EDAF35: OPERATING SYSTEMS

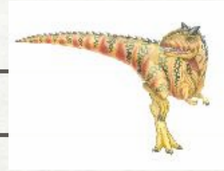
MODULE 3
PROCESSES, THREADS

CONTENTS

MODULE 3

- “Process” concept, features and operations
- Inter-process communication (IPC)
- Examples from common OS
- “Thread” concept, relation to processes
- Multithreading and OS
- Examples of threading APIs

CHAPTER 3



CHAPTER 4

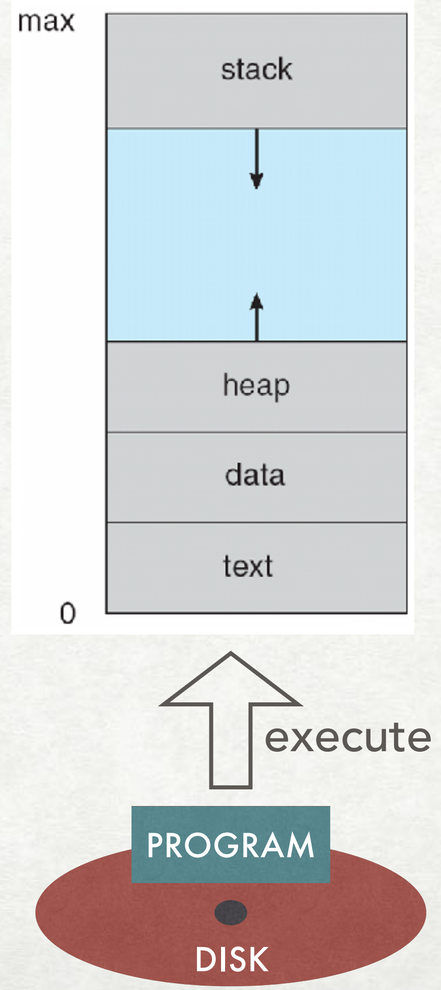
PROCESSES (CH3)

A FEW DEFINITIONS

PROCESSES

- job (batch systems), task/user program (time sharing system) = process
- **process** — "a program in execution"
- sequence of instructions (text), data (heap, stack,…), current instruction to execute (PC), other state info, etc.

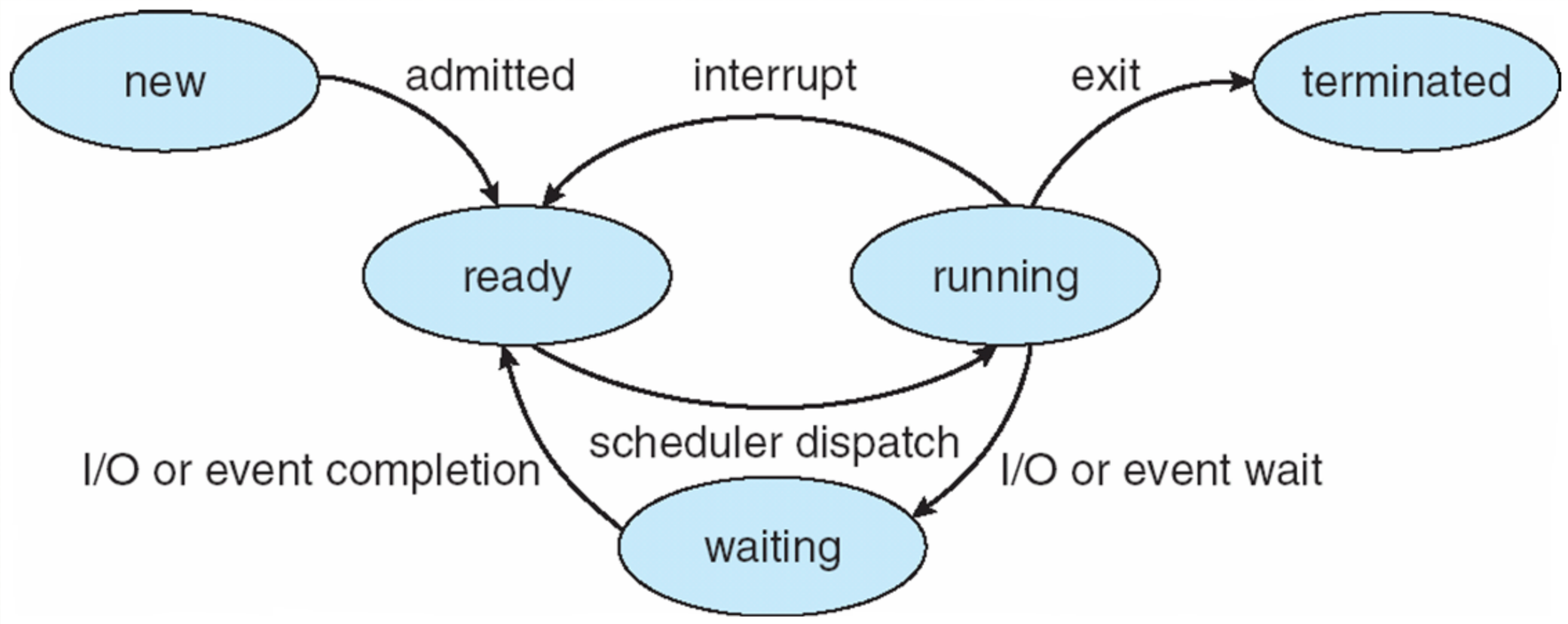
A process (in memory)



PROCESS STATE

PROCESSES

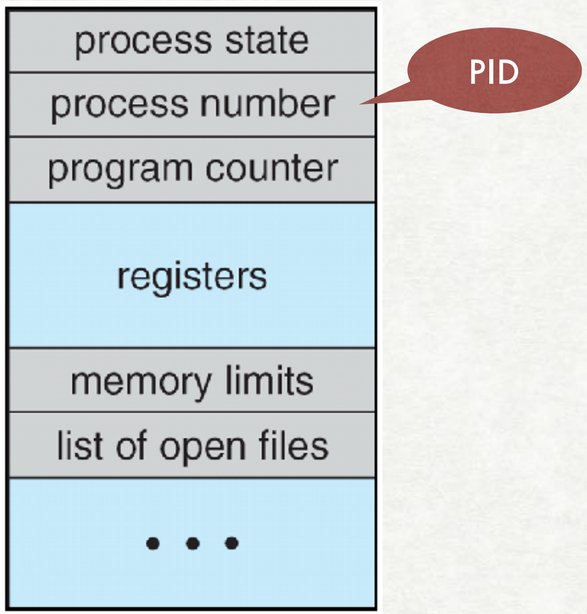
- multiple processes on the same CPU
- only one active (running) at any time



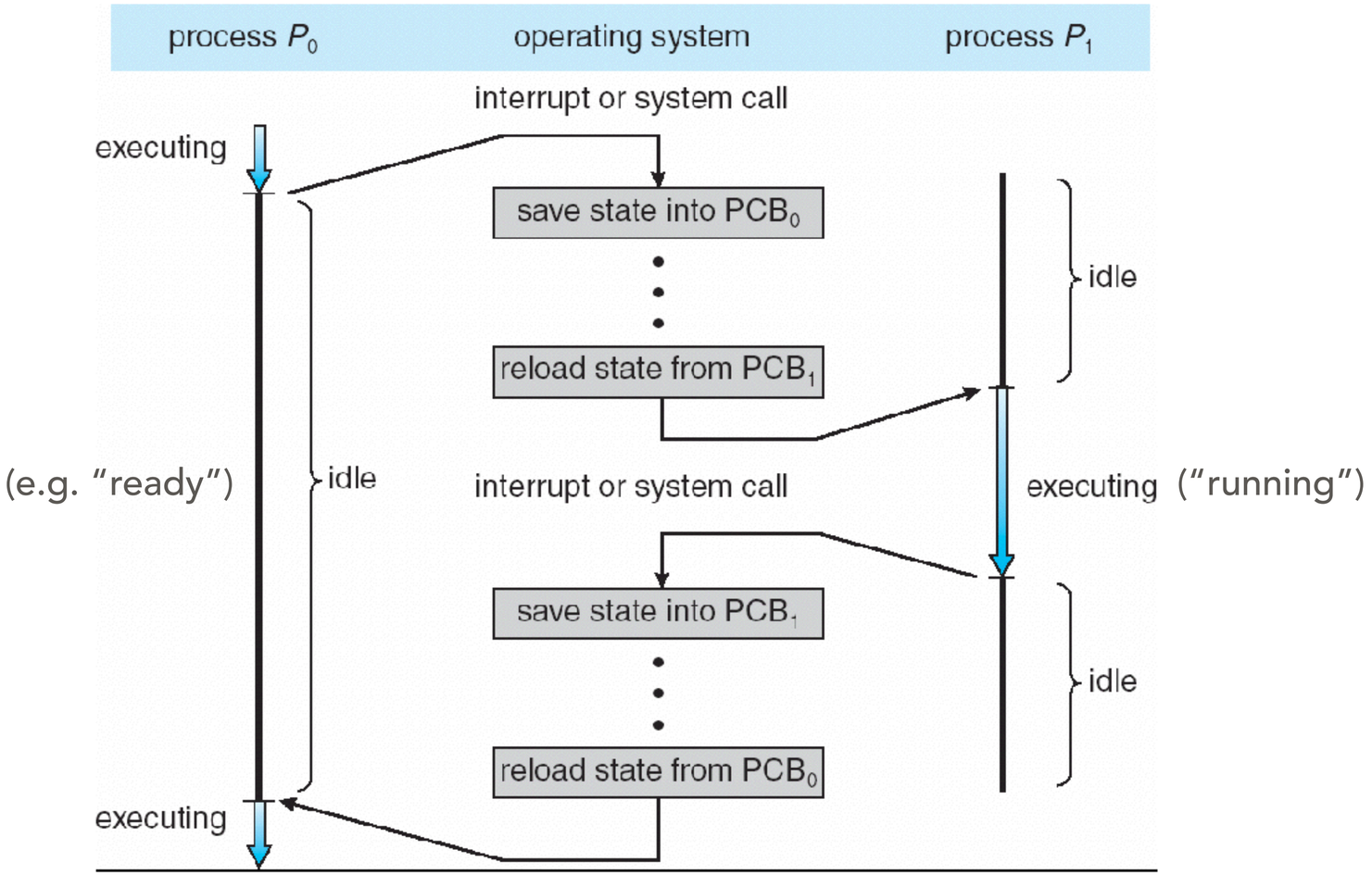
PCB AND CONTEXT SWITCHES

PROCESSES

Process Control Block (PCB)



Context Switch from P0 to P1 and back

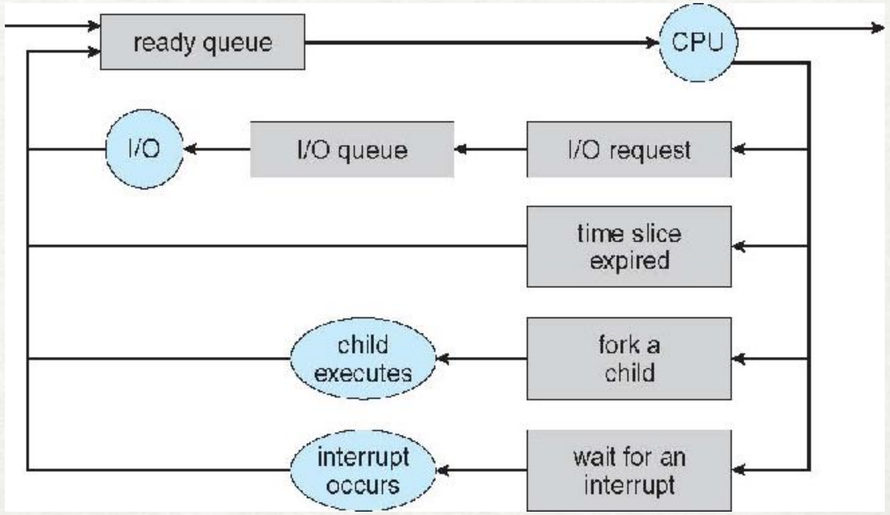


PROCESS SCHEDULING

PROCESSES

- **multiprogramming** — keep several processes in memory and run them concurrently
- a goal: maximize the use of the CPU
- processes migrate among different queues: **job queue** — all processes, **ready queue** — waiting to run, **device queue** — waiting for a particular I/O device
- schedulers — selects which process to run next
- good mix of I/O-bound and CPU-bound processes

more in Module 5

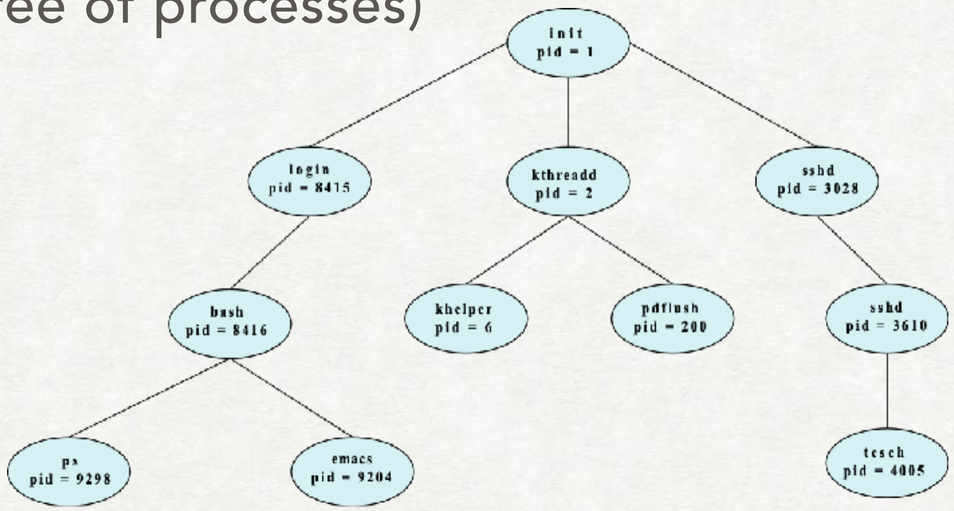


A “queuing diagram” — helper tool

OPERATIONS ON
PROCESSES

CREATE

- “parent” creates “children” (tree of processes)



- how do they execute relative to each other? (`wait`)
- what happens to the parent’s resources?

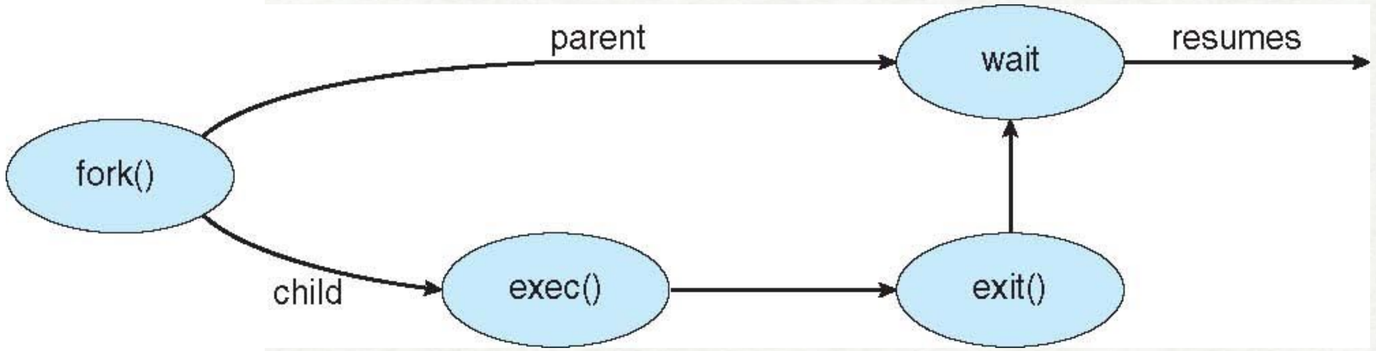
TERMINATE

- normally — execute last instruction, produces a exit code (main `return` or `exit`)
- parent terminates a child (identified via “pid”, given on creation)
- choice: whole branch or only one?
- zombie vs. orphan processes

CHECK MAN PAGES FOR:
`FORK, EXEC, WAIT, EXIT, PS, KILL`

FORKING PROCESSES IN UNIX (C)

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

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```



`fork()` returns both in child (0) and in parent (child pid)

`exec()` replaces the process' memory with a new process!

(instructions after the exec line are not run)

INTERPROCESS COMMUNICATION (IPC)

PROCESSES

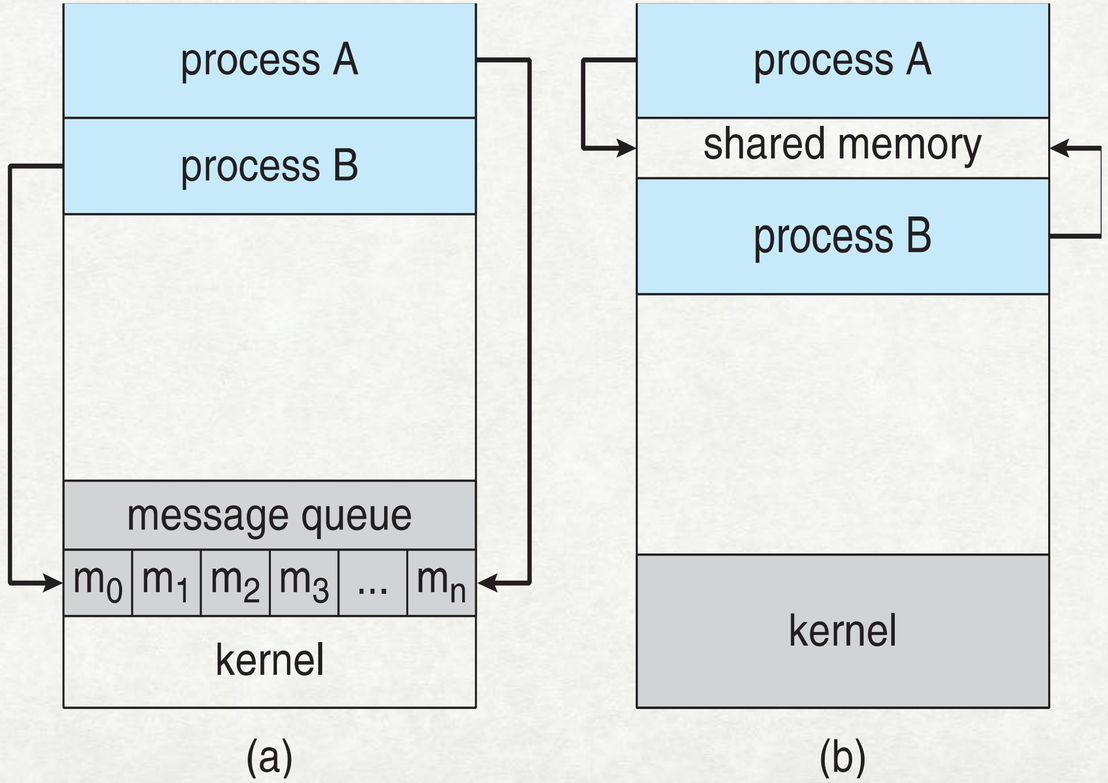
- independent vs. cooperating processes:
  sharing information, computation speedup, modularity, convenience

- two basic IPC types:

  (a) message passing

  (b) shared memory

<u>advantages and drawbacks?</u>



AN EXAMPLE: POSIX SHARED MEMORY

PROCESSES

- Create shared memory segment (“everything is a file in UNIX”):

```
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```

— other processes use it to open an existing segment.

- Set the size of the object: `ftruncate(shm_fd, 4096);`
- Map it to memory:

```
ptr = mmap(0, 4096, PROT_READ, MAP_SHARED, shm_fd, 0);
```

- Write to/read from the shared memory:

```
sprintf(ptr, "Writing to shared memory”);
```

- Remove the segment: `shm_unlink(name);`

STANDARD FILE OPERATIONS

…check also man pages…

IPC – MESSAGE PASSING

PROCESSES

- more structured and controlled than shared memory
- goto model for distributed systems
- **operations** — `send(message,…), receive(message,…)`
- **naming** — direct (process-to-process) vs. indirect communication (via mailboxes)
- **synchronization** — blocking (synchronous) vs. non-blocking (asynchronous)
- **buffering** — zero/bounded/unbounded capacity

AN EXAMPLE: MACH MESSAGE PASSING

PROCESSES

- even system calls are messages
- Kernel and Notify mailboxes (**ports**) in each task
- three system calls:
  `msg_send(), msg_receive(), msg_rpc()`
- ports created via:
  `port_allocate()`
- send and receive — flexible
  - e.g. on full mailbox choose to:
    - wait indefinitely
    - wait at most n milliseconds
    - return immediately
    - temporarily cache a message

COMMUNICATIONS IN CLIENT-SERVER SYSTEMS

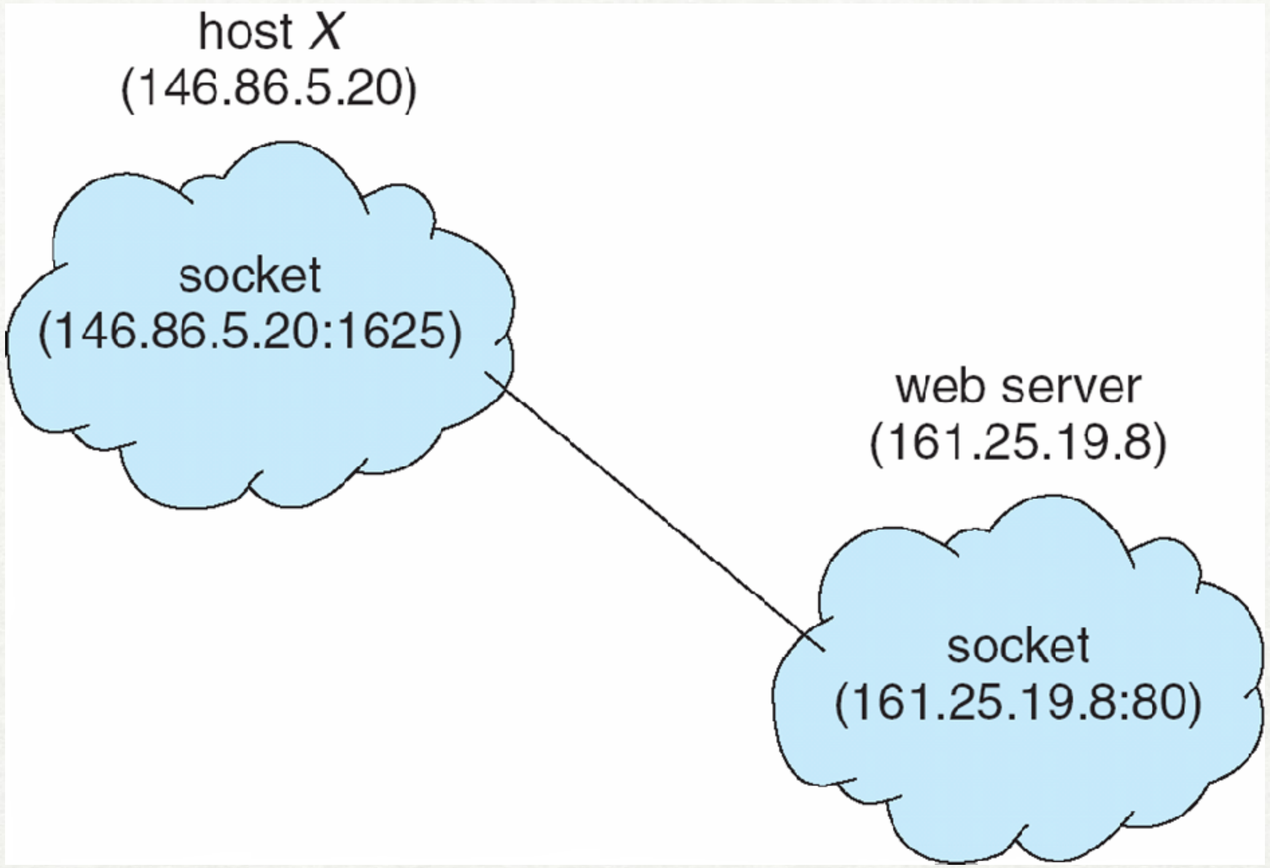
PROCESSES

- Sockets
- Remote Procedure Calls (RPC) / Remote Method Invocations (Java RMI)
- Pipes

focus of Networking and Web Programming courses — see book for more details

SOCKET COMMUNICATION

PROCESSES



SOCKETS IN JAVA

PROCESSES

- Three types of sockets:
  - connection oriented (Transmission Control Protocol **TCP**) — messages arrive in order as sent
  - connectionless (User Datagram Protocol **UDP**) — no order guarantees
  - multicast — send data to several recipients

A Date Server

```
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);
      }
   }
}
```

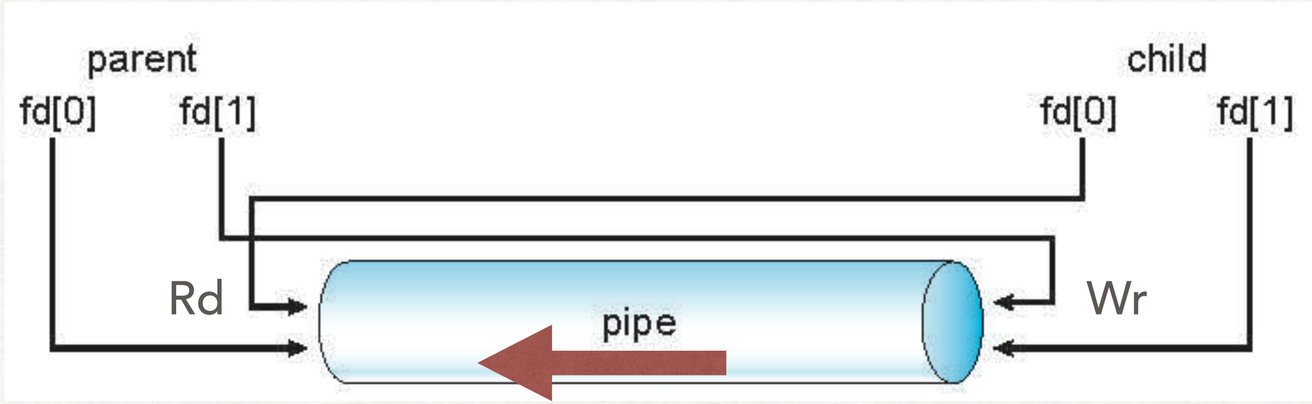
see book for the client code

PIPES

PROCESSES

- another IPC mechanism, originally from UNIX

- **choices**:
  uni- or bi-directional?
  full or half duplex?
  parent—child based or not?
  local or network based?



- ordinary (anonymous) vs. named pipes

see UNIX and Windows code samples in the book

A SIMPLE UNIX PIPE EXAMPLE

PROCESSES

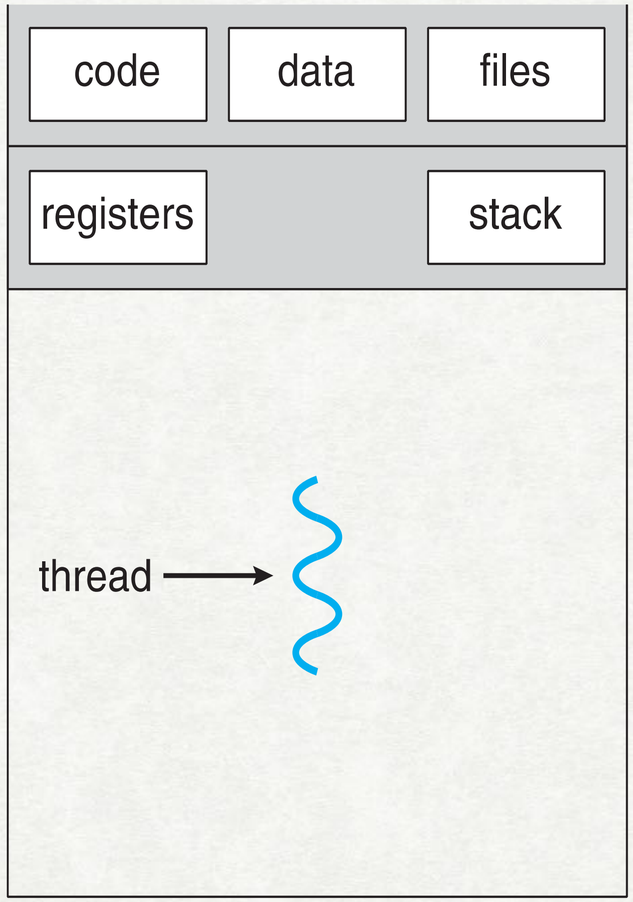
```
int fd[2];
pid_t pid;

pipe(fd);
pid = fork();

if(pid == 0) {
   dup2(fd[0], STDIN_FILENO);
   close(fd[1]);
   exec(<whatever>);
} else {
   close(fd[0]);
   …
```

Figure this out by checking the man pages!

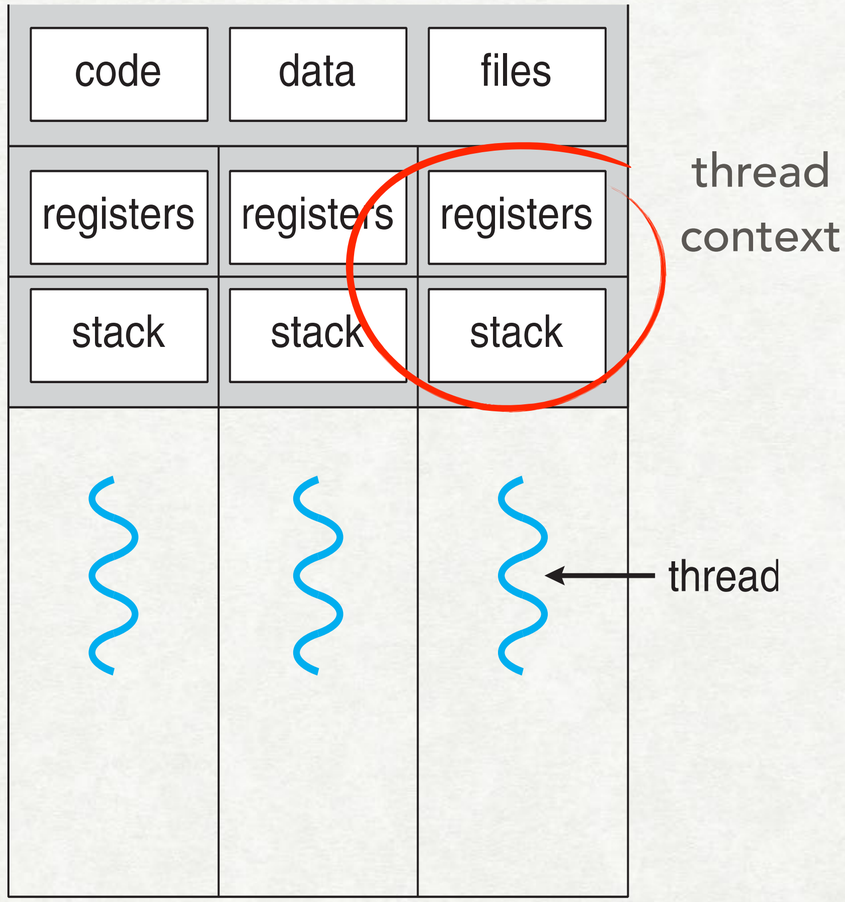
THREADS (CH4)

SINGLE- VS. MULTI-THREADED PROCESSES

THREADS



single-threaded process



multithreaded process

WHY MULTIPLE THREADS?

THREADS

- **Responsiveness** – part of a process can block while other parts still run (e.g. GUI)
- **Resource Sharing** – process resources are shared (no IPC needed)
- **Economy** – cheaper than processes, thread switching lower overhead
- **Scalability** – multithreaded processes can take advantage of multiprocessors

MULTICORE PROGRAMMING

THREADS

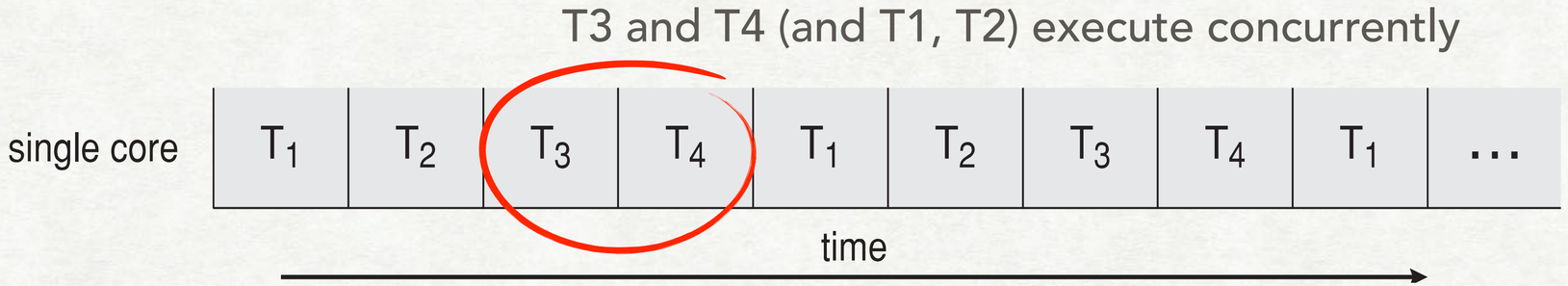
More challenging to efficiently use the architecture:

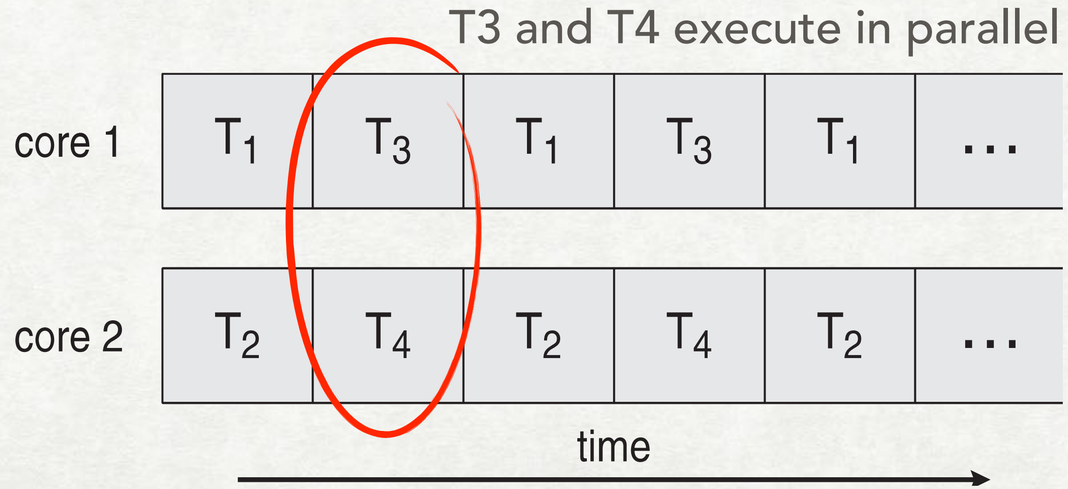
- divide activities
- balance
- divide the data
- handle dependencies
- test and debug

— see Jonas Skeppstedt course, EDAN26

CONCURRENCY VS. PARALLELISM

THREADS





PARALLELISM AND PERFORMANCE

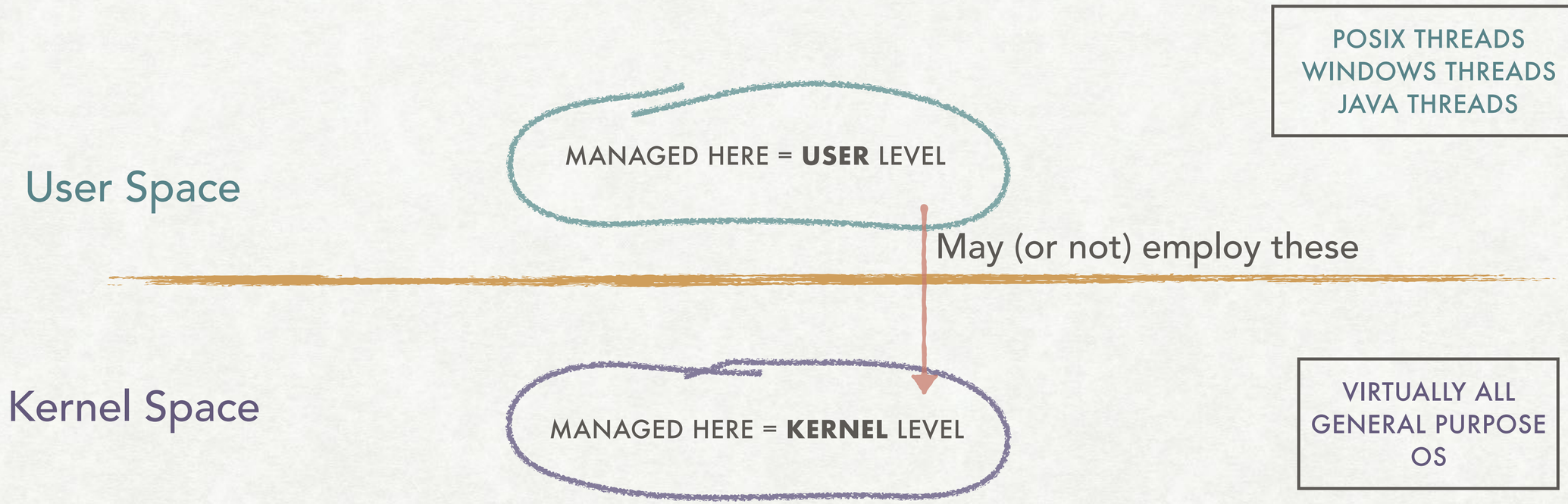
$$speedup \leq \frac{1}{S + \frac{1 - S}{N}}$$

AMDAHL'S LAW

The serial portion of an application (S) has a disproportionate effect on performance when adding additional cores (N).

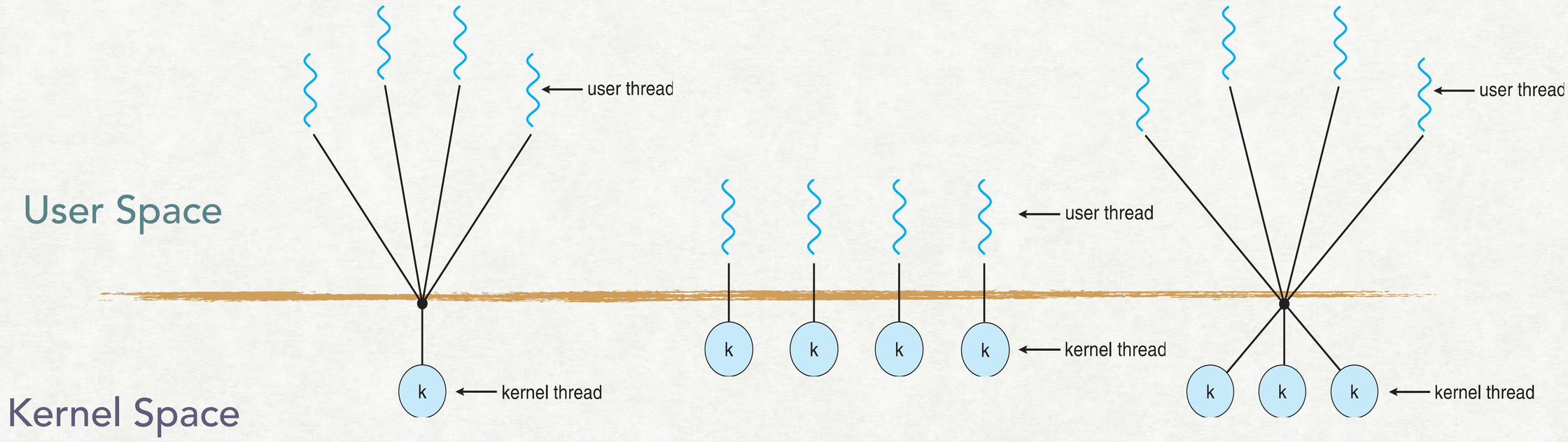
TYPE OF THREADS IN AN OS

THREADS



MULTITHREADING MODELS

MAPPING THREADS



1 many-to-one

2 one-to-one

3 many-to-many

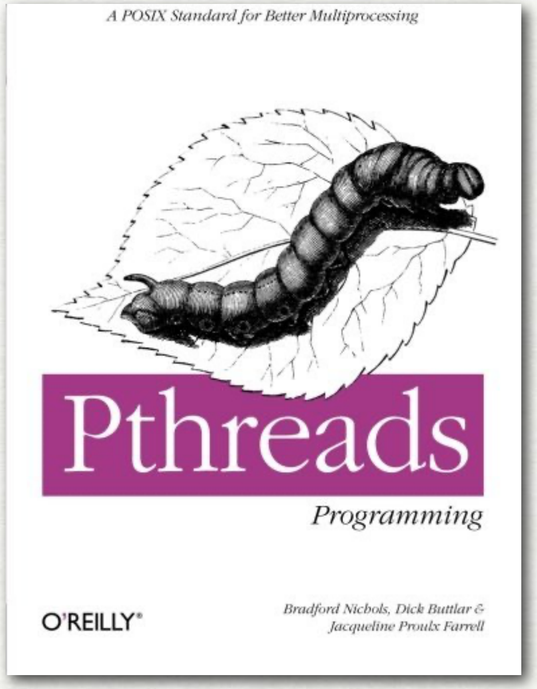
MOST COMMON:
WINDOWS, LINUX

4 bind only some

PTHREADS

THREAD LIBRARIES

- **interface/specification**, not implementation
- may be implemented as user or kernel level
- POSIX standard API, IEEE Std 1003.1c—1995
- thread creation and synchronization
- common in Unix-like OS (BSD, Mac OS X, Linux,…)
- some ports for Windows



PTHREADS EXAMPLE

THREAD LIBRARIES

```
#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;
   }
```

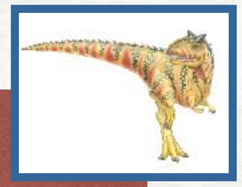
```
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

IMPLICIT THREADING

PROGRAMMING WITH THREADS

Can we decouple programming **functionality** from **thread management**?

text book



THREAD POOLS

OPENMP

GRAND CENTRAL DISPATCH

others

THREADING BUILDING BLOCKS (C++ LIB)

JAVA.UTIL.CONCURRENT (JAVA LIB)

see also Patrik Persson's course, EDAP10 — Concurrent Programming

THREADING ISSUES

- `fork()` and `exec()` semantics with threads
- signal handling — which thread(s)?
- thread cancellation — async. vs deferred
- thread local storage
- scheduler activations

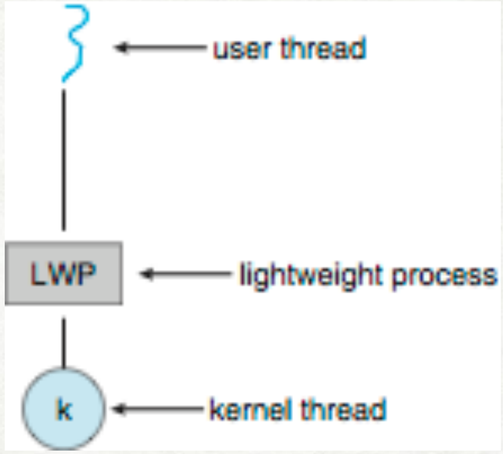
see Unix kill and pthread_kill

see pthread_cancel and pthread_testcancel

see C11 thread_local and pthread_key_*

SCHEDULER ACTIVATIONS

THREADS



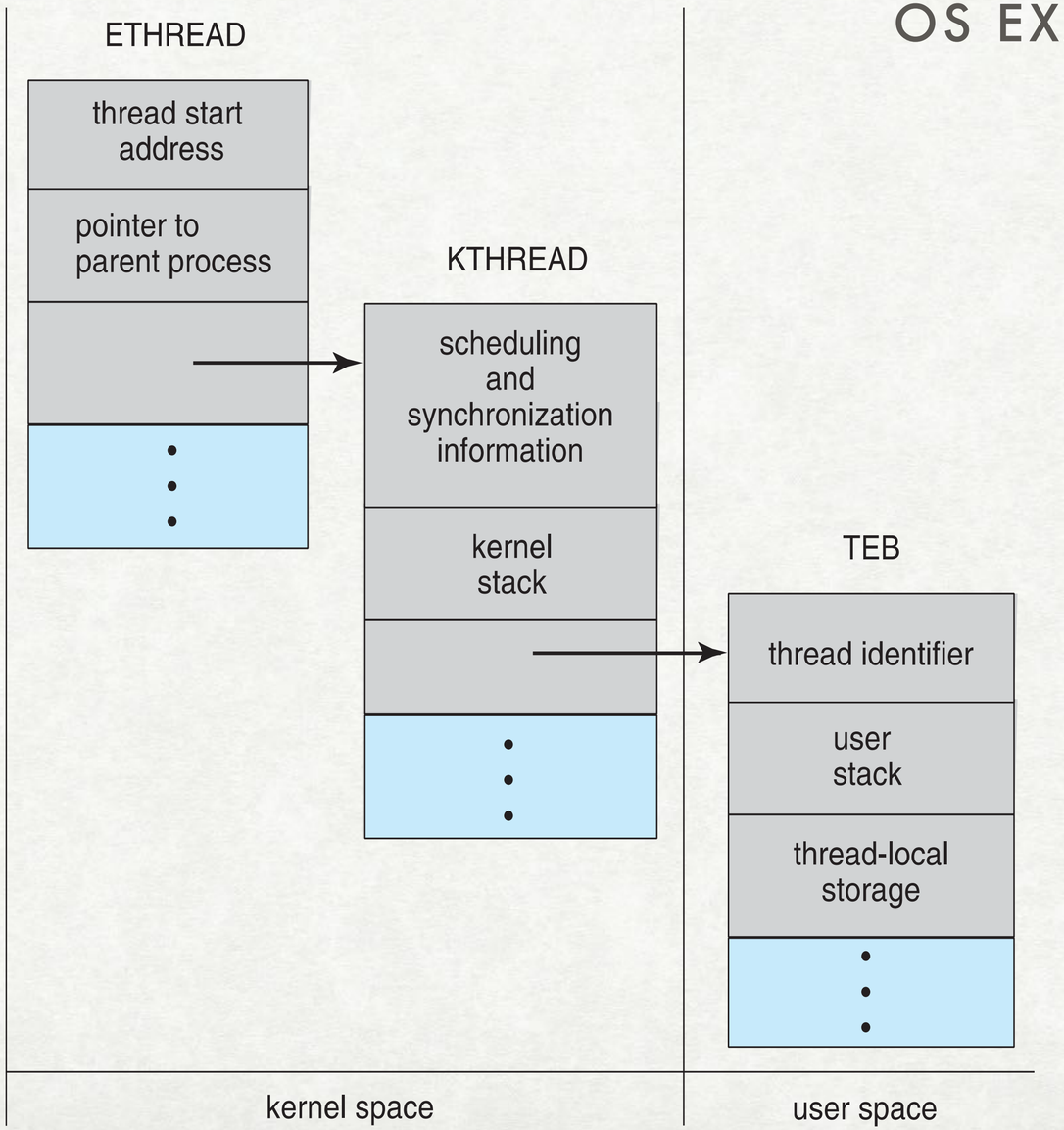
- "many-to-many"— how many k-threads?
- **lightweight process** (LWP) — intermediate level data structure
- kernel: LWP attached to a k-thread
  (blocks if k-thread blocks)
- user: LWP is virtual processor
  (u-threads can be scheduled on it)

http://www.cs.washington.edu/homes/bershad/Papers/p53-anderson.pdf

- **scheduler activation** — scheme for communicating between u-thread lib and kernel
- **upcalls** — kernel informs u-thread lib about k- events (e.g."LWP about to block")

WINDOWS THREADS

OS EXAMPLES



- Windows API — Win 98, NT, 2000, XP, 7
- kernel-level, one-to-one
- executive thread block (ETHREAD), kernel thread block (KTHREAD), thread environment block (TEB)
- separate kernel & user stacks

LINUX THREADS

OS EXAMPLES

- called **tasks** (= threads = processes)
- remember “one-to-one” model
- `clone(…)`,`clone3(…)` — like `fork()`, but finer control of what is shared

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.

see `man clone`

END OF MODULE 3