CHAPTER 3 - PROCESS CONCEPT

OBJECTIVES

- Introduce a process → a program in execution → basis of all computation
- Describe features of processes: scheduling, creation, termination, communication
- Explore interprocess communication using shared memory and message passing
- Describe communication in client-server systems

PROCESS CONCEPT

THE PROCESS

- Batch system –> **jobs**
- Time-shared systems –> **user programs** or **tasks**

💡 Textbook: uses **job** and **process** interchangeably

Process –> a program in execution

process execution must progress in sequential fashion

THE PROCESS - PARTS

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

PROGRAM VS PROCESS

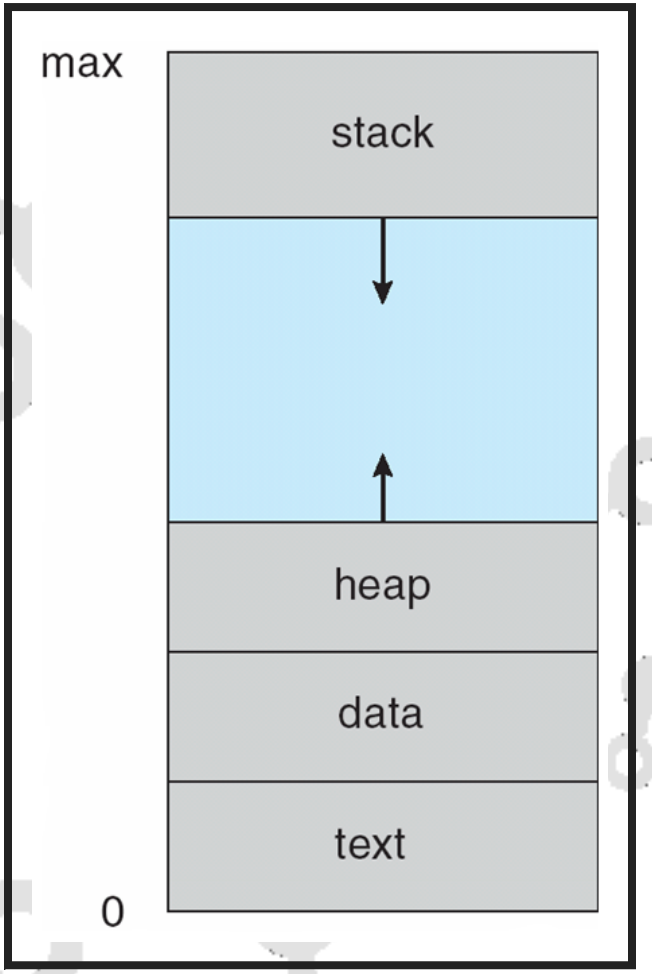
Program is passive entity stored on disk *(executable file)*, **process** is active

Program becomes **process** when executable file loaded into memory

Execution starts by GUI mouse clicks, command line entry of its name, etc

One program can be several processes → Consider multiple users executing the same program

PROCESS IN MEMORY

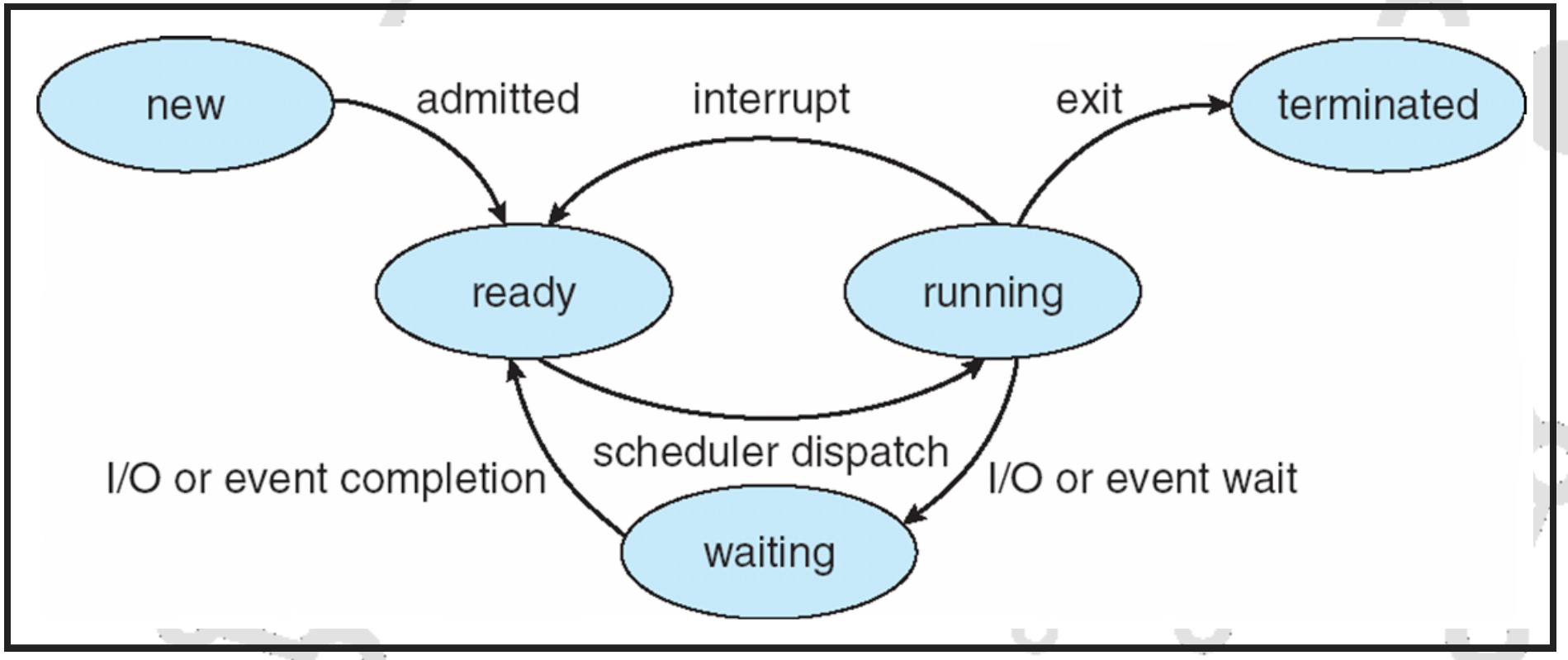


PROCESS STATE

As a process executes, it changes state

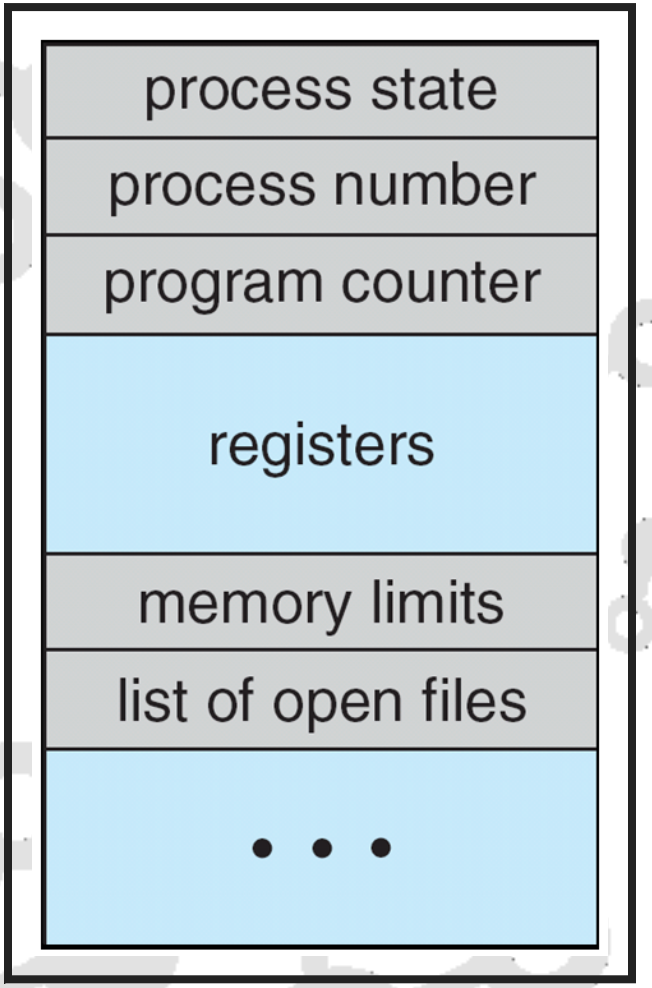
- **new:** The process is being created
- **running:** Instructions are being executed
- **waiting:** The process is waiting for some event to occur
- **ready:** The process is waiting to be assigned to a processor
- **terminated:** The process has finished execution

DIAGRAM OF PROCESS STATE



PROCESS CONTROL BLOCK

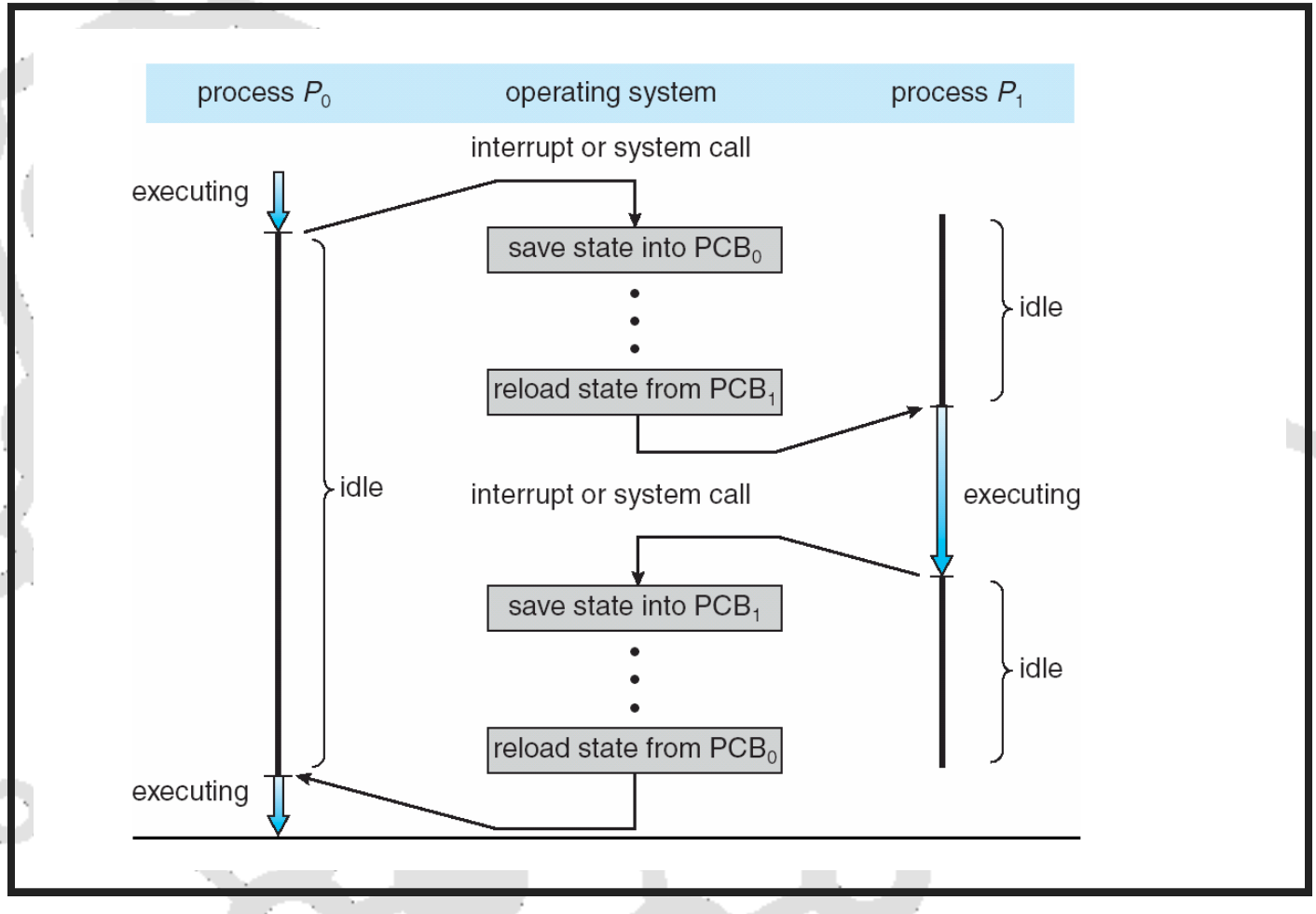
Information associated with each process



PROCESS STATE

- **Process state:** running, waiting, etc
- **Program counter:** location of instruction to next execute
- **CPU registers:** contents of all process-centric registers
- **CPU scheduling info:** priorities, scheduling queue pointers
- **Memory-management info:** memory allocated to process
- **Accounting info:** CPU used, clock time elapsed since start, time limits
- **I/O status info:** I/O devices allocated to process

CPU SWITCH FROM PROCESS TO PROCESS



THREADS

So far, process has a single thread of execution

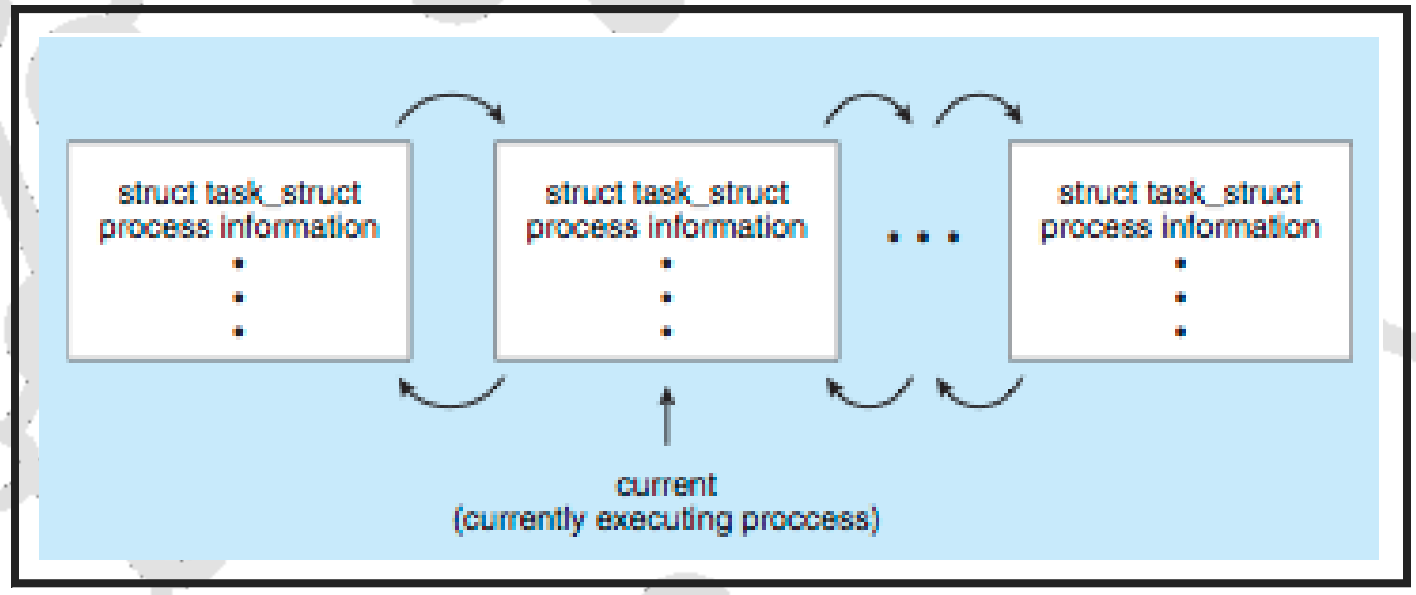
Consider having multiple program counters per process

- Multiple locations can execute at once
  - Multiple threads of control → threads
- Must then have storage for thread details, multiple program counters in PCB

PROCESS REPRESENTATION

Process Representation in Linux

Represented by the C structure `task_struct`



PROCESS REPRESENTATION IN LINUX

task_struct

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

PROCESS SCHEDULING

PROCESS SCHEDULING

Maximize CPU use, quickly switch processes onto CPU for time sharing

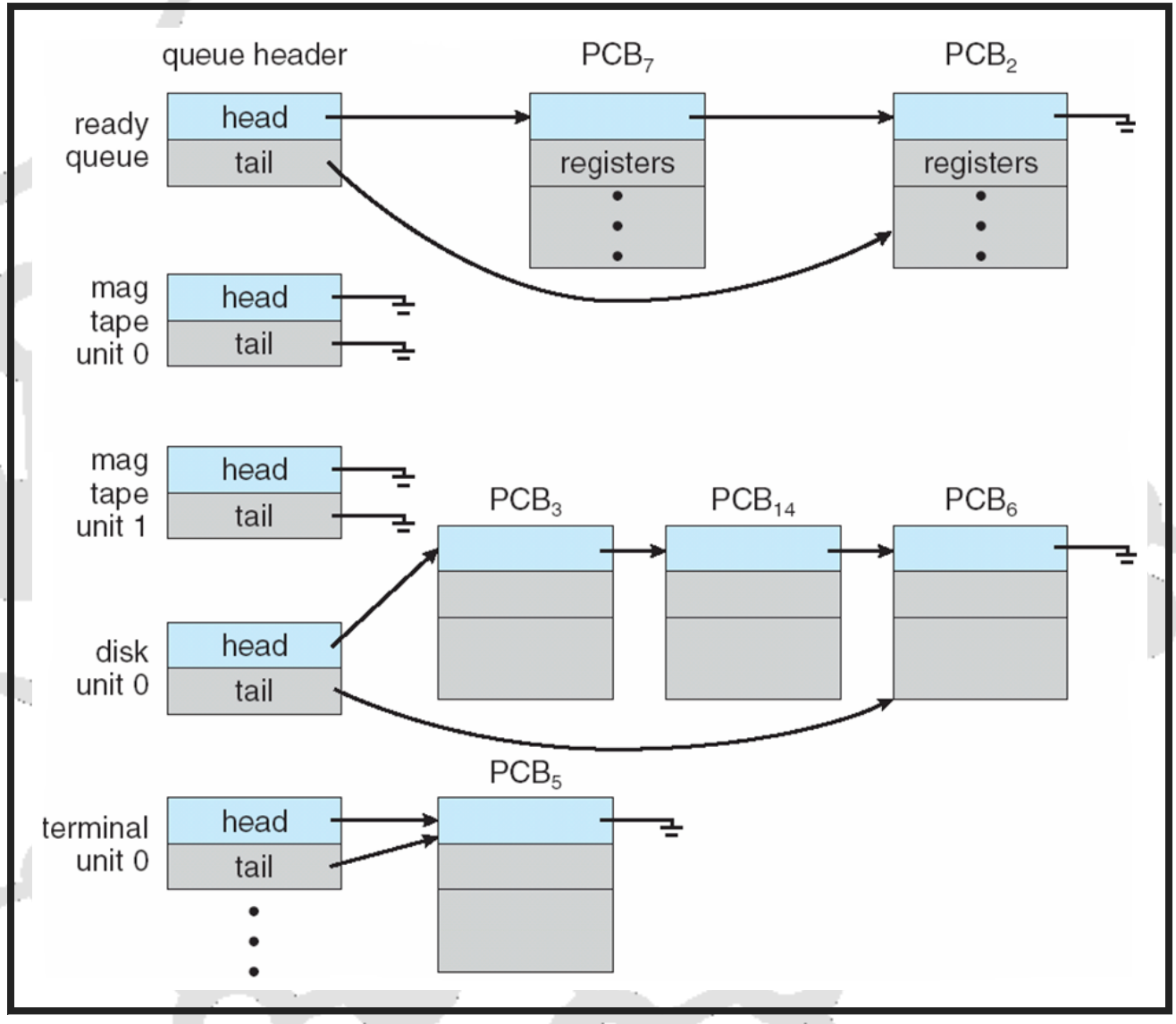
Process scheduler selects among available processes for next execution on CPU

Maintains scheduling queues of processes

SCHEDULING QUEUS

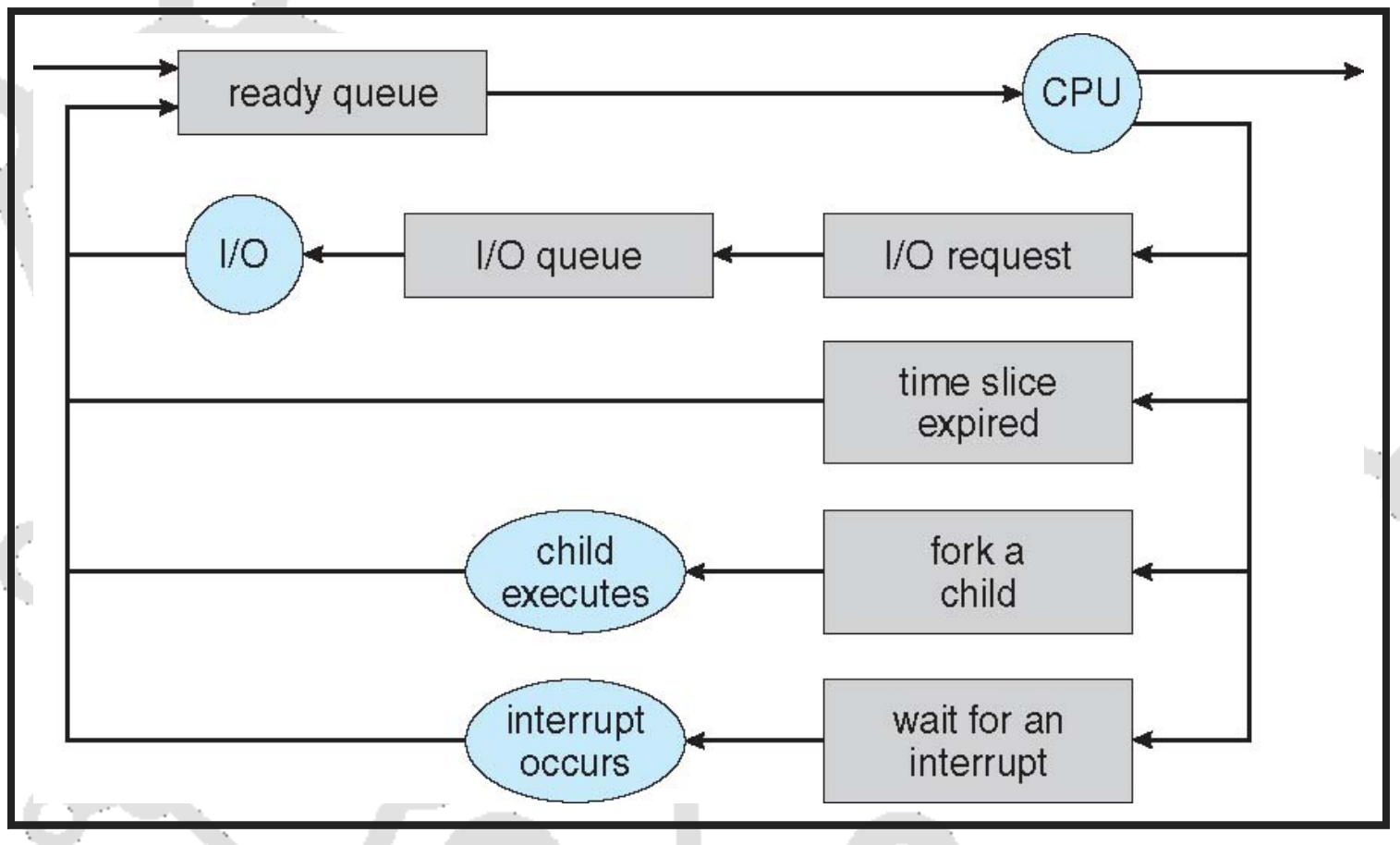
- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- **Processes** migrate among the various queues

SCHEDULING QUEUS



SCHEDULING QUEUES

Queuing diagram represents queues, resources, flows



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
  - Sometimes the only scheduler in a system

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

SCHEDULERS

Processes can be described as either:

- **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
- **CPU-bound process** – spends more time doing computations; few very long CPU bursts

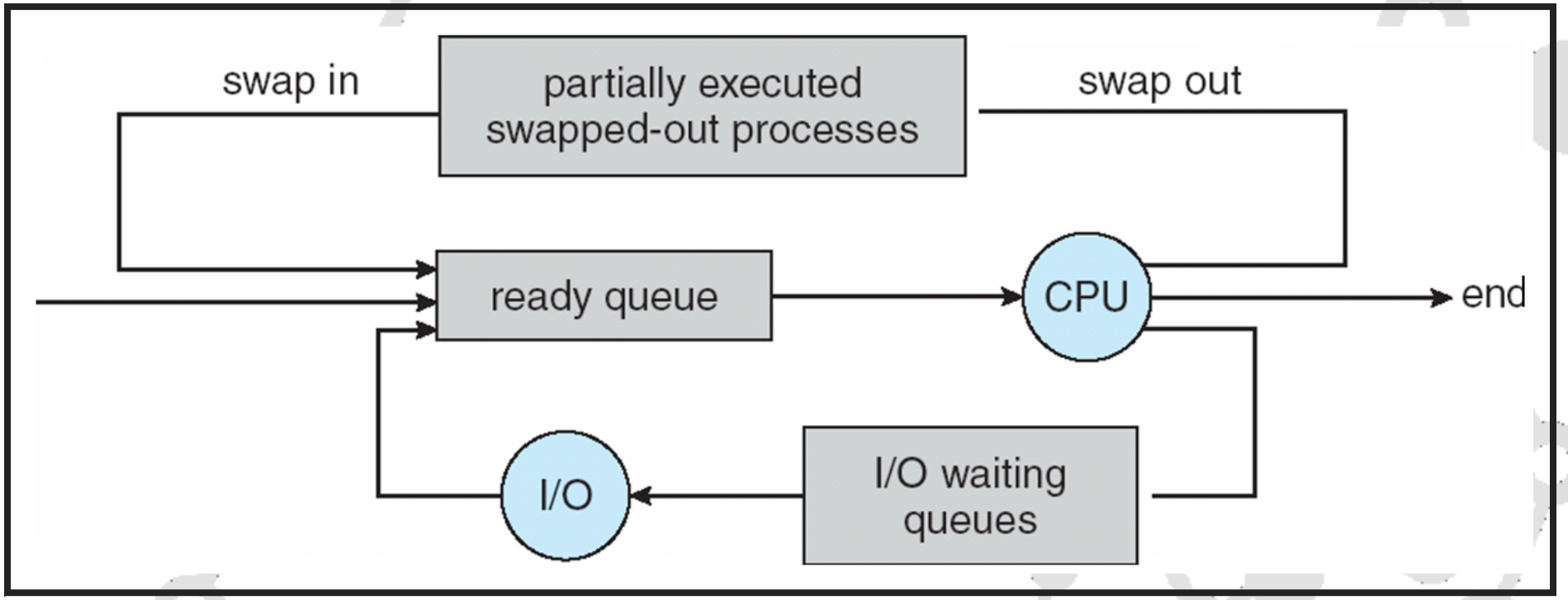
Long-term scheduler strives for good process mix

MEDIUM TERM SCHEDULING

Medium-term scheduler can be added if degree of multiple programming needs to decrease

Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**

MEDIUM TERM SCHEDULING



MULTITASKING IN MOBILE SYSTEMS

Some systems / early systems allow only one process to run, others suspended

IOS

Due to screen real estate, user interface limits iOS provides for a

- Single foreground process- controlled via user interface
- Multiple background processes– in memory, running, but not on the display, and with limits
- Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback

ANDROID

Android runs foreground and background, with fewer limits

- Background process uses a service to perform tasks
- Service can keep running even if background process is suspended
- Service has no user interface, small memory use

CONTEXT SWITCH

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

Context of a process represented in the PCB

CONTEXT SWITCH

Context-switch time is overhead; the system does no useful work while switching

- The more complex the OS and the PCB → longer the context switch

Time dependent on hardware support

- Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once

OPERATIONS ON PROCESSES

OPERATIONS ON PROCESSES

System must provide mechanisms for process creation, termination, and so on

PROCESS CREATION

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

Generally, process identified and managed via a process identifier (pid)

RESOURCE SHARING OPTIONS

- Parent and children share all resources
- Children share subset of parent’s resources
- Parent and child share no resources

EXECUTION OPTIONS

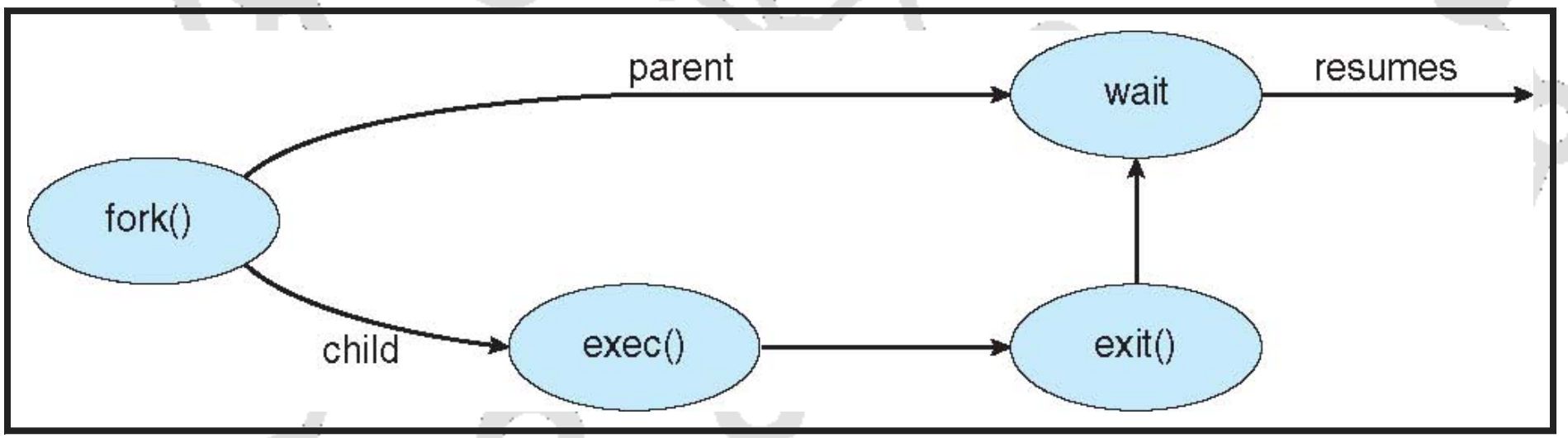
- Parent and children execute concurrently
- Parent waits until children terminate

ADDRESS SPACE

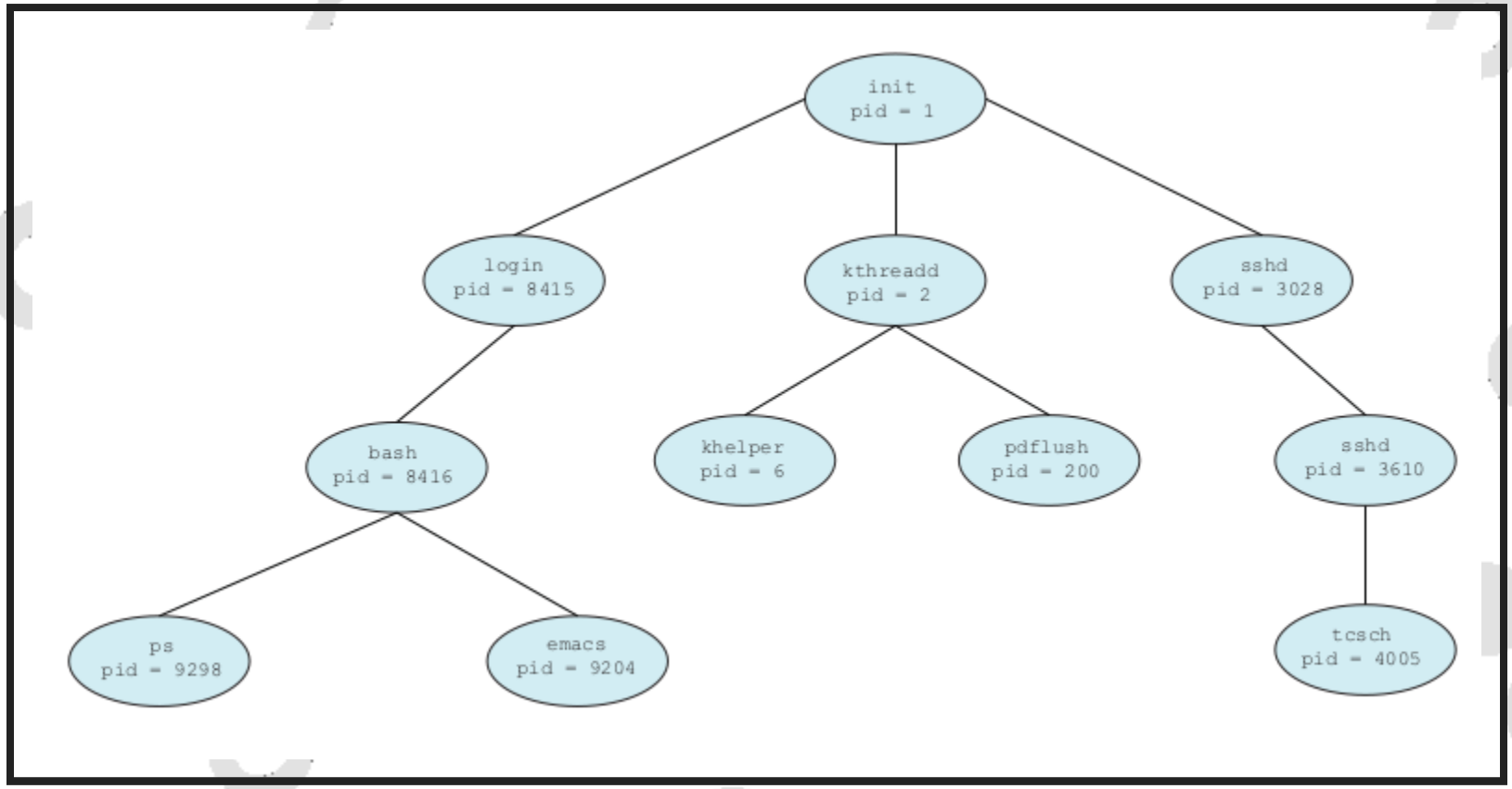
- Child duplicate of parent
- Child has a program loaded into it

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



A TREE OF PROCESSES IN LINUX



C PROGRAM FORKING SEPARATE PROCESS

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main() {
    pid t pid;
    pid = fork(); /* fork a child process */
    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 */
        wait(NULL);/* parent will wait for child to complete */
        printf("Child Complete");
    }
}
```

CREATING - WINDOWS API

```
#include <stdio.h>
#include <windows.h>
int main(VOID) {
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    ZeroMemory(&si, sizeof(si)); /* allocate memory */
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));
    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
        "C: \\ WINDOWS \\ system32 \\ mspaint.exe", /* command */
        NULL, /* don't inherit process handle */
        NULL, /* don't inherit thread handle */
        FALSE, /* disable handle inheritance */
        0, /* no creation flags */
        NULL, /* use parent's environment block */
```

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 operating system

PROCESS TERMINATION

Parent may terminate execution of children processes (`abort()`)

- Child has exceeded allocated resources
- Task assigned to child is no longer required
- If parent is exiting
  - Some operating systems do not allow child to continue if its parent terminates
  - All children terminated → cascading termination

PROCESS TERMINATION

Wait for termination, returning the pid:

```
pid_t pid;
int status;
pid = wait(&status);
```

- If no parent waiting, then terminated process is a **zombie**
- If parent terminated, processes are **orphans**

INTERPROCESS COMMUNICATION

INTERPROCESS COMMUNICATION

Processes within a system may be independent or cooperating

- Independent process cannot affect or be affected by the execution of another process
- Cooperating process can affect or be affected by other processes, including sharing data

INTERPROCESS COMMUNICATION

- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience

INTERPROCESS COMMUNICATION

- Cooperating processes need interprocess communication (IPC)
- Two models of IPC
  - Shared memory
  - Message passing

MULTIPROCESS ARCHITECTURE – CHROME BROWSER

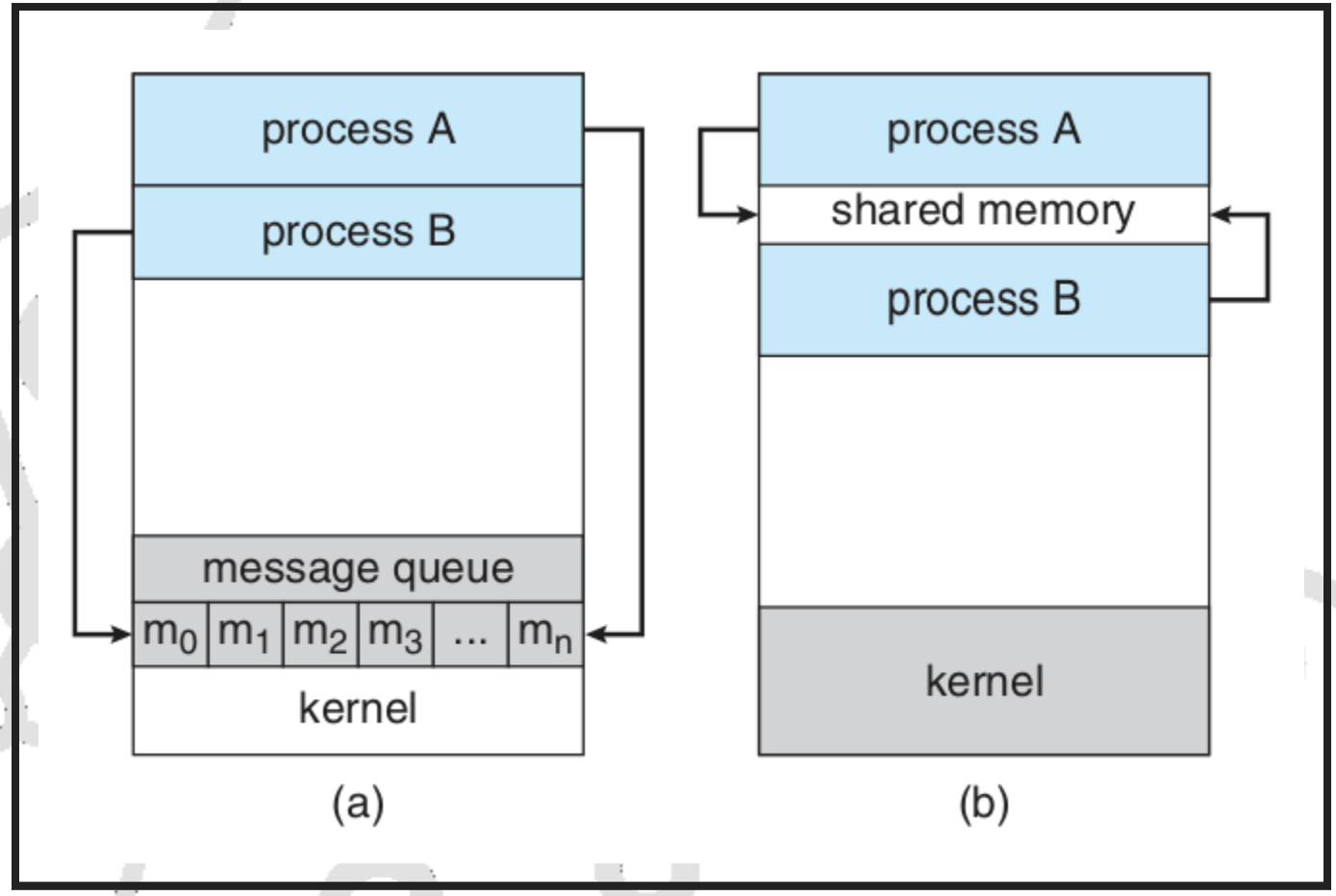
- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash

MULTIPROCESS ARCHITECTURE – CHROME BROWSER

- Google Chrome Browser is multiprocess with 3 categories
  1. Browser process manages user interface, disk and network I/O
  2. Renderer process renders web pages, deals with HTML, Javascript, new one for each website opened
     - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  3. Plug-in process for each type of plug-in

MULTIPROCESS ARCHITECTURE – CHROME BROWSER

COMMUNICATIONS MODELS



PRODUCER-CONSUMER PROBLEM

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

- unbounded-buffer places no practical limit on the size of the buffer
- bounded-buffer assumes that there is a fixed buffer size

SHARED-MEMORY SYSTEMS

Shared data

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

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

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

BOUNDED-BUFFER – PRODUCER

```
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 ;
}
```

BOUNDED-BUFFER – CONSUMER

```
item next_consumed;
while (true) {
    while (in == out)
        ; /* do nothing */
    next consumed = buffer[out];
    out = (out + 1) % BUFFER SIZE ;
    /* consume the item in next consumed */
}
```

MESSAGE-PASSING SYSTEMS

Mechanism for processes to communicate and to synchronize their actions

Message system – processes communicate with each other without resorting to shared variable

INTERPROCESS COMMUNICATION

- 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

INTERPROCESS COMMUNICATION

- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., direct or indirect, synchronous or asynchronous, automatic or explicit buffering)

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

DIRECT COMMUNICATION

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

INDIRECT COMMUNICATION

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?

INDIRECT COMMUNICATION

Solutions

- Allow a link to be associated with at most two processes
- Allow only one process at a time to execute a receive operation
- Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

SYNCHRONIZATION

Message passing may be either blocking or non-blocking

Blocking is considered synchronous

- Blocking send has the sender block until the message is received
- Blocking receive has the receiver block until a message is available

SYNCHRONIZATION

Non-blocking is considered asynchronous

- Non-blocking send has the sender send the message and continue
- Non-blocking receive has the receiver receive a valid message or null

SYNCHRONIZATION

Different combinations possible

If both send and receive are blocking, we have a rendezvous

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 – 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 waits

EXAMPLES OF IPC SYSTEMS

POSIX SHARED MEMORY

- Process first creates shared memory segment

```
shm_fd = shm_open(name, O CREAT | O RDRW, 0666);
```

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

```
ftruncate(shm fd, 4096);
```

- Now the process could write to the shared memory

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

POSIX PRODUCER

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

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

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

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

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

```
    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;
}
```

POSIX CONSUMER

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

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

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

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

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

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

    return 0;
}
```

MACH (1)

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()

MACH (2)

- Send and receive are flexible, for example four options if mailbox full:
  1. Wait indefinitely
  2. Wait at most n milliseconds
  3. Return immediately
  4. Temporarily cache a message

WINDOWS (!)

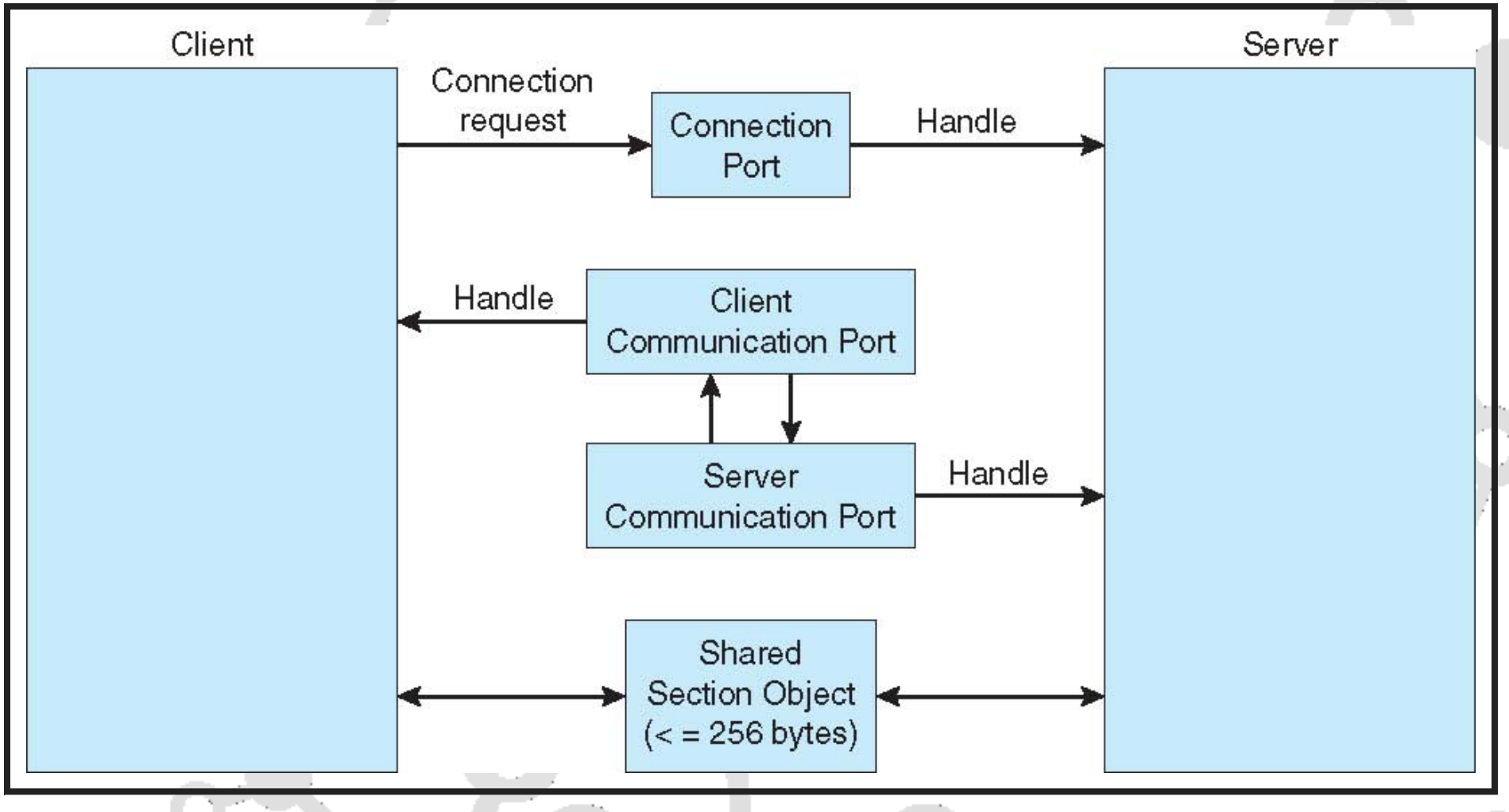
Message-passing centric via advanced local procedure call (LPC) facility

- Only works between processes on the same system
- Uses ports (like mailboxes) to establish and maintain communication channels

WINDOWS (2)

- Communication works as follows:
  - The client opens a handle to the subsystem's connection port object.
  - The client sends a connection request.
  - The server creates two private communication ports and returns the handle to one of them to the client.
  - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

LOCAL PROCEDURE CALLS IN WIN XP



COMMUNICATION IN CLIENT – SERVER SYSTEMS

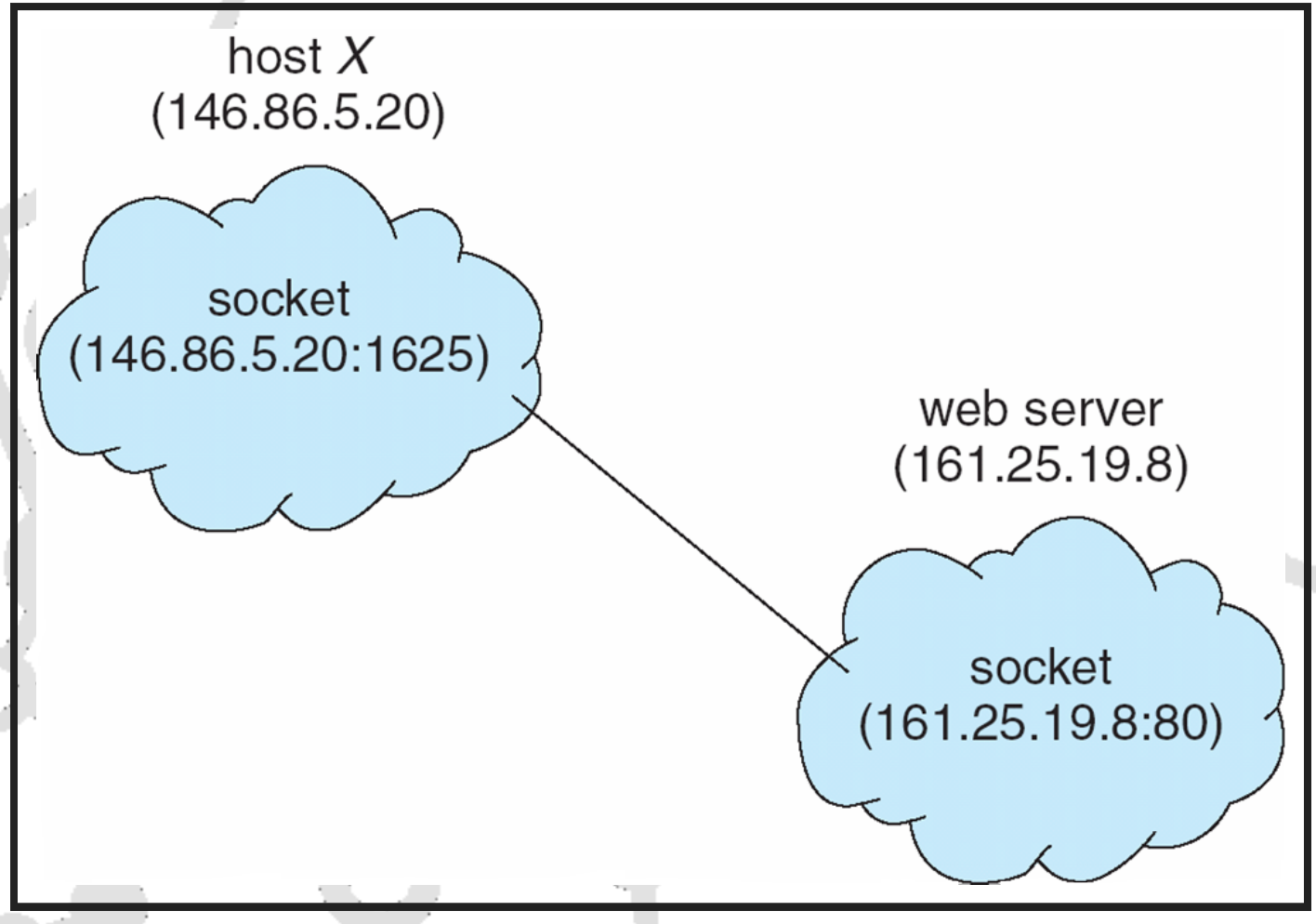
COMMUNICATION IN CLIENT – SERVER SYSTEMS

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

SOCKETS

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

SOCKET COMMUNICATION



SOCKETS IN JAVA

Three types of sockets

- Connection-oriented (TCP)
- Connectionless (UDP)
- MulticastSocket class – data can be sent to multiple recipients

SOCKETS IN JAVA

```
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);
      }
   }
}
```

REMOTE PROCEDURE CALLS

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

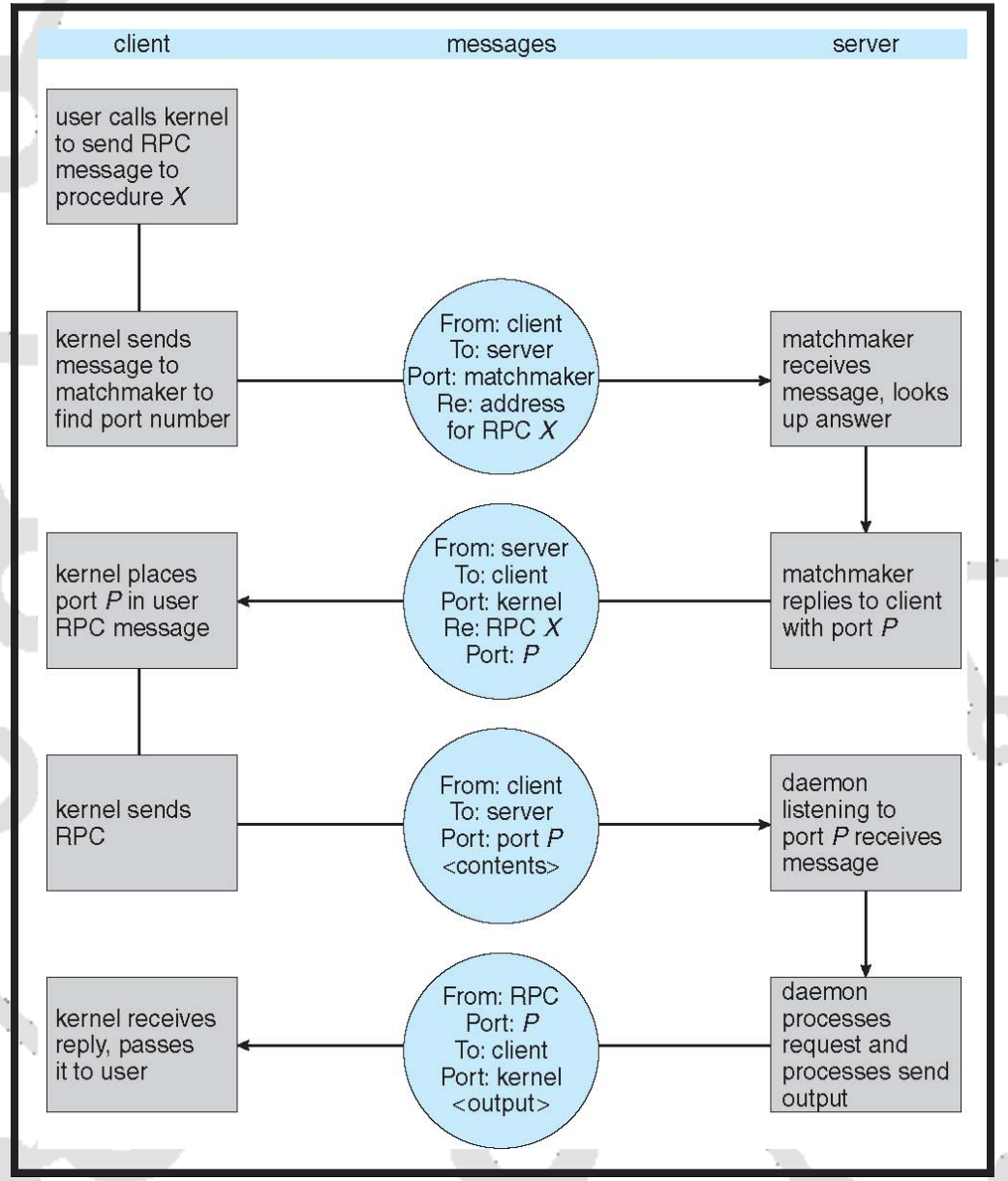
Again uses ports for service differentiation

- Stubs – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and marshalls the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server

REMOTE PROCEDURE CALLS

- On Windows, stub code compile from specification written in Microsoft Interface Definition Language (MIDL)
- Data representation handled via External Data Representation (XDL) format to account for different architectures → Big-endian and little-endian
- Remote comm. → more failure scenarios than local
- Messages can be delivered exactly once rather than at most once
- OS typically provides a rendezvous/matchmaker service to connect client and server

EXECUTION OF RPC



PIPES

Acts as a conduit allowing two processes to communicate

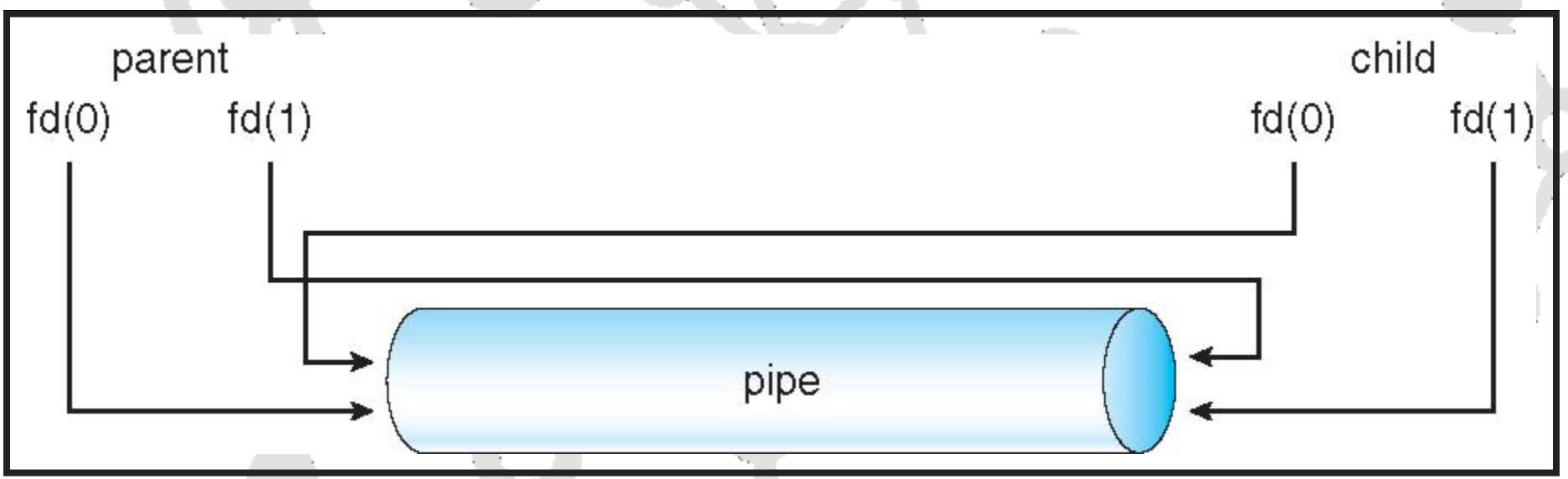
- Issues
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e. parent-child) between the communicating processes?
  - Can the pipes be used over a network?

ORDINARY PIPES

- 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

ORDINARY PIPES

- Require parent-child relationship between communicating processes
- Windows calls these anonymous pipes



NAMED PIPES

Named Pipes are more powerful than ordinary pipes

Communication is bidirectional

No parent-child relationship is necessary between the communicating processes

Several processes can use the named pipe for communication

Provided on both UNIX and Windows systems

QUESTIONS

BONUS

- Exam question number 2: **Process Concept and Multithreaded Programming**