CSCI-GA.2250-001

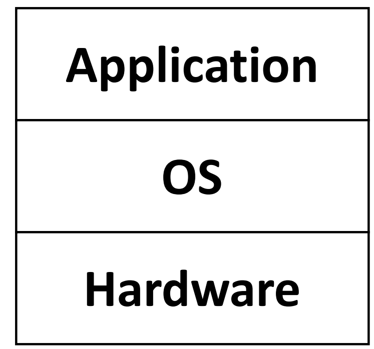
# Operating Systems

## Structure of Operating Systems

Hubertus Franke
frankeh@cims.nyu.edu

# Recap: What is an OS ?

- Code that:
  - Sits between programs & hardware
  - Sits between different programs
  - Sits betweens different users
- Job of OS:
  - Manage hardware resources
    - Allocation, protection, reclamation, virtualization
  - Provide services to app. How ? → system call
    - Abstraction, simplification, standardization



# System Call

- Invoked via non-priviliged instruction
  - TRAP
  - Treated often like an interupt, but its "somewhat" different

- Synchronous transfer control

- Side-effect of executing a trap in userspace is that an exception is raised and program execution continues at a prescribed instruction in the kernel → syscall_handler()

# Traps / Exception / Interrupt

- Understand similarity and differences to interrupt and exception

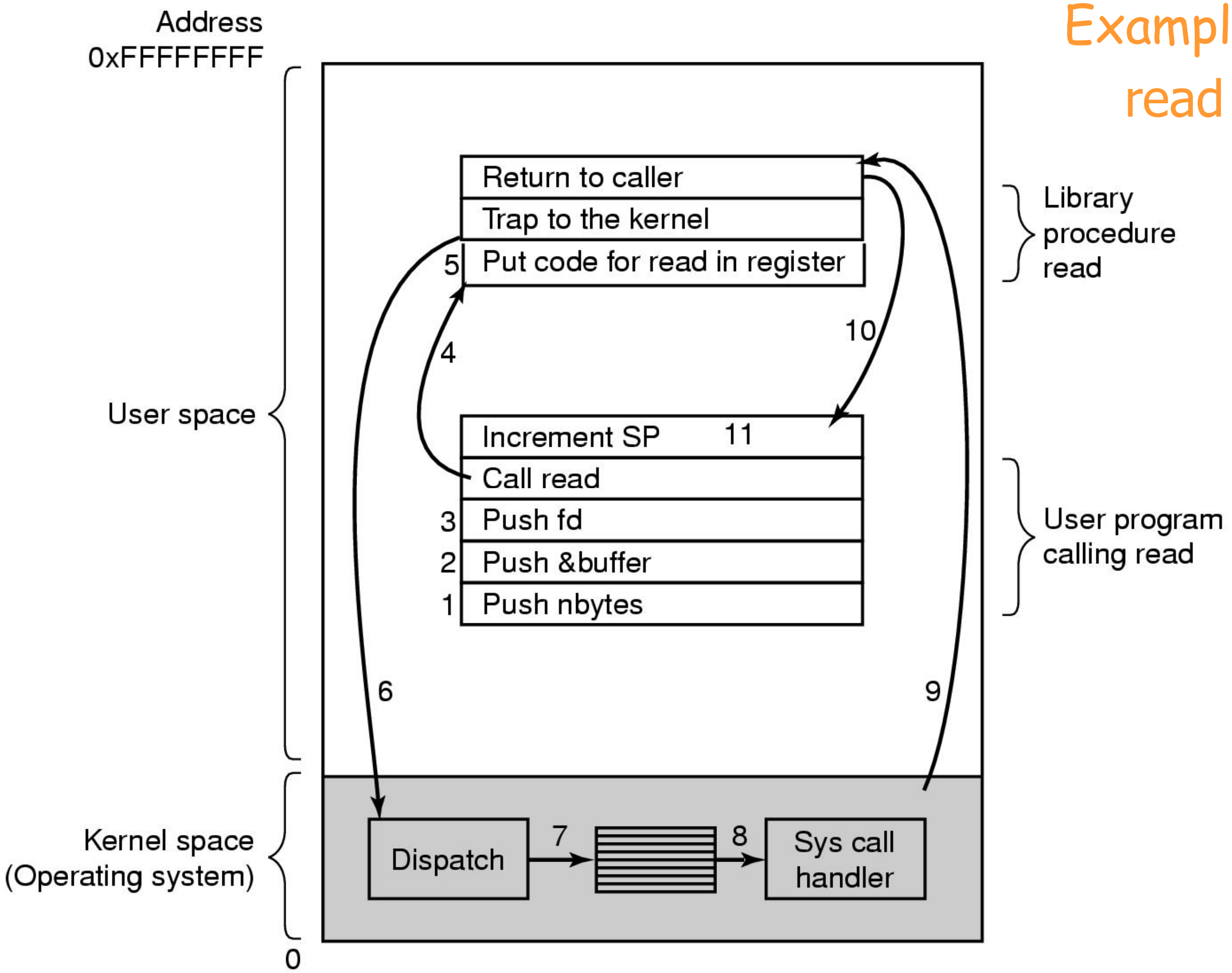
- Interrupts:
  - *asynchronous* :Triggered by an event from a "device"
- Traps
  - *Synchronous:* triggered by "trap instruction" for syscall
- *Exception:*
  - *Synchronous:* triggered by a "fault condition" of an instruction condition

- <u>They all end up in the so called "interrupt handling"</u>

# Steps in Making a System Call

Example:

read (fd, buffer,nbytes)



# Operating Systems Structure (Chapter 1)

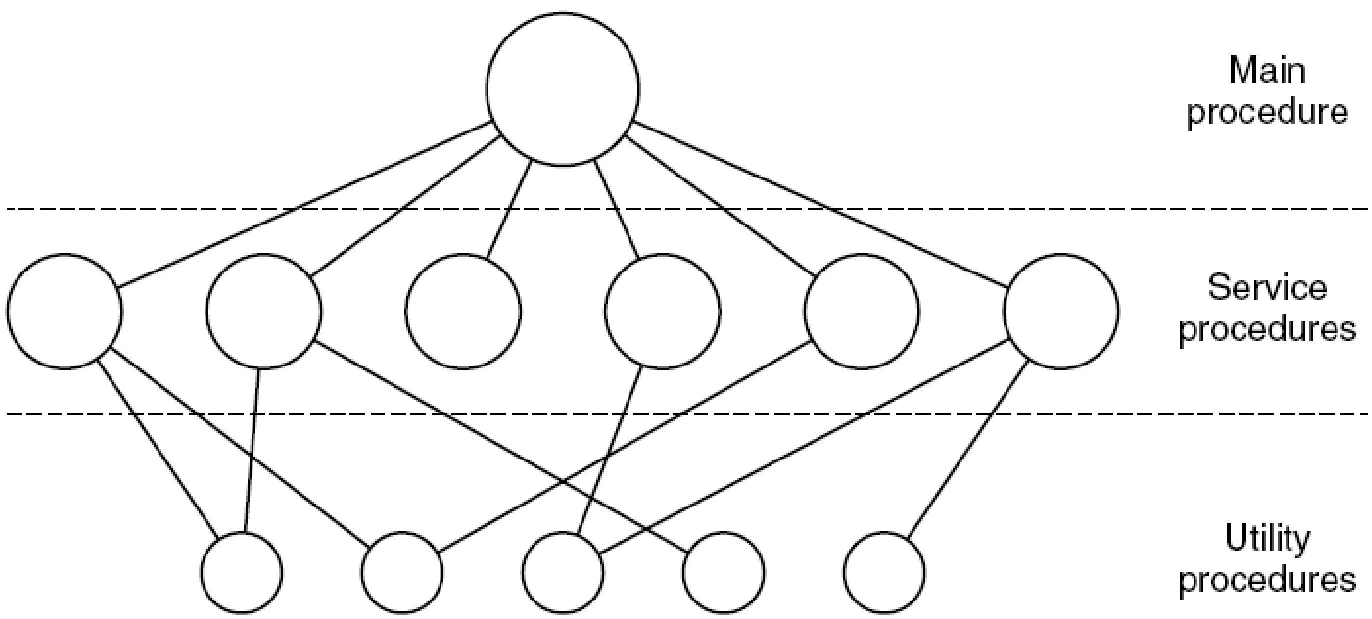
Monolithic systems – basic structure

1. A main program that invokes the requested service procedure.
2. A set of service procedures that carry out the system calls.
3. A set of utility procedures that help the service procedures.

# Monolithic Systems

By far the most common OS organization

A simple structuring model for a monolithic system.



# Layered Systems

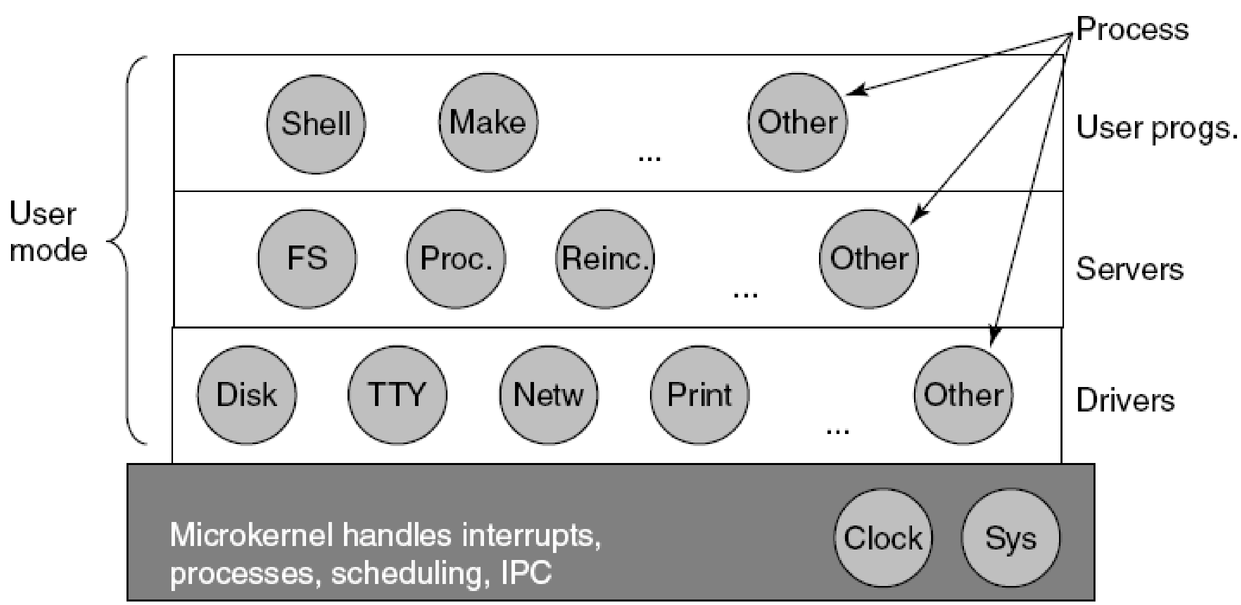
- Layer-n services are comprised of services provided by Layer-(n-1)..
- Structure of the THE operating system (Dijkstra 1968)

| Layer | Function |
|---|---|
| 5 | The operator |
| 4 | User programs |
| 3 | Input/output management |
| 2 | Operator-process communication |
| 1 | Memory and drum management |
| 0 | Processor allocation and multiprogramming |

- THE used this approach as a design AID
- Multics Operating System relied on Hardware Protection to enforce layering

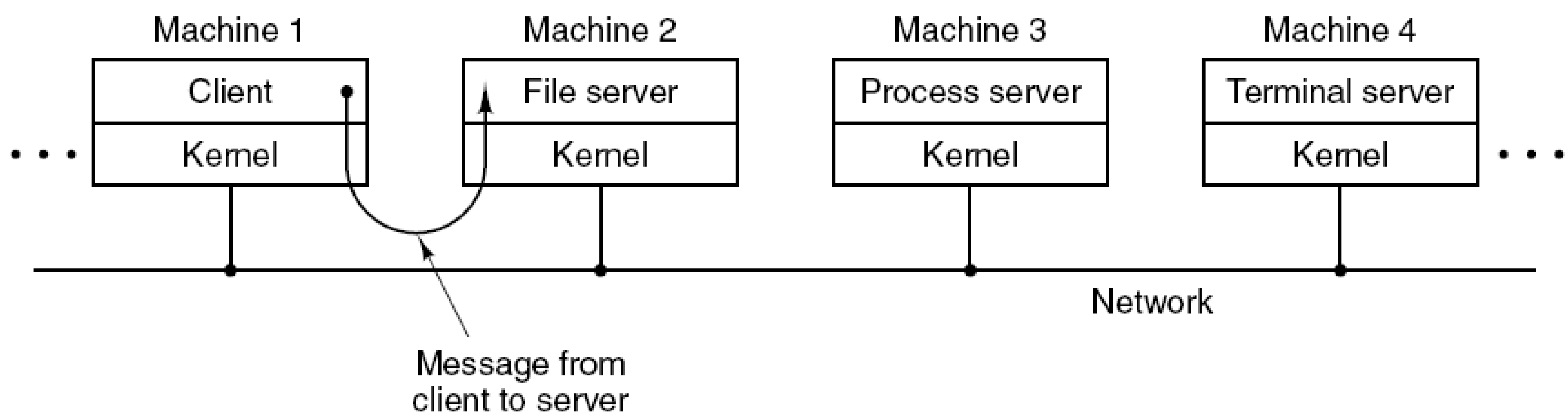
# Microkernels

- Microkernels move the layering boundaries between kernel and userspace
- Move only most rudimentary services to kernel
- Move other services to Userspace
- Higher Overhead, but more flexibility, higher robustness
  - Minix, L4, K42,
  - Minix (Tanenbaum is only 3200 lines of C and 800 lines assembler)



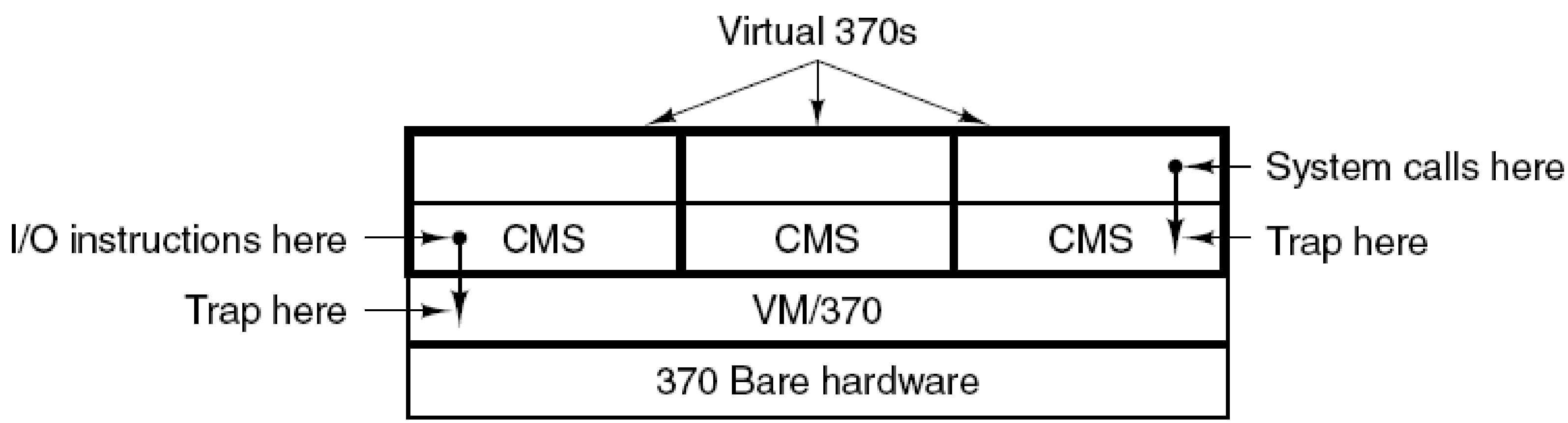
# Client-Server Model

- Assumes generic network model (network, bus)
- Communication via message passing



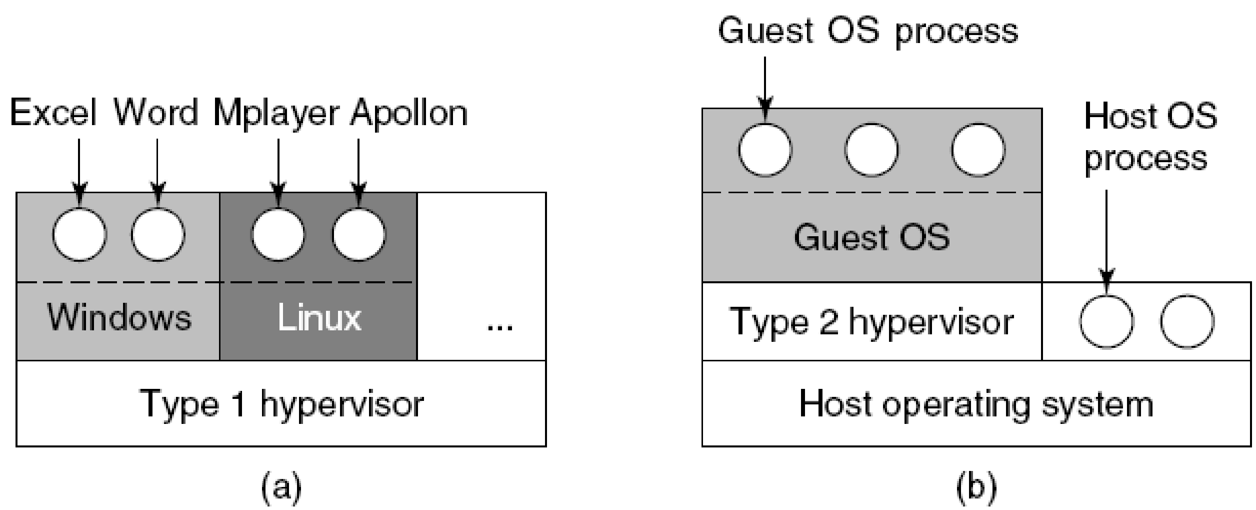
# Virtual Machines (1)

- VM/370: Timesharing system should be comprised of:
  - Multiprogramming
  - extended machine with more interface than bare HW
  - Completely separate these two functions
- Provides ability to "self-virtualize"
- Beginning of "modern day" virtualization technology



# Virtual Machines (2)

- A type 1 hypervisor (like virtual machine monitor)
  - Priviliged instructions are trapped and “emulated”
- A type 2 hypervisor (runs on top of a host OS)
  - Unmodified ( trapped )
  - Modified ( paravirtualization )



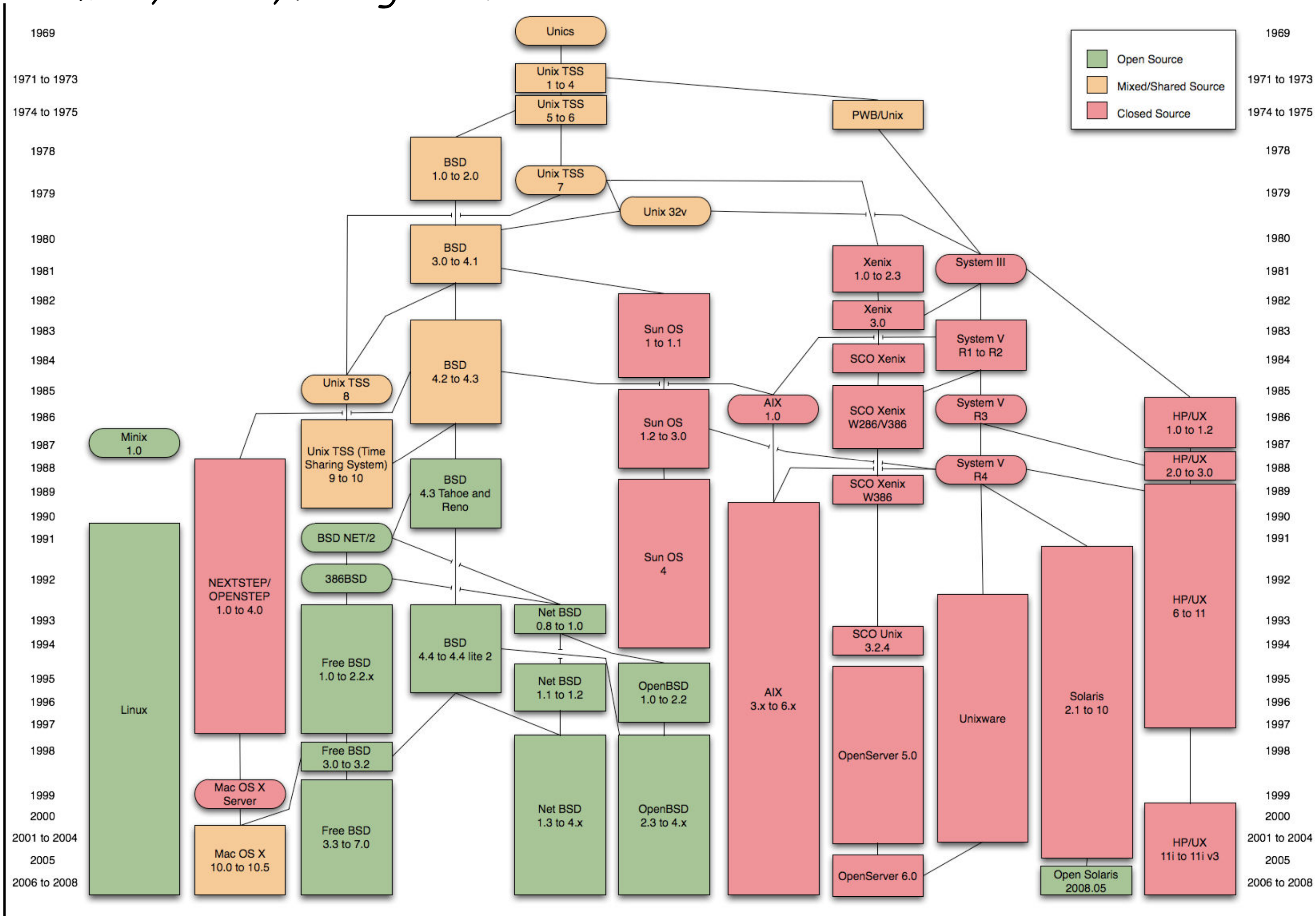
# Other areas of virtual machines usage

- Java virtual machines
- Dynamic scripting languages (e.g. Python)

- Typically define a instruction set that is "interpreted" by the associate virtual machine
  - JVM, PVM
  - Modern system then JIT (Just In Time) compile the VM instructions into native code.

# History of the UNIX Operating System
(source: wikipedia)

Bell Labs: *Thomson, Richie, Kernigan et.al*



CSCI-GA.2250-001

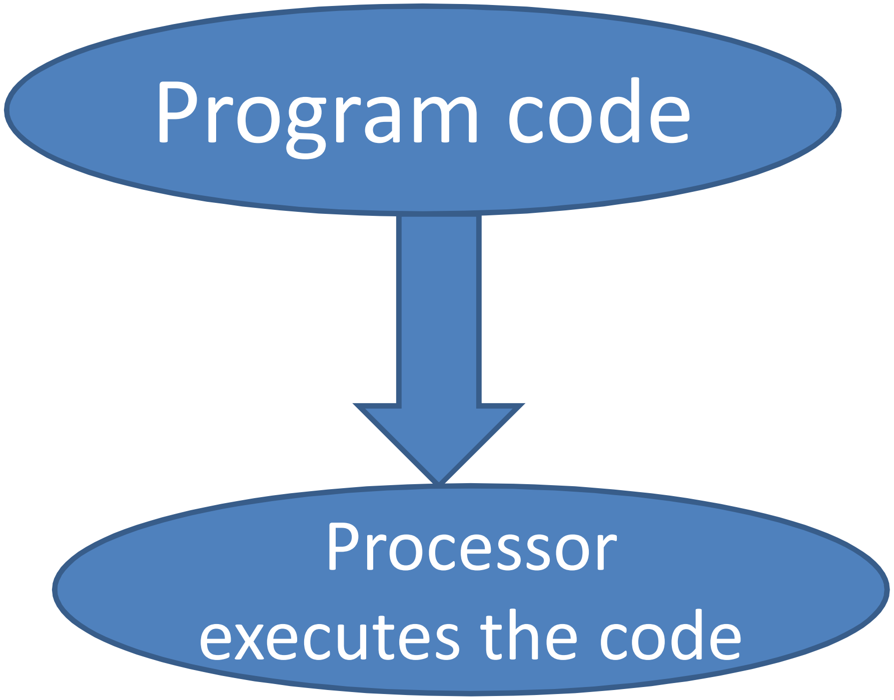
# Operating Systems

## Lecture 2 (b):
## Processes and Threads - Part 1

Hubertus Franke
frankeh@cims.nyu.edu

# OS Management of Application Execution

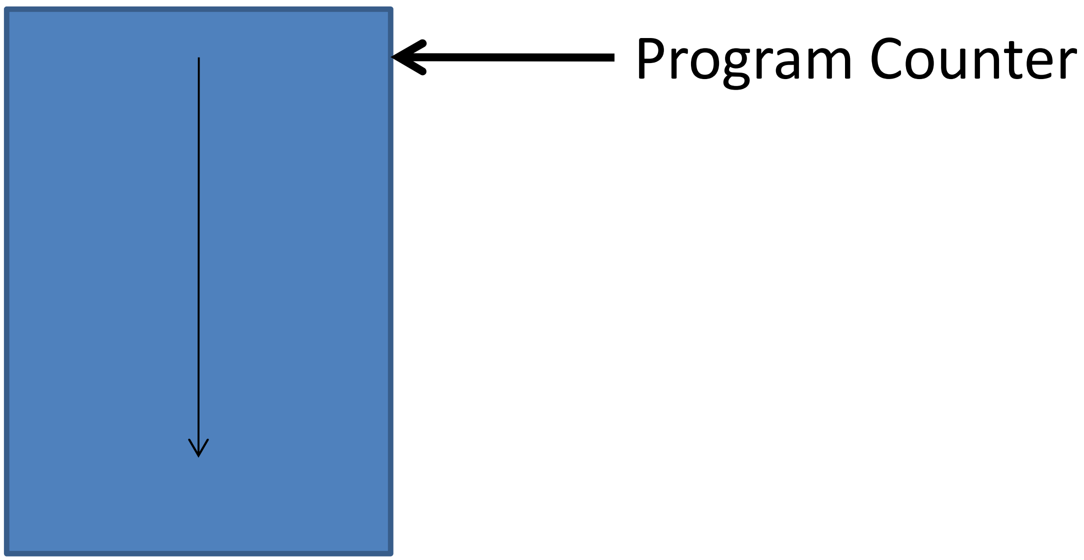
- Resources are made available to multiple applications
- The processor is switched among multiple applications so all will appear to be progressing
- The processor and I/O devices can be used efficiently




When the processor begins to execute the program code, we refer to this executing entity as a ***process***

# What Is a Process?

An abstraction of a running program



# The Process Model

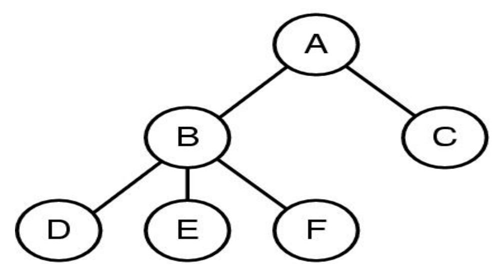
- A process is an instance of an executing program and includes
  - Program counter
  - Registers
  - Variables
  - …
- A process has a program, input, output, and state.

If a program is running twice, does it count as two processes? or one?

# Process: a running program

- A process includes
  - Address space
  - Process table entries (state, registers)
    - Open files, thread(s) state, resources held

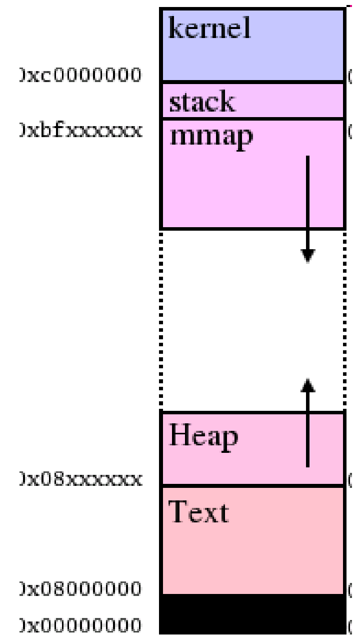
- A process tree
  - A created two child processes, B and C
  - B created three child processes, D, E, and F

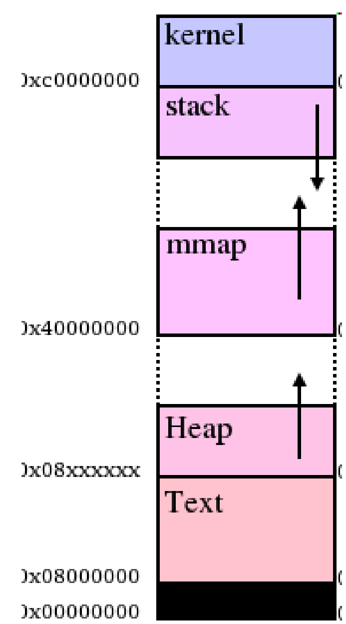


# Address Space

- Defines where sections of data and code are located in 32 or 64 address space
- Defines protection of such sections
- ReadOnly, ReadWrite, Execute

- Confined "private" addressing concept
  - ➔ requires form of address virtualization



# Process Element

- While the program is executing, this process can be uniquely characterized by a number of elements, including:

**Where do we store all this info?**

identifier

state

priority

program counter

memory pointers

context data

I/O status information

accounting information

# Process Control Block

- Contains the process elements
- It is possible to interrupt a running process and later resume execution as if the interruption had not occurred
- Created and managed by the operating system
- Key tool that allows support for multiple processes

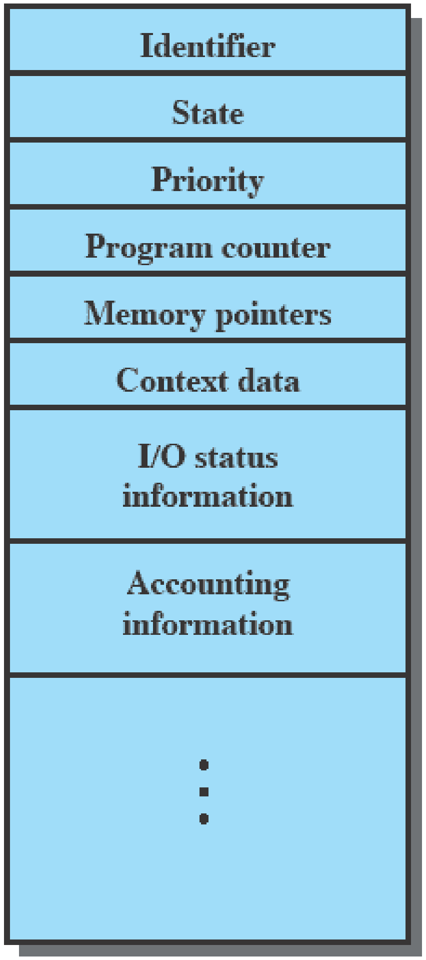
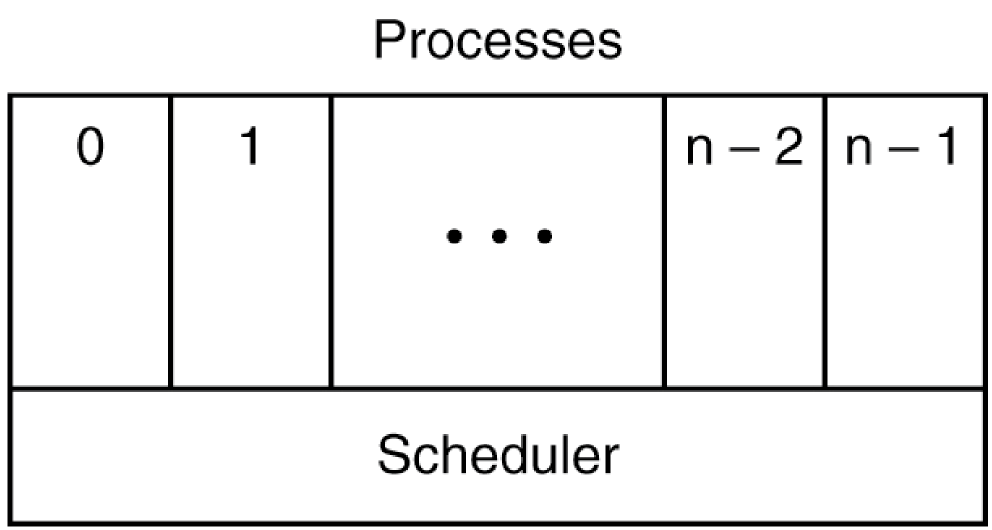
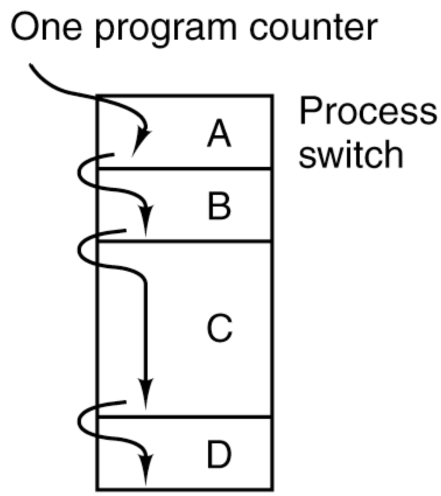



Figure 3.1 Simplified Process Control Block

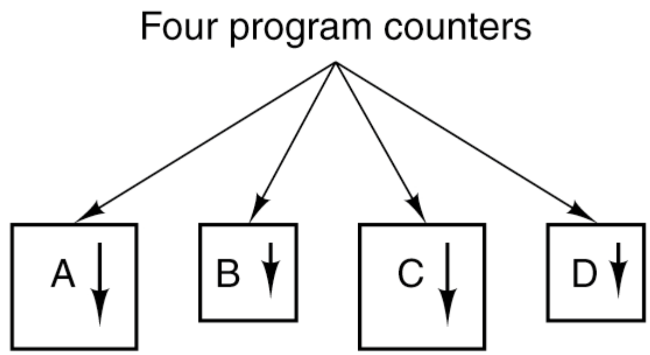
# Multiprogramming

- One CPU and several processes
- CPU switches from process to process quickly

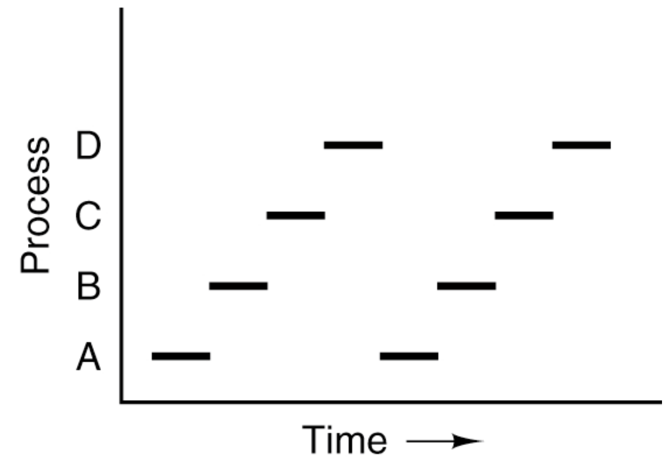





(a)



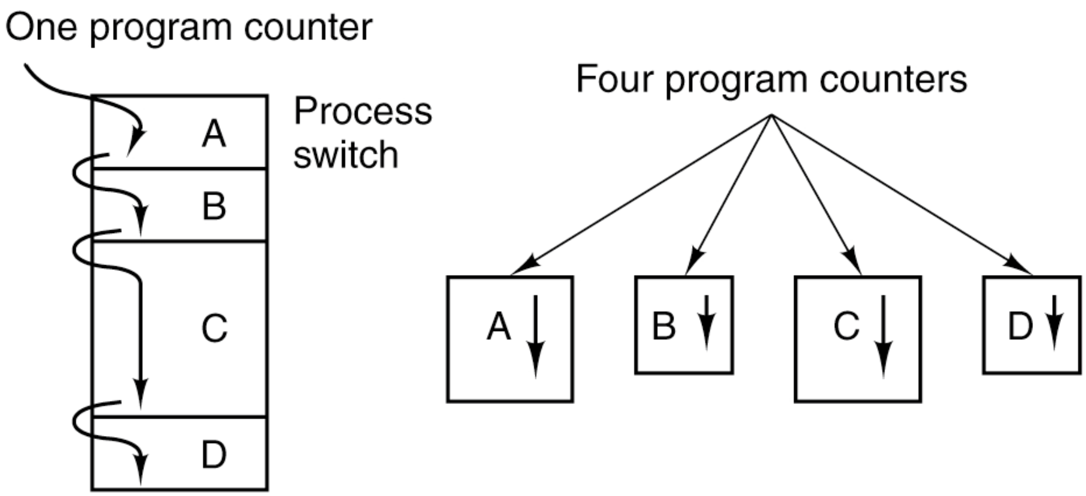

(b)




**What Really Happens**

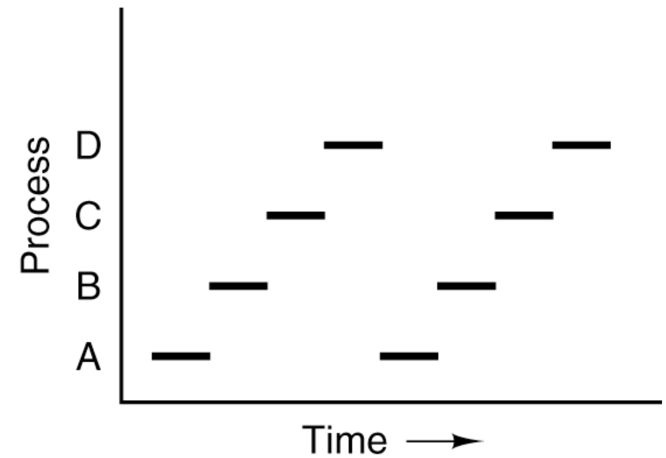
**What We Think It Happens**

If we run the same program several times, will we get the same execution time?

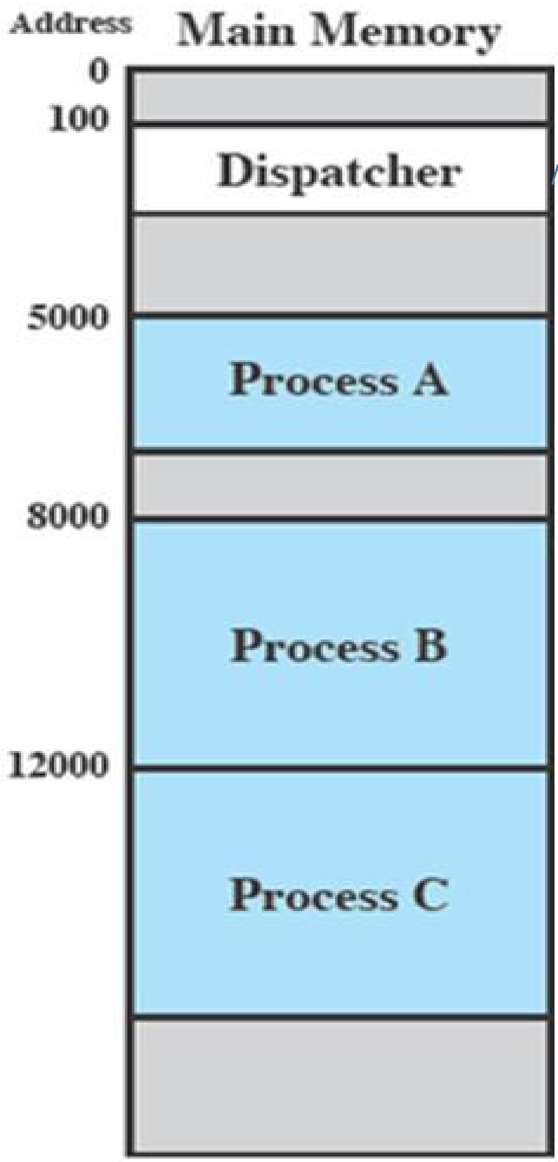



(a) (b)

**What Really Happens** **What We Think It Happens**

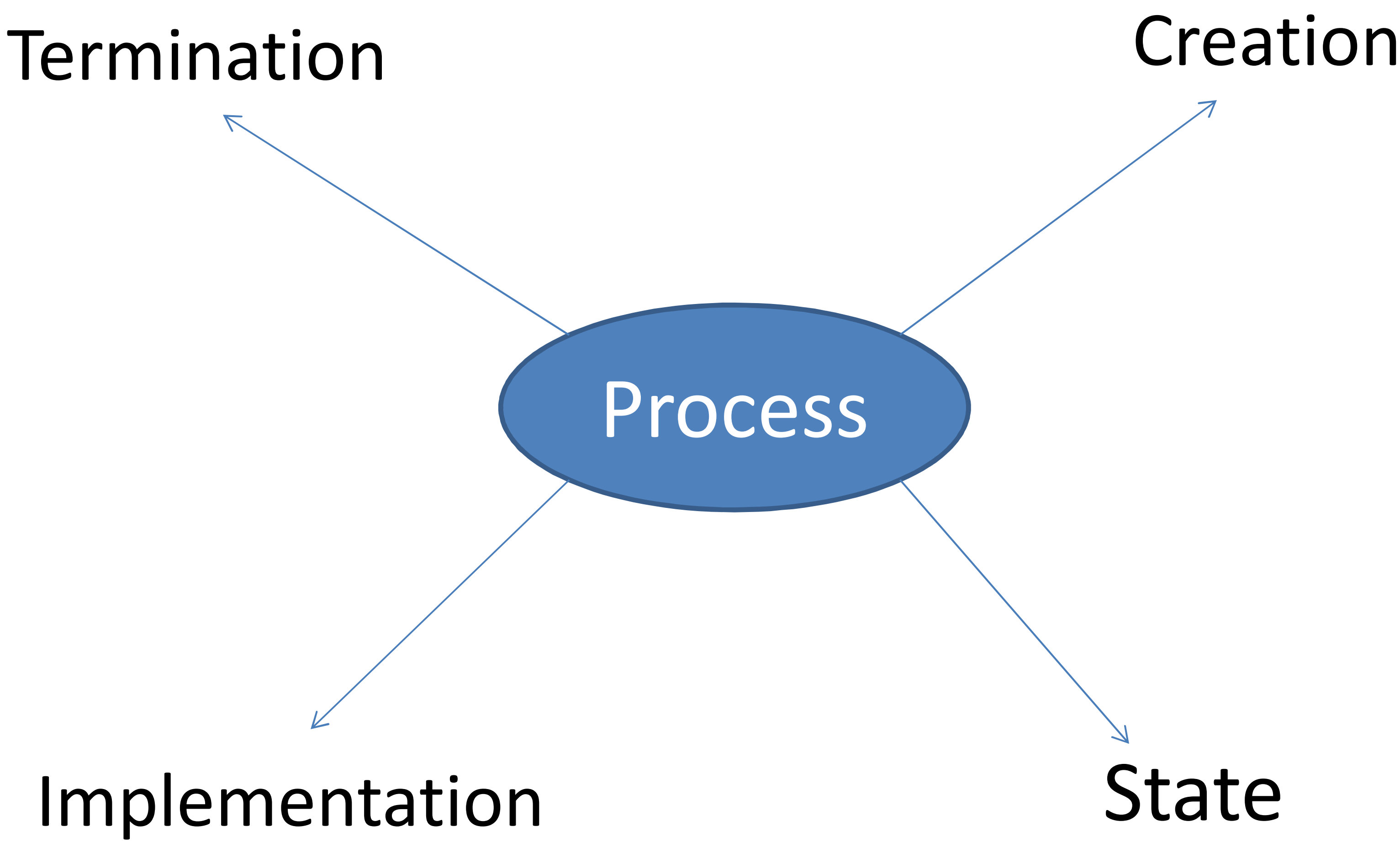


# Example




| | |
|---|---|
| 1 | 5000 |
| 2 | 5001 |
| 3 | 5002 |
| 4 | 5003 |
| 5 | 5004 |
| 6 | 5005 |
| ------------------- Timeout | |
| 7 | 100 |
| 8 | 101 |
| 9 | 102 |
| 10 | 103 |
| 11 | 104 |
| 12 | 105 |
| 13 | 8000 |
| 14 | 8001 |
| 15 | 8002 |
| 16 | 8003 |
| -------------- I/O Request | |
| 17 | 100 |
| 18 | 101 |
| 19 | 102 |
| 20 | 103 |
| 21 | 104 |
| 22 | 105 |
| 23 | 12000 |
| 24 | 12001 |
| 25 | 12002 |
| 26 | 12003 |
| 27 | 12004 |
| 28 | 12005 |
| ------------------- Timeout | |
| 29 | 100 |
| 30 | 101 |
| 31 | 102 |
| 32 | 103 |
| 33 | 104 |
| 34 | 105 |
| 35 | 5006 |
| 36 | 5007 |
| 37 | 5008 |
| 38 | 5009 |
| 39 | 5010 |
| 40 | 5011 |
| ------------------- Timeout | |
| 41 | 100 |
| 42 | 101 |
| 43 | 102 |
| 44 | 103 |
| 45 | 104 |
| 46 | 105 |
| 47 | 12006 |
| 48 | 12007 |
| 49 | 12008 |
| 50 | 12009 |
| 51 | 12010 |
| 52 | 12011 |
| ------------------- Timeout | |

Small program that switches the processor from one process to another (also called Scheduler)


Termination
Creation
Process
Implementation
State

# Process Creation

- System initialization
  - At boot time
  - Foreground
  - Background (daemons)
- Execution of a process creation system call by a running process
- A user request
- A batch job
- Created by OS to provide a service
- Interactive logon

# Process Termination

- Normal exit (voluntary)
- Error exit (voluntary)
- Fatal error (involuntary)
- Killed by another process (involuntary)

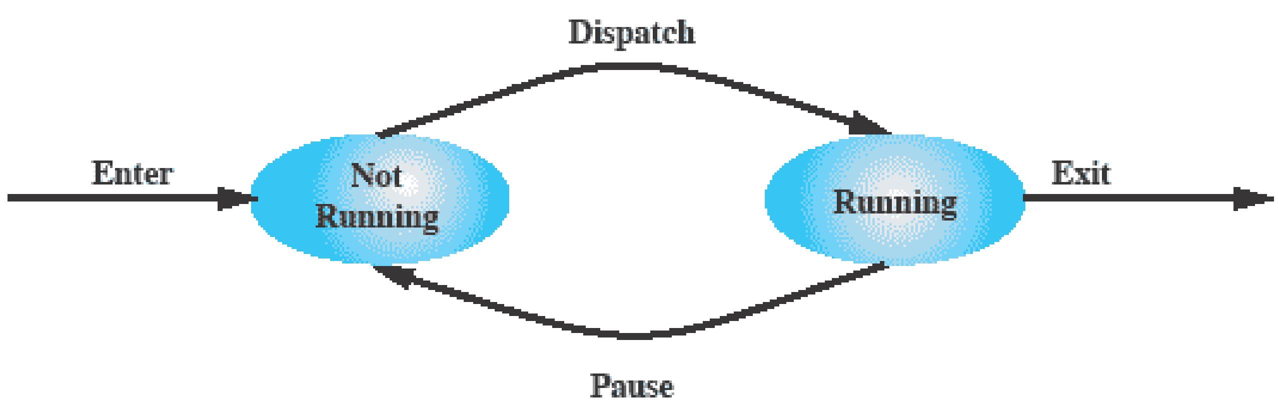
# Process Termination: More Scenarios

| | |
|---|---|
| Normal completion | The process executes an OS service call to indicate that it has completed running. |
| Time limit exceeded | The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input. |
| Memory unavailable | The process requires more memory than the system can provide. |
| Bounds violation | The process tries to access a memory location that it is not allowed to access. |
| Protection error | The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file. |
| Arithmetic error | The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate. |
| Time overrun | The process has waited longer than a specified maximum for a certain event to occur. |
| I/O failure | An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer). |
| Invalid instruction | The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data). |
| Privileged instruction | The process attempts to use an instruction reserved for the operating system. |
| Data misuse | A piece of data is of the wrong type or is not initialized. |
| Operator or OS intervention | For some reason, the operator or the operating system has terminated the process (e.g., if a deadlock exists). |
| Parent termination | When a parent terminates, the operating system may automatically terminate all of the offspring of that parent. |
| Parent request | A parent process typically has the authority to terminate any of its offspring. |

# Process State

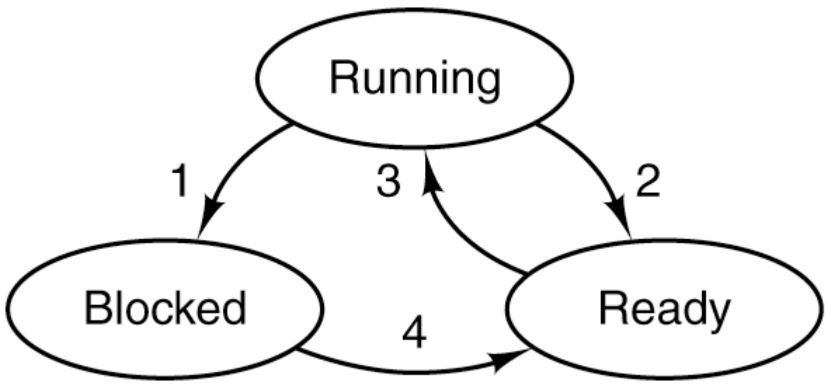
- Depending on the implementation, there can be several possible state models.

- The Simplest one: Two-state diagram




(a) State transition diagram

# Process State: Three-State Model




1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

# Process State
# Five-State Model

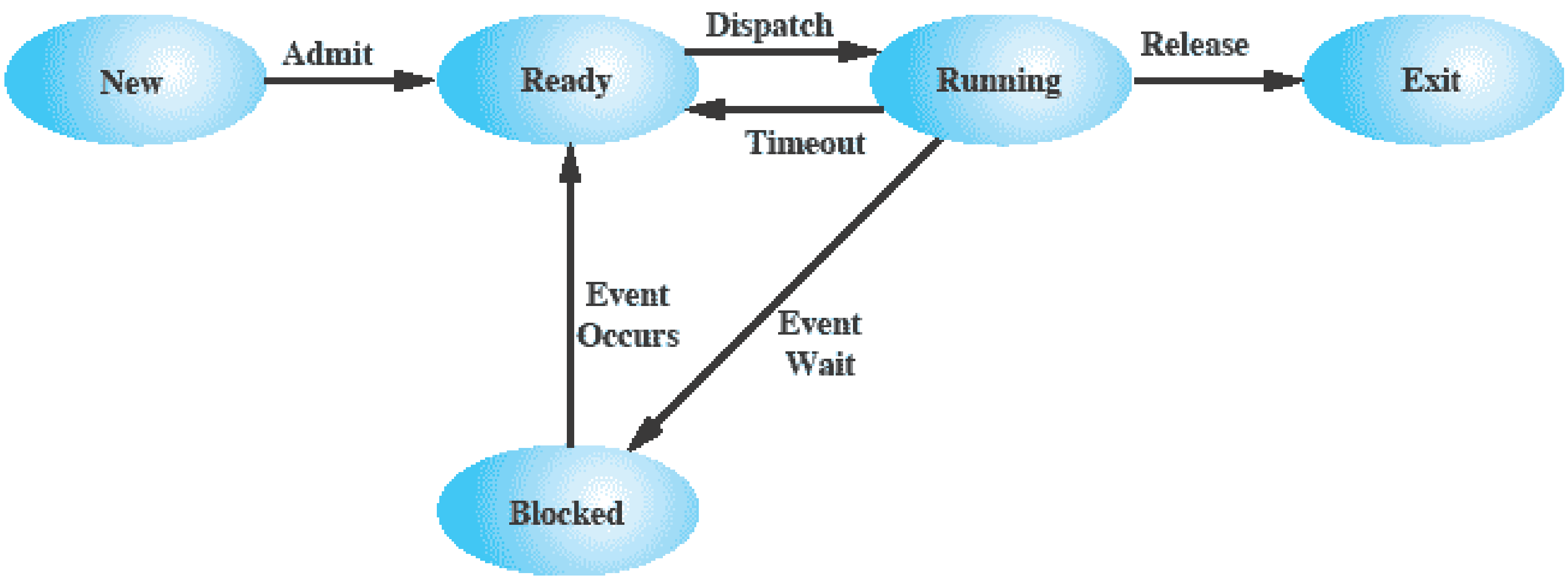
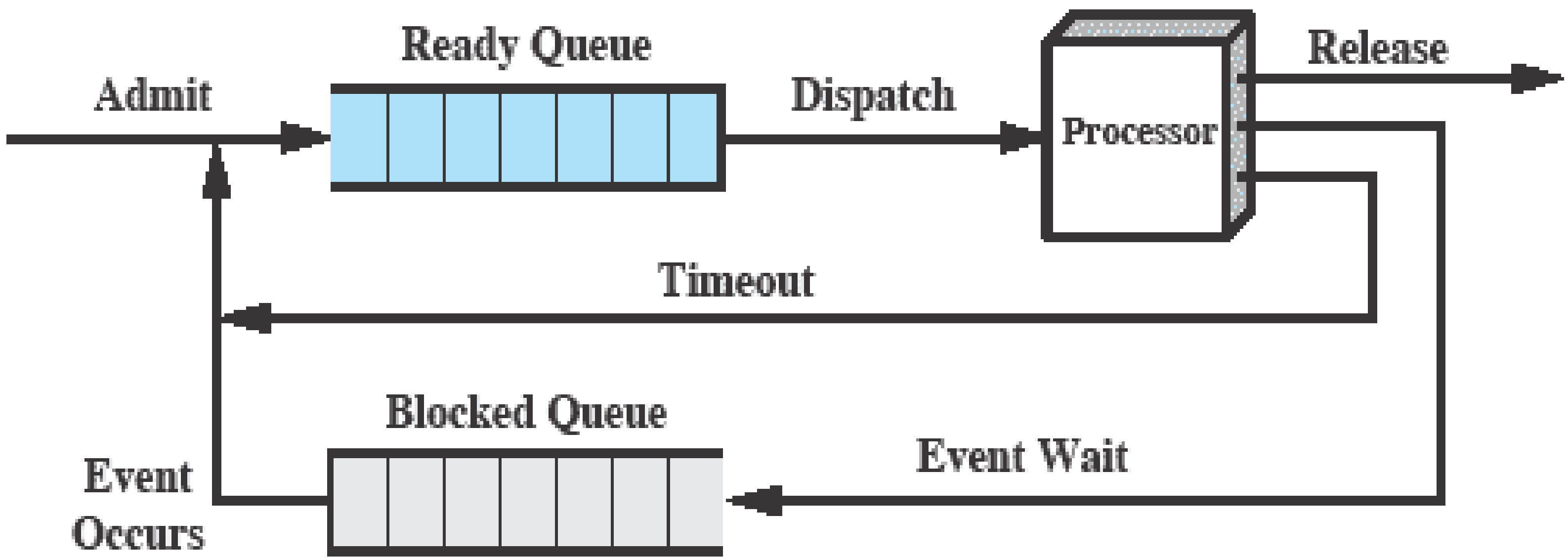


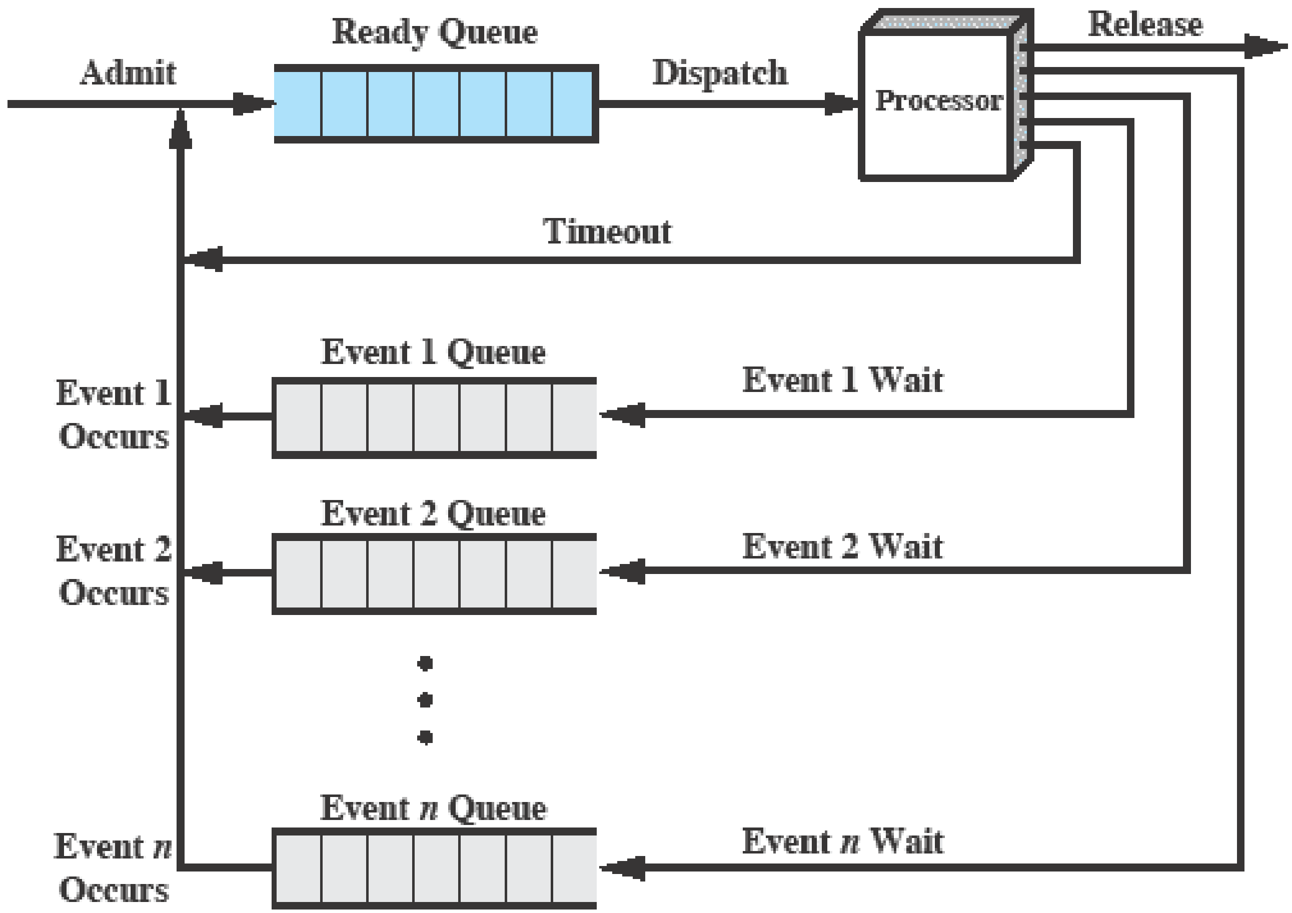

Figure 3.6 Five-State Process Model

# Using Queues to Manage Processes



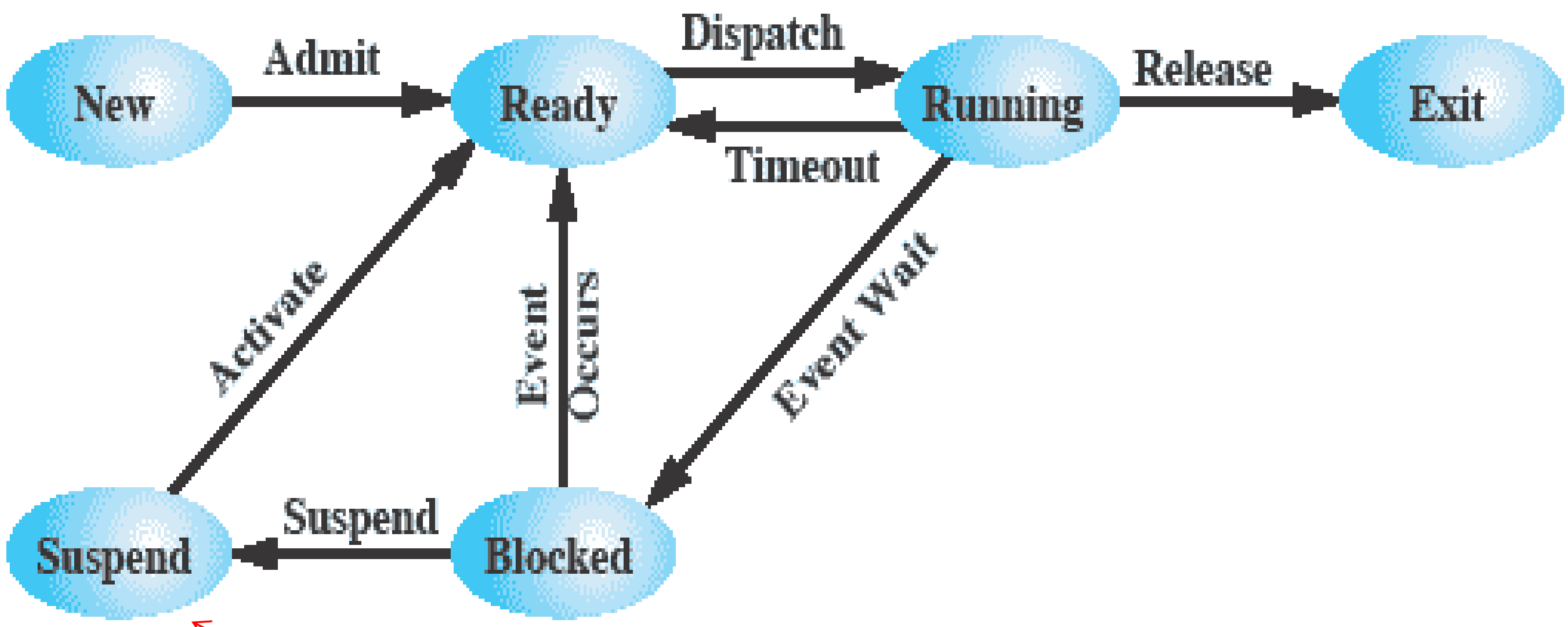

(a) Single blocked queue

# Using Queues to Manage Processes



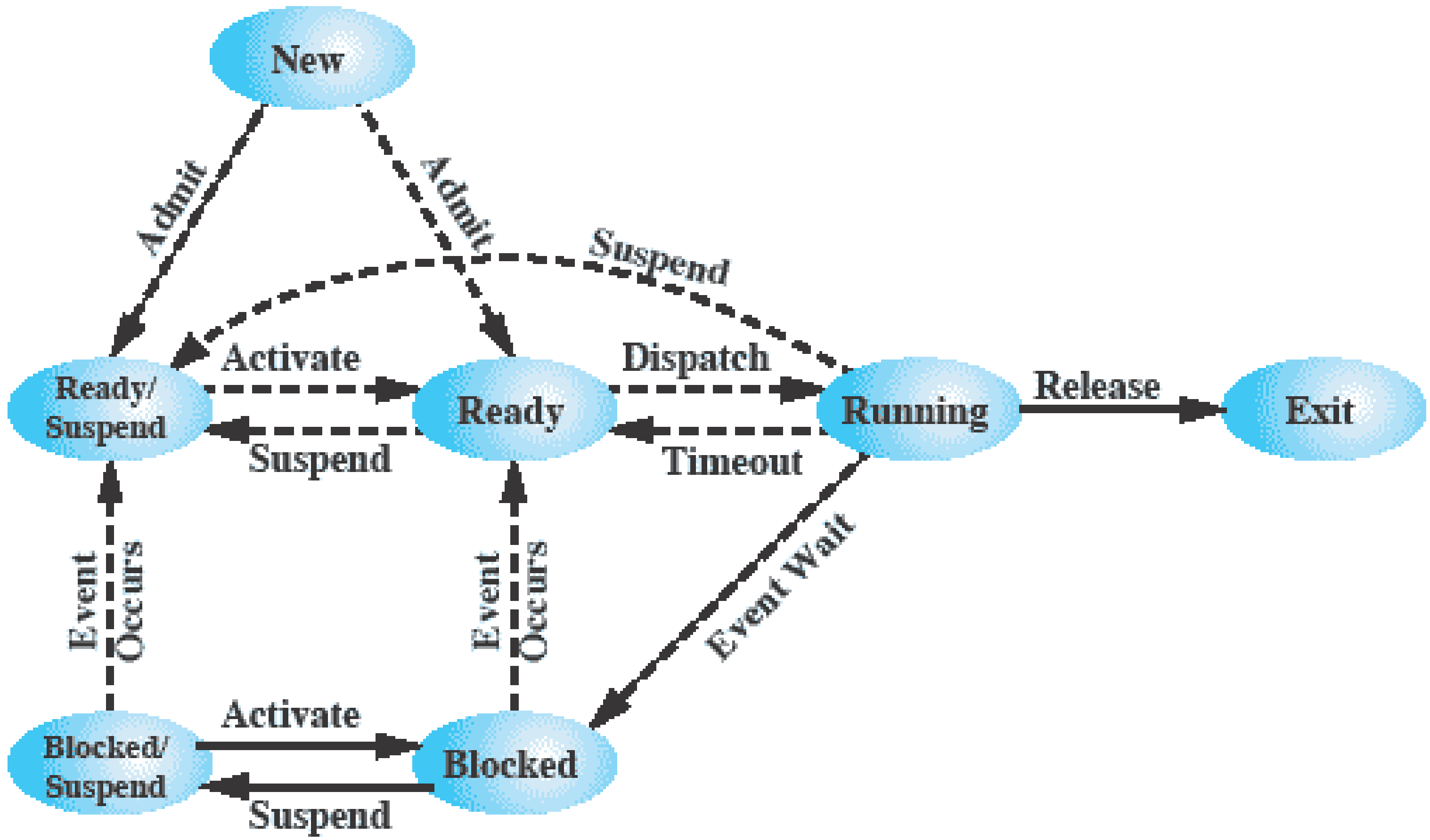

(b) Multiple blocked queues

# One Extra State!




Swapped to disk

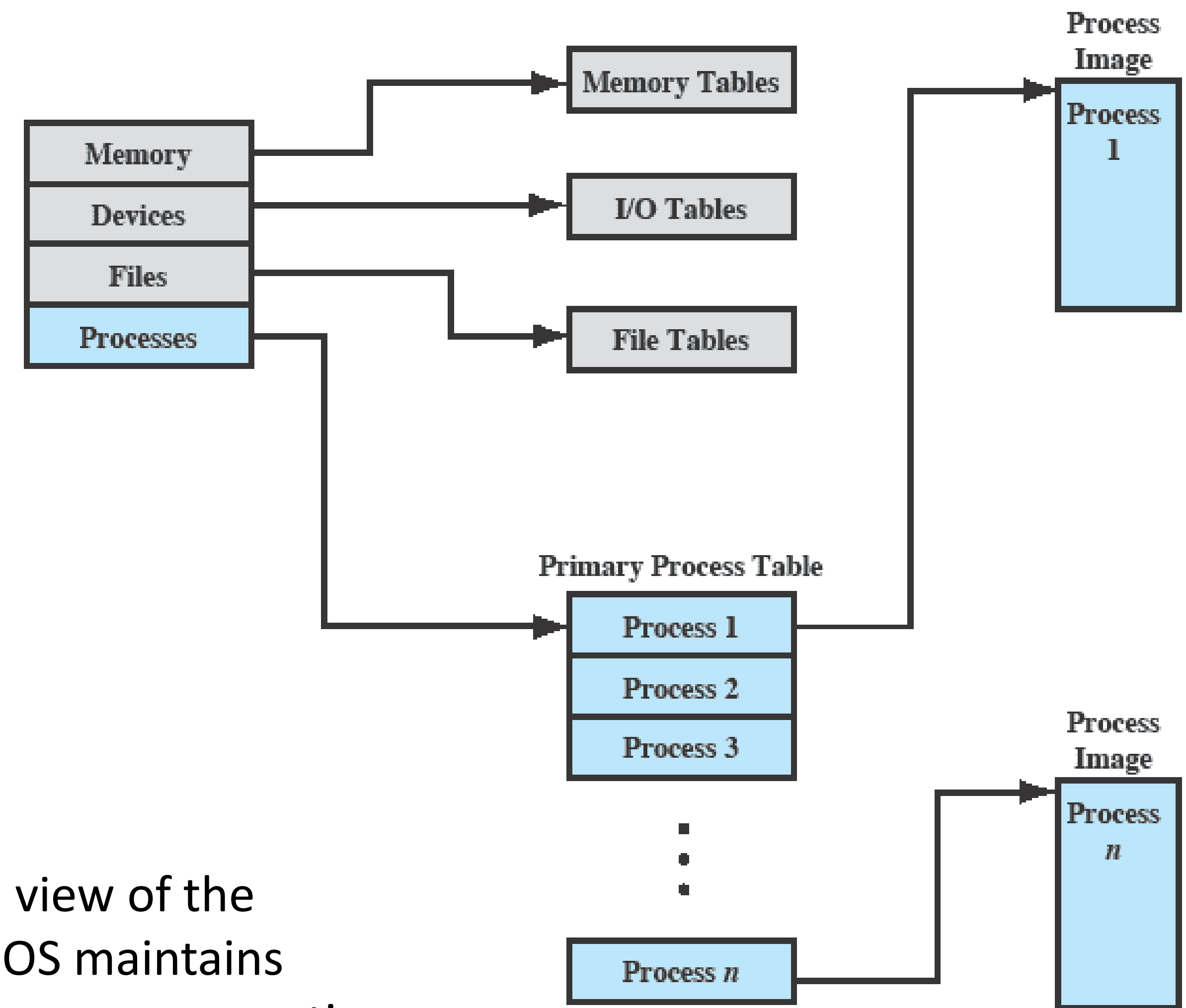
# One Extra State!



# Implementation of Processes

- OS maintains a Control table (also called process table)
- An array of structures (or a hash table)
- One entry per process

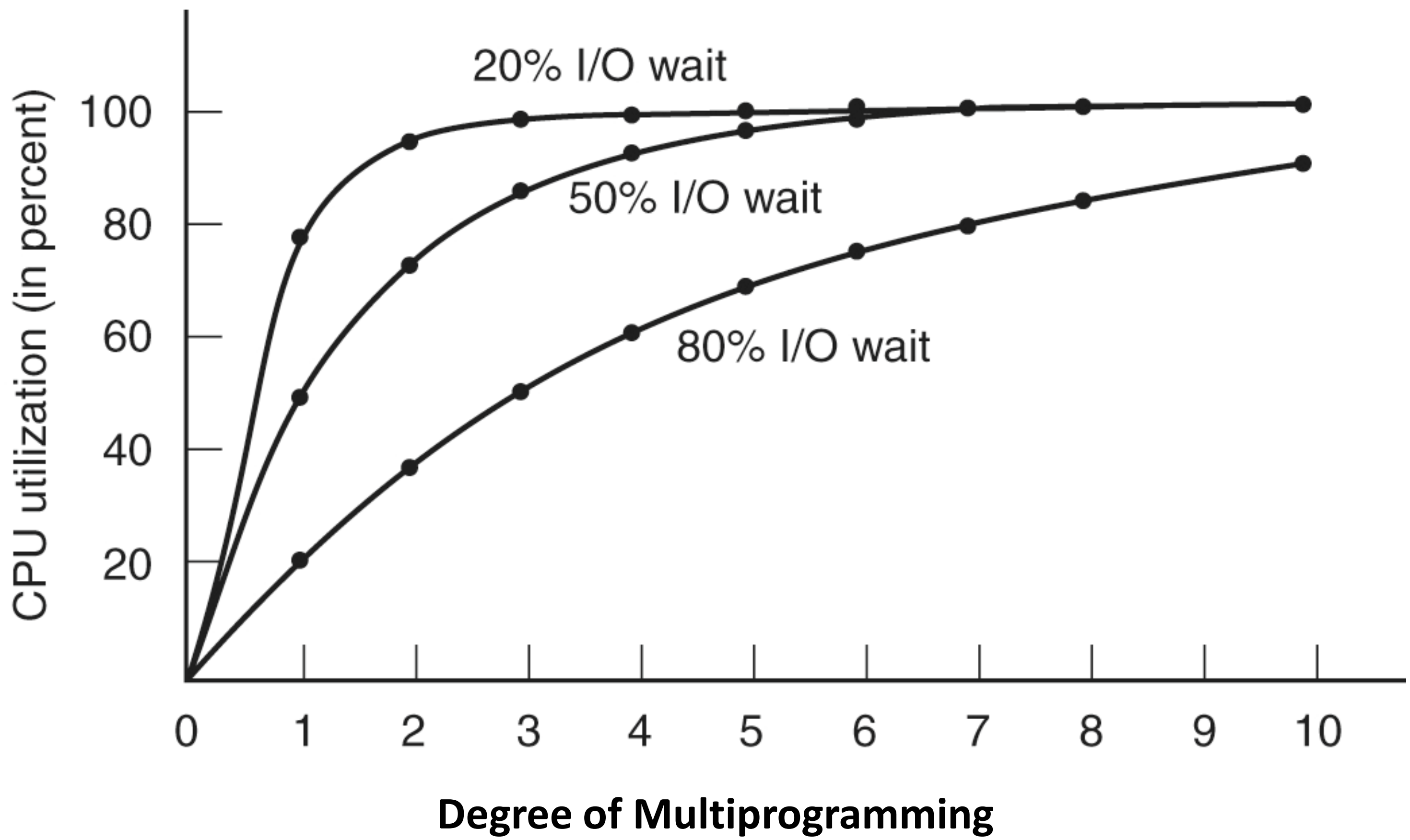
| Process management | Memory management | File management |
| --- | --- | --- |
| Registers | Pointer to text segment info | Root directory |
| Program counter | Pointer to data segment info | Working directory |
| Program status word | Pointer to stack segment info | File descriptors |
| Stack pointer | | User ID |
| Process state | | Group ID |
| Priority | | |
| Scheduling parameters | | |
| Process ID | | |
| Parent process | | |
| Process group | | |
| Signals | | |
| Time when process started | | |
| CPU time used | | |
| Children's CPU time | | |
| Time of next alarm | | |




Conceptual view of the tables that OS maintains in order to manage execution of processes on resources.

# Simple Modeling of Multiprogramming

- A process spends fraction *p* waiting for I/O
- Assume *n* processors in memory at once
- The probability that all processes are waiting for I/O at once is $p^n$
- So -> CPU Utilization = $1 - p^n$




Multiprogramming lets processes use the CPU when it would otherwise become idle.

# How to multiprogramming

- Really a question of how to increase concurrency.
- Example Webserver:
  - If single process, every system call that blocks will block forward progress
  - Let's discuss !!!!!!!

# Solution #1

- Multiple Processes

- What's the issue ?
  - Resource consumption
  - Who owns perceived single resource:
    - E.g. webserver port 80 / 1080 / 8080 ????

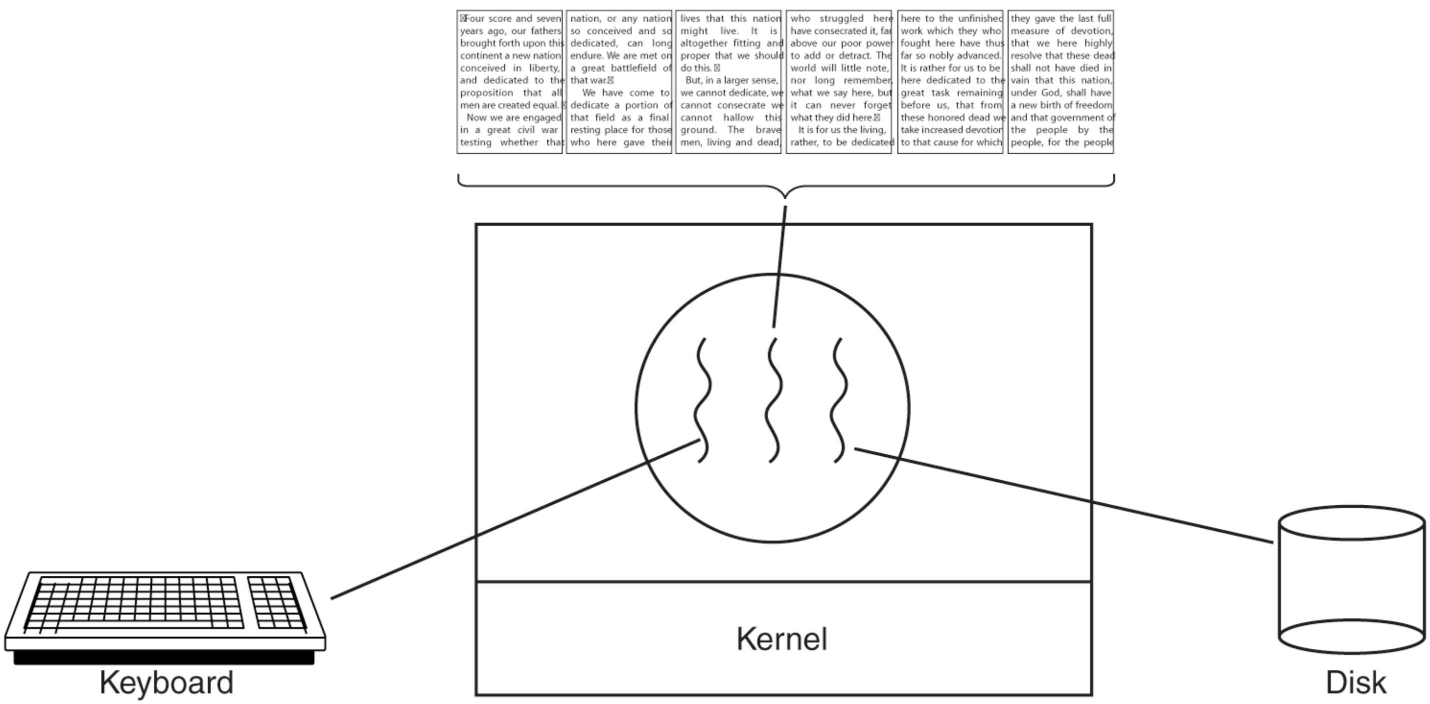
# Threads

- Multiple threads of control within a process
- All threads of a process share the same address space

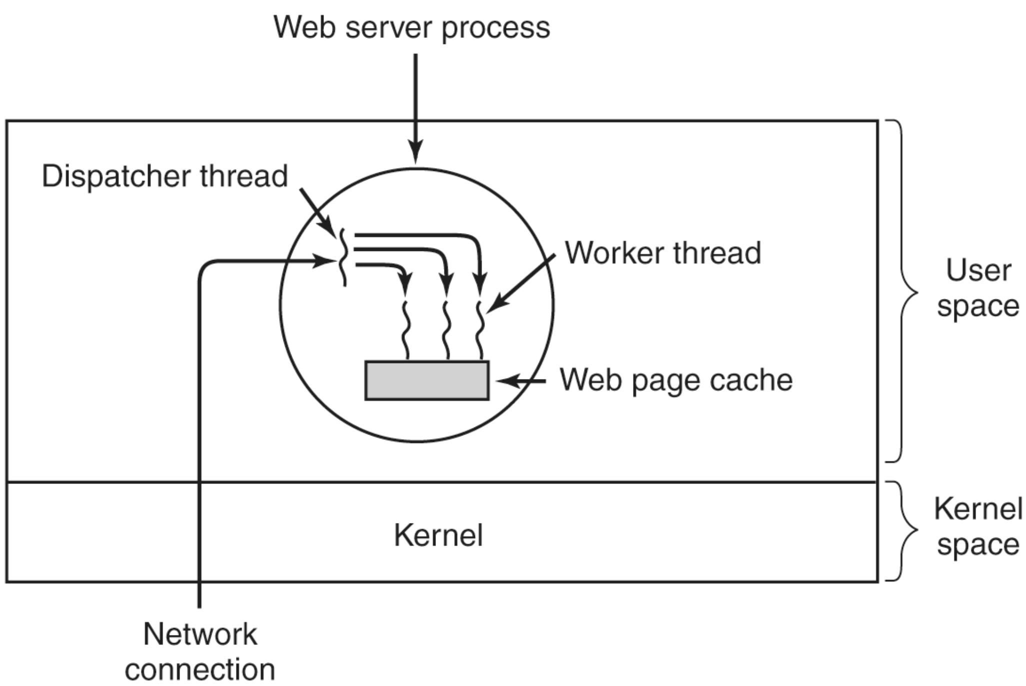
# Why Threads?

- For some applications many activities can happen at once
  - With threads, programming becomes easier
  - Benefit applications with I/O and processing that can overlap
- Lighter weight than processes
  - Faster to create and restore

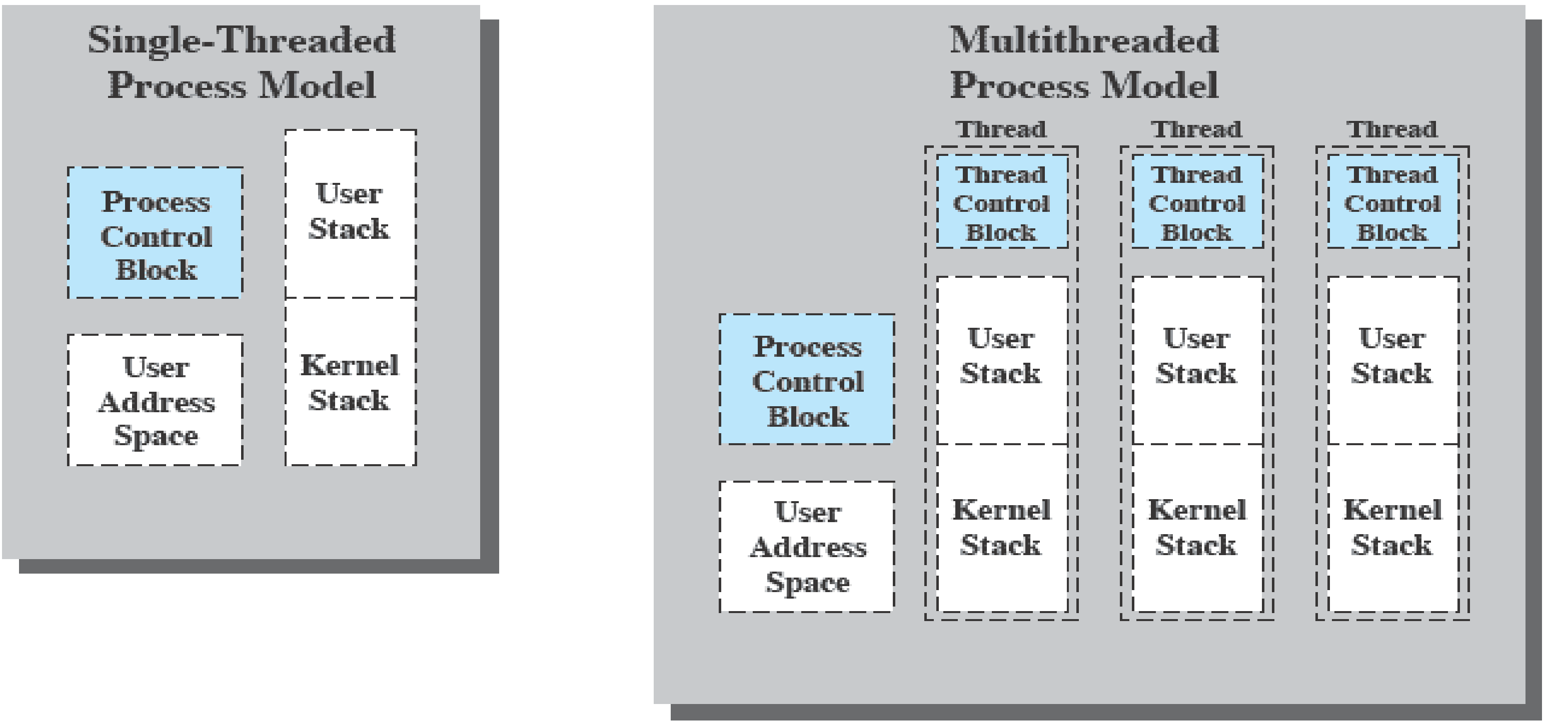
# Example 1:
# A Word Processor



# Example 2: Multithreaded Web Server



# Processes vs. Threads



# Processes vs Threads

- Process groups resources
- Threads are entities scheduled for execution on CPU
- No protections among threads (unlike processes) [Why?]
- Thread can be in any of several states: running, blocked, ready, and terminated

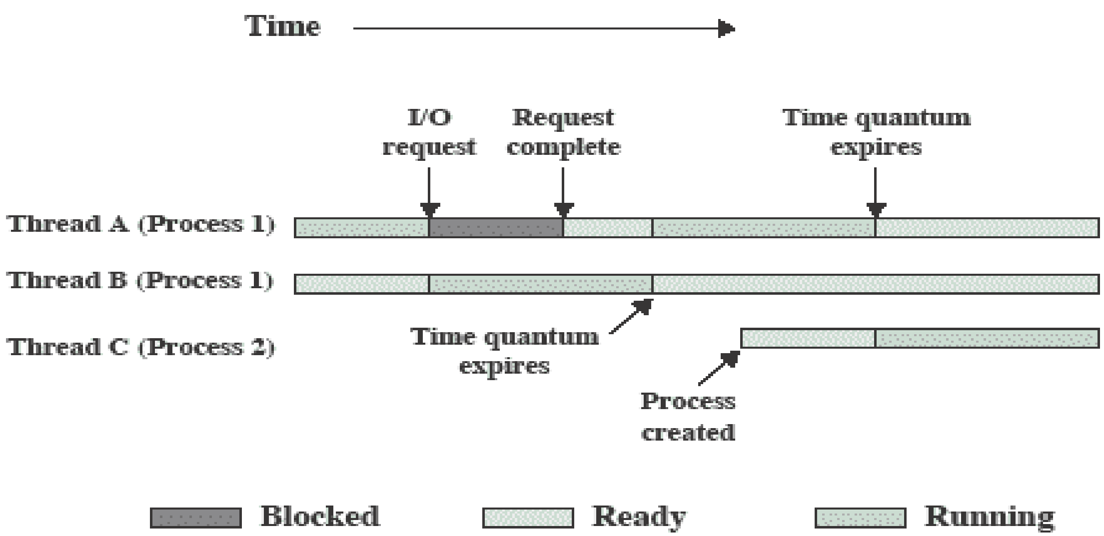
# Processes vs Threads

- The unit of dispatching is referred to as a ***thread*** or ***lightweight process***
- The unit of resource ownership is referred to as a ***process*** or ***task***
- ***Multithreading*** - The ability of an OS to support multiple, concurrent paths of execution within a single process

# Processes vs Threads

- Process is the unit for resource allocation and a unit of protection.
- Process has its own address space.
- A thread has:
  - an execution state (Running, Ready, etc.)
  - saved thread context when not running
  - an execution stack
  - some per-thread static storage for local variables
  - access to the memory and resources of its process (all threads of a process share this)

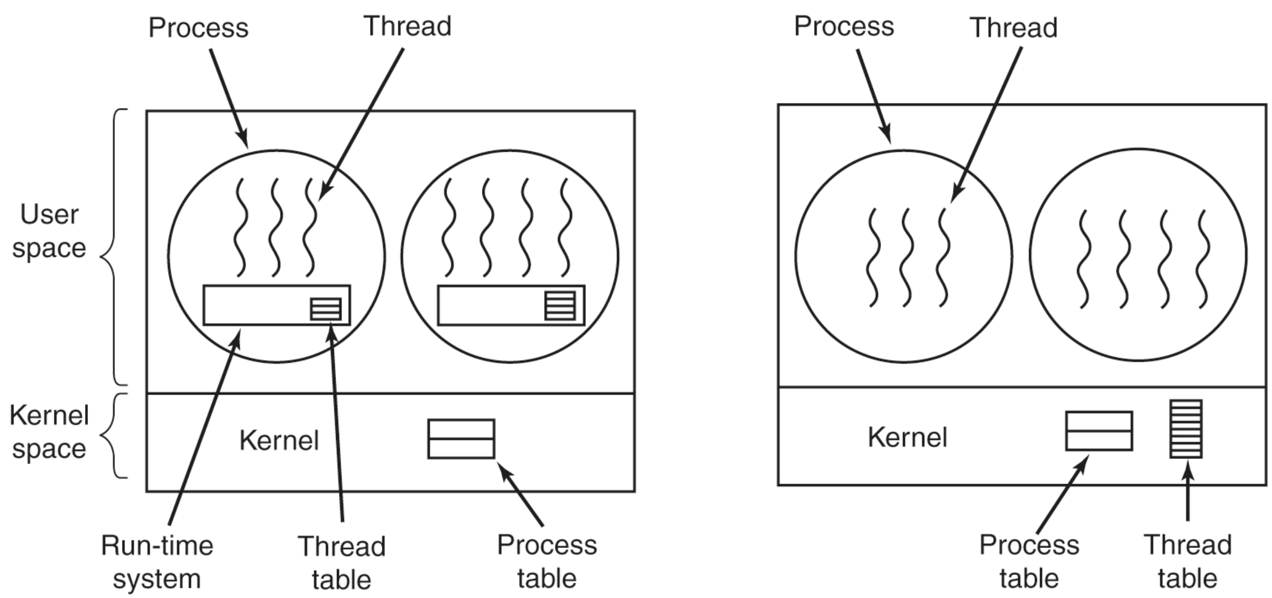
# Multithreading on Uniprocessor System



# Where to Put The Thread Package?

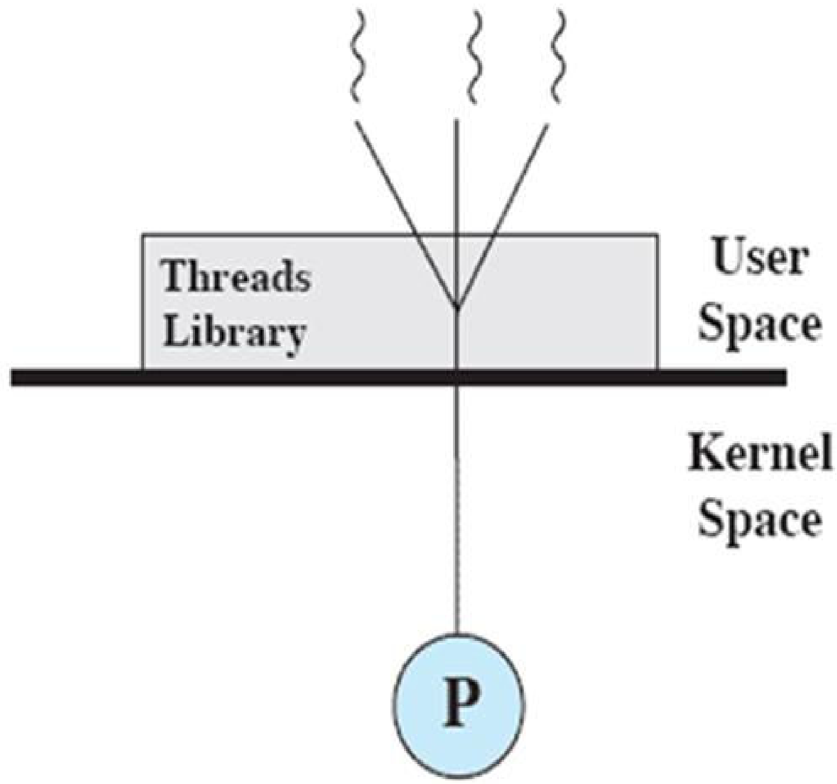
User space

Kernel space



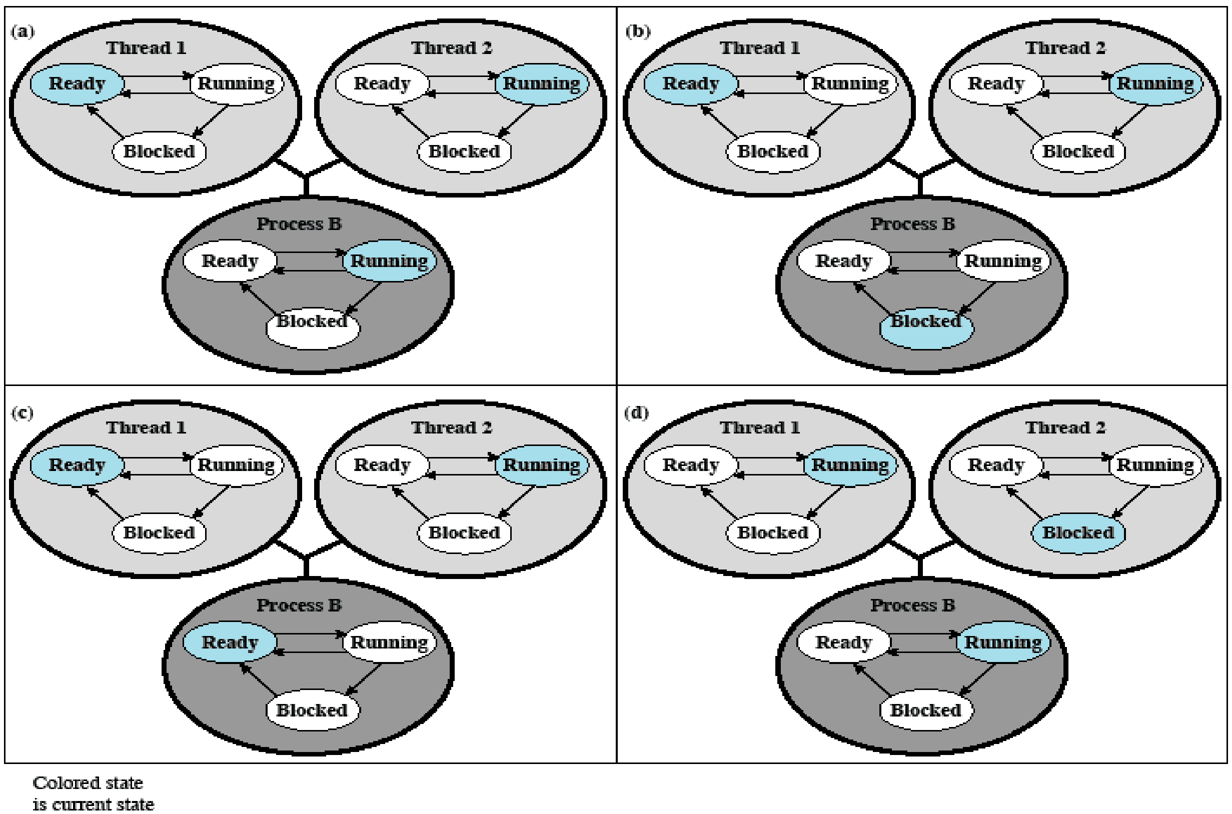
# User-Level Threads (ULT)

- All thread management is done by the application
- The kernel is not aware of the existence of threads



# User-Level Threads (ULTs)

- The kernel continues to schedule the process as a unit and assigns a single execution state .



# Implementing Threads in User Space

- Threads are implemented by a library
- Kernel knows nothing about threads
- Each process needs its own private thread table
- Thread table is managed by the runtime system

# User-Level Threads (ULTs)

**Advantages**

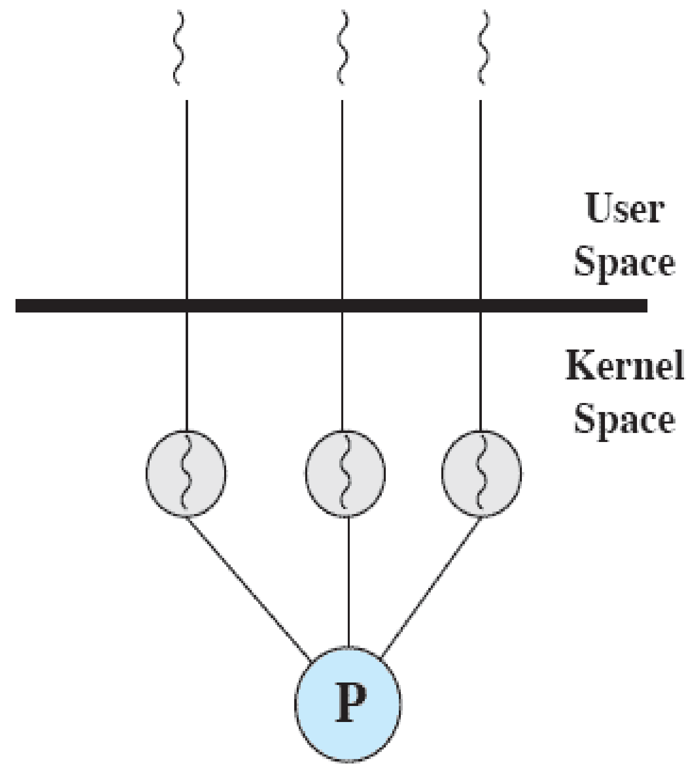
- Thread switch does not require kernel-mode.
- Scheduling (of threads) can be application specific.
- Can run on any OS.
- Scale better

**Disadvantages**

- A system-call by one thread can block all threads of that process.
- Page fault blocks the whole process
- In pure ULT, multithreading cannot take advantage of multiprocessing

# Kernel-Level Threads (KLTs)

- Thread management is done by the kernel
- no thread management is done by the application
- Windows OS is an example of this approach



# Kernel-Level Threads (KLTs)

**Advantages**

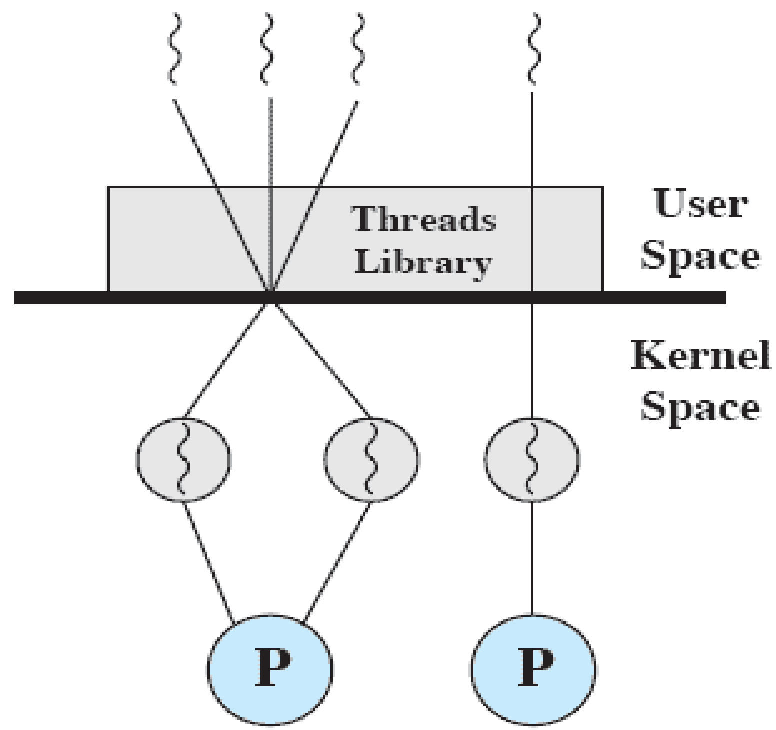
- The kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread in a process is blocked, the kernel can schedule another thread of the same process
- Kernel routines can be multithreaded

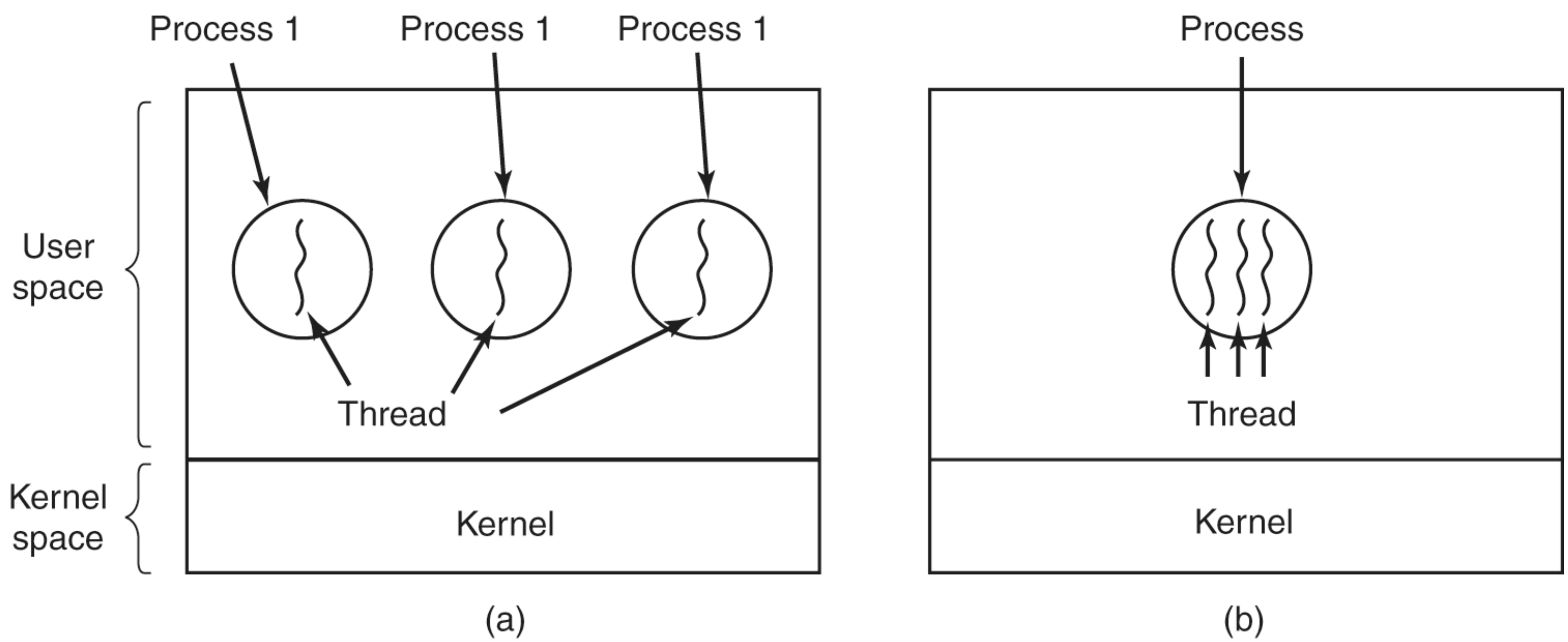
**Disadvantages**

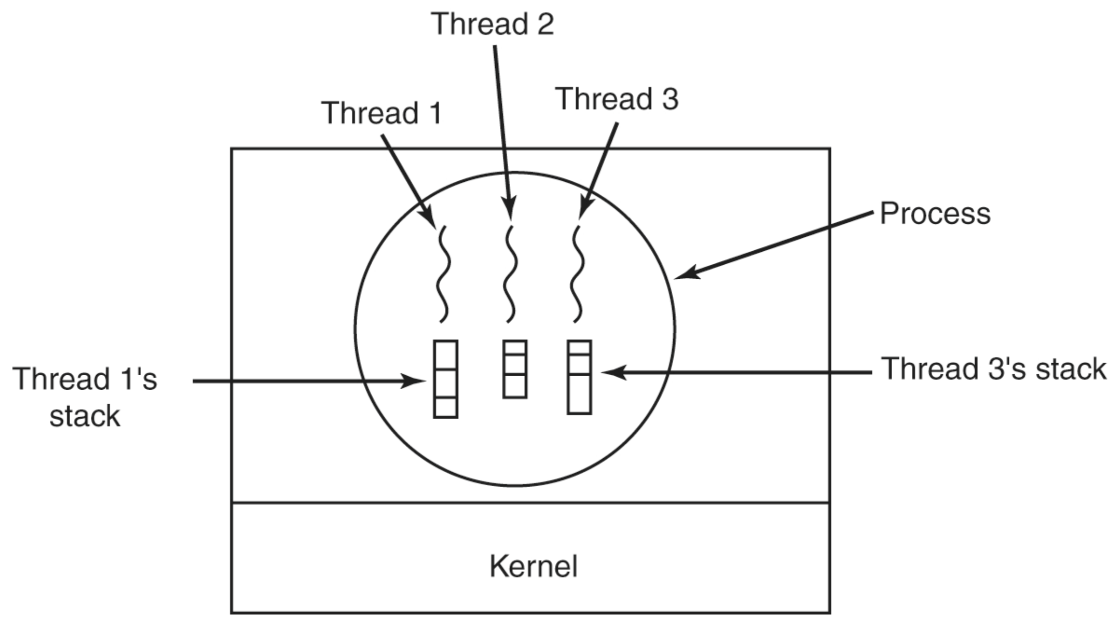
- The transfer of control from one thread to another within the same process requires a mode switch to the kernel

# Combined (Hybrid) Approach

- Thread creation is done completely in user space.
- Bulk of scheduling and synchronization of threads is by the application (i.e. user space).
- Multiple ULTs from a single application are mapped onto (smaller or equal) number of KLTs.
- Solaris is an example




Process 1
Process 1
Process 1
User space
Thread
Kernel space
Kernel
(a)
Process
Thread
Kernel
(b)




Each thread has its own stack (Why?).

# Implementing Threads in Kernel Space

- Kernel knows about and manages the threads
- No runtime is needed in each process
- Creating/destroying/(other thread related operations) a thread involves a system call

# Implementing Threads in Kernel Space

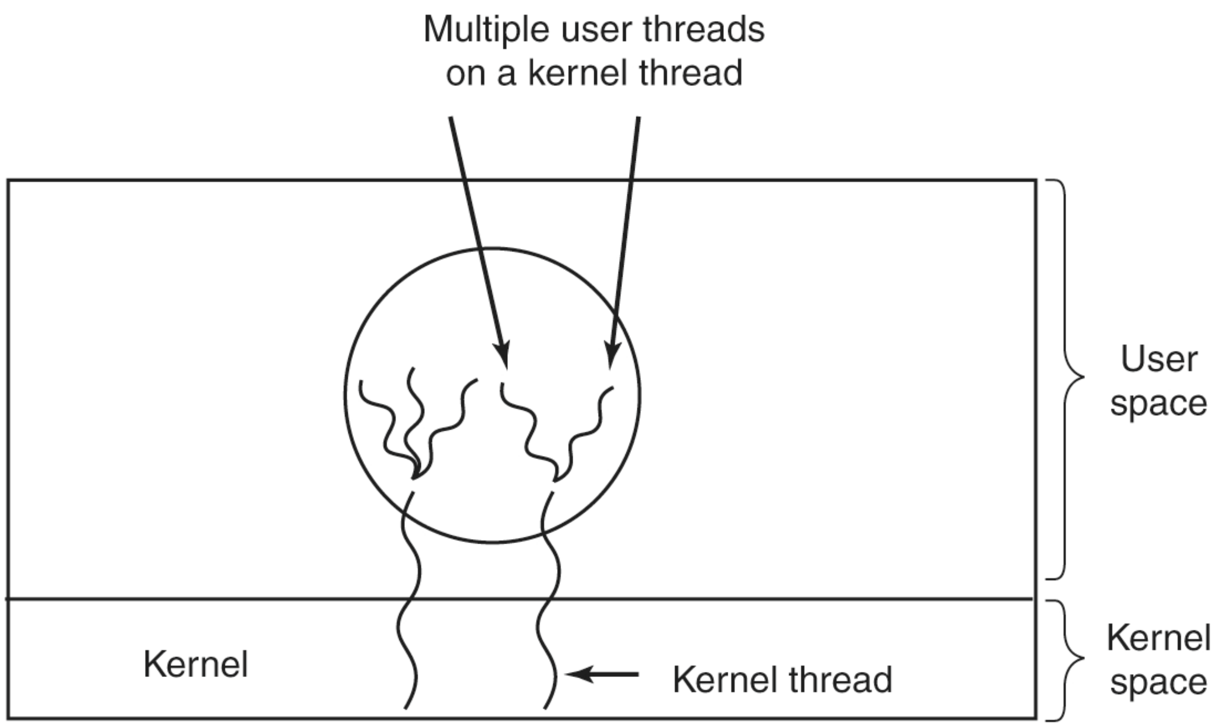
**Advantages**

- When a thread blocks (due to page fault or blocking system calls) the OS can execute another thread from the same process

**Disadvantages**

- Cost of system call is very high

# Hybrid Implementation



# PCB vs TCB

- Process Control Block handles global process resources
- Thread Control Block handles thread execution resources

| Per process items | Per thread items |
|---|---|
| Address space | Program counter |
| Global variables | Registers |
| Open files | Stack |
| Child processes | State |
| Pending alarms | |
| Signals and signal handlers | |
| Accounting information | |

- pids vs. tid

```
frankeh@NYU2:~$ ps -edfmT | grep --color=no -e "^[f|U]" | head
UID        PID  SPID  PPID  C STIME TTY          TIME CMD
frankeh   1445     -  1431  0 11:46 ?        00:00:00 init --user
frankeh      -  1445     -  0 11:46 -        00:00:00 -
frankeh   1500     -     1  0 11:46 ?        00:00:00 /usr/bin/VBoxClient --clipboard
frankeh      -  1500     -  0 11:46 -        00:00:00 -
frankeh      -  1519     -  0 11:46 -        00:00:00 -
frankeh   1508     -     1  0 11:46 ?        00:00:00 /usr/bin/VBoxClient --display
frankeh      -  1508     -  0 11:46 -        00:00:00 -
frankeh      -  1523     -  0 11:46 -        00:00:00 -
frankeh   1512     -     1  0 11:46 ?        00:00:00 /usr/bin/VBoxClient --seamless
```

# Different Naming Conventions

- Thread Models are also knows as general ratio of user threads over kernels threads
- 1:1 : each user thread == kernel thread
- M:1 : user level thread mode
- M:N : hybrid model

# How are threads created ?

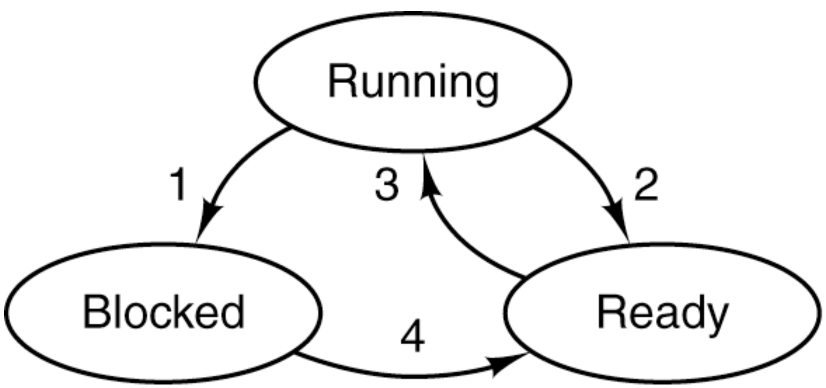
```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg  );
```

Assuming 1:1 model:

a) Allocates a new stack via malloc
b) Calls clone() [ see later ] to create a new schedulable thread
c) Sets the threads stack pointer to (a)
d) Calls  (*start_routine)(arg)

# Thread State Model:

- What changes to the state model in a kernel based thread model ?
- Really replace "process" with "thread" and you are basically there.
- Often we interchangeably use thread and process scheduling.




Thread

1. ~~Process~~ blocks for input
2. Scheduler picks another ~~process~~
3. Scheduler picks this ~~process~~
4. Input becomes available

```
#>  sed –e "s/[P|p]rocess/thread/g"
```

# Some Unix Details

- Creation of a new process:   fork()
- Executing a program in that new process
- Signal notifications

# fork()

**#include <unistd.h>**

**pid_t fork(void);**

## Description

**fork**() creates a new process by duplicating the calling process. The new process, referred to as the *child*, is an exact duplicate of the calling process, referred to as the *parent*, except for the following points:

*
The child has its own unique process ID, and this PID does not match the ID of any existing process group (***setpgid***(*2*)).

*
The child's parent process ID is the same as the parent's process ID.

*
The child does not inherit its parent's memory locks (***mlock***(*2*), ***mlockall***(*2*)).

*
Process resource utilizations (***getrusage***(*2*)) and CPU time counters (***times***(*2*)) are reset to zero in the child.

*
The child's set of pending signals is initially empty (***sigpending***(*2*)).

*
The child does not inherit semaphore adjustments from its parent (***semop***(*2*)).

*
The child does not inherit record locks from its parent (***fcntl***(*2*)).

*
The child does not inherit timers from its parent (***setitimer***(*2*), ***alarm***(*2*), ***timer_create***(*2*)).

*
The child does not inherit outstanding asynchronous I/O operations from its parent (***aio_read***(*3*), ***aio_write***(*3*)), nor does it inherit any asynchronous I/O contexts from its parent (see ***io_setup***(*2*)).

# fork()

```c
#include <stdio.h>
#include <unistd.h>
int main(int argc, char **argv)
{
    pid_t pid = fork();
    if (pid == 0)
    {
        // child process
    }
    else if (pid > 0)
    {
        // parent process
     }
    else
    {
        // fork failed
        printf("fork() failed!\n");
        return 1;
    }
```

Creates new PCB and Address Space

Child runs now here

Parent continues here

# execv()

execl, execlp, execle, execv, execvp, execvpe - execute a file

## Synopsis

**#include <unistd.h>**

**extern char **environ;**

**int execl(const char** **path*, **const char** **arg*, **...);**
**int execlp(const char** **file*, **const char** **arg*, **...);**
**int execle(const char** **path*, **const char** **arg*,
**..., char * const** *envp*[]);
**int execv(const char** **path*, **char *const** *argv*[]);
**int execvp(const char** **file*, **char *const** *argv*[]);
**int execvpe(const char** **file*, **char *const** *argv*[],
**char *const** *envp*[]);

## Description

The **exec**() family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for **execve**(2). (See the manual page for **execve**(2) for further details about the replacement of the current process image.)

The initial argument for these functions is the name of a file that is to be executed.

The *const char *arg* and subsequent ellipses in the **execl**(), **execlp**(), and **execle**() functions can be thought of as *arg0*, *arg1*, ..., *argn*. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the filename associated with the file being executed. The list of arguments *must* be terminated by a NULL pointer, and, since these are variadic functions, this pointer must be cast *(char *) NULL*.

The **execv**(), **execvp**(), and **execvpe**() functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers *must* be terminated by a NULL pointer.

The **execle**() and **execvpe**() functions allow the caller to specify the environment of the executed program via the argument *envp*. The *envp* argument is an array of pointers to null-terminated strings and *must* be terminated by a NULL pointer. The other functions take the environment for the new process image from the external variable *environ* in the calling process.

# execv()

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char **argv)
{
    pid_t pid = fork();
    if (pid == 0)
    {
        execv(path,executablename)
    }
    else if (pid > 0)
    {
        int status;
        waitpid(pid,&status,option)
     }
    else
    {
        // fork failed
        printf("fork() failed!\n");
        return 1;
    }
```

→ Creates new PCB and Address Space

→ Child starts new program image of new process

→ Parent waits for child process to finish

# clone()

**NAME** top

```
clone, __clone2 - create a child process
```

**SYNOPSIS** top

```
/* Prototype for the glibc wrapper function */

#define _GNU_SOURCE
#include <sched.h>

int clone(int (*fn)(void *), void *child_stack,
          int flags, void *arg, ...
          /* pid_t *ptid, struct user_desc *tls, pid_t *ctid */ );

/* Prototype for the raw system call */

long clone(unsigned long flags, void *child_stack,
          void *ptid, void *ctid,
          struct pt_regs *regs);
```

**DESCRIPTION** top

clone() creates a new process, in a manner similar to fork(2).

This page describes both the glibc clone() wrapper function and the underlying system call on which it is based. The main text describes the wrapper function; the differences for the raw system call are described toward the end of this page.

Unlike fork(2), clone() allows the child process to share parts of its execution context with the calling process, such as the memory space, the table of file descriptors, and the table of signal handlers. (Note that on this manual page, "calling process" normally corresponds to "parent process". But see the description of CLONE_PARENT below.)

One use of clone() is to implement threads: multiple threads of control in a program that run concurrently in a shared memory space.

See:

http://man7.org/linux/man-pages/man2/clone.2.html

For what things can be clone.

This is the bases on how threads are created.

Linux/Unix internally uses objects to create processes / threads

How these objects interrelate depends on fork/clone calls

(see blackboard).

# Signal()

- Means to "signal a process"
- There are a set of signals that can be sent to a process (some require permissions).
- Process indicates which signal it wants to catch and provides a call back function
- When the signal is to be sent (event or "kill <signal> pid") the kernel delivers that signal.

| *No.* | *Short name* | *What it means* |
|---|---|---|
| 1 | SIGHUP | If a process is being run from terminal and that terminal suddenly goes away then the process receives this signal. "HUP" is short for "**hang up**" and refers to hanging up the telephone in the days of telephone modems. |
| 2 | SIGINT | The process was "**int**errupted". This happens when you press Control+C on the controlling terminal. |
| 3 | SIGQUIT | |
| 4 | SIGILL | **Ill**egal instruction. The program contained some machine code the CPU can't understand. |
| 5 | SIGTRAP | This signal is used mainly from within debuggers and program tracers. |
| 6 | SIGABRT | The program called the `abort()` function. This is an emergency stop. |
| 7 | SIGBUS | An attempt was made to access memory incorrectly. This can be caused by alignment errors in memory access etc. |
| 8 | SIGFPE | A **f**loating **p**oint **e**xception happened in the program. |
| 9 | SIGKILL | The process was explicitly killed by somebody wielding the `kill` program. |
| 10 | SIGUSR1 | Left for the programmers to do whatever they want. |
| 11 | SIGSEGV | An attempt was made to access memory not allocated to the process. This is often caused by reading off the end of arrays etc. |
| 12 | SIGUSR2 | Left for the programmers to do whatever they want. |
| 13 | SIGPIPE | If a process is producing output that is being fed into another process that consume it via a **pipe** ("`producer \| consumer`") and the consumer dies then the producer is sent this signal. |
| 14 | SIGALRM | A process can request a "wake up call" from the operating system at some time in the future by calling the `alarm()` function. When that time comes round the wake up call consists of this signal. |
| 15 | SIGTERM | The process was explicitly killed by somebody wielding the `kill` program. |
| *16* | *unused* | |
| 17 | SIGCHLD | The process had previously created one or more **child** processes with the `fork()` function. One or more of these processes has since died. |
| 18 | SIGCONT | (To be read in conjunction with SIGSTOP.)<br>If a process has been paused by sending it SIGSTOP then sending SIGCONT to the process wakes it up again ("**cont**inues" it). |
| 19 | SIGSTOP | (To be read in conjunction with SIGCONT.)<br>If a process is sent SIGSTOP it is paused by the operating system. All its |

# Signal()

```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>

void sig_handler(int signo) {
   if (signo == SIGINT)
      printf("received SIGINT\n");
}

int main(void) {

  // install the handler
  if (signal(SIGINT, sig_handler) == SIG_ERR)
      printf("\ncan't catch SIGINT\n");

  // A long long wait so that we can easily issue a
  // signal to this process

  while(1)  sleep(1);
  return 0;
}
```

When signal is delivered:
* kernel stops threads in process
* kernel "adds a stack frame"
* kernel "switches IPC to *sig_handler*
* kernel continues thread
* thread will continue with *sig_handler*
* Thread on completion will call back to kernel

| *No.* | *Short name* | *What it means* |
|---|---|---|
| | | state is preserved ready for it to be restarted (by SIGCONT) but it doesn't get any more CPU cycles until then. |
| 20 | SIGTSTP | Essentially the same as SIGSTOP. This is the signal sent when the user hits Control+Z on the terminal. (SIGTSTP is short for "**t**erminal **st**o**p**") The only difference between SIGTSTP and SIGSTOP is that pausing is only the *default* action for SIGTSTP but is the *required* action for SIGSTOP. The process can opt to handle SIGTSTP differently but gets no choice regarding SIGSTOP. |
| 21 | SIGTTIN | The operating system sends this signal to a backgrounded process when it tries to read **in**put from its terminal. The typical response is to pause (as per SIGSTOP and SIFTSTP) and wait for the SIGCONT that arrives when the process is brought back to the foreground. |
| 22 | SIGTTOU | The operating system sends this signal to a backgrounded process when it tries to write **ou**tput to its terminal. The typical response is as per SIGTTIN. |
| 23 | SIGURG | The operating system sends this signal to a process using a network connection when "**urg**ent" out of band data is sent to it. |
| 24 | SIGXCPU | The operating system sends this signal to a process that has exceeded its CPU limit. You can cancel any CPU limit with the shell command "`ulimit -t unlimited`" prior to running `make` though it is more likely that something has gone wrong if you reach the CPU limit in `make`. |
| 25 | SIGXFSZ | The operating system sends this signal to a process that has tried to create a file above the file size limit. You can cancel any file size limit with the shell command "ulimit -f unlimited" prior to running `make` though it is more likely that something has gone wrong if you reach the file size limit in `make`. |
| 26 | SIGVTALRM | This is very similar to SIGALRM, but while SIGALRM is sent after a certain amount of real time has passed, SIGVTALRM is sent after a certain amount of time has been spent running the process. |
| 27 | SIGPROF | This is also very similar to SIGALRM and SIGVTALRM, but while SIGALRM is sent after a certain amount of real time has passed, SIGPROF is sent after a certain amount of time has been spent running the process and running system code on behalf of the process. |
| 28 | SIGWINCH | (Mostly unused these days.) A process used to be sent this signal when one of its windows was resized. |
| 29 | SIGIO | (Also known as SIGPOLL.) A process can arrange to have this signal sent to it when there is some input ready for it to process or an output channel has become ready for writing. |
| 30 | SIGPWR | A signal sent to processes by a power management service to indicate that power has switched to a short term emergency power supply. The process |

# Conclusions

- Process is one the most central concept in OS
- Process vs Thread (understand difference)
  - Process is a resource container with at least one thread of execution
  - Thread is a unit of execution that lives in a process (no thread without a process)
  - Threads share the resources of the owning process.
- Multiprogramming vs multithreading