PROCESSES, THREADS AND CONCURRENCY

Chapter 3 and 4

OBJECTIVES - PROCESSES

- Introduce a process → a program in execution
- Describe features of processes
- Interprocess Communication
- Describe communication in client-server systems

OBJECTIVES

- Introduce notion of thread → a fundamental unit of CPU utilization
  - forms the basis of multithreaded computer systems
- Pthreads, Windows, and Java API thread libraries
- Strategies that provide implicit threading
- OS support for threads in Windows and Linux

CHAPTER 3 - PROCESSES

THE PROCESS

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

Process –> a program in execution

PROGRAM VS PROCESS

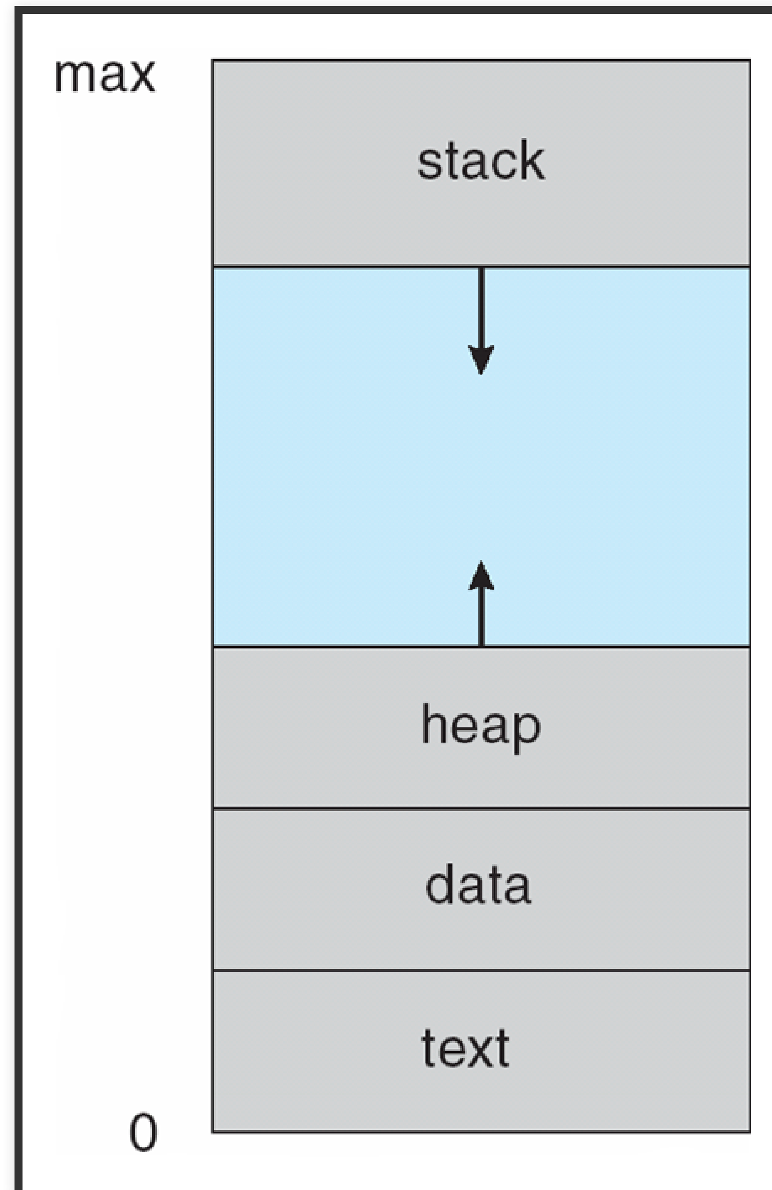
Program is passive entity stored on disk *(executable file)*, **process** is active

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

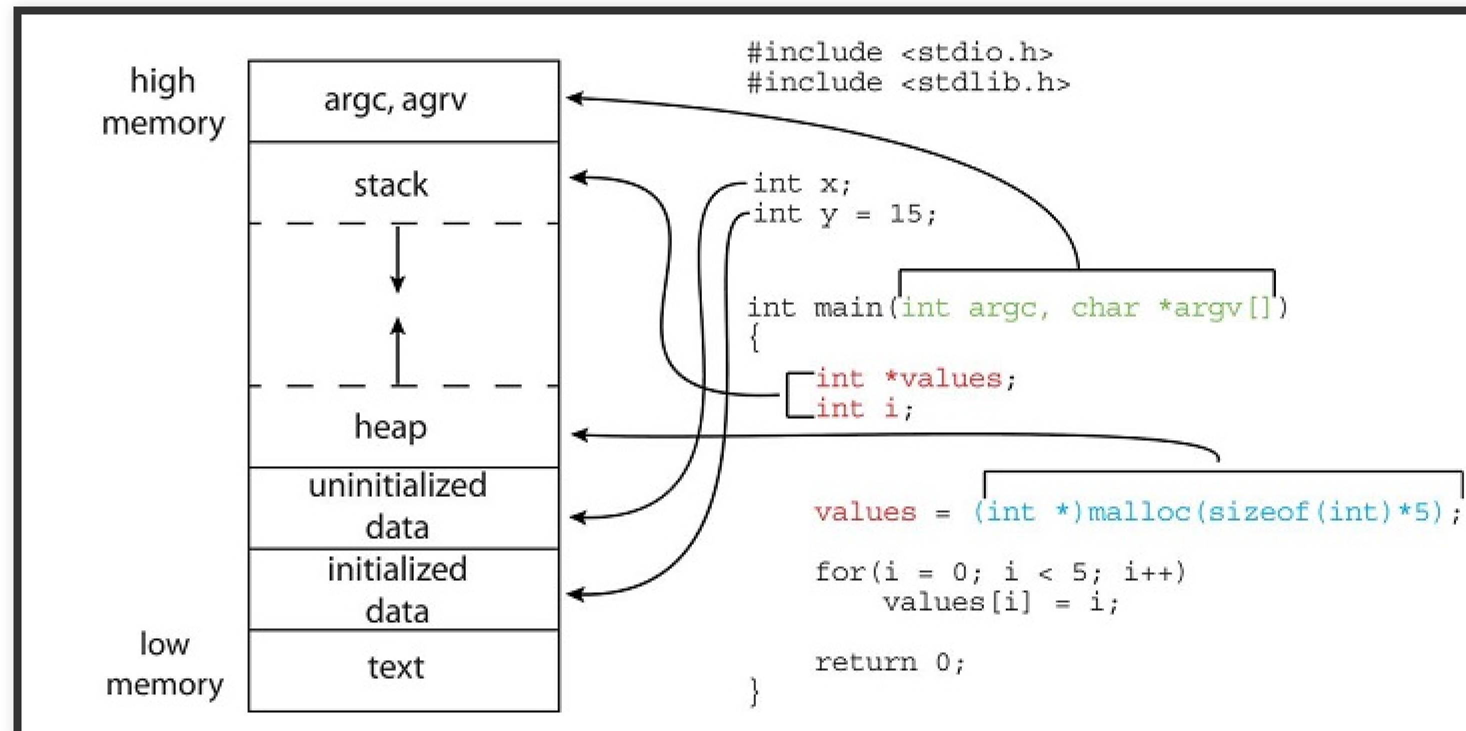
Execution starts by GUI mouse clicks, command line entry of its name, etc

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

THE PROCESS - PARTS



THE PROCESS - C PROGRAM

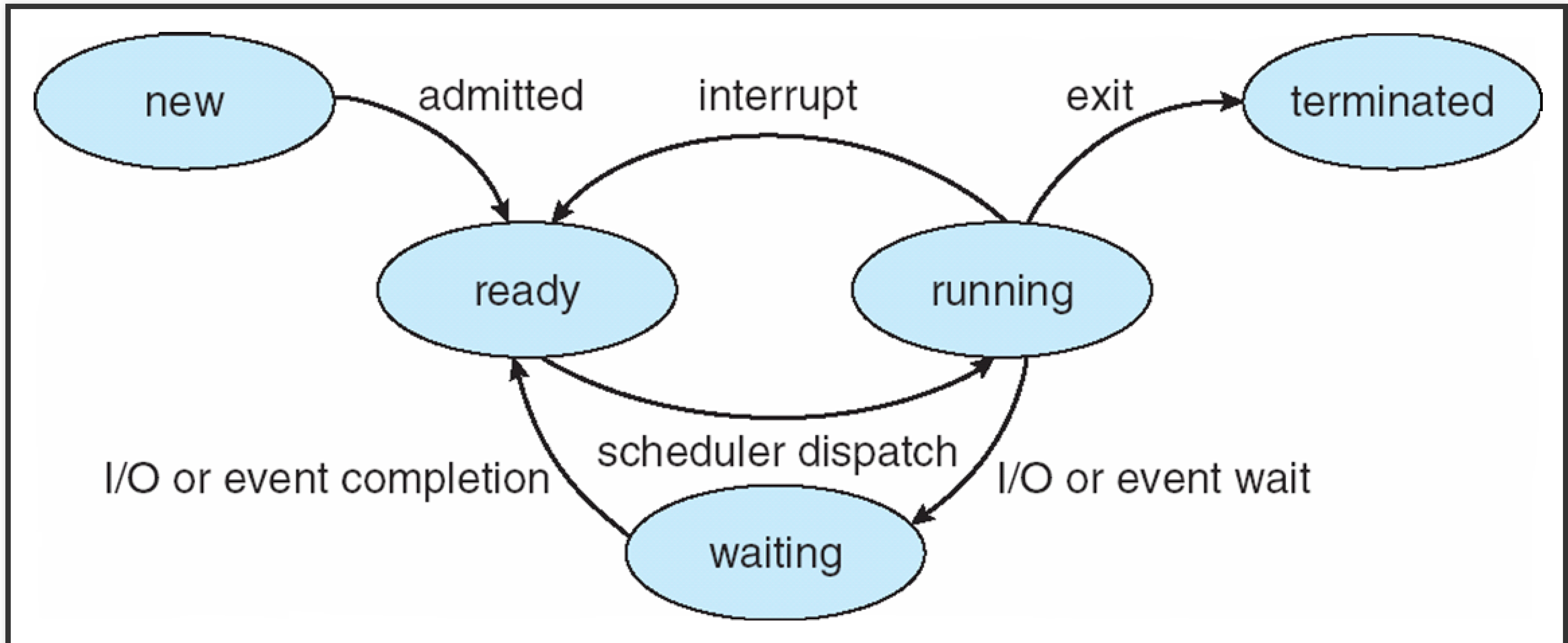


SKELETON PROGRAM -FIRST PROJECT

```
size program
```

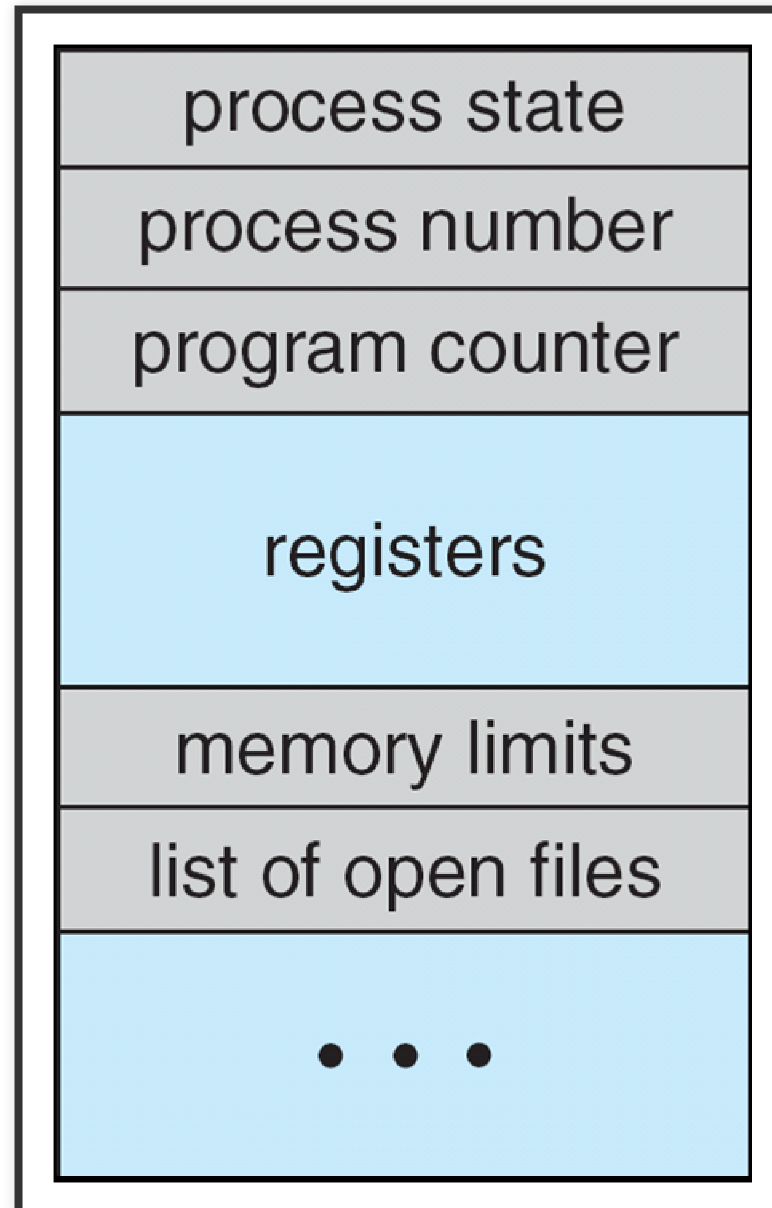
```
   text    data     bss     dec     hex filename
   1857     552       8    2417     971 program
```

PROCESS STATE



PROCESS CONTROL BLOCK

Information associated with each process



THREADS

So far, process has a single thread of execution

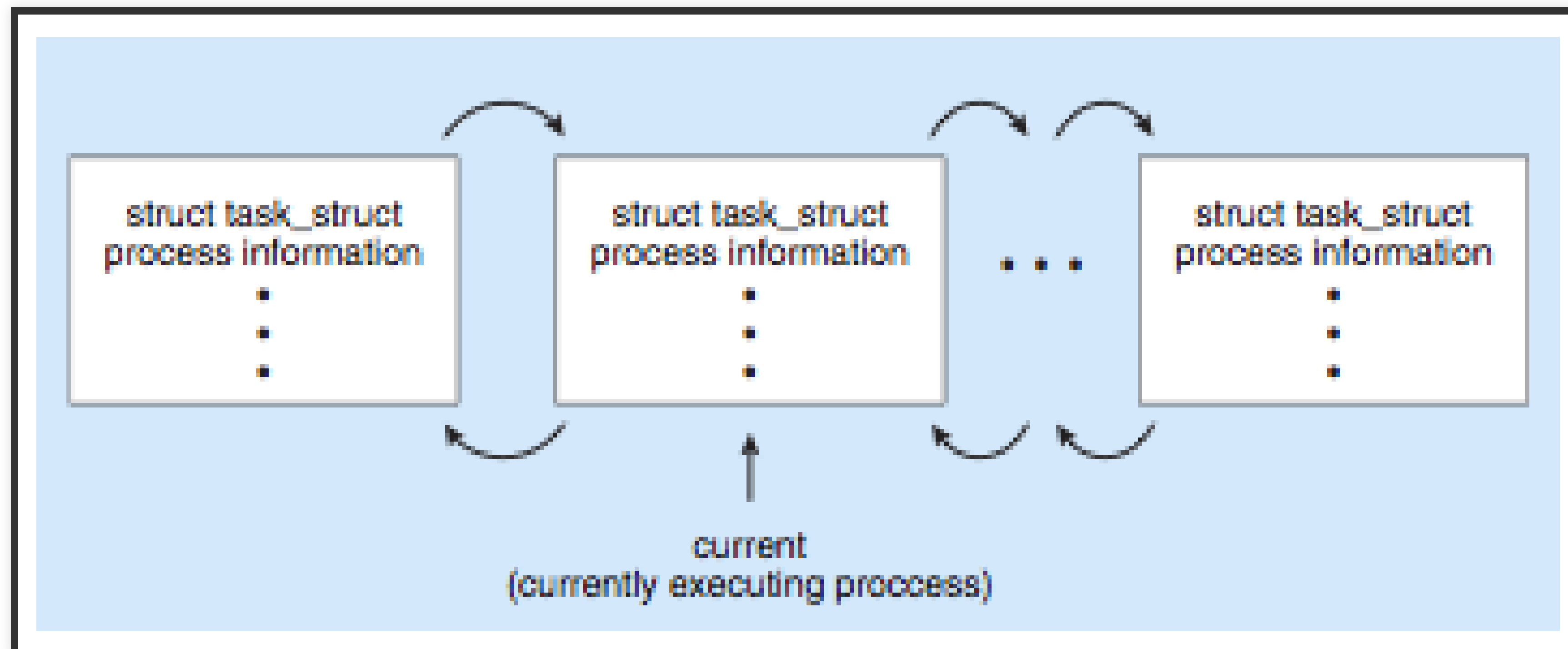
Consider having multiple program counters per process

- 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

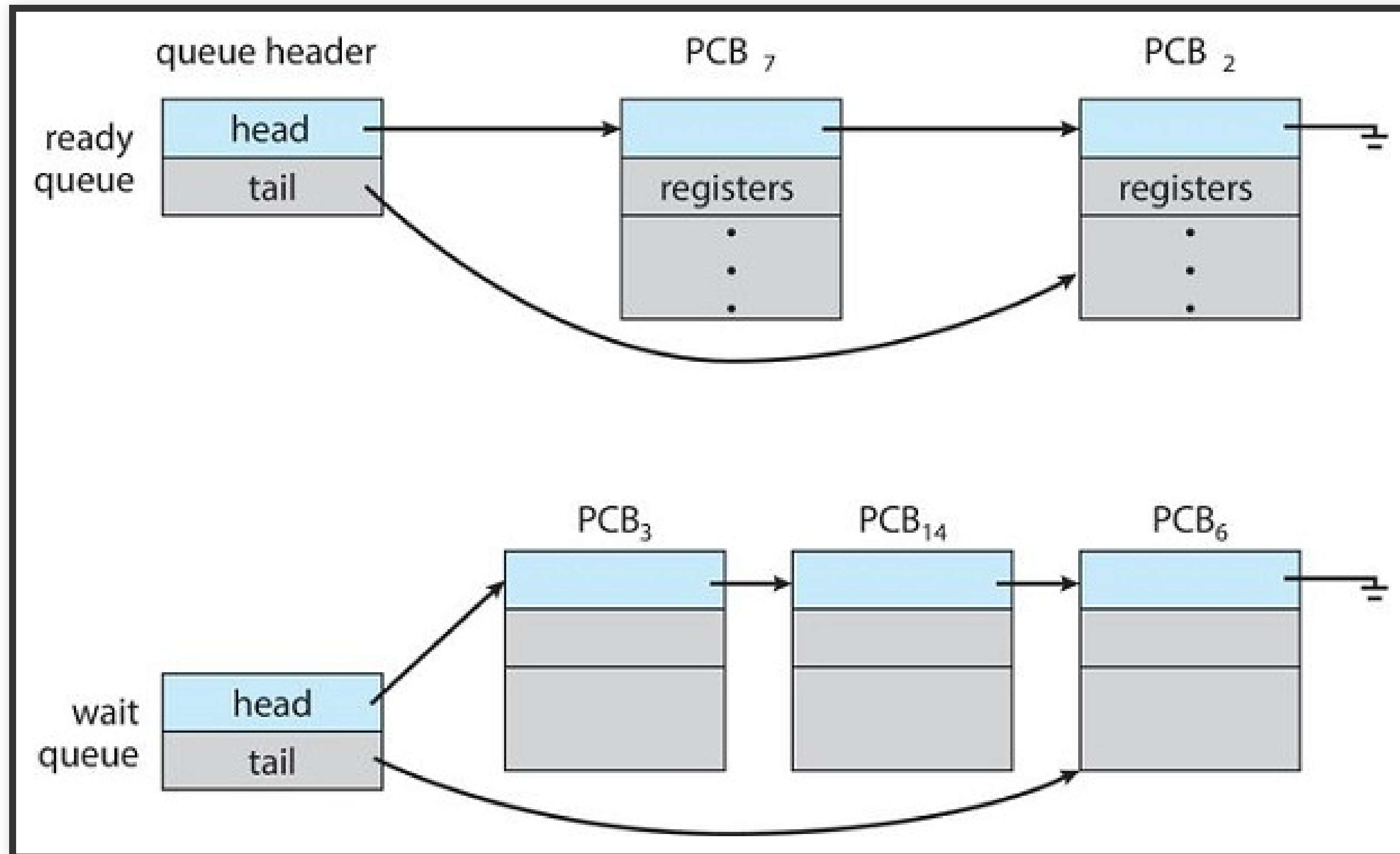
The number of processes currently in memory is known as the **degree of multiprogramming**.

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

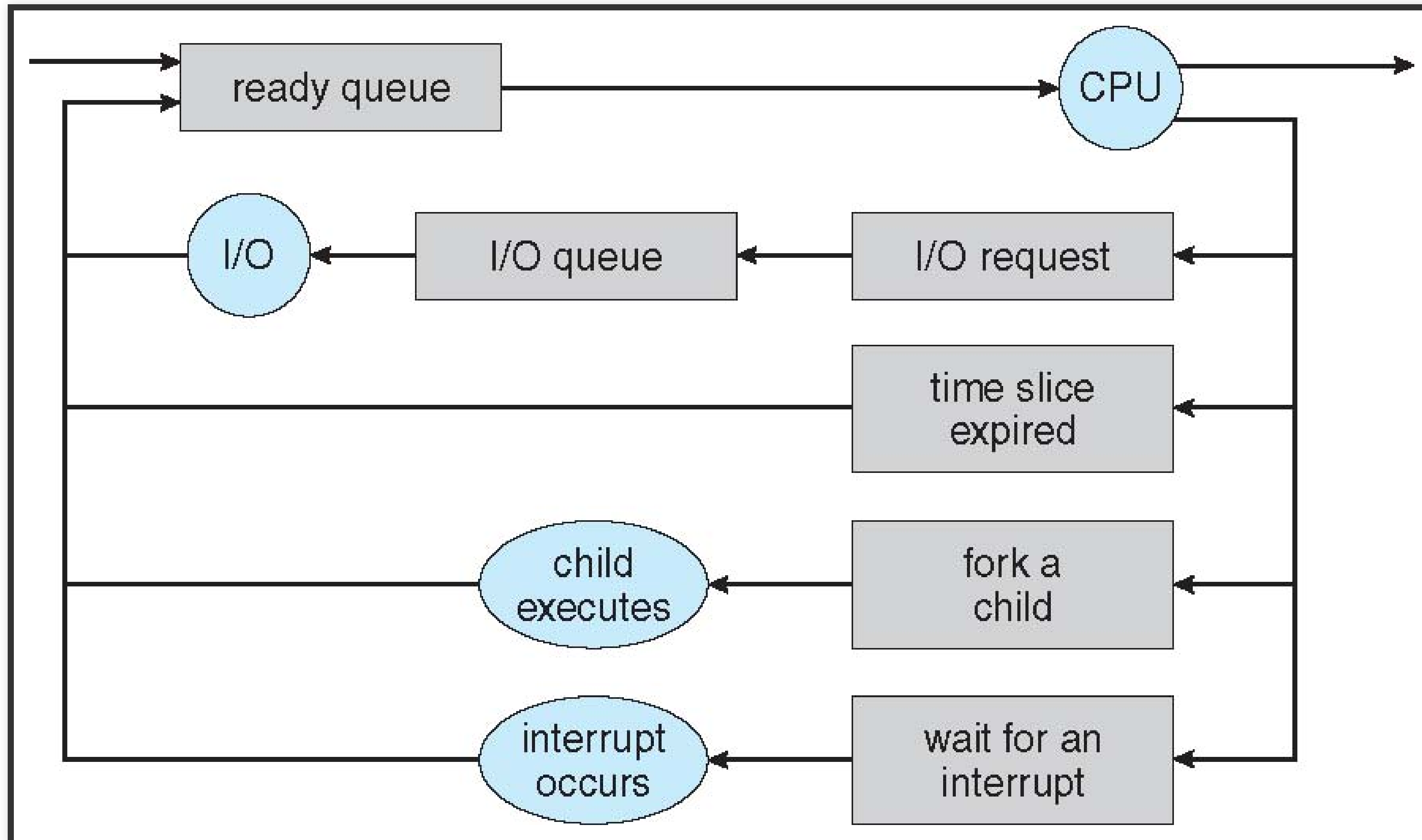
Processes migrate among the various queues

SCHEDULING QUEUES



SCHEDULING QUEUES

Queuing diagram represents queues, resources, flows



CPU SCHEDULING

The role of the CPU scheduler is to select from among the processes that are in the ready queue and allocate a CPU core to one of them.

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 scheduler executes at least once every 100 milliseconds, although typically much more frequently.

SWAPPING

Key idea: Temporarily remove process from memory and active contention of CPU by placing process on disk

Reduce degree of multiprogramming.

Later reintroduce to memory and continue execution.

Done when memory has been overcommitted (More in Chapter 9)

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

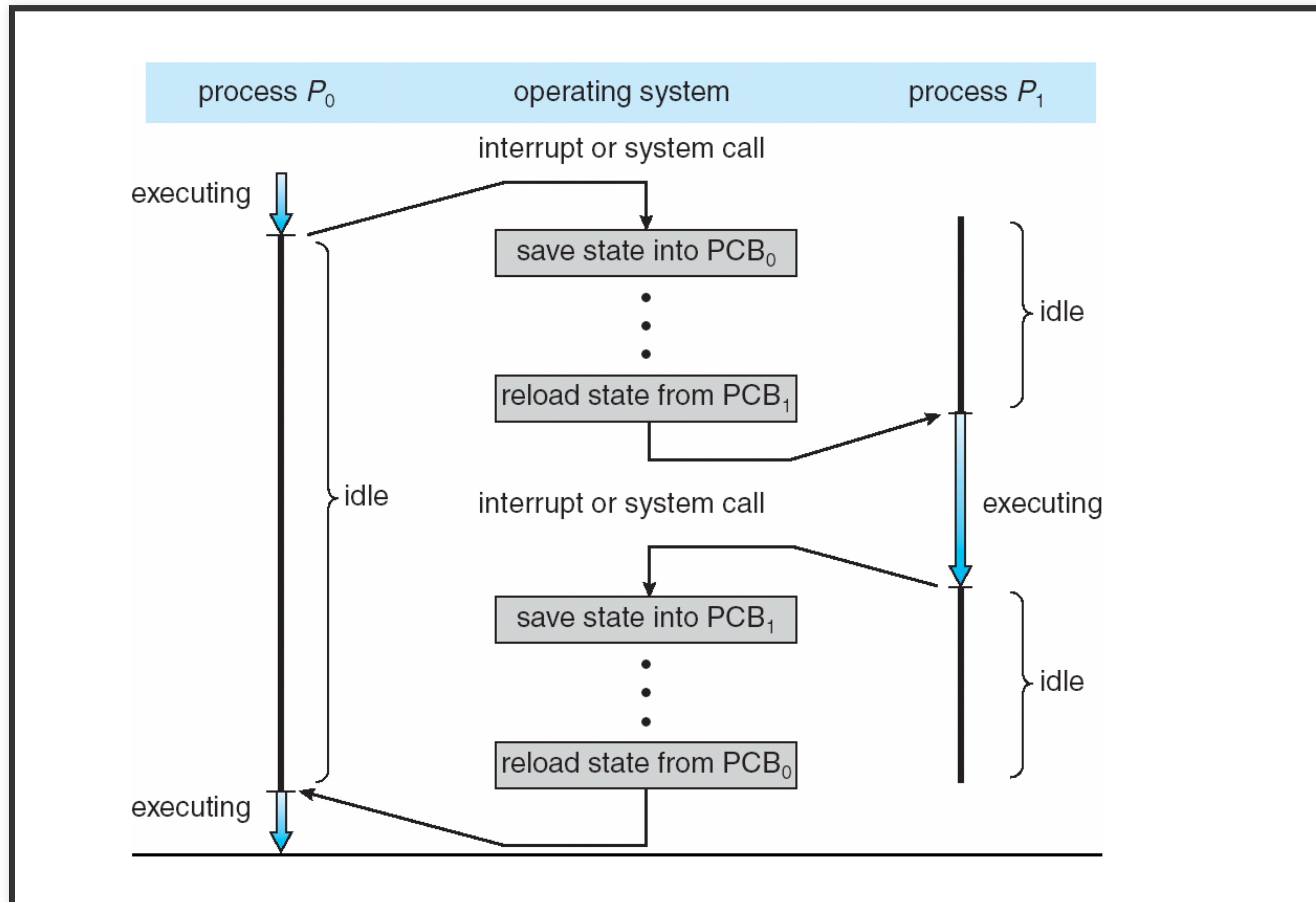
Context-switch time is overhead; the system does no useful work while switching

- 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

CPU SWITCH FROM PROCESS TO PROCESS



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

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

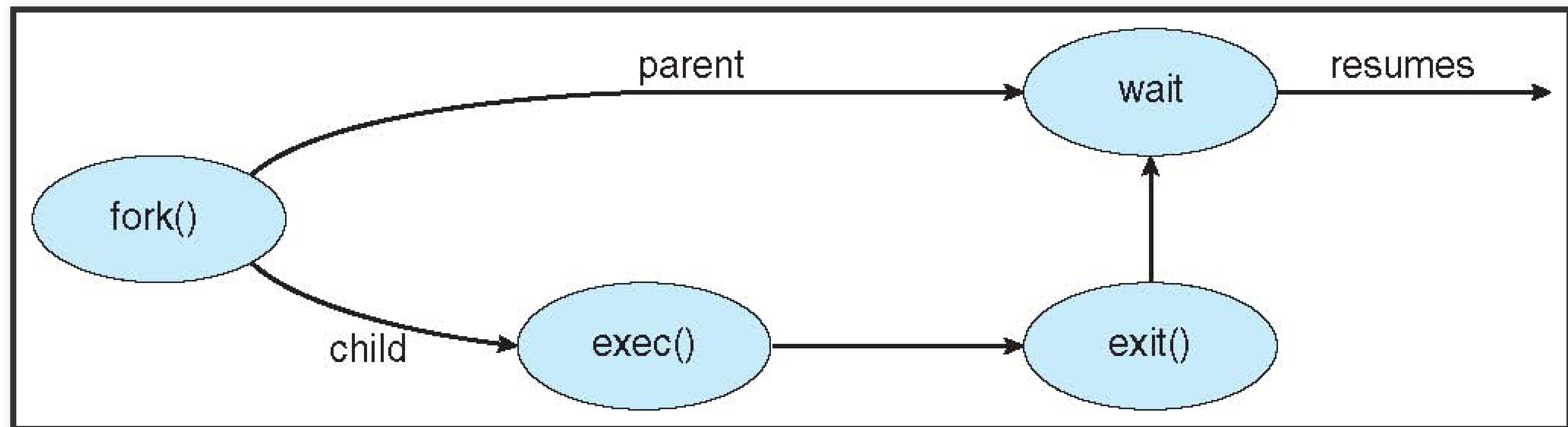
- Parent and children execute concurrently
- Parent waits until children terminate

ADDRESS SPACE

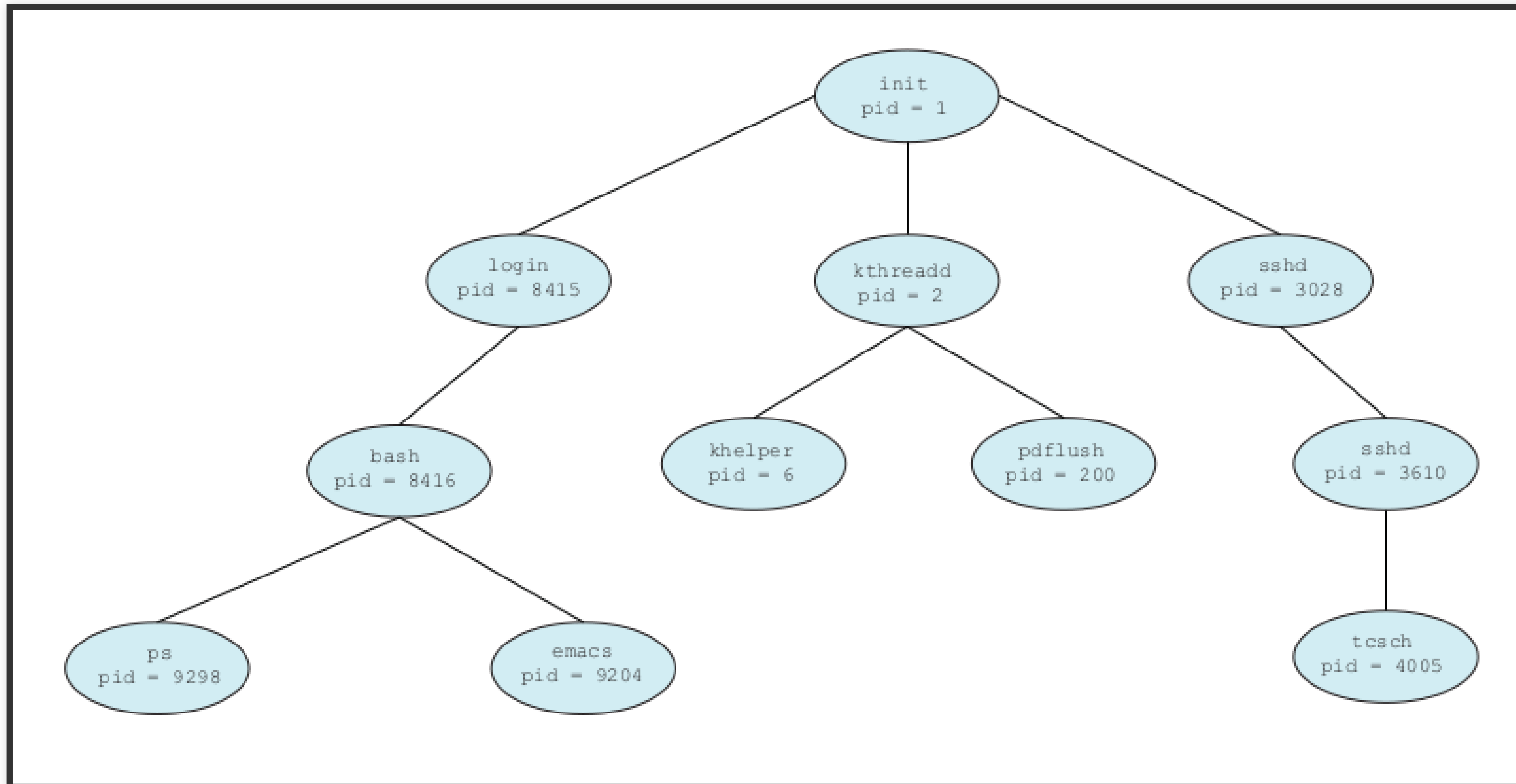
- Child duplicate of parent
- Child has a program loaded into it

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 */
        NULL, /* use parent’s existing directory */
        &si, &pi)) {
        fprintf(stderr, "Create Process Failed");
        return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");
    CloseHandle(pi.hProcess); /* close handles */
    CloseHandle(pi.hThread);
}
```

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**

ANDROID PROCESS HIERARCHY

Limited resources ⇒ May have to terminate process to reclaim. Least important is terminated

1. Foreground process (visible on screen)
2. Visible process (activity foreground is referring to)
3. Service process (similar to background, but apparent to user)
4. Background process - not visible to user
5. Empty process (No active component associated with any application)

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

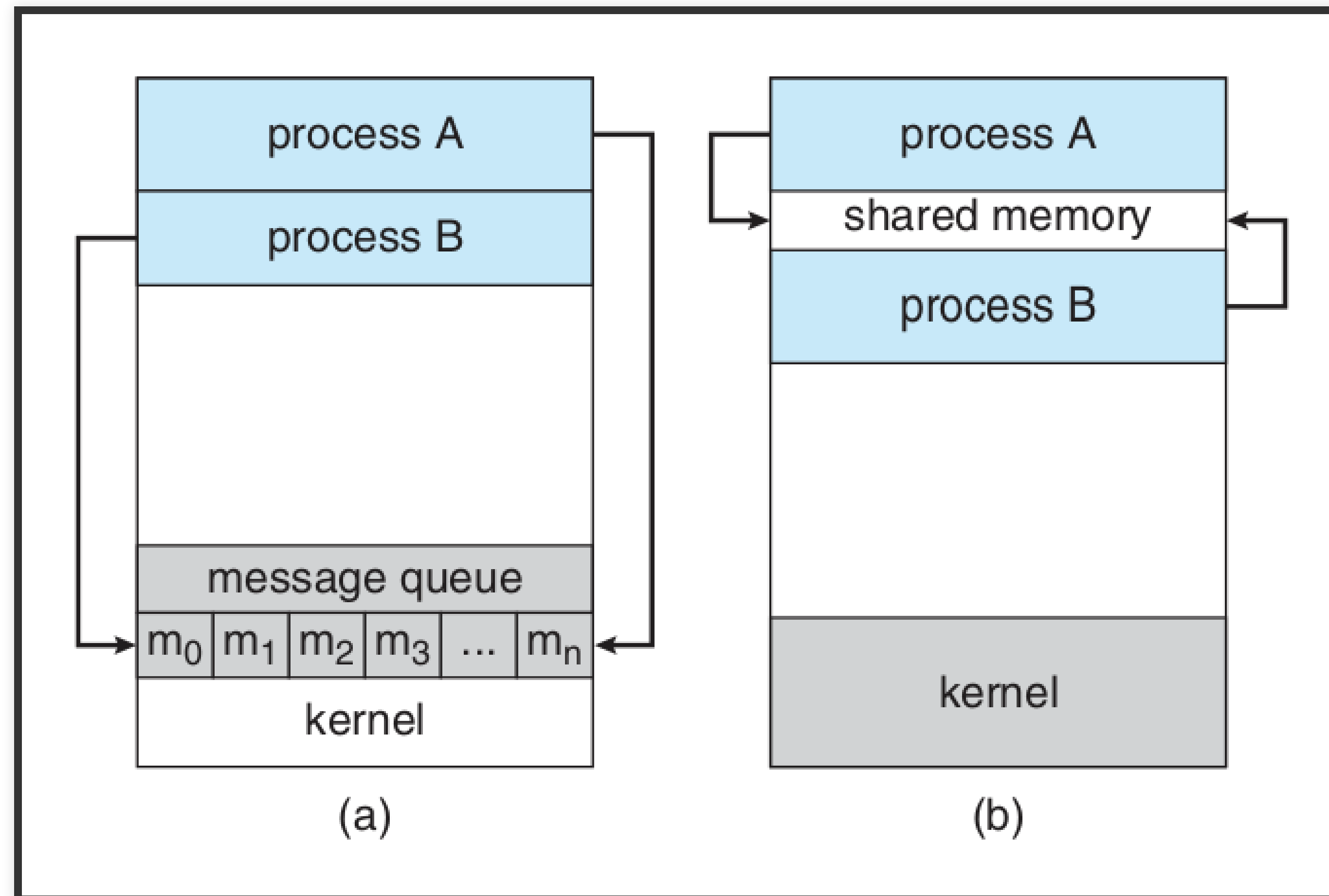
INTERPROCESS COMMUNICATION

Cooperating processes need interprocess communication (IPC)

Two models of IPC

- Shared memory
- Message passing

COMMUNICATIONS MODELS



PRODUCER-CONSUMER PROBLEM

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

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

IPC IN SHARED-MEMORY SYSTEMS

Requires communicating processes to establish a region of shared memory

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 */
}
```

IPC IN MESSAGE PASSING SYSTEMS

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

MESSAGE-PASSING OPERATIONS

- 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
  - Shared memory, hardware bus
- Logical
  - 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 - NAMING

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 - NAMING

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

DIRECT COMMUNICATION - NAMING

Asymmetry in addressing

- `send(P, message)` —Send a message to process `P`.
- `receive(id, message)` — Receive a message from any process. `id` is set to the name of the process with which communication has taken place.

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:

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

POSIX PRODUCER

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

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

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

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

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

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

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```

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 communication, 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 (1)

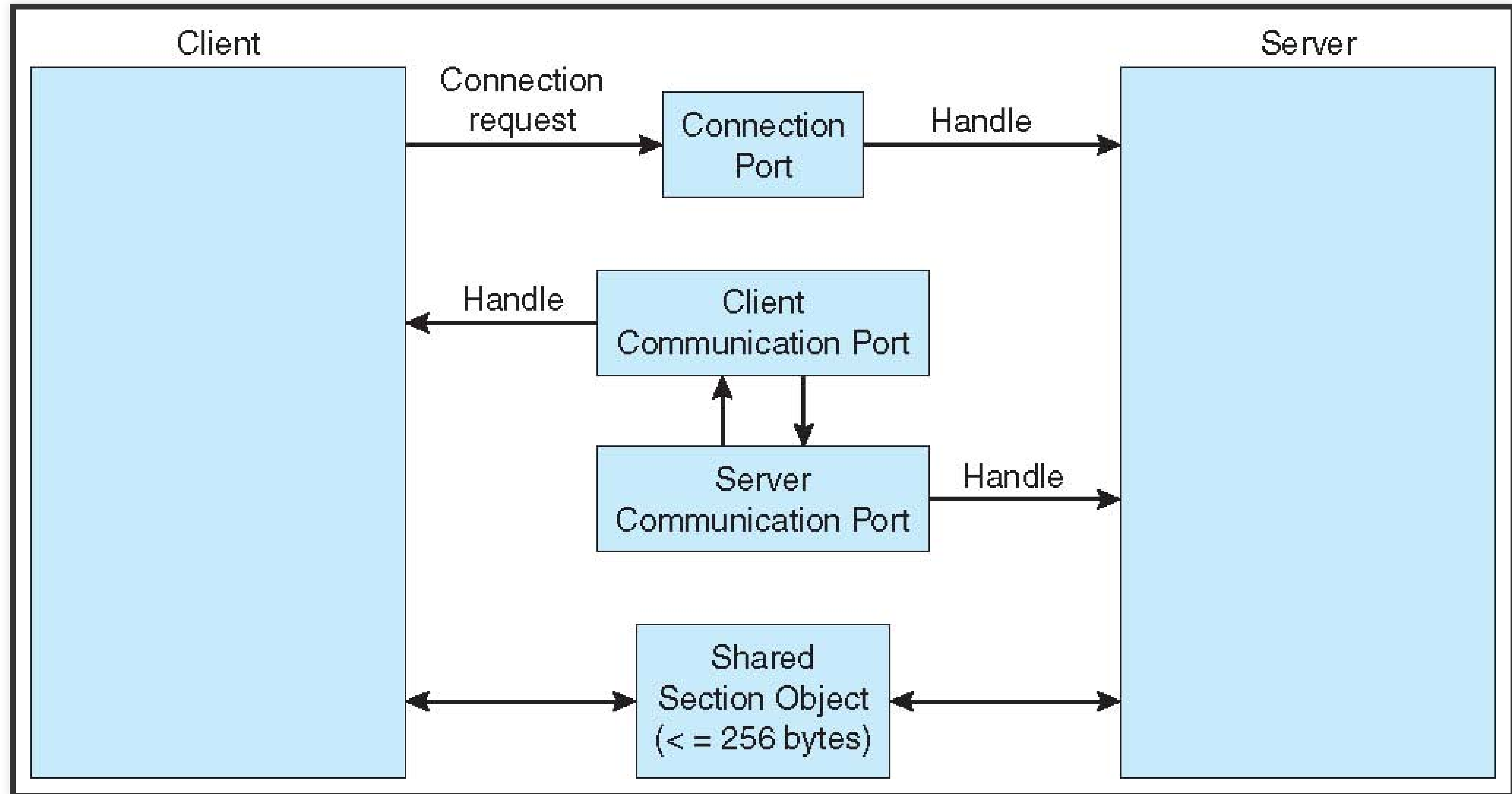
Message-passing centric via advanced local procedure call (ALPC) facility

- 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 WINDOWS



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

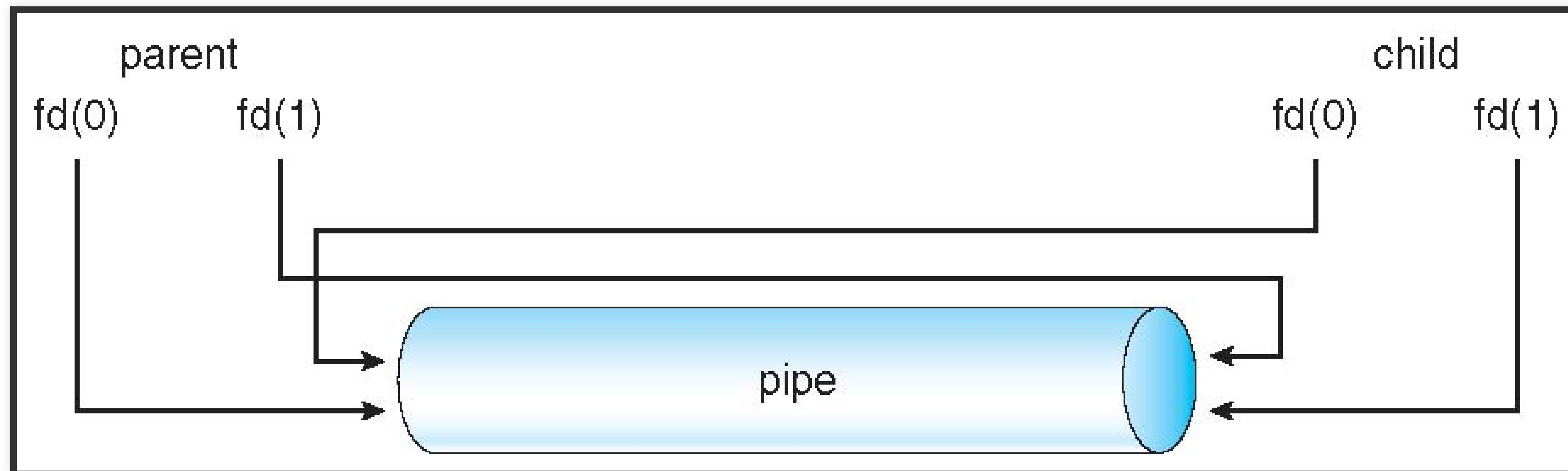
- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the write-end of the pipe)
- Consumer reads from the other end (the read-end of the pipe)
- Ordinary pipes are therefore unidirectional

Unix creation:

```
pipe(int fd[])
```

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

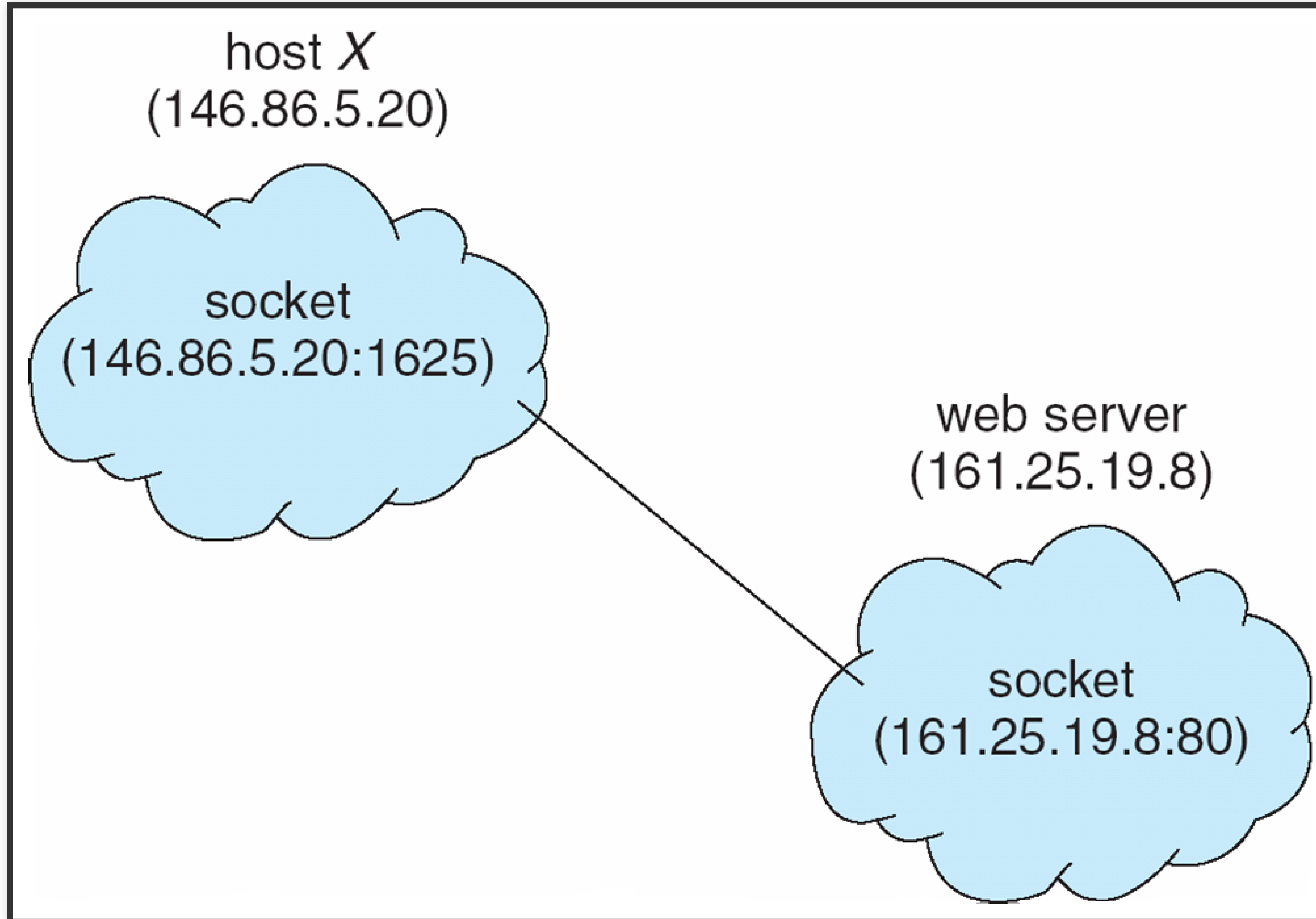
COMMUNICATION IN CLIENT – SERVER SYSTEMS

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

SOCKETS

- A socket is defined as an endpoint for communication
- Concatenation of IP address and port – 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

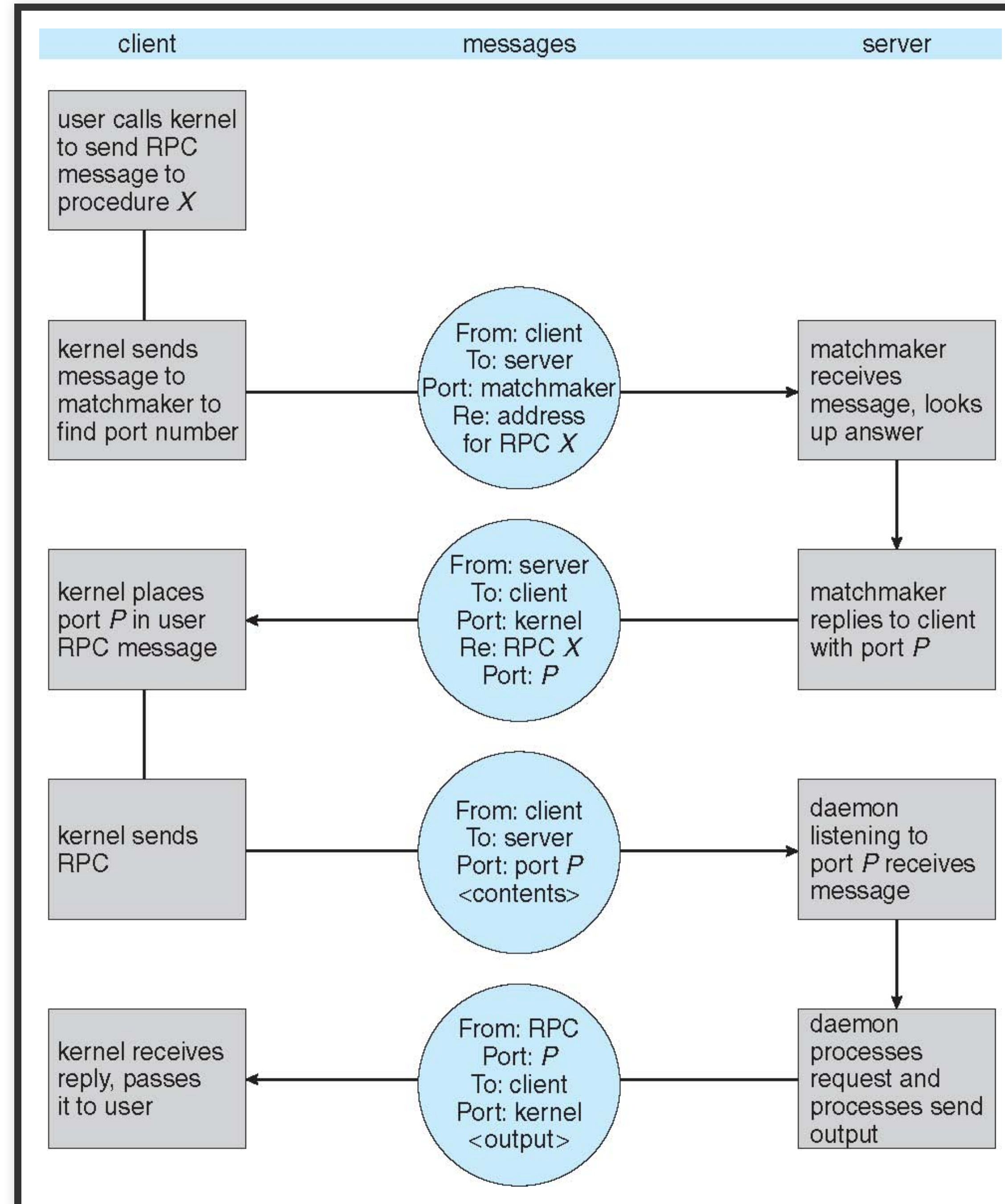
Again uses ports for service differentiation

- **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



CHAPTER 4 - THREADS AND CONCURRENCY

MOTIVATION

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request

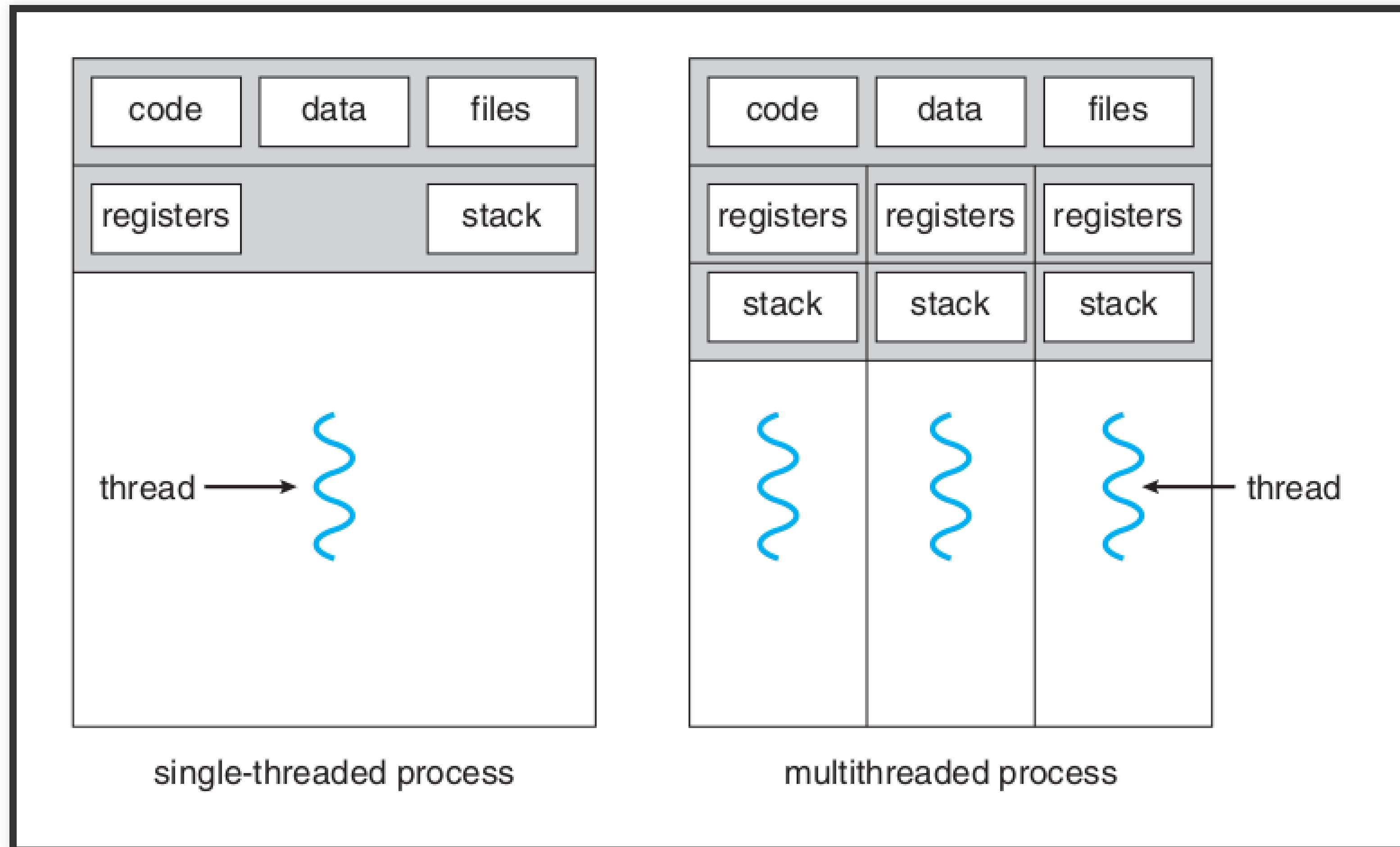
MOTIVATION

- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

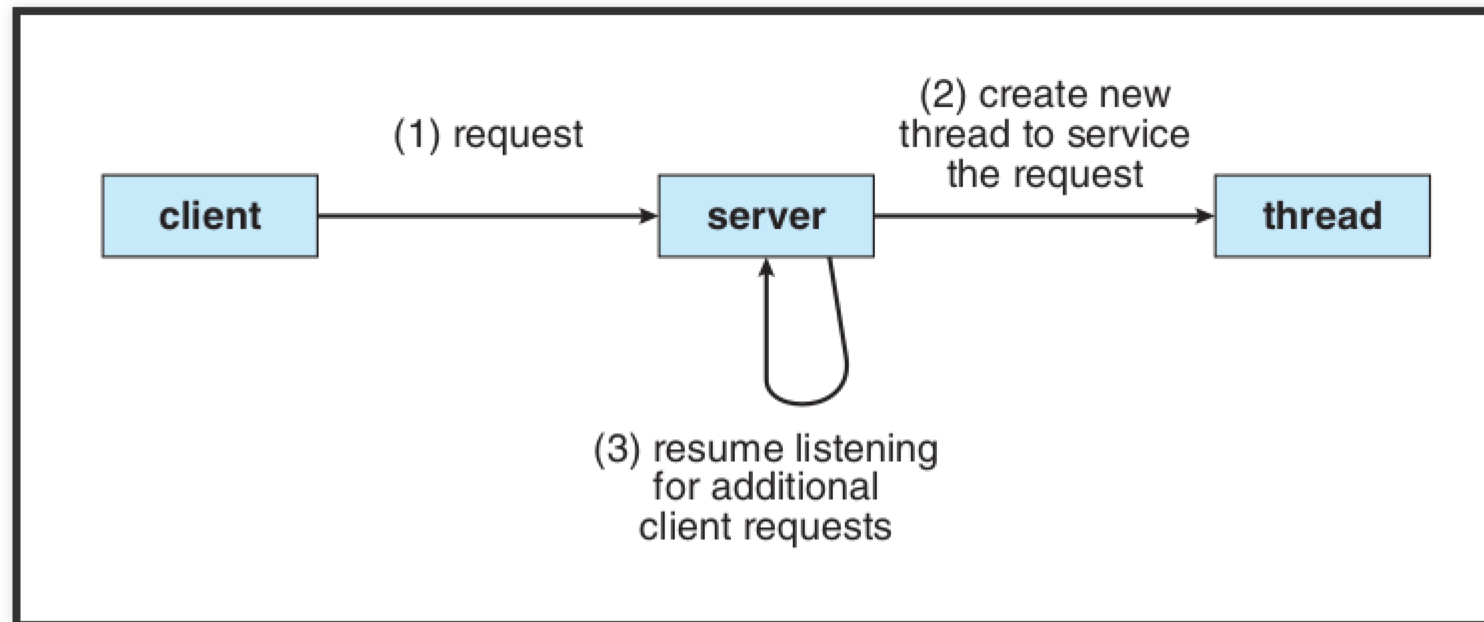
THREAD

- Basic unit of computation
  - Thread ID
  - Program counter
  - Register set
  - Stack
  - Shares
- Shares code, data, OS resources with other threads

SINGLE AND MULTITHREADED PROCESSES



MULTITHREADED SERVER ARCHITECTURE



BENEFITS

- Responsiveness
- Resource Sharing
- Economy
- Scalability

MULTICORE PROGRAMMING

MULTICORE PROGRAMMING

Multicore or multiprocessor systems putting pressure on programmers, challenges include:

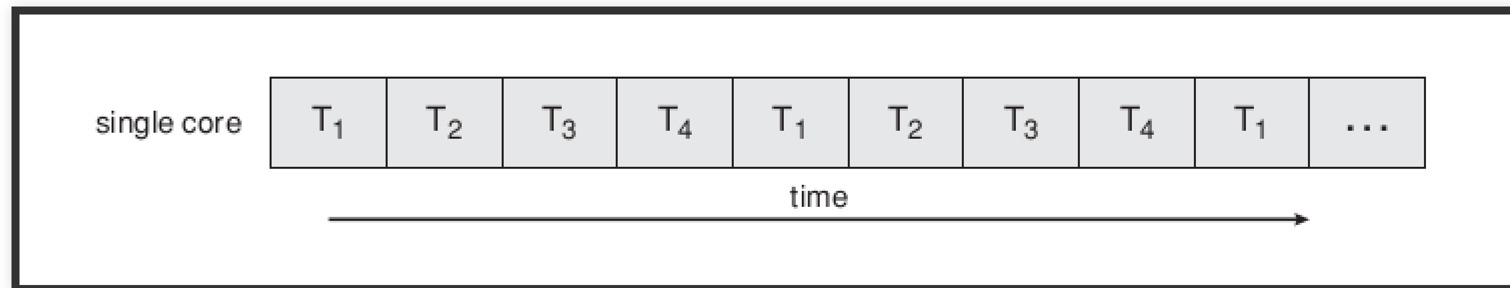
- Dividing activities
- Balance
- Data splitting
- Data dependency
- Testing and debugging

PARALLELISM VS CONCURRENCY

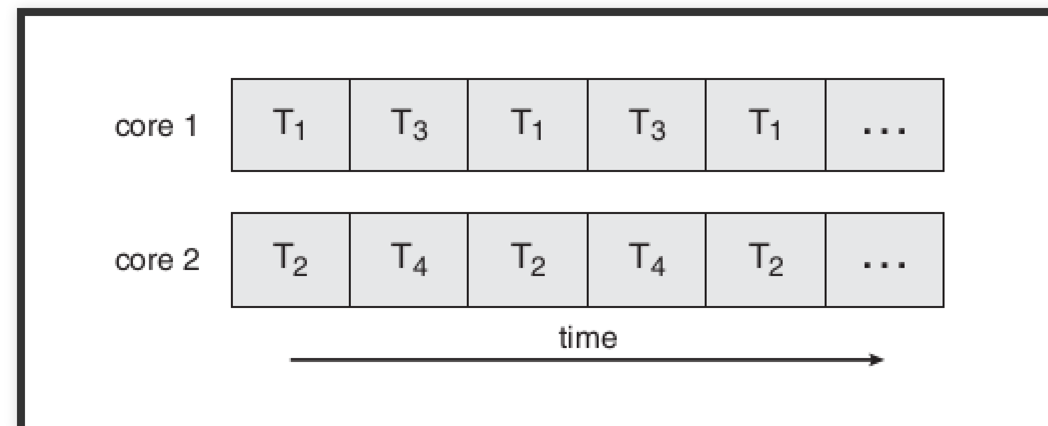
- Parallelism implies a system can perform more than one task simultaneously
- Concurrency supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

PARALLELISM VS CONCURRENCY

Concurrent execution on single-core system:



Parallelism on a multi-core system:



TYPES OF PARALLELISM

- **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
- **Task parallelism** – distributing threads across cores, each thread performing unique operation As # of threads grows, so does architectural support for threading

Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

AMDAHL'S LAW

Identifies performance gains from adding additional cores to an application that has both serial and parallel components

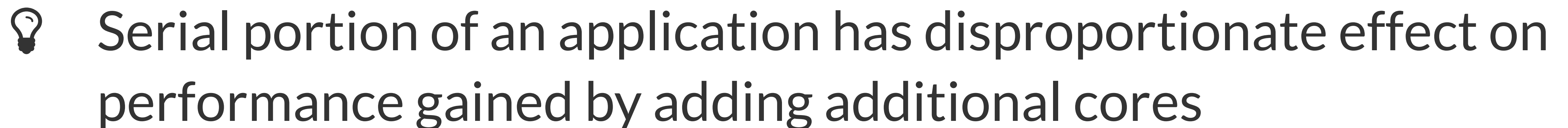
- S is serial portion
- N processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

AMDAHL'S LAW

If application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times

- As N approaches infinity, speedup approaches 1 / S



AMDAHL'S LAW

amdahls

MULTITHREADING MODELS

USER THREADS

User threads - management done by user-level threads library

Three primary thread libraries:

- POSIX Pthreads
- Windows threads
- Java threads

KERNEL THREADS

Kernel threads - Supported by the Kernel

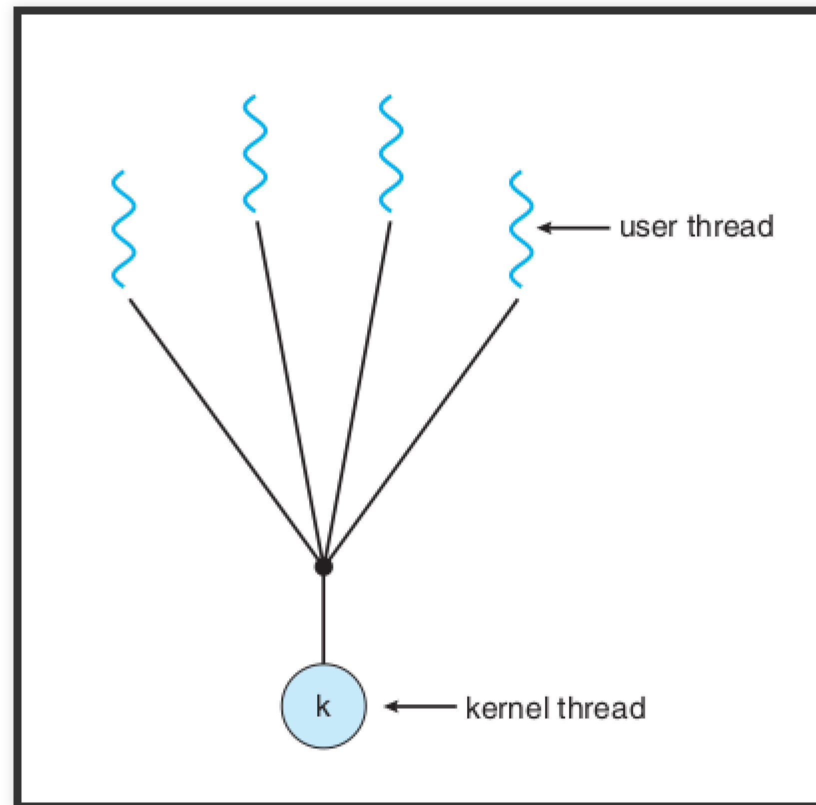
Examples – virtually all general purpose operating systems, including:

- Windows
- Solaris
- Linux
- Tru64 UNIX
- Mac OS X

MULTITHREADING MODELS

- Many-to-One
- One-to-One
- Many-to-Many

MANY-TO-ONE

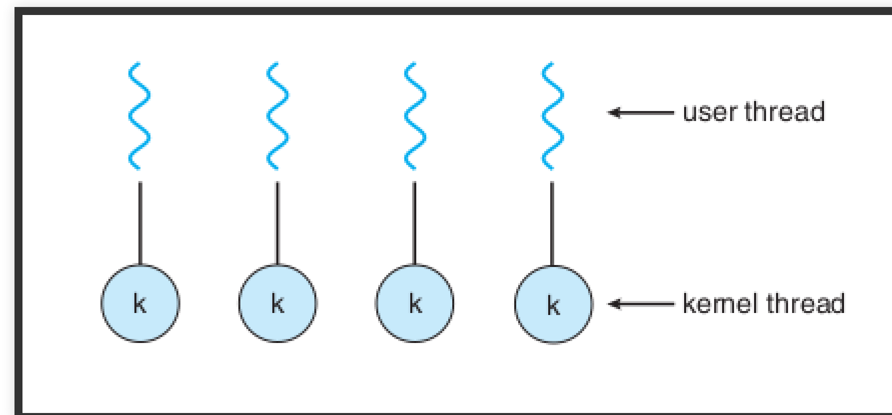


MANY-TO-ONE

Many user-level threads mapped to single kernel thread

- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

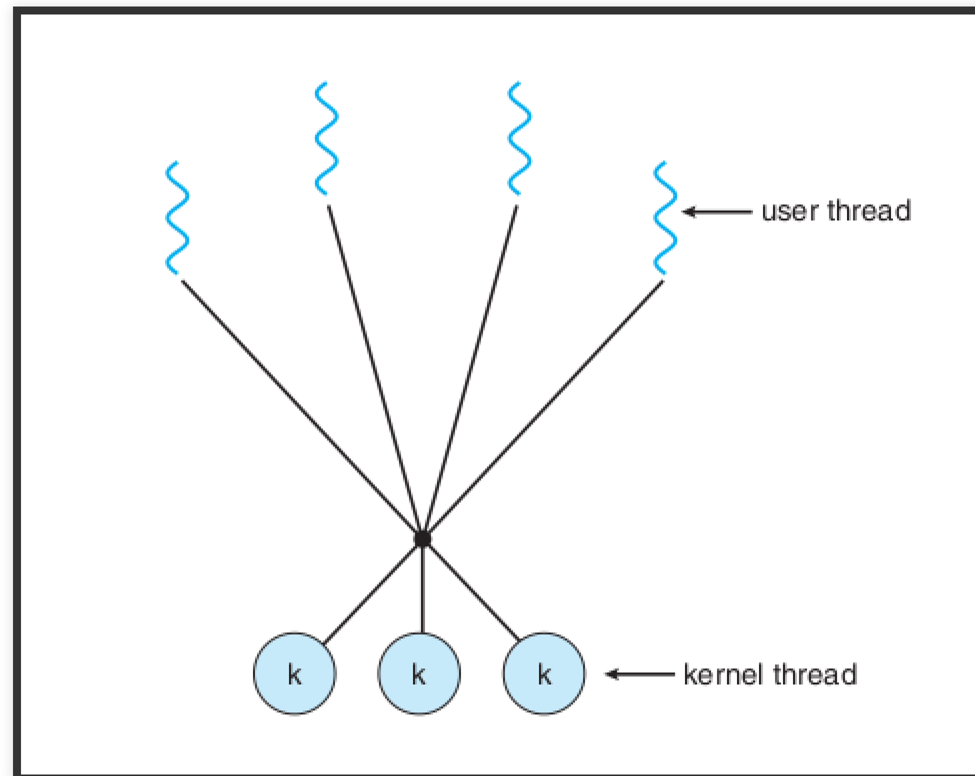
ONE-TO-ONE



ONE-TO-ONE

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

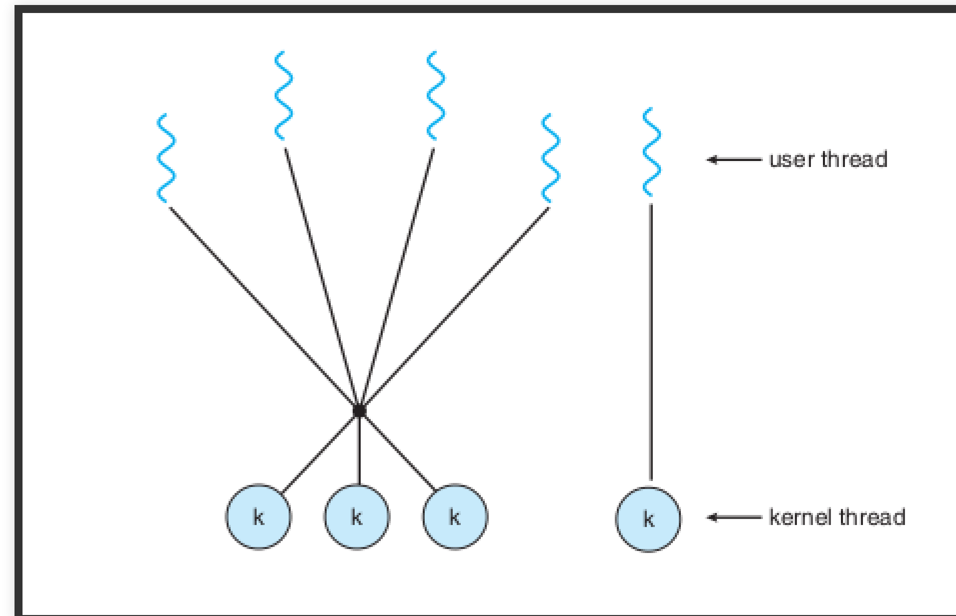
MANY-TO-MANY



MANY-TO-MANY

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the ThreadFiber package

TWO-LEVEL MODEL



TWO-LEVEL MODEL

- Similar to M:M, except that it allows a user thread to be bound to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

THREAD LIBRARIES

ASYNCHRONOUS AND SYNCHRONOUS THREADING

Asynchronous threading

Once the parent creates a child thread, the parent resumes its execution ⇒ parent and child execute concurrently and independently of one another.

Synchronous threading

Occurs when the parent thread creates one or more children and then must wait for all of its children to terminate before it resumes.

THREAD LIBRARIES

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

PTHREADS

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- Specification, not implementation
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

PTHREADS EXAMPLE

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[]) {
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value> \n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%d must be >= 0 \n",atoi(argv[1]));
        return -1;
    }

    /* get the default attributes */
    pthread_attr init(&attr);
    /* create the thread */
    pthread_create(&tid,&attr,runner,argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid,NULL);
}
```

PTHREADS CODE FOR JOINING 10 THREADS

```c
#define NUM THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM THREADS];

for (int i = 0; i < NUM THREADS; i++) {
    pthread_join(workers[i], NULL);
}
```

WIN API MULTITHREADED

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

DWORD Sum; /* data is shared by the thread(s) */
/* the thread runs in this separate function */
DWORD_WINAPI Summation(LPVOID Param) {
    DWORD_Upper = *(DWORD*)Param;
    for (DWORD i = 0; i <= Upper; i++) {
        Sum += i;
    }
    return 0;
}
int main(int argc, char *argv[]) {
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;
    if (argc != 2) {
        fprintf(stderr,"An integer parameter is required \ n");
        return -1;
    }
    Param = atoi(argv[1]);
    if (Param < 0) {
        fprintf(stderr,"An integer >= 0 is required \ n");
    return -1;
    }
    /* create the thread */
    ThreadHandle = CreateThread(
```

JAVA THREADS

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

```
public interface Runnable  {
    public abstract void run();
}
```

JAVA MULTITHREADED PROGRAM

```
class Sum {
    private int sum;
    public int getSum() {
            return sum;
    }
    public void setSum(int sum) {
        this.sum = sum;
    }
}
```

JAVA MULTITHREADED PROGRAM

```
class Summation implements Runnable  {
    private int upper;
    private Sum sumValue;
    public Summation(int upper, Sum sumValue) {
        this.upper = upper;
        this.sumValue = sumValue;
    }
    public void run() {
        int sum = 0;
        for (int i = 0; i <= upper; i++)
            sum += i;
        sumValue.setSum(sum);
    }
}
```

JAVA MULTITHREADED PROGRAM

```java
public class Driver {
  public static void main(String[] args) {
    if (args.length > 0) {
      if (Integer.parseInt(args[0]) < 0) {
        System.err.println(args[0] + " must be >= 0.");
      } else {
        Sum sumObject = new Sum();
        int upper = Integer.parseInt(args[0]);
        Thread thrd = new Thread(new Summation(upper, sumObject));
        thrd.start();
        try {
          thrd.join();
          System.out.println("The sum of "+upper+" is "+sumObject.getSum());
        } catch (InterruptedException ie) { }
      }
    } else {
      System.err.println("Usage: Summation <integer value>");
    }
  }
}
```

IMPLICIT THREADING

IMPLICIT THREADING

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers

IMPLICIT THREADING

- Four methods explored
  - Thread Pools
  - Fork Joins
  - OpenMP
  - Grand Central Dispatch
- Other methods include Microsoft Threading Building Blocks (TBB), `java.util.concurrent` package

THREAD POOLS

Create a number of threads in a pool where they await work

Advantages:

- Usually slightly faster to service a request with an existing thread than create a new thread
- Allows the number of threads in the application(s) to be bound to the size of the pool
- Separating task to be performed from mechanics of creating task allows different strategies for running task
    - i.e. Tasks could be scheduled to run periodically

THREAD POOLS

Windows API supports thread pools:

```
DWORD WINAPI PoolFunction(AVOID Param) {
    /*
     * This function runs as a separate thread
     */
}
```

GPARS (GROOVY)

```groovy
@Grab(group='org.codehaus.gpars', module='gpars', version='1.2.1')

// Calculate the n'th fibonacci number
def fibonnacci( n ) {
        if( n < 2) {
                return 1
        }
        fibonnacci(n-1) + fibonnacci(n-2)
}

groovyx.gpars.GParsPool.withPool {
        (1..50).eachParallel { int i ->
                def tId = Thread.currentThread().getId()
                println "${i} fib number: ${fibonnacci(i)} ${tId}"
        }
}
```

FORK JOIN

Main parent thread creates/forks child threads and waits for them to terminate and join with them

fork join 1

FORK JOIN IN JAVA

```
Task(problem)
    if problem is small enough
        solve the problem directly
    else
        subtask1 = fork(new Task(subset of problem)
        subtask2 = fork(new Task(subset of problem)
        result1 = join(subtask1)
        result2 = join(subtask2)
        return combined results
```

FORK JOIN IN JAVA

fork join 2

OPENMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- **Identifies parallel regions** – blocks of code that can run in parallel

OPENMP EXAMPLE

```c
#include <omp.h>
#include <stdio.h>

// Compile using:
// gcc -fopenmp  omp.c
int main(int argc, char *argv[]) {

    #pragma omp parallel
    {
        printf("I am a parallel region.\n");
    }
    return 0;
}
```

OPENMP EXAMPLE

```c
#include <omp.h>
#include <stdio.h>

// Compile using:
// gcc -fopenmp  omp2.c
int main(int argc, char *argv[]) {

    int i;

    #pragma omp parallel for
    for( i = 0; i < 20; i++ ) {
        printf("I am a parallel for loop: %i.\n", i);
    }
    return 0;
}
```

GRAND CENTRAL DISPATCH

- Apple technology for Mac OS X and iOS operating systems
- Extensions to C, C++ languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Block is in `"^{ }"`
  - `^{ printf("I am a block"); }`
- Blocks placed in dispatch queue
  - Assigned to available thread in thread pool when removed from queue

GRAND CENTRAL DISPATCH

- Two types of dispatch queues:
  - **serial** – blocks removed in FIFO order, queue is per process, called main queue
    - Programmers can create additional serial queues within program
  - **concurrent** – removed in FIFO order but several may be removed at a time
- three system wide queues with priorities low, default, high

THREADING ISSUES

THREADING ISSUES

- Semantics of `fork()` and `exec()` system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

SEMANTICS

Semantics of `fork()` and `exec()`

- Does `fork()` duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork
- `Exec()` usually works as normal – replace the running process including all threads

SIGNAL HANDLING

- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals

1. Signal is generated by particular event
2. Signal is delivered to a process
3. Signal is handled by one of two signal handlers:
    1. default
    2. user-defined

SIGNAL HANDLING

Every signal has default handler that kernel runs when handling signal

- User-defined signal handler can override default
- For single-threaded, signal delivered to process

SIGNAL HANDLING

Where should a signal be delivered for multi-threaded?

- Deliver the signal to the thread to which the signal applies
- Deliver the signal to every thread in the process
- Deliver the signal to certain threads in the process
- Assign a specific thread to receive all signals for the process

THREAD CANCELLATION

- Terminating a thread before it has finished
- Thread to be canceled is target thread
- Two general approaches:
  - Asynchronous cancellation terminates the target thread immediately
  - Deferred cancellation allows the target thread to periodically check if it should be cancelled

THREAD CANCELLATION

Pthread code to create and cancel a thread:

```
pthread_t tid;

/* Create the thread*/
pthread_create(&tid, 0, worker, NULL);
...
/* Cancel the thread */
pthread_cancel(tid);
```

THREAD CANCELLATION

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

THREAD CANCELLATION

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches cancellation point
    - I.e. `pthread_testcancel()`
    - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals

THREAD-LOCAL STORAGE

- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

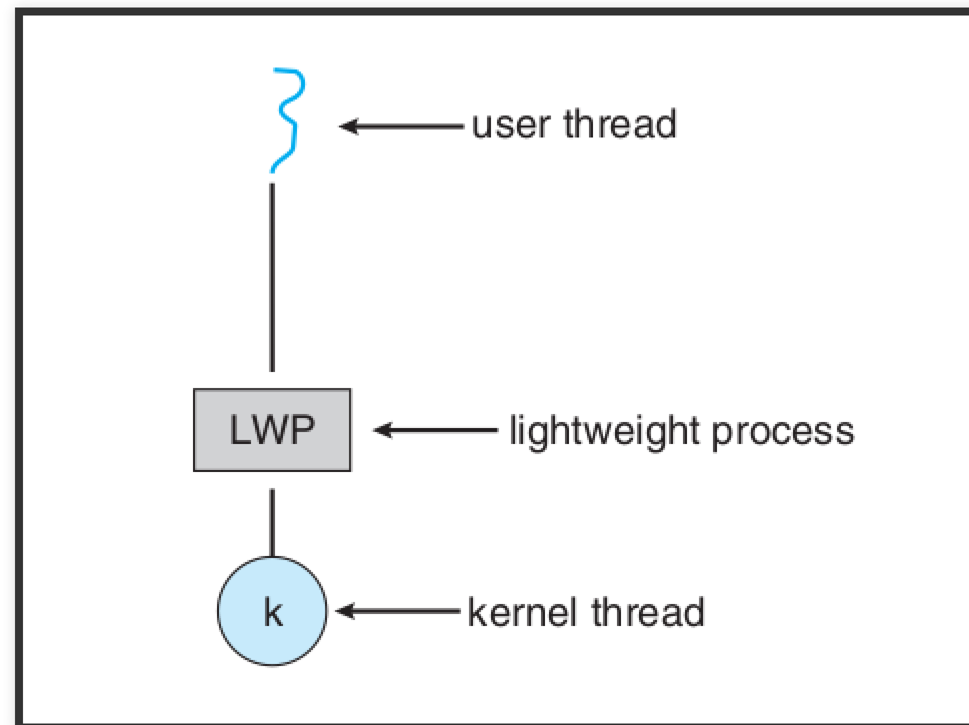
THREAD-LOCAL STORAGE

- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to static data
  - TLS is unique to each thread

SCHEDULER ACTIVATIONS

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – lightweight process (LWP)
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?

SCHEDULER ACTIVATIONS



SCHEDULER ACTIVATIONS

- Scheduler activations provide upcalls - a communication mechanism from the kernel to the upcall handler in the thread library
- This communication allows an application to maintain the correct number kernel threads

OPERATING SYSTEM EXAMPLES

OPERATING SYSTEM EXAMPLES

- Windows Threads
- Linux Thread

WINDOWS THREADS

Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7

- Implements the one-to-one mapping, kernel-level

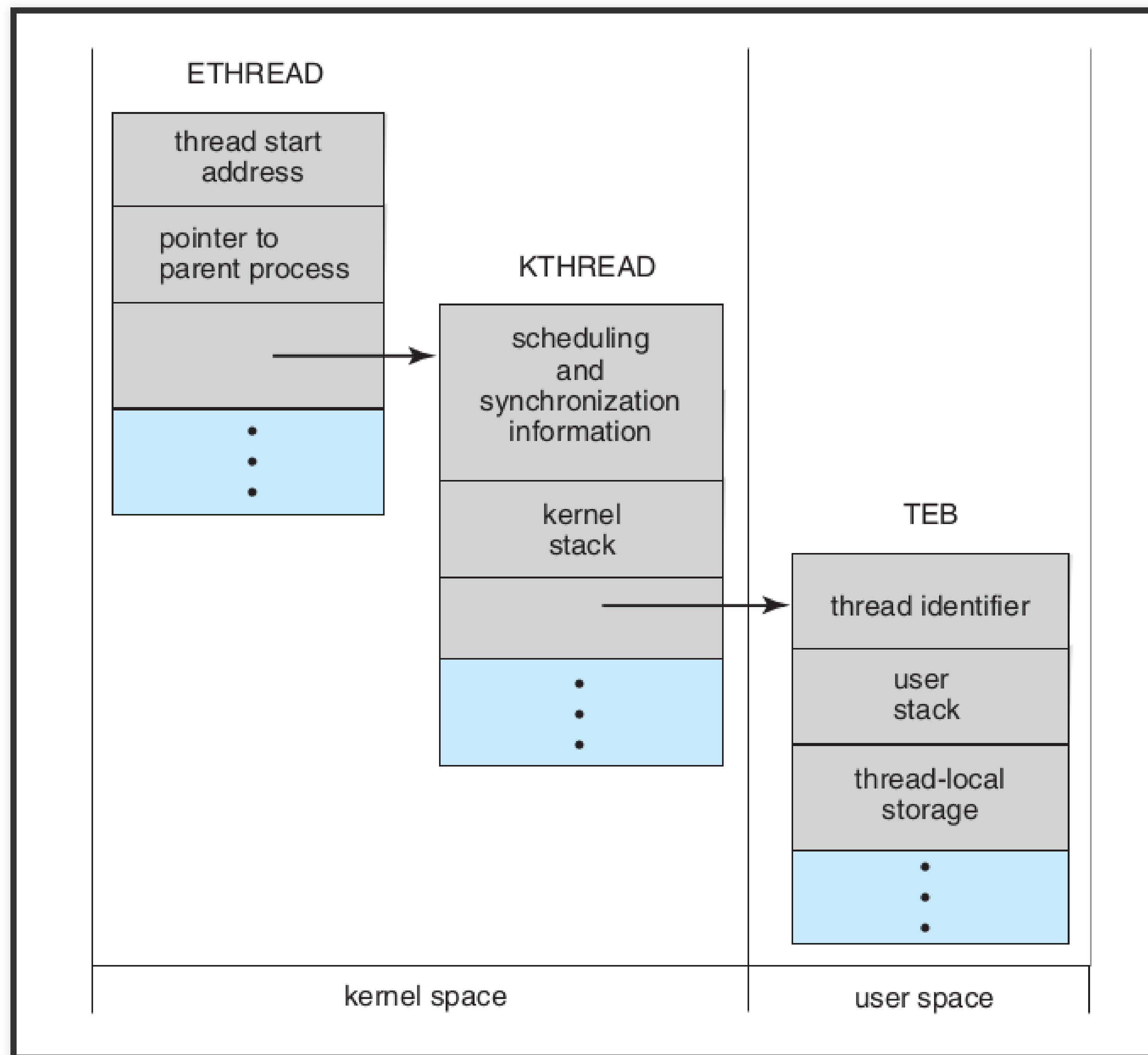
WINDOWS THREADS

- Each thread contains
  - A thread id
  - Register set representing state of processor
  - Separate user and kernel stacks for when thread runs in user mode or kernel mode
  - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the context of the thread

WINDOWS THREADS

- The primary data structures of a thread include:
  - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
  - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
  - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space

WIN THREADS DATA STRUCTURES


ETHREAD
thread start address
pointer to parent process
KTHREAD
scheduling and synchronization information
kernel stack
TEB
thread identifier
user stack
thread-local storage
kernel space
user space

LINUX THREADS

Linux refers to them as tasks rather than threads or processes, as it does not distinguish between them.

- Process creation is done through `fork()` system call
- Thread creation is done through `clone()` system call

LINUX THREADS

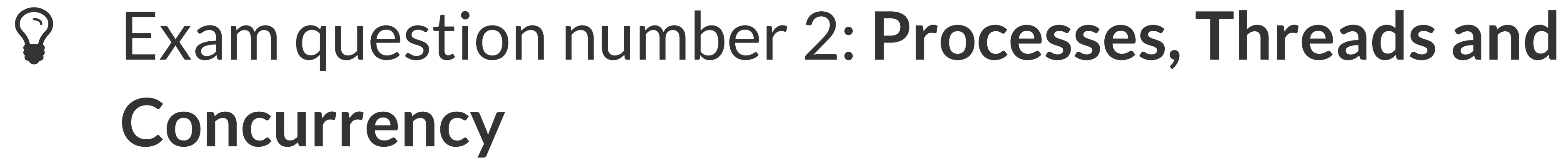
- `clone()` allows a child task to share the address space of the parent task (process)
  - Flags control behavior

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- `struct task_struct` points to process data structures (shared or unique)

QUESTIONS

BONUS



QUESTIONS

BONUS