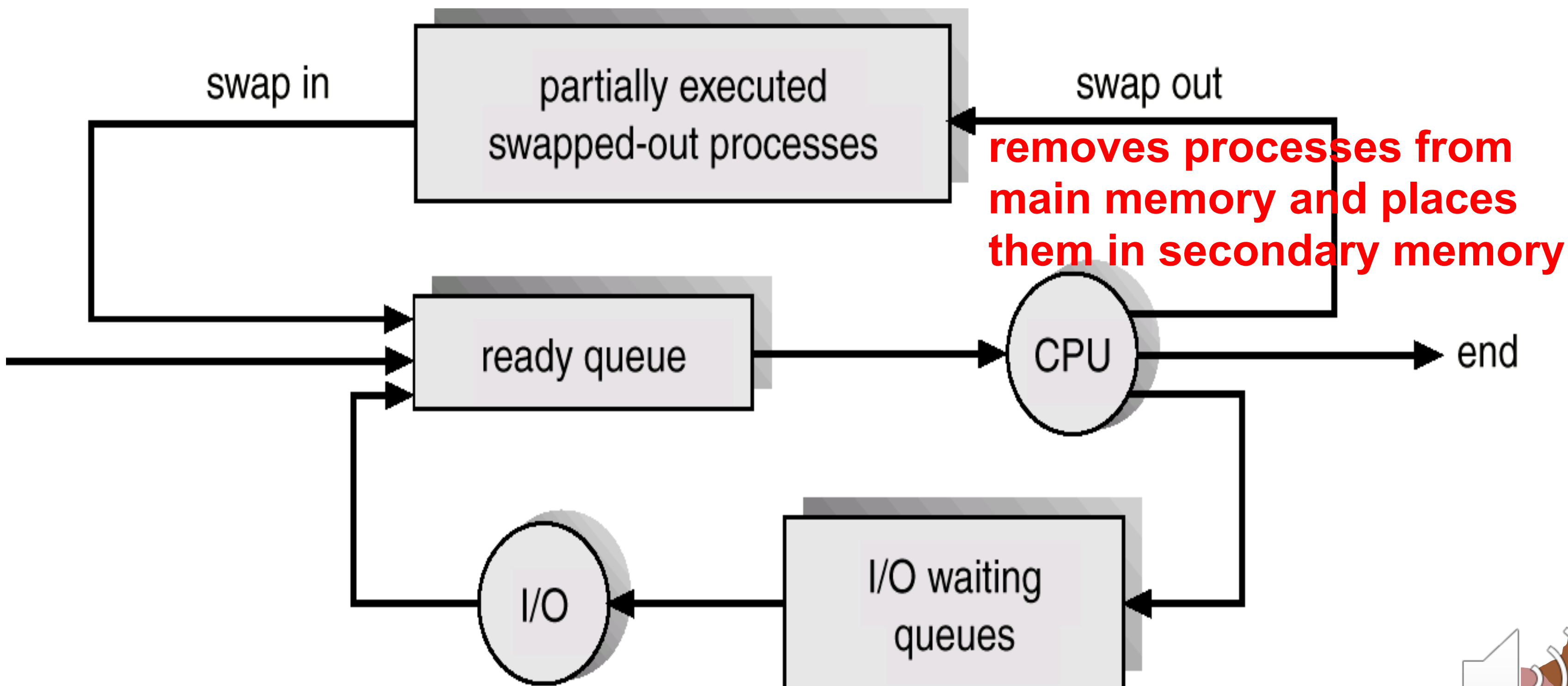
- **Discussion**: If you design a CPU scheduler, which type of processes will you give a higher priority of granting CPU resource? CPU-bound processes (a), or I/O-bound processes (b)?



- Which process should have preferred access to CPU
- Why?

Addition of Medium-Term Scheduling

- The resource needs of a process may vary during its runtime. When the system resources become insufficient, some processes may need to swap out



http://en.wikipedia.org/wiki/Scheduling_(computing)#Medium-term_scheduling

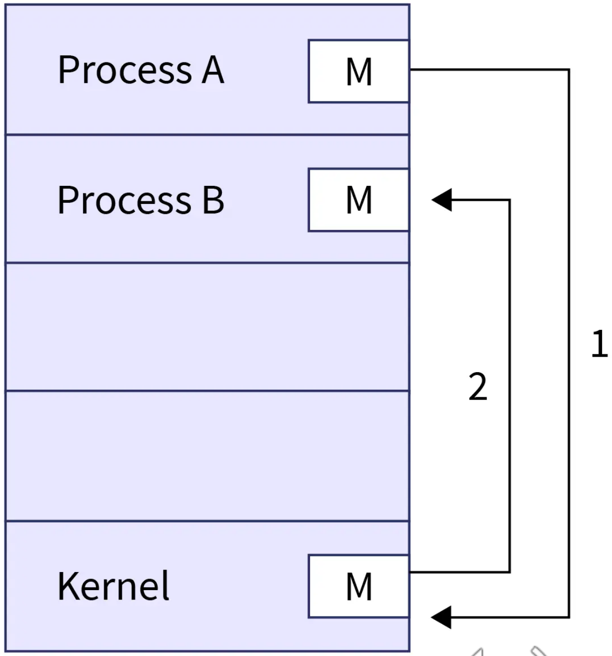
Chapter 3: Processes

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

Cooperating Processes

- *Independent* process cannot affect or be affected by the execution of another process.

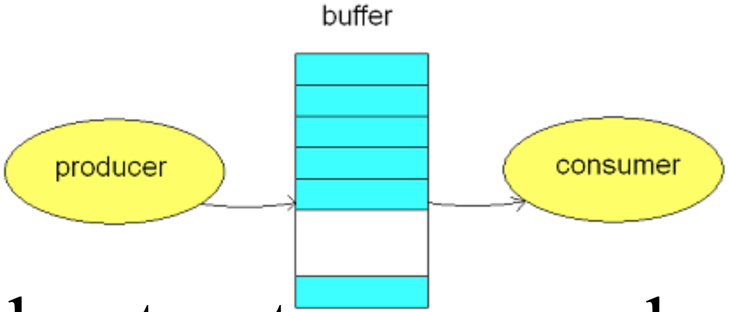
- *Cooperating* process can affect or be affected by the execution of another process



- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

A Common Cooperating Pattern: Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process.



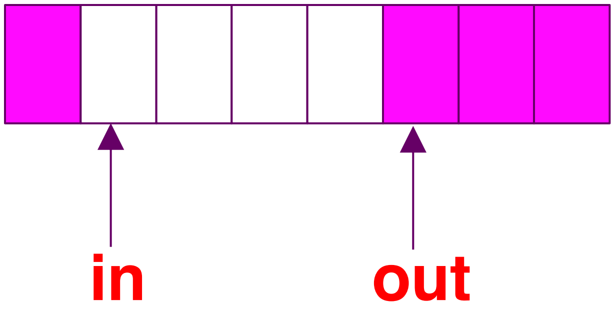
- A buffer is used to hold not-yet-consumed products (for example, kafka for unbounded buffer, redis for bounded buffer)
  - *unbounded-buffer* places no practical limit on the size of the buffer, e.g., a buffer on disk with large space
  - *bounded-buffer* assumes that there is a fixed buffer size, e.g., a buffer in main memory with limited space

Bounded-Buffer – Share-memory Solution

Shared Data

```
#define BUF_LEN 10
Typedef struct {
    . . .
} item;
item buffer[BUF_LEN];
int in = 0, out = 0;
```

Circular queue



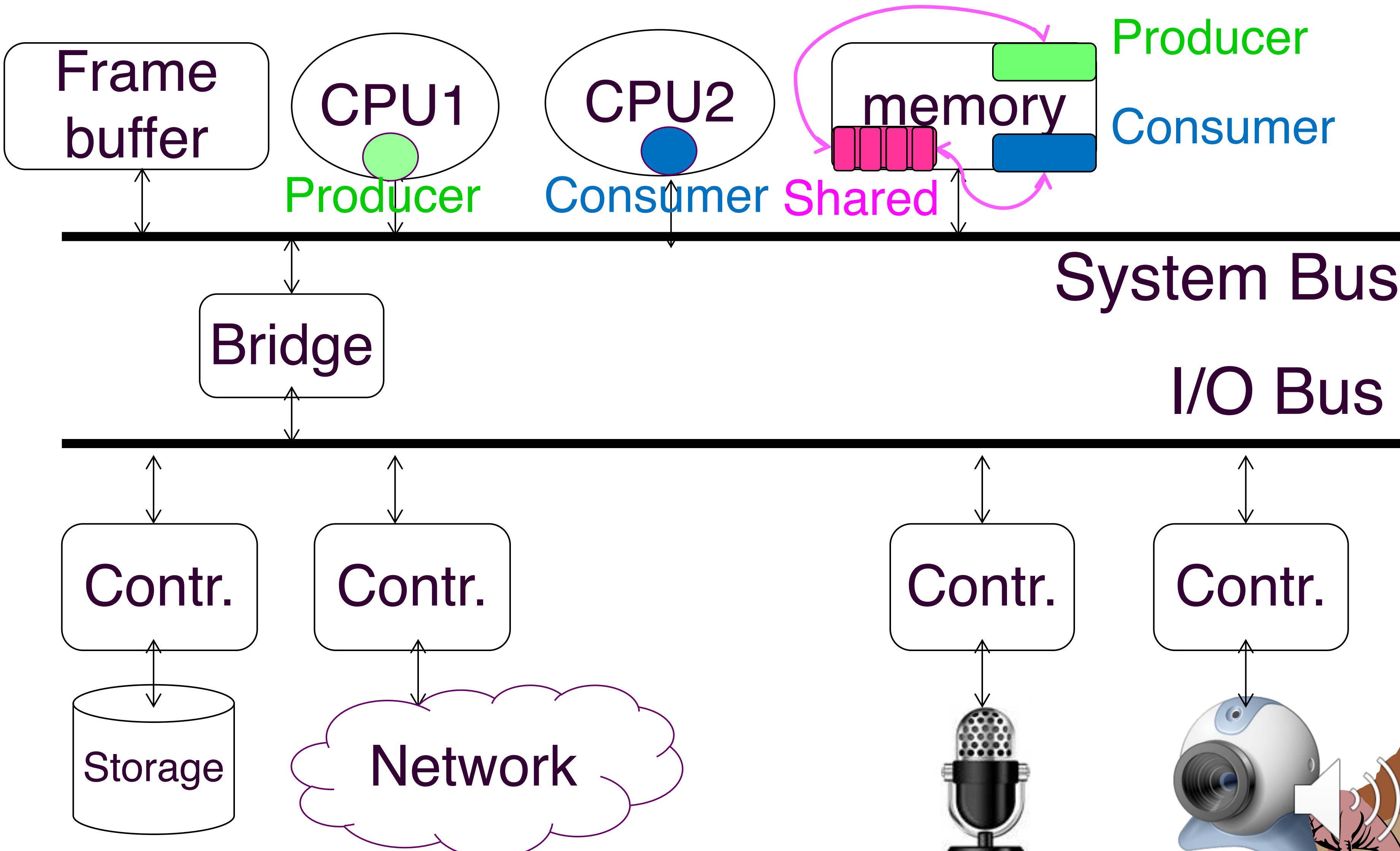
Producer Process

```
item nextProduced;
while (1) {
  while (((in+1)%BUF_LEN) == out)
    ; /* do nothing */
  buffer[in] = nextProduced;
  in = (in + 1) % BUF_LEN;
}
```

Consumer Process

```
item nextConsumed;
while (1) {
  while (in == out)
      ; /* do nothing */
  nextConsumed = buffer[out];
  out = (out + 1) % BUF_LEN;
}
```

Implementation of Communication Link by Shared Memory



Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions.

- Message-passing system – processes communicate with each other without resorting to shared variables.

Interprocess Communication (Cont.)

- 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., logical properties)

Implementation Questions

- 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?

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.
  - Two proc 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
  - 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

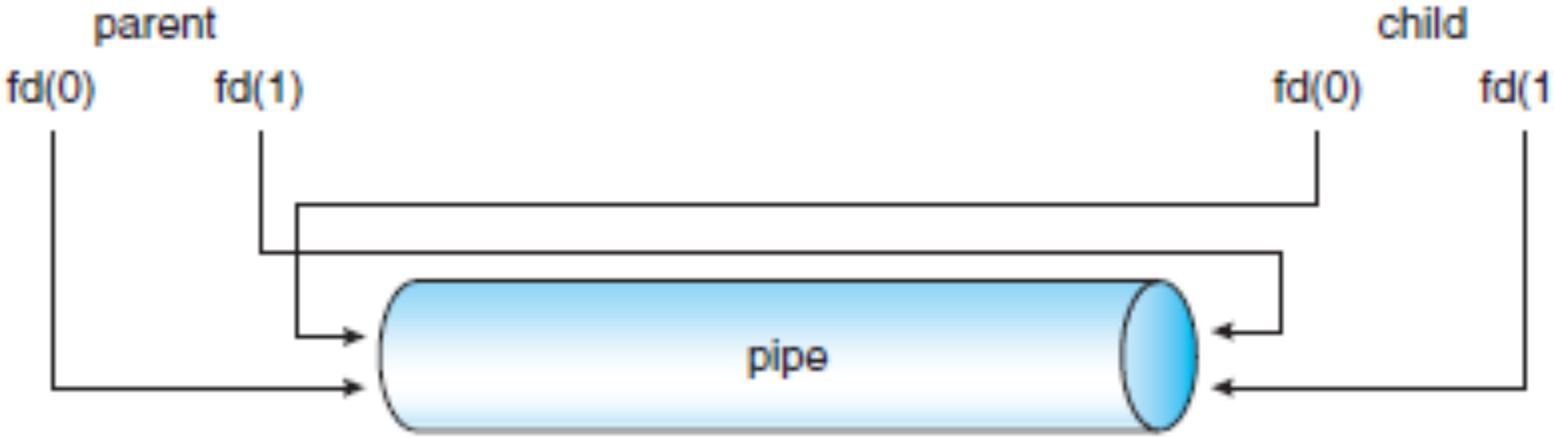
- Message passing may be either blocking or non-blocking.
- **Blocking** is considered **synchronous**
- **Non-blocking** is considered **asynchronous**
- **send** and **receive** primitives may be either blocking or non-blocking.

Buffering

- Queue of messages attached to the link; implemented in one of three ways.
  1. Zero capacity – 0 messages
     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 blocks.

Pipes in Unix

- UNIX pipes are implemented in a similar way, but with the **pipe**() system call.
  - The output of one process is connected to an in-kernel pipe.
  - The input of another process is connected to that same pipe.
  - E.g.,
    - ✓ ls | wc

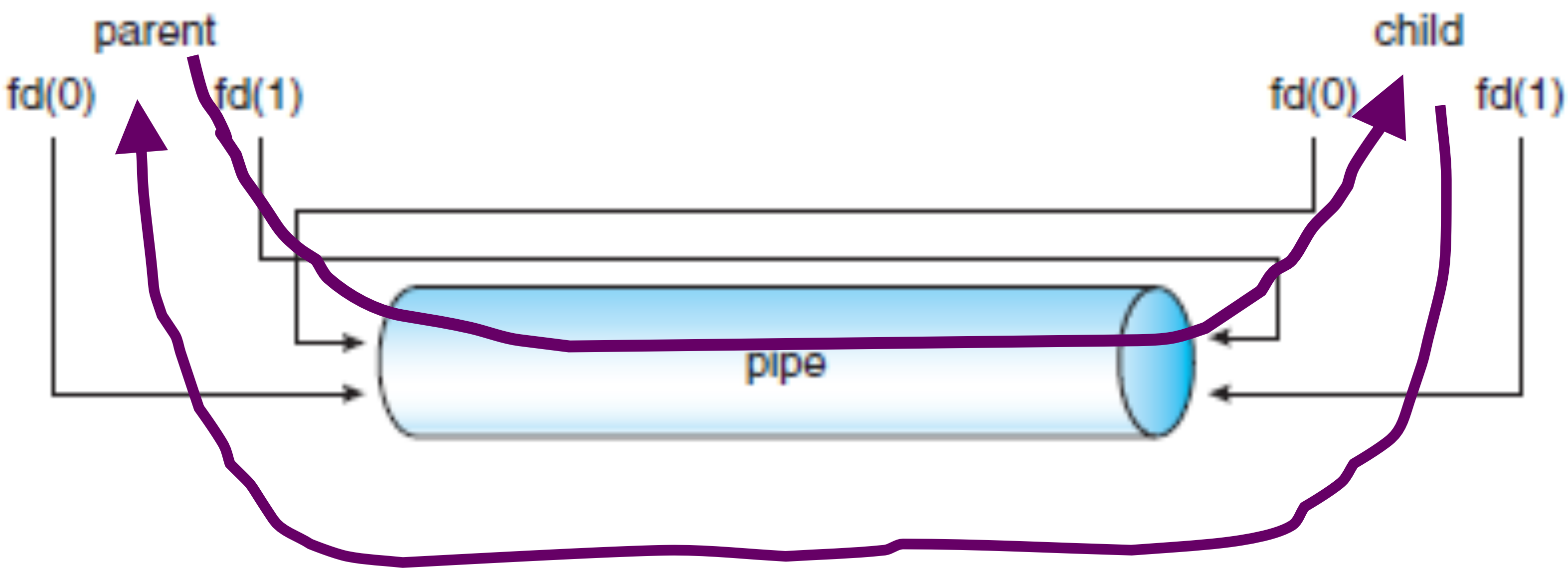



```
/* Used to store two ends of
the pipe */
int fd[2];

/* create the pipe */
if (pipe(fd)==-1) {
    fprintf(stderr, ”error");
    return 1;
}

pid = fork();
```

```
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]);
}
```

Discussion

- What if the parent wants to write something to child, while child also wants to write something to parent?

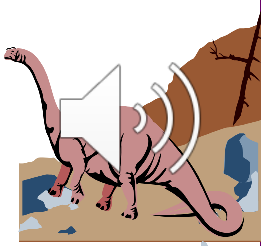


HOW?

- Hints, ordinary pipes are unidirectional

Chapter 3: Processes

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

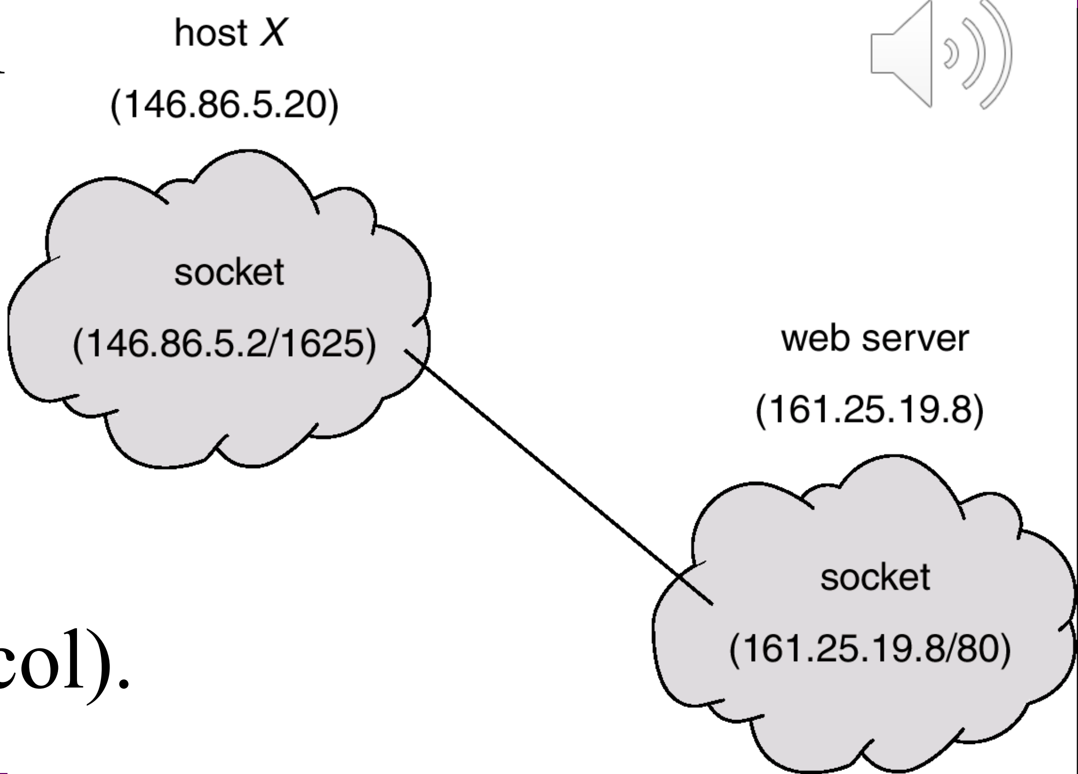


Client-Server Communication

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

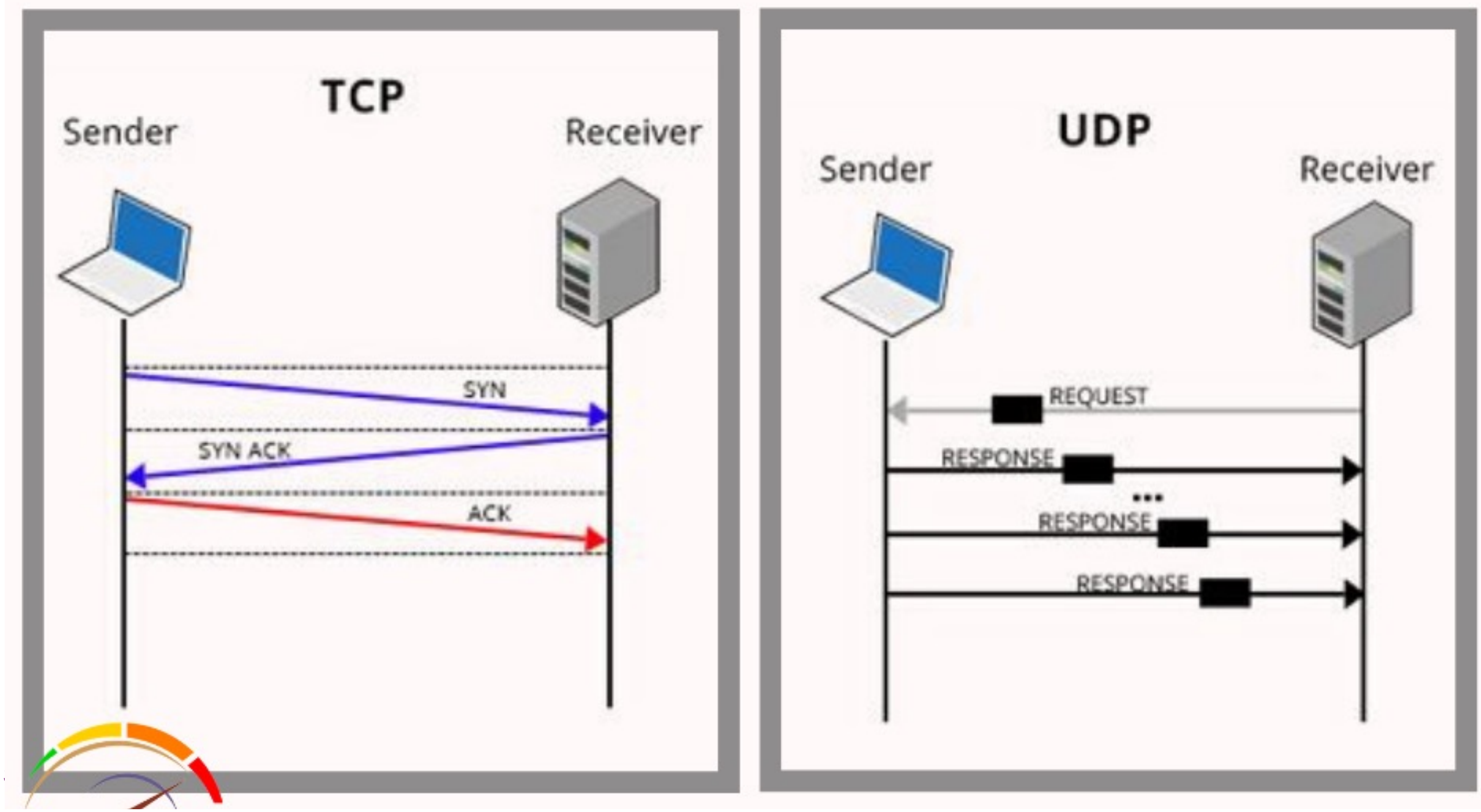
Sockets

- A socket is defined as an *endpoint for communication*
- Concatenation of IP address and port, e.g., socket **161.25.19.8:1625** is port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets.
- In the TCP/IP protocol suite, there are two transport-layer protocols: TCP (Transport Control Protocol) and UDP (User Datagram Protocol).



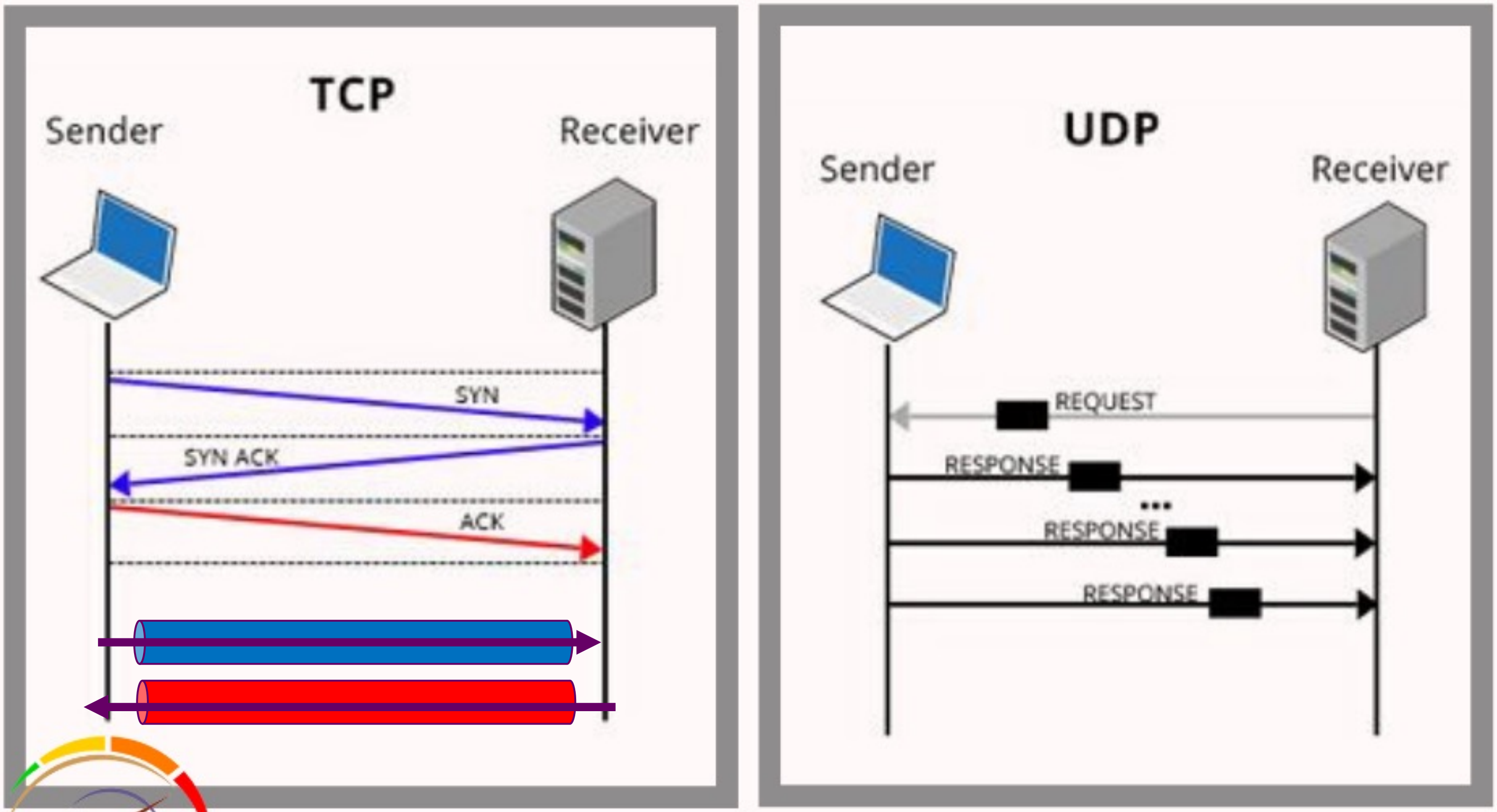
TCP vs. UDP Sockets (1)

- TCP sits on top of the IP layer, and provides a *reliable and ordered* communication channel between applications running on networked computers



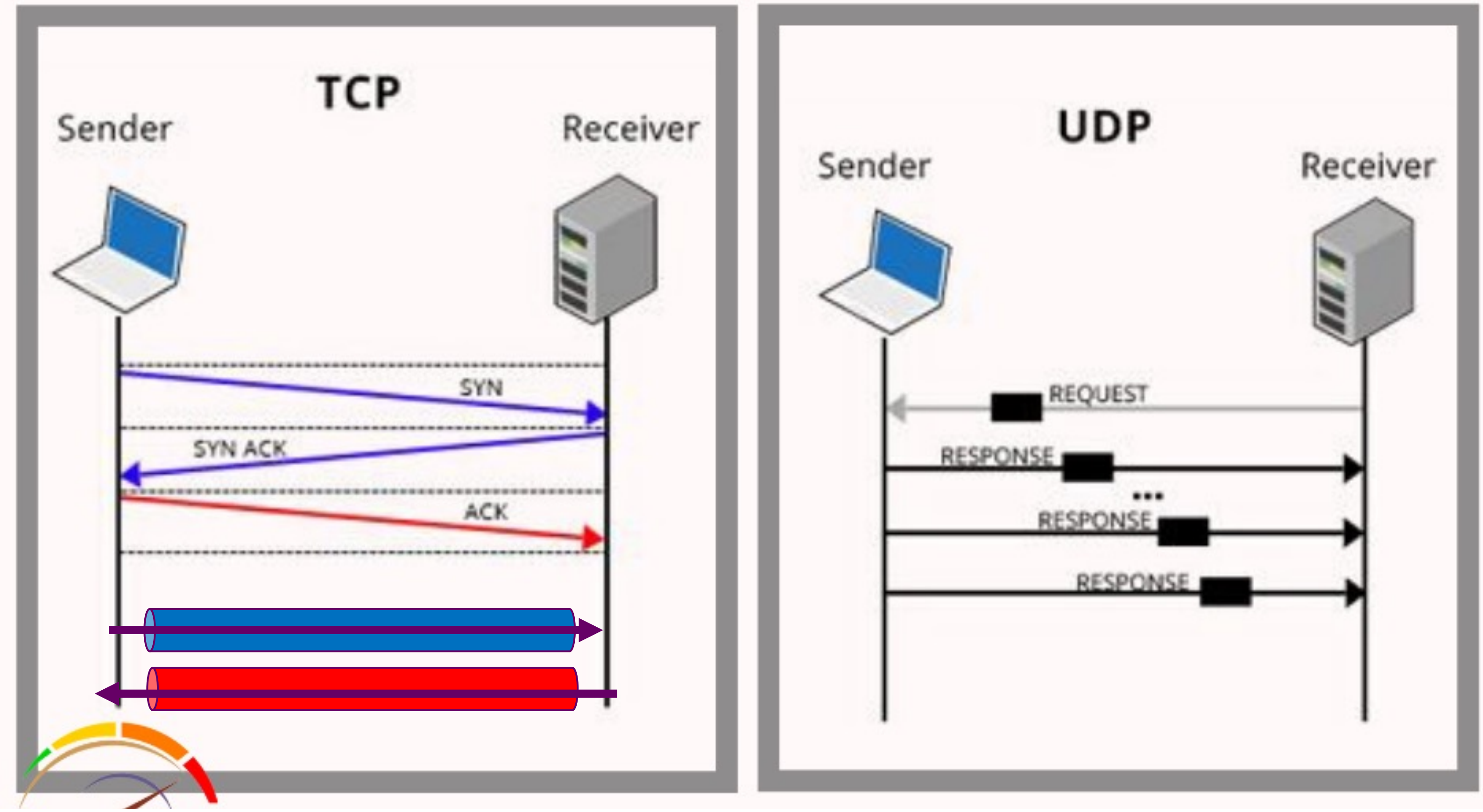
TCP vs. UDP Sockets (2)

- Conceptually, we can imagine a TCP connection as two pipes between two communicating applications, one for each direction: data put into a pipe from one end will be delivered to the other end.



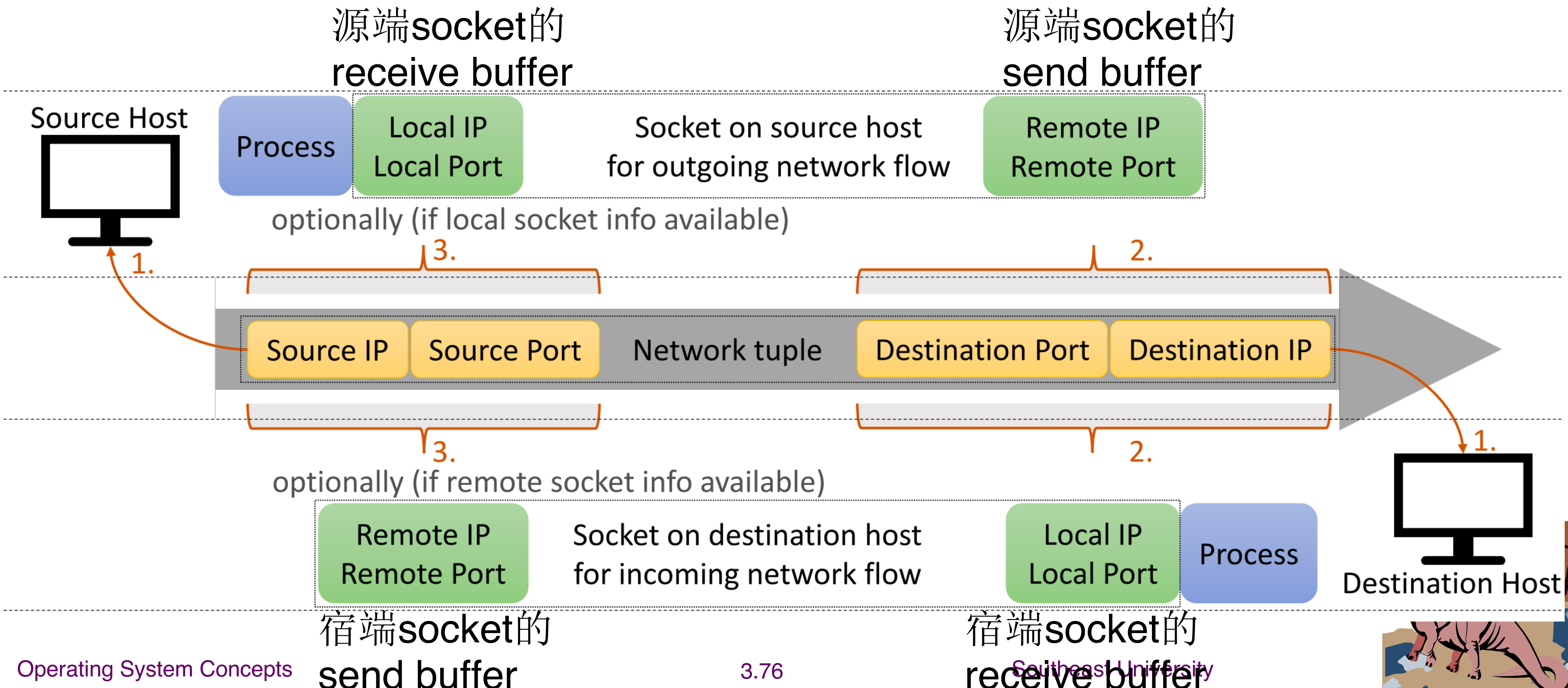
TCP vs. UDP Sockets (3)

- UDP does not provide reliability or ordered communication, but it is lightweight with lower overhead, and is thus good for applications that do *not require reliability or order*



Data Transmission: Under the Hood

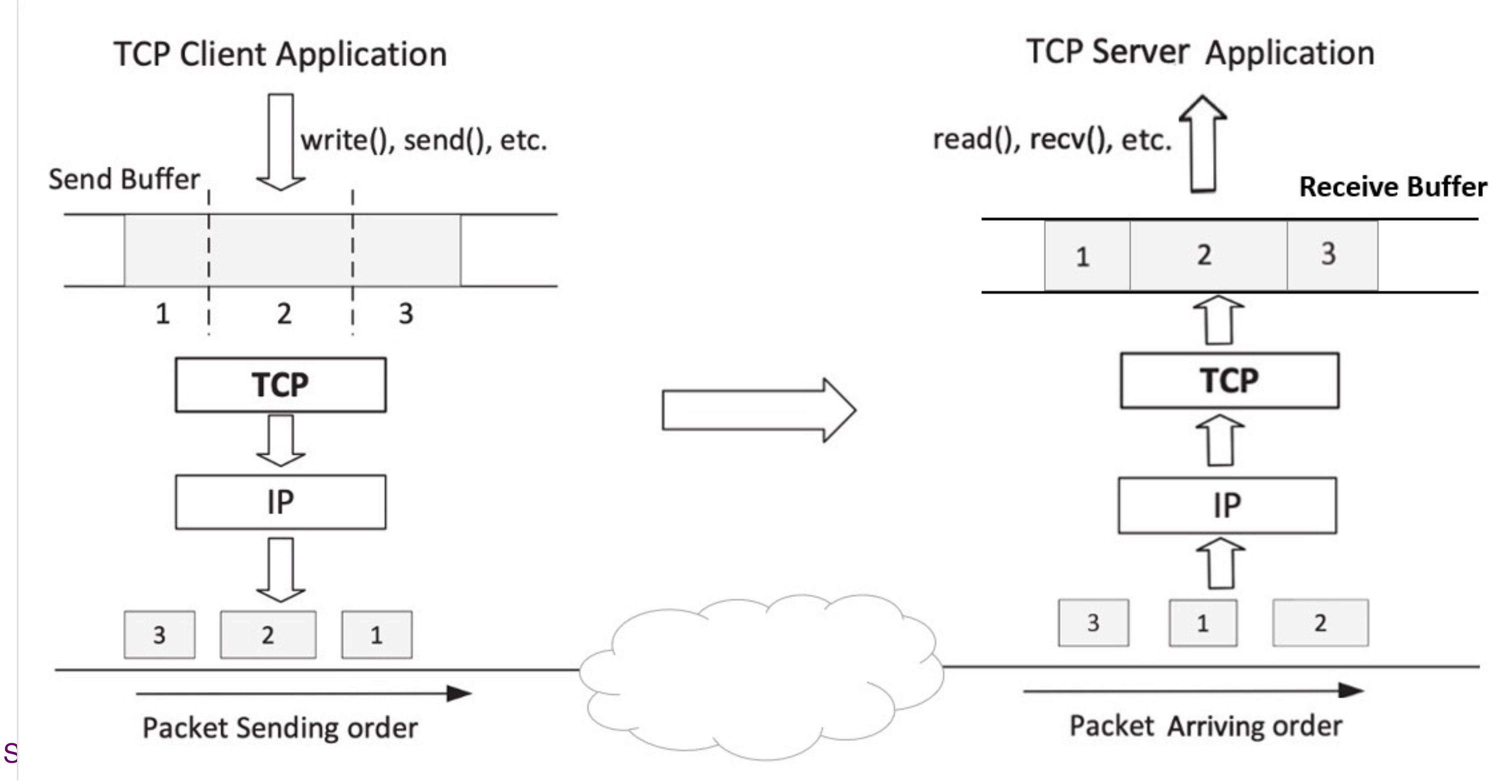
- TCP is duplex: Once a connection is established, OS allocates two buffers for each end, one for sending data (send buffer), and other for receiving data (receive buffer)



POSIX APIs for TCP connections

- APIs for TCP connections provide developers with interfaces to establish, manage, and terminate TCP communication between networked applications.

- socket() – Create a socket.
- bind() – Bind a socket to an IP addr. and port.
- listen() – Listen for incoming connections.
- accept() – Accept incoming connections
- connect() – Initiate a connection.
- send() / recv() – Send and receive data.
- close() – Close the socket.



An Example: TCP Client Program

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
int main() {
    // Step 1: Create a socket
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    // Step 2: Set the destination information
    struct sockaddr_in dest;
    memset(&dest, 0, sizeof(struct sockaddr_in));
    dest.sin_family = AF_INET;  // IPv4
    dest.sin_addr.s_addr = inet_addr("10.0.2.69");
    dest.sin_port = htons(9090);
```

An Example: TCP Client Program

```
    // Step 3: Connect to the server
    connect(sockfd, (struct sockaddr *)&dest,
    sizeof(struct sockaddr_in));

    // Step 4: Send data to the server
    char *buffer1 = "Hello Server!\n";
    char *buffer2 = "Hello Again!\n";
    write(sockfd, buffer1, strlen(buffer1));
    write(sockfd, buffer2, strlen(buffer2));

    // Step 5: Close the connection
    close(sockfd);
    return 0;
}
```

An Example: TCP Server Program

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
int main()
{
    int sockfd, newsockfd;
    struct sockaddr_in my_addr, client_addr;
    char buffer[100];
    // Step 1: Create a socket
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    // Step 2: Bind to a port number
    memset(&my_addr, 0, sizeof(struct sockaddr_in))
```

An Example: TCP Server Program

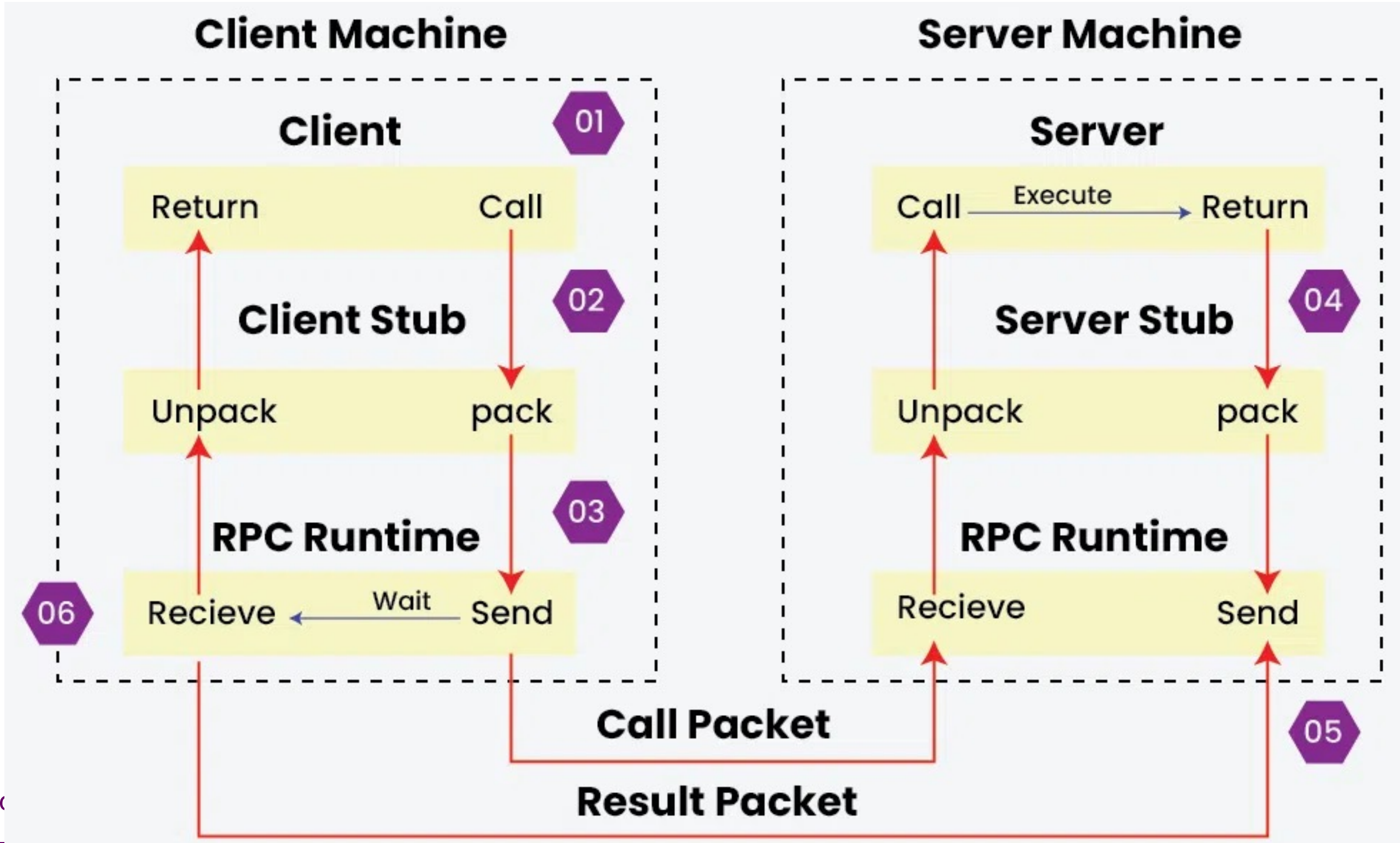
```
    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(9090);
    bind(sockfd, (struct sockaddr *)&my_addr,
sizeof(struct sockaddr_in));
    // Step 3: Listen for connections
    listen(sockfd, 5);
    // Step 4: Accept a connection request
    int client_len = sizeof(client_addr);
    while (1) {
        newsockfd = accept(sockfd, (struct sockaddr
*)&client_addr, &client_len);
        if (fork() == 0) { // The child process
            close (sockfd);
            // Read data.
            memset(buffer, 0, sizeof(buffer));
            int len = read(newsockfd, buffer, 100);
```

An Example: TCP Server Program

```
            printf("Received %d bytes.\n%s\n", len, buffer);
            close (newsockfd);
            return 0;
        } else { // The parent process
            close (newsockfd);
        }
    }
    // Step 5: Read data from the connection
    memset(buffer, 0, sizeof(buffer));
    int len = read(newsockfd, buffer, 100);
    printf("Received %d bytes: %s", len, buffer);
    // Step 6: Close the connection
    close(newsockfd); close(sockfd);
}
```

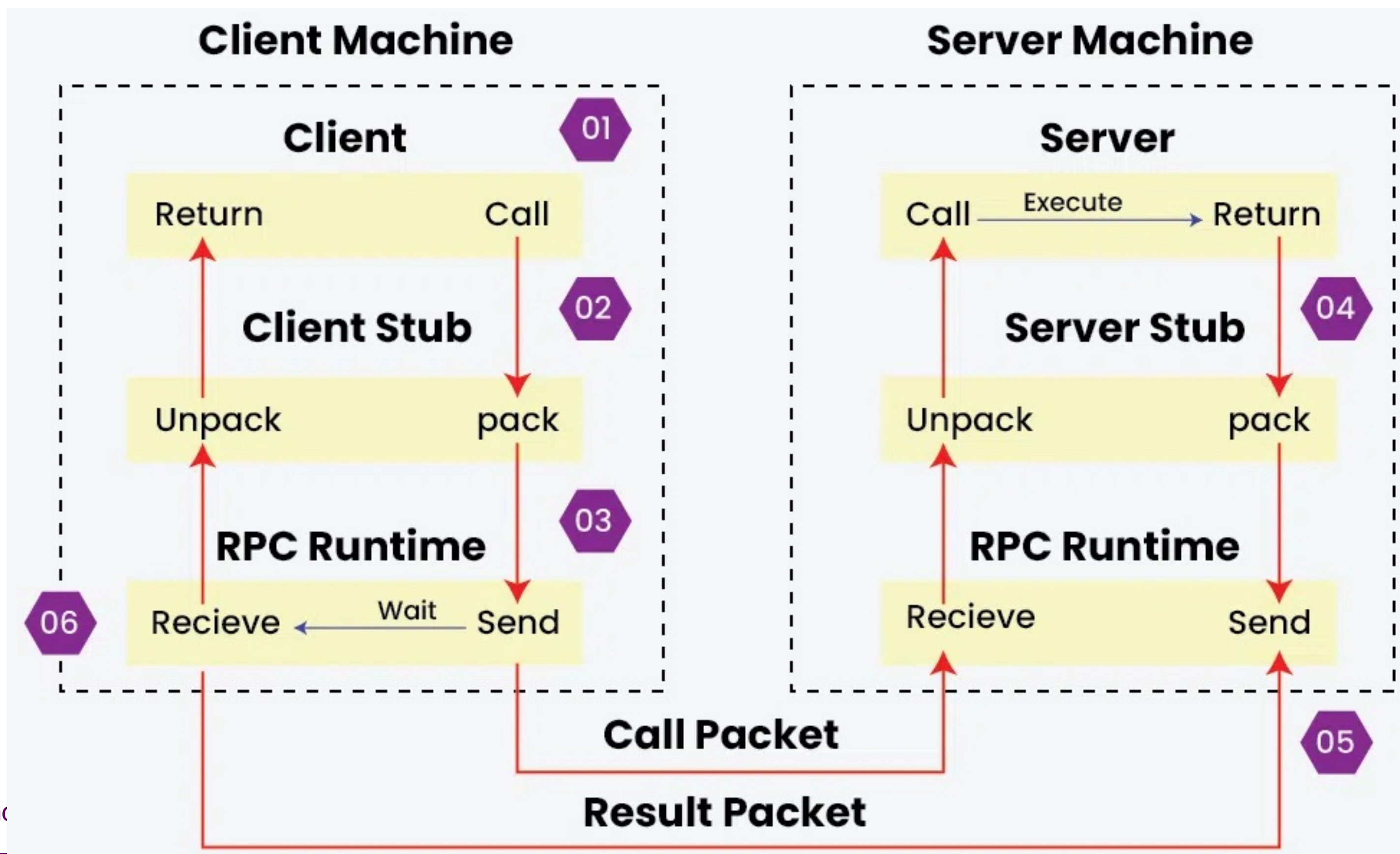
Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.



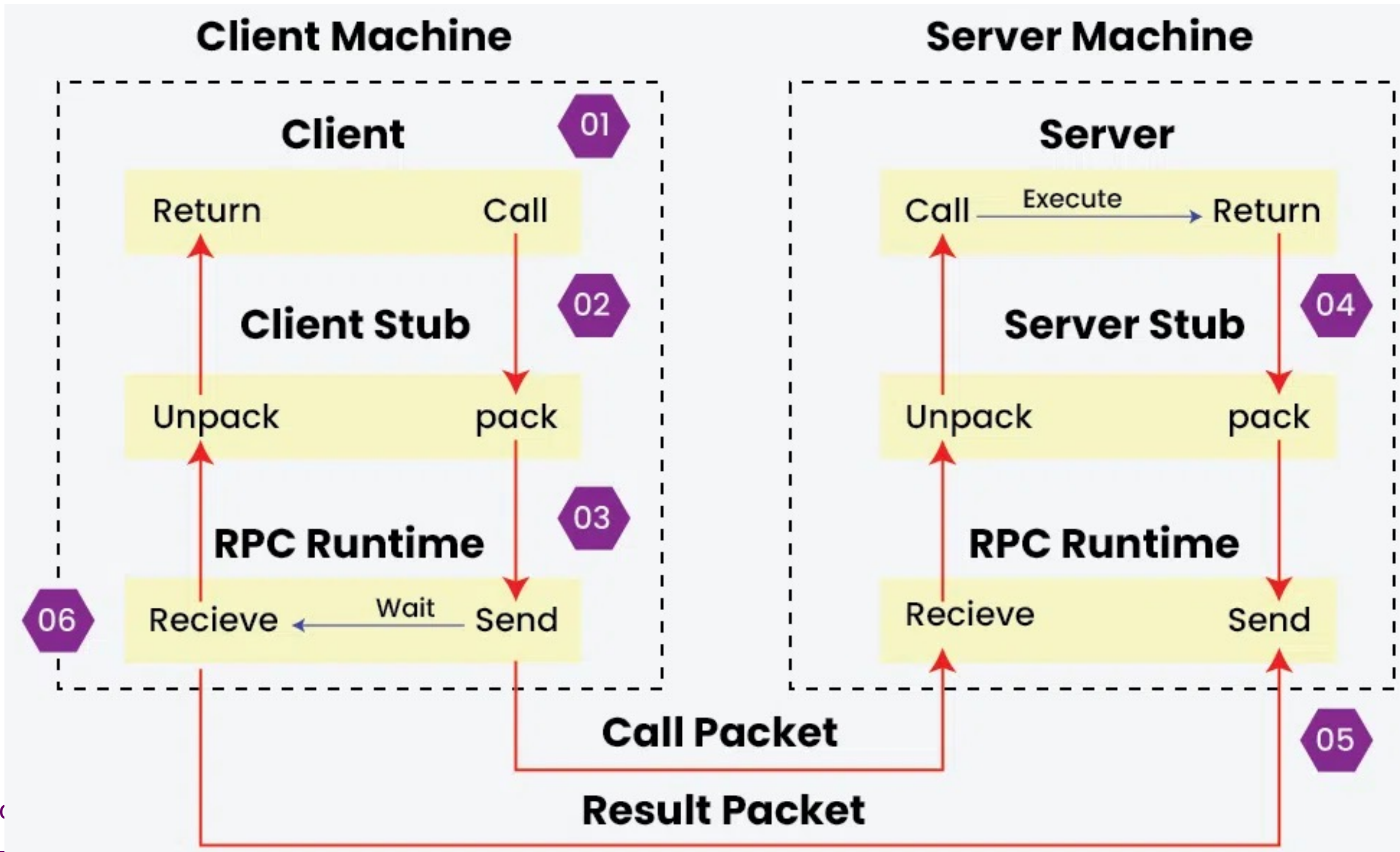
The RPC is invoked by the client. BUT…

- A client-side proxy, called stub, is used to represent the actual procedure on the server.
- The client-side stub locates the server and *marshalls* the parameters.

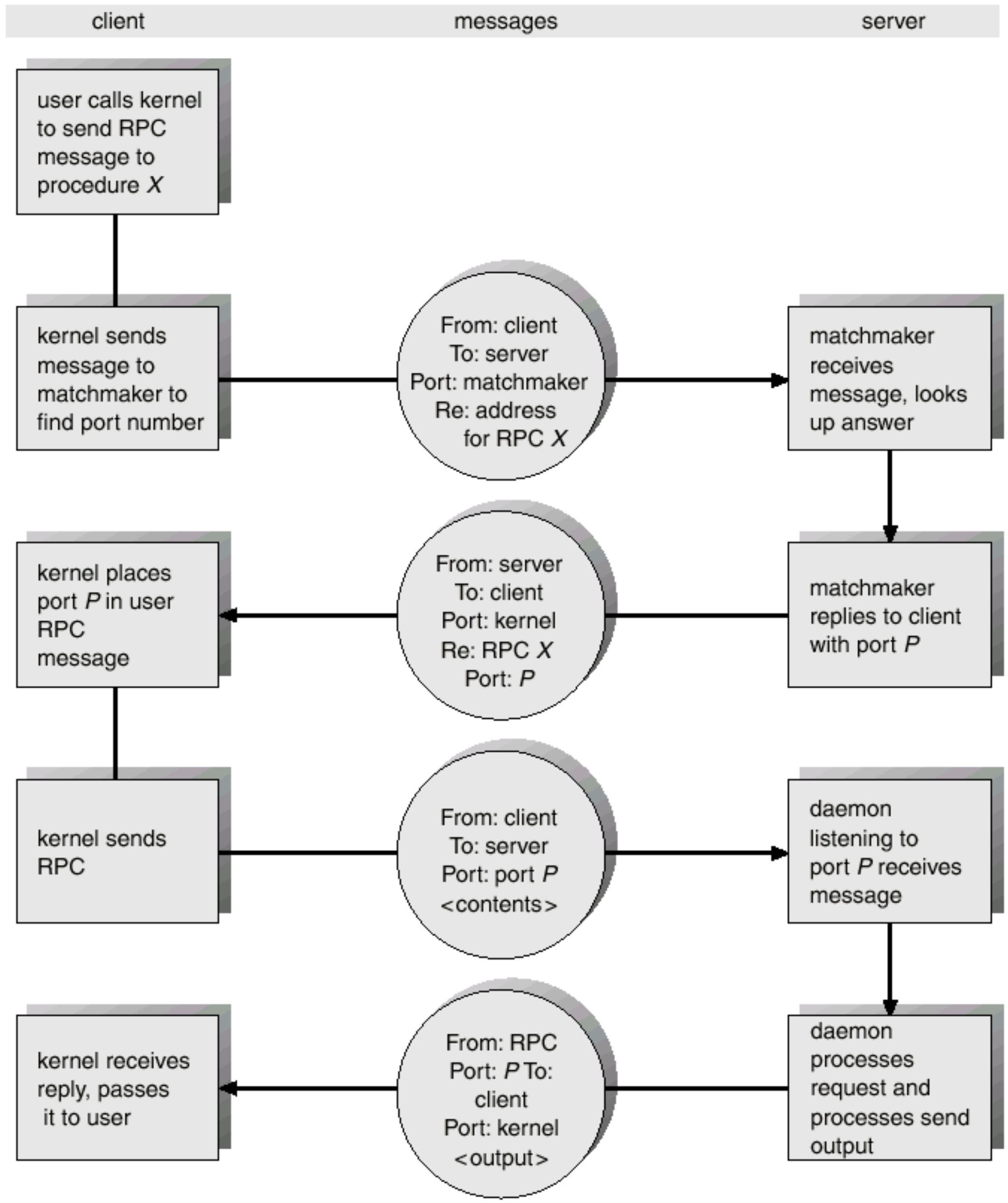


The RPC is invoked by the client. BUT…

- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server.

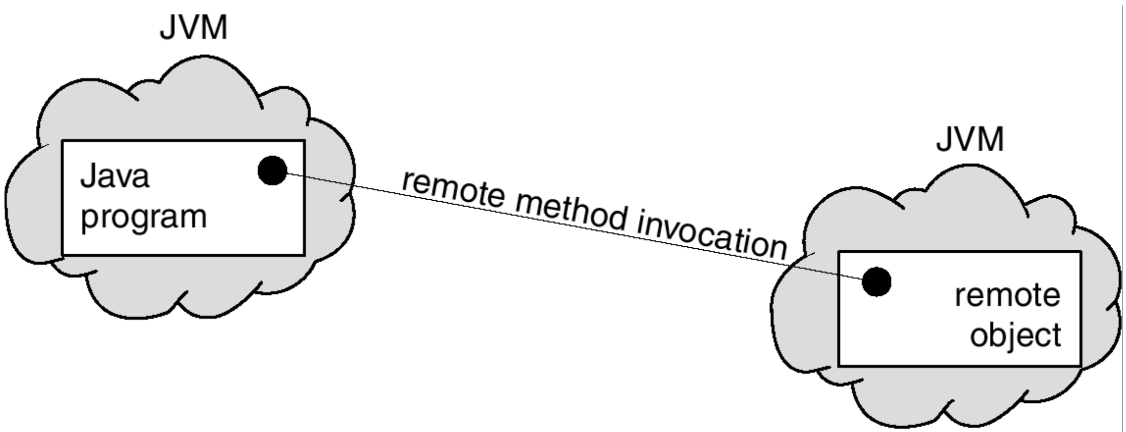


Execution Steps of RPC

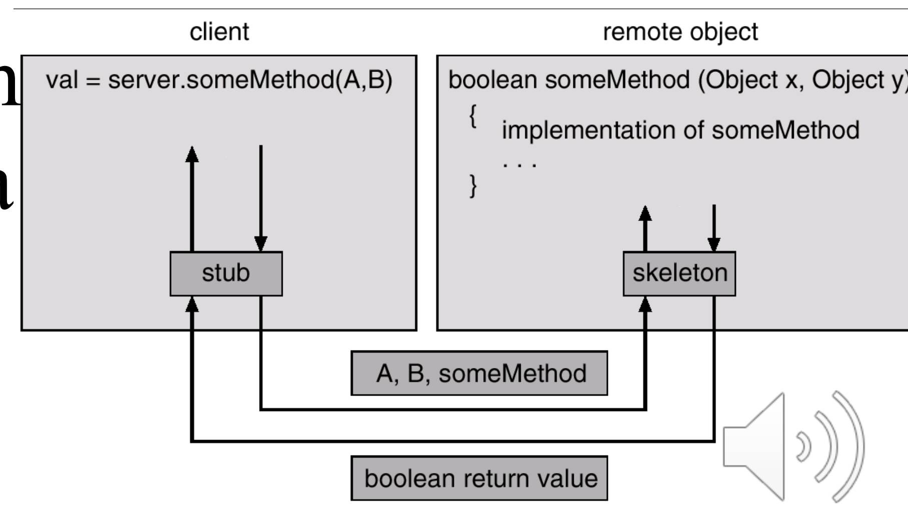


Remote Method Invocation

- 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.



- Marshalling Parameters