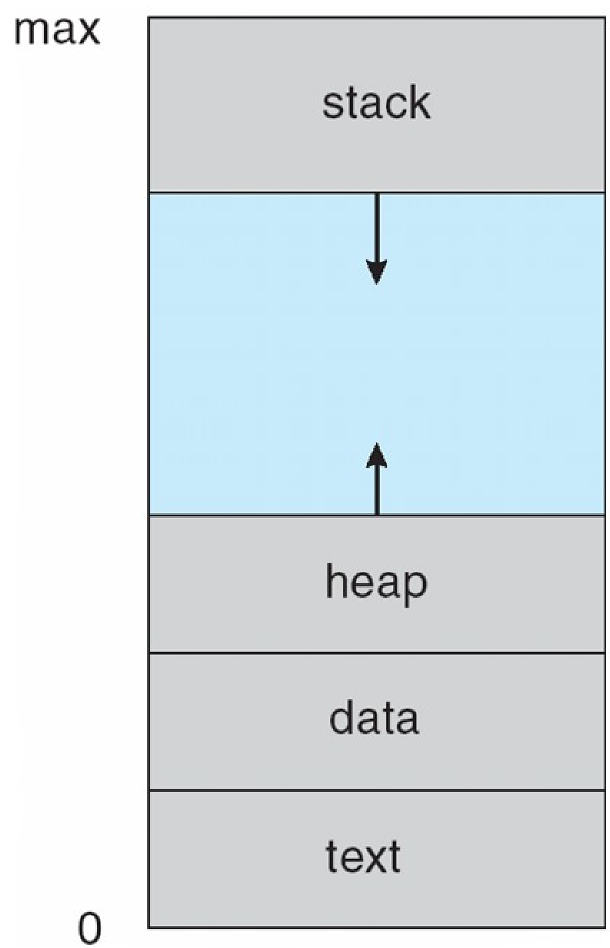
# Process

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
  - Process – a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program counter
  - stack
  - data section
- Topics:
  - Operations in Process
  - Scheduling
  - Interprocess Communication

# Process in Memory

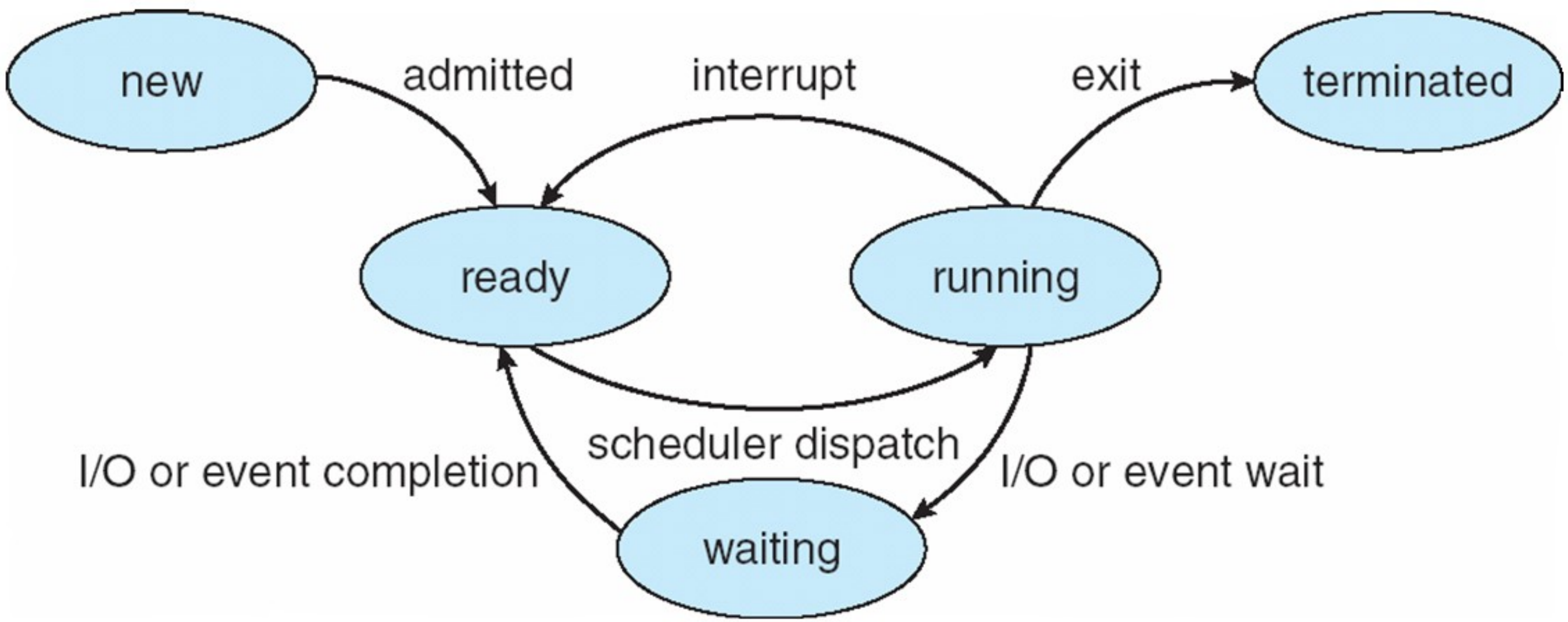


# Process State

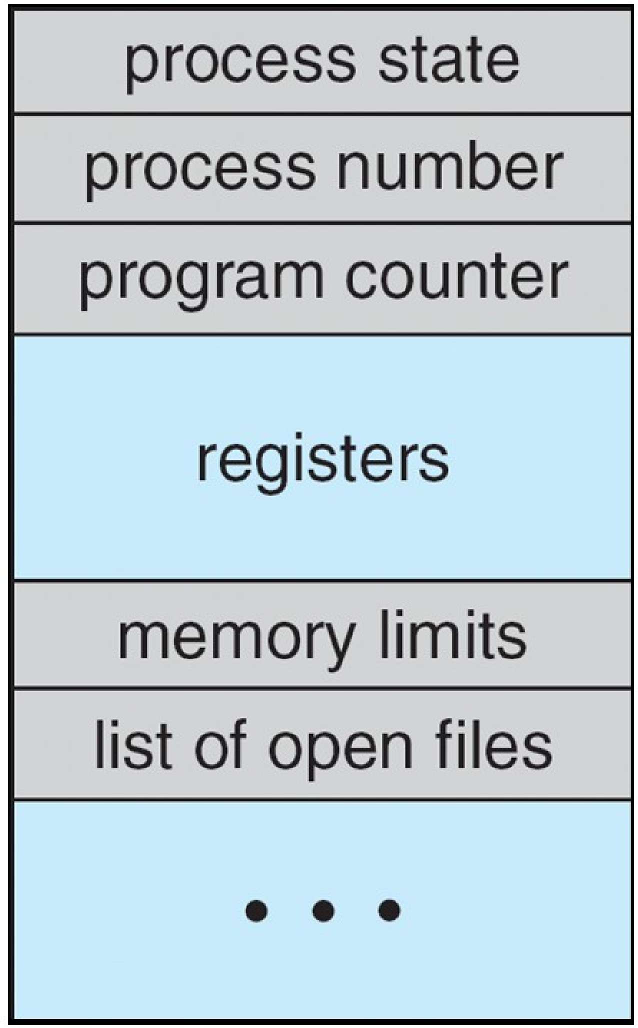
As a process executes, it changes *state*

- **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

# Process States and Transition



# Process Control Block (PCB)



# Context Switch

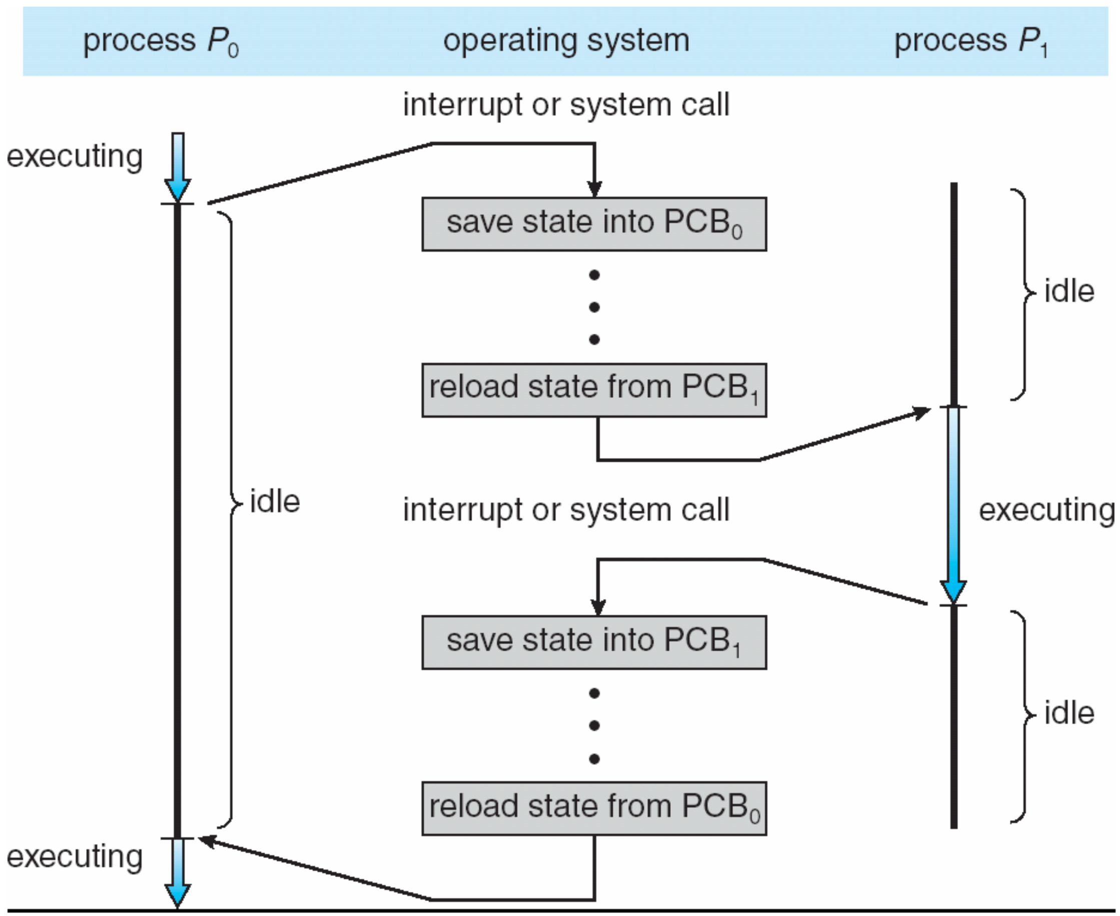
When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a **context switch**.

**Context** of a process represented in the PCB

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

Time dependent on hardware support

# CPU Switch From Process to Process



# 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**)

Options in Resource sharing

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

Options Execution

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

# Process Creation (Cont.)

Options n 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

# Unix *Fork/Exec/Exit/Wait* Example

int pid = fork();

*Create a new process that is a clone of its parent.*

exec*(“program” *[, argvp, envp]*);

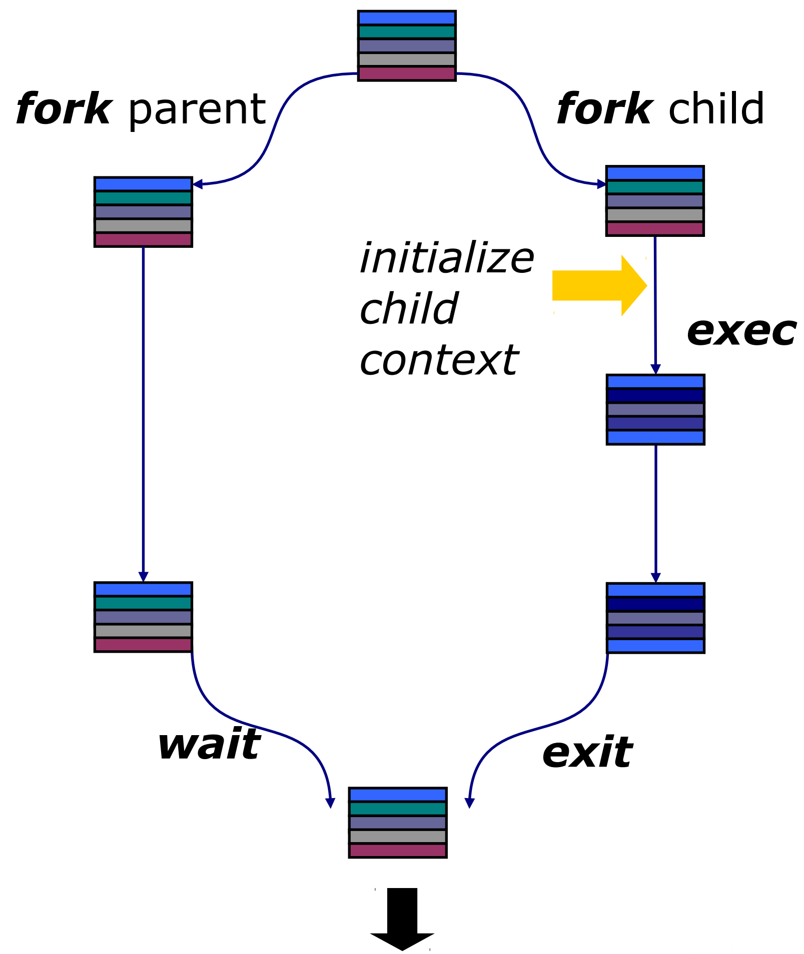
*Overlay the calling process virtual memory with a new program, and transfer control to it.*

exit(status);

*Exit with status, destroying the process.*

int pid = wait*(&status);

*Wait for exit (or other status change) of a child.*



# Example: Process Creation in Unix

*The **fork** syscall returns <u>twice</u>: it returns a zero to the child and the child process ID (pid) to the parent.*

```
int pid;
int status = 0;

if (pid = fork()) {
        /* parent */
        .....
        pid = wait(&status);
} else {
        /* child */
        .....
        exit(status);
}
```

*Parent uses **wait** to sleep until the child exits; **wait** returns child pid and status.*

***Wait** variants allow wait on a specific child, or notification of stops and other signals.*

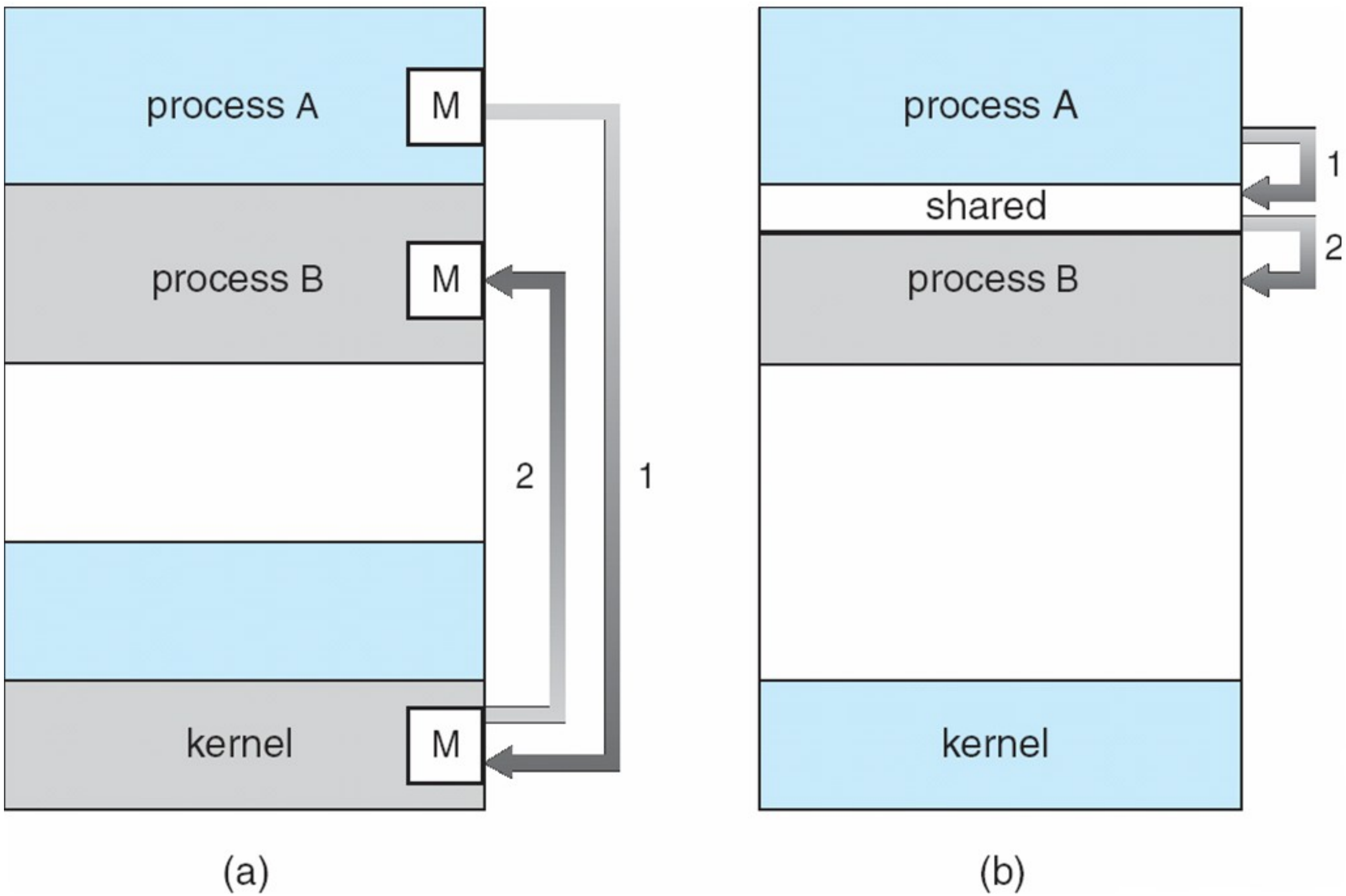
# C Program Forking Separate Process

```
int main()
{
int  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
          fprintf(stderr, "Fork Failed");
          exit(-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");
          exit(0);
    }
}
```

# 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
- 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 system do not allow child to continue if its parent terminates
      - All children terminated - **cascading termination**

# Communications Models: Shared memory or Message Passing



# Synchronization

Message passing may be either blocking or non-blocking

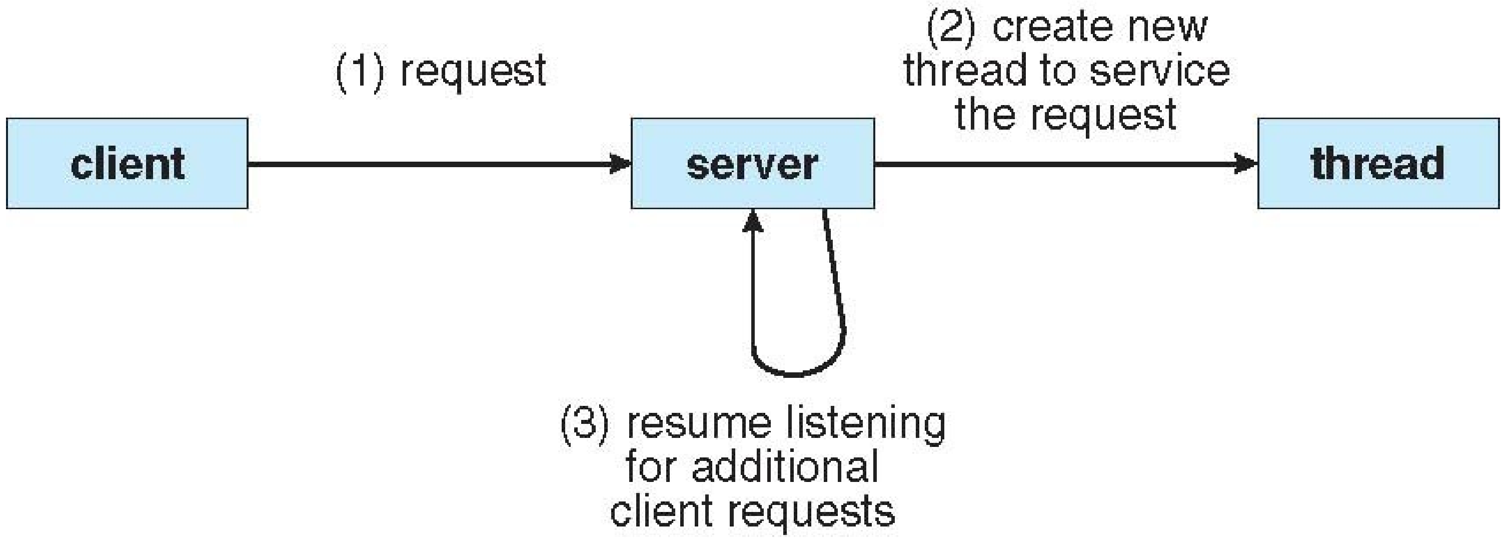
**Blocking** is considered **synchronous**

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

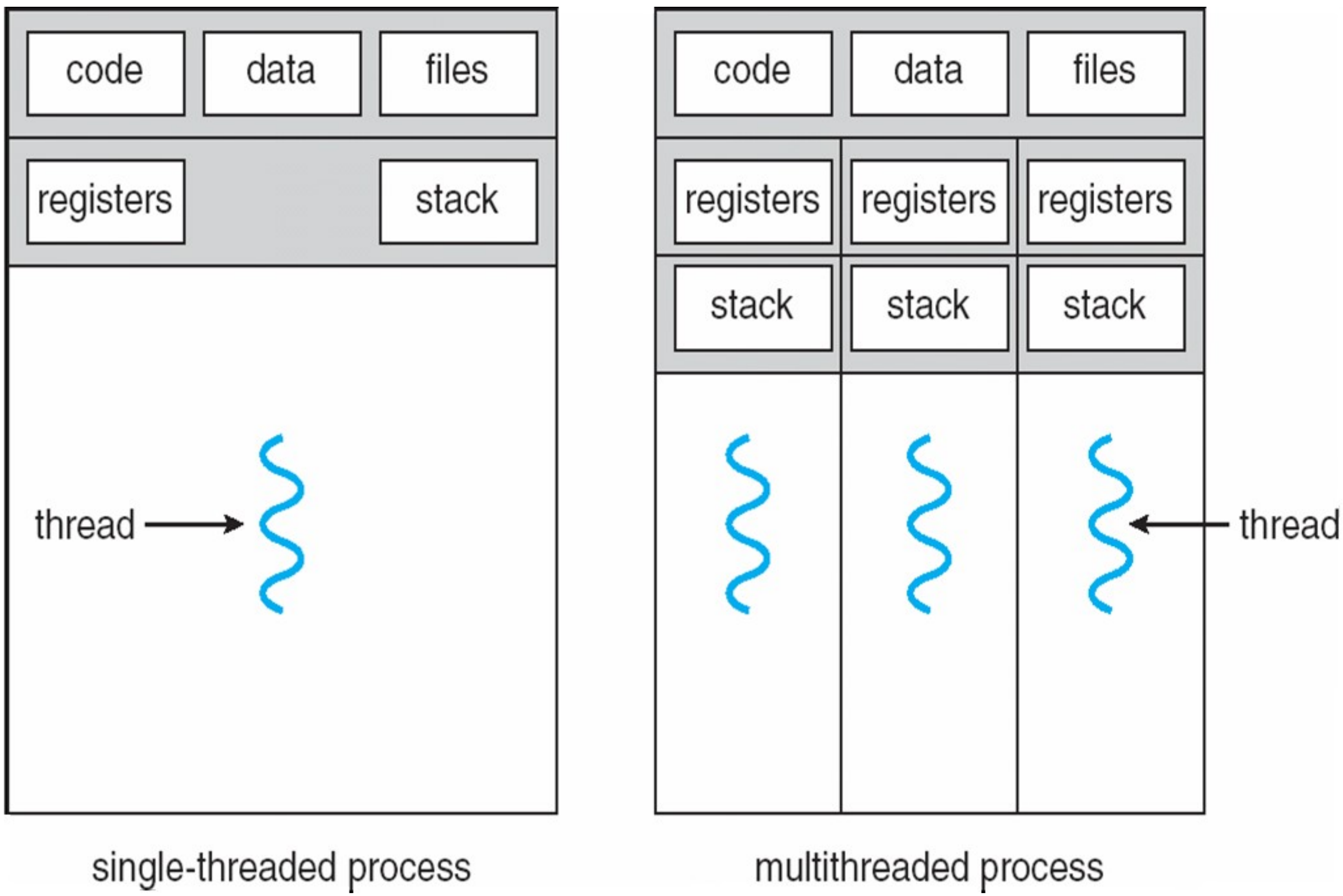
**Non-blocking** is considered **asynchronous**

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

# Motivation for multi-threaded servers



# Single and Multithreaded Processes



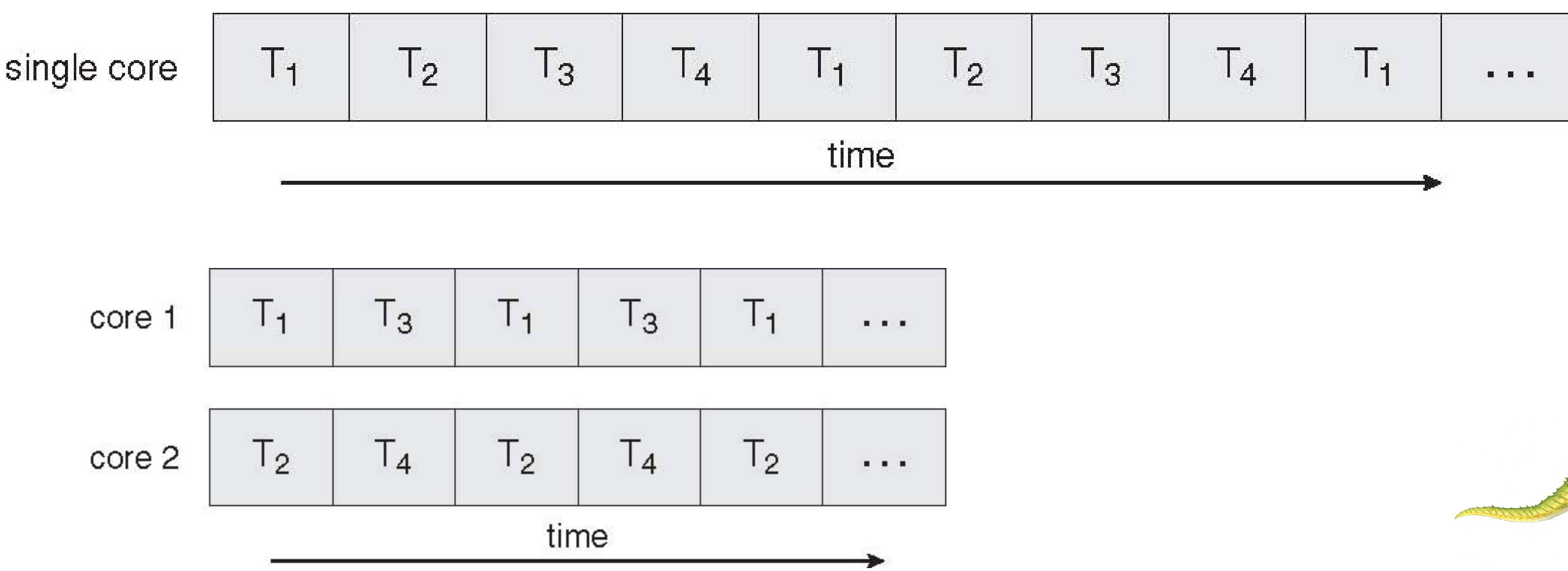
# Benefits

Responsiveness

Resource Sharing

Economy

Scalability



# Kernel Threads

- Recognized and supported by the OS Kernel
- OS explicitly performs  scheduling and context switching of kernel threads
- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

# User Threads

- Thread management done by user-level threads library
  - OS kernel does not know/recognize there are multiple threads running in a user program.
  - The user program (library) is responsible for scheduling and context switching of its threads.

- Three primary thread libraries:
  - POSIX **Pthreads**
  - Win32 threads
  - Java threads

# User- vs. Kernel-level Threads

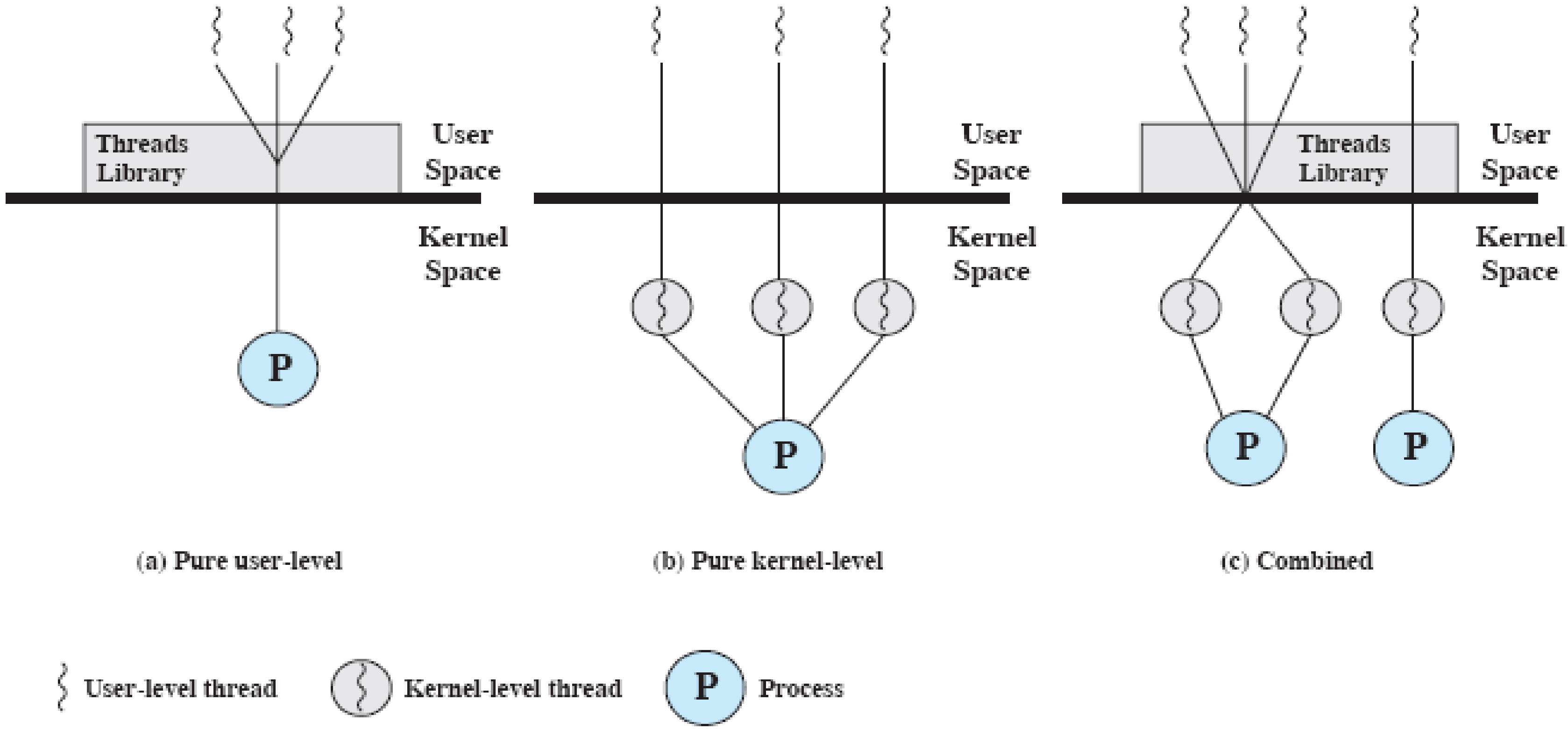



Figure 4.6 User-Level and Kernel-Level Threads

From W. Stallings, Operating Systems, 6th Edition

# Pthreads

May be provided either as user-level or kernel-level

A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization

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)

# 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