CSCI-GA.2250-001

Operating Systems

Processes and Threads

Hubertus Franke
frankeh@cims.nyu.edu

Details of Lecture

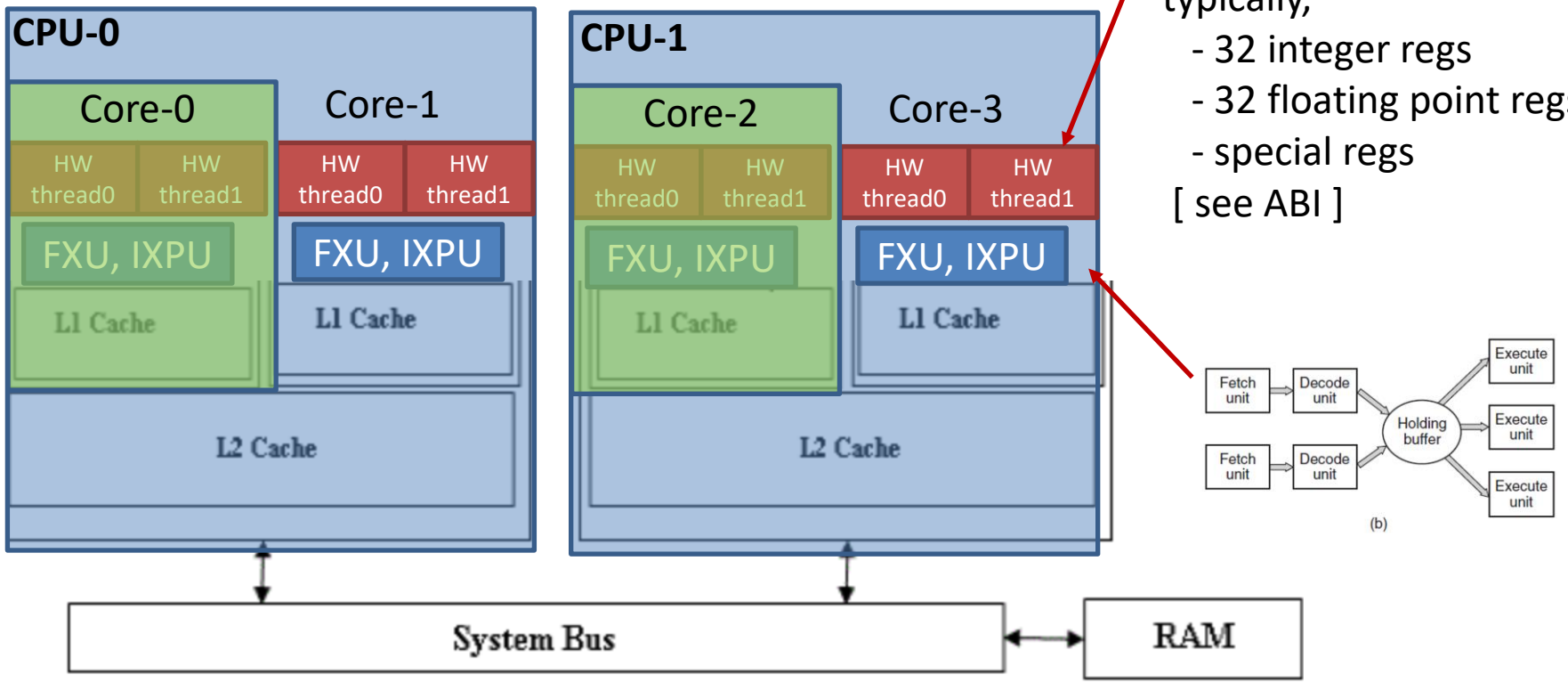
- Process Model
- Process Creation (fork , exec)
- Signals
- Process State / Transition Models
- Multi-programming
- Threads

What is exactly a "processor" from an OS perspective

OS can schedule an independent unit of execution (process or thread) onto each Hardware-Thread (HW-Thread)

Each HW-thread exposes register set typically,
- 32 integer regs
- 32 floating point regs
- special regs

[see ABI]

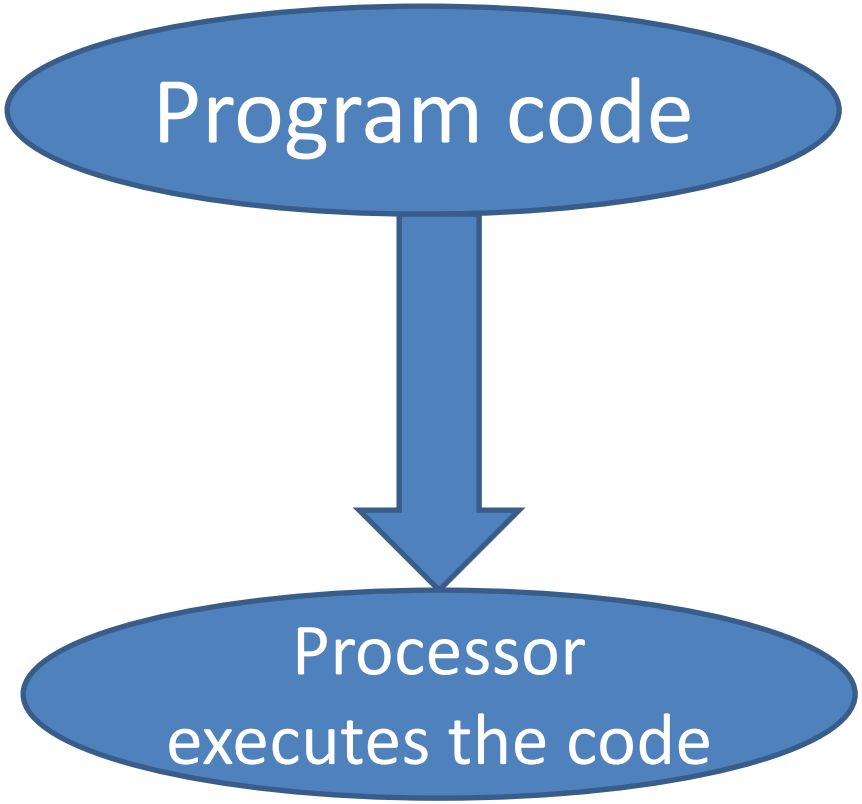


OS Management of Application Execution

- Resources are made available to multiple applications
- Instruction and registers are not virtualized, applications run natively on the hardware.
- Hence, a "processor" or "core" or "hw-thread" can only run one unit of execution (process/thread) at a time.
- The processor is switched among multiple units of execution, so all will appear to be progressing (albeit potentially at reduced speed)

  this "switch" is called a "*context switch*" and is initiated by the operating system, more on this later

- The processor and I/O devices can be used efficiently
  - When application performs I/O, the processor can be used for a different application.

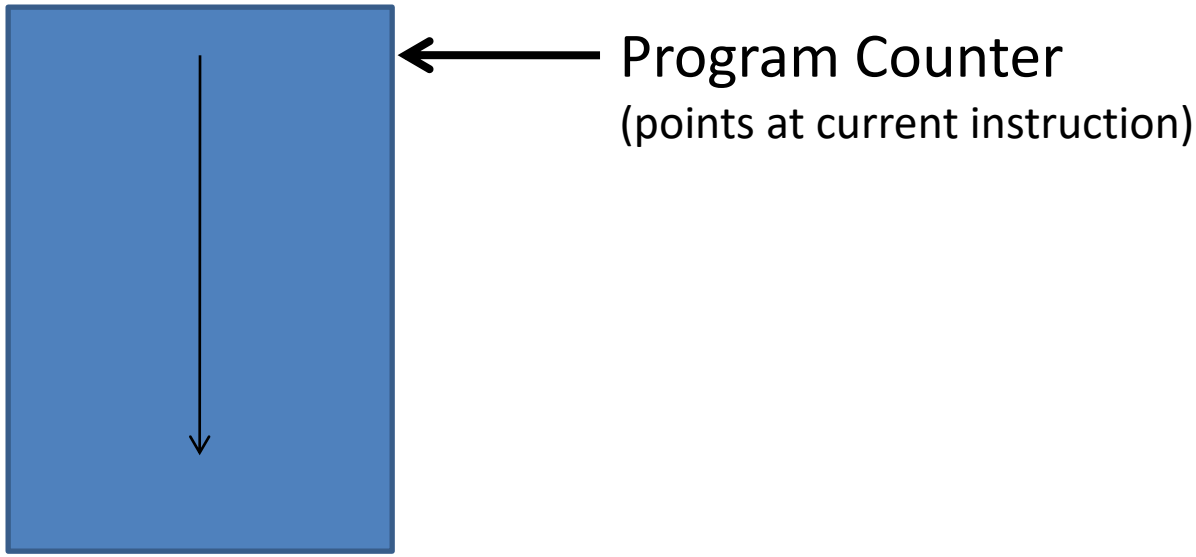


Resides on storage
(just a sequence of bytes
[code , data])

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 has a program, input, output, and state (data).
- A process is an instance of an executing program and includes
  - Variables (memory)
  - Code
  - Program counter (really hardware resource)
  - Registers (- " -)
  - …

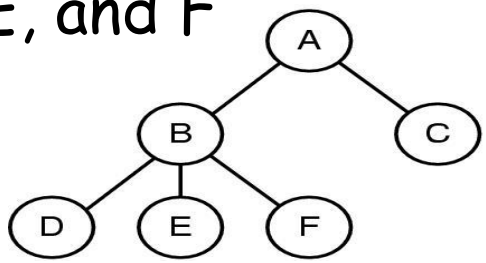
If a program is running twice, does it count as two processes? or one?

Process: a running program

- A process includes
  - Process State (state, registers save areas)
    - Open files, thread(s) state, resources held
  - Address space (view of its private memory)

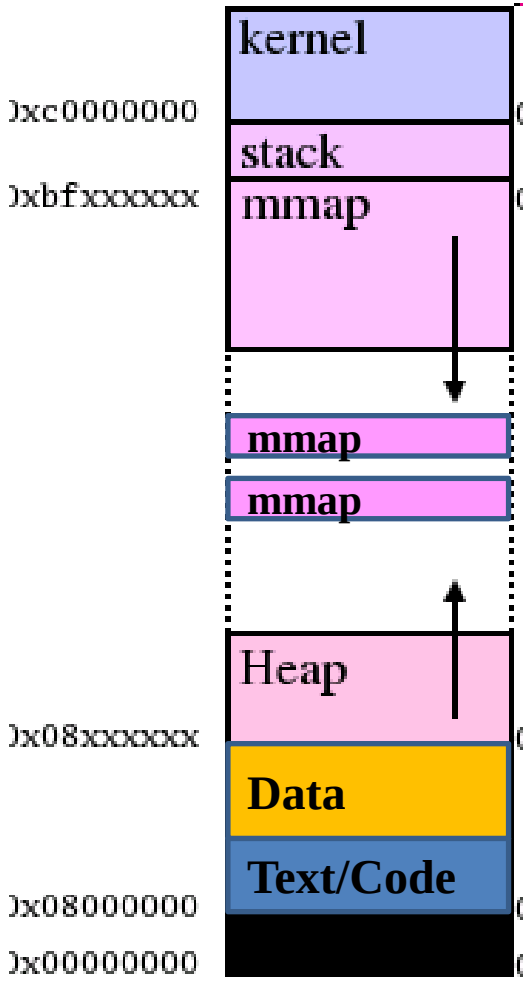
  All state is accessible as an entry in the process (entry) table

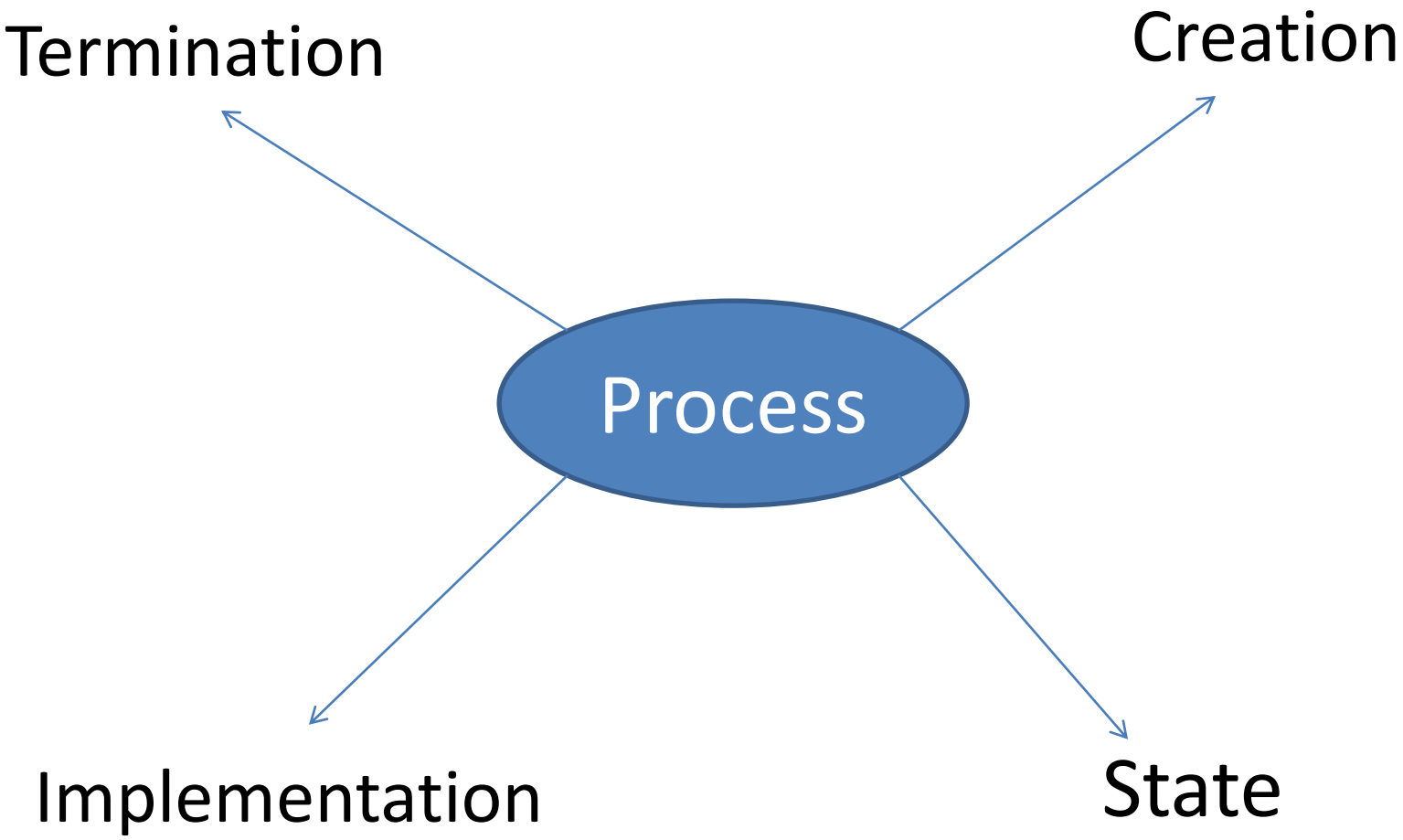
- A process tree
  - A created two child processes, B and C
  - B created three child processes, D, E, and F



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 (will get to it during memory memory management)




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 login

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.

Implementation of Processes

- OS maintains a process table

```
Process* proc_table[];  // not visible to user
```

- An array (or a hash table) of structures
- One entry per process
  - pid is the uniq id the user can refer too in system calls (handle)

`proc_table[pid]` →

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		

Implementation of Processes: Process Control Block (PCB)

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

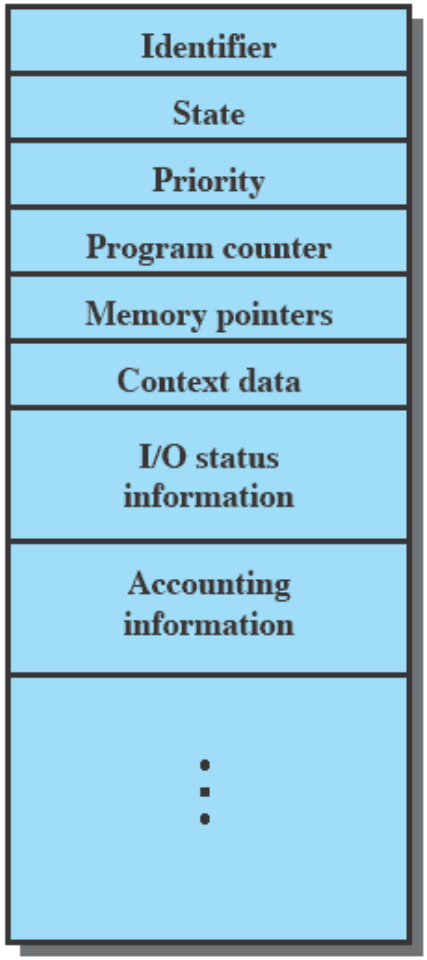
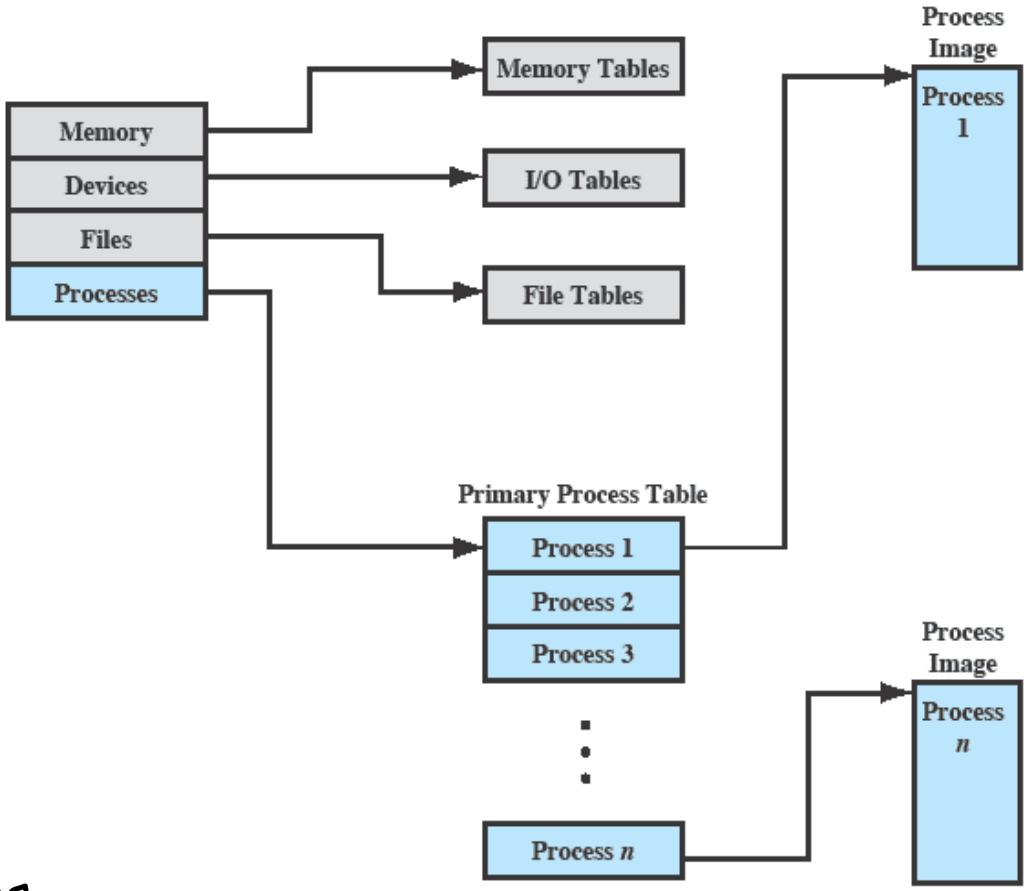


Figure 3.1 Simplified Process Control Block

OS objects related to process

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

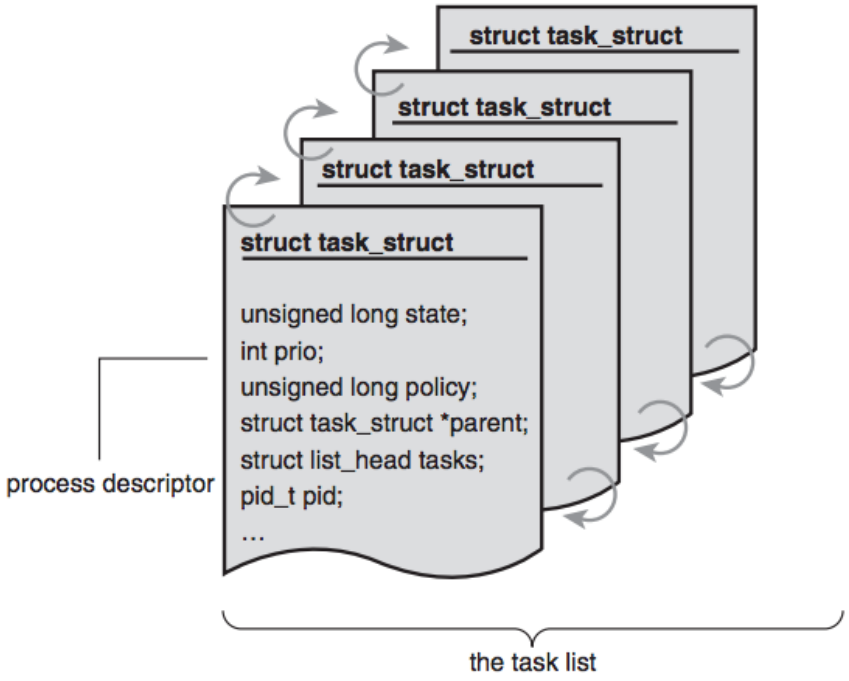


Though there is one PCB per process, there is a "mesh" of multiple objects to allow for sharing between processes where appropriate
- Memory tables, io-tables, ….

Linux Process (Task) object

- Linux Process ~= struct task

~1.5KB + 2*stacks (user + kernel)



Need two separate stacks per process
to ensure isolation between
Kernel and User
(recall systemcall)

user: stack growth till out of space
Kernel: 4KB or 8KB

Some Unix Details

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

- The kernel boot manually creates ONE process (the boot process (pid=0))
- In Linux , pid=0 spawns the init process (pid=1 and kernel threads)

  all regular processes are created by fork() from the init process (pid=1)

```
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 11:11 ?        00:00:02 /sbin/init
root           2       0  0 11:11 ?        00:00:00 [kthreadd]
root           3       2  0 11:11 ?        00:00:00 [rcu_gp]
root           4       2  0 11:11 ?        00:00:00 [rcu_par_gp]
root           6       2  0 11:11 ?        00:00:00 [kworker/0:0H-kblockd]
root           7       2  0 11:11 ?        00:00:00 [kworker/0:1-events]
root           8       2  0 11:11 ?        00:00:00 [kworker/u8:0-events_power_efficient]
root           9       2  0 11:11 ?        00:00:00 [mm_percpu_wq]
root          10       2  0 11:11 ?        00:00:00 [ksoftirqd/0]
```

```
systemd+     626       1  0 11:12 ?        00:00:00 /lib/systemd/systemd-resolved
systemd+     627       1  0 11:12 ?        00:00:00 /lib/systemd/systemd-timesyncd
root         642       1  0 11:12 ?        00:00:00 /usr/sbin/haveged --Foreground --verbose=1 -w 1024
root         670       1  0 11:12 ?        00:00:00 /usr/lib/accountsservice/accounts-daemon
root         671       1  0 11:12 ?        00:00:00 /usr/sbin/acpid
```

```
root        1280       1  0 11:12 ?        00:00:00 /usr/sbin/lightdm
root        1296       1  0 11:12 ?        00:00:00 /usr/sbin/VBoxService --pidfile /var/run/vboxadd-service.sh
root        1298    1280  0 11:12 tty7     00:00:01 /usr/lib/xorg/Xorg -core :0 -seat seat0 -auth /var/run/lightdm
/root/:0 -nolisten tcp vt7 -novtswitch
rtkit       1358       1  0 11:12 ?        00:00:00 /usr/libexec/rtkit-daemon
root        1385       1  0 11:12 ?        00:00:00 /usr/lib/bluetooth/bluetoothd
root        1416    1280  0 11:12 ?        00:00:00 lightdm --session-child 12 19
frankeh     1445       1  0 11:12 ?        00:00:00 /lib/systemd/systemd --user
frankeh     1448    1445  0 11:12 ?        00:00:00 (sd-pam)
```

fork()

```
#include <unistd.h>

pid_t fork(void);
```

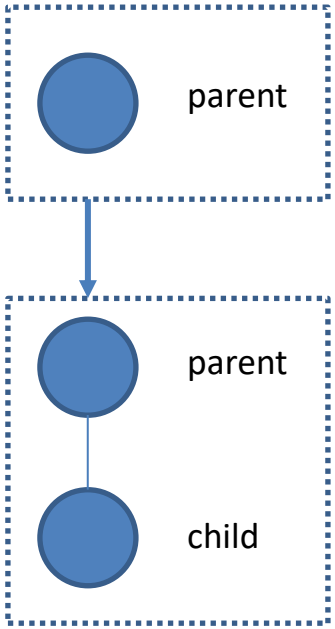
Description

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)).

Only **LOGICALLY** duplicates the whole process, otherwise way to expensive (will cover in MemMgmt)

!



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;
    }
}
```

syscall that creates new PCB and duplicates Address Space

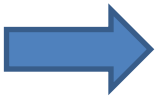
Child runs now here

Parent continues here

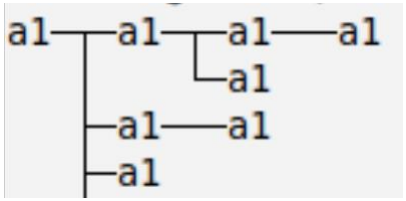
fork()

```c
int main(int argc, char **argv)
{
    for (int i=0; i<3; i++) {
        // observe <i>
        fork();
    }
    sleep(10);
}
```

Resulting Process Tree

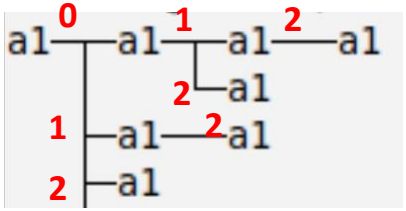


pstree –c <pid-of-first-a1>



On fork() the entire process state is copied (memory, stack , ..) !

Consider variable <i> at “//observe <i>”



Now consider the classical forkbomb():

```c
int main(int argc, char **argv)
{
    for (;;)
        fork();
}
```

It will reender your machine unresponsive in seconds.

fork() is fast !!!

- It is fast because we don't really make a full copy of the process's address space.
- Simple program to measure execution time

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

struct timeval startt, endt, difft;
struct timeval sumt;

#define HOWMANY (5)

int main(int argc, char* argv[])
{
    int pid, i;

    timerclear(&sumt);
    gettimeofday(&startt,NULL);
    gettimeofday(&endt,NULL);
    timersub(&endt, &startt, &difft);
    printf("overhead %ld.%06ld\n\n", difft.tv_sec, difft.tv_usec);

    for (i=0 ; i< HOWMANY ; i++ ) {
       gettimeofday(&startt,NULL);
       pid = fork();
       gettimeofday(&endt,NULL);
       timersub(&endt, &startt, &difft);
       printf("%4d %4d %ld.%06ld\n", getpid(), pid, difft.tv_sec, difft.tv_usec);
       if (pid)
          usleep(10000);
       else
          exit(0);
       timeradd(&sumt, &difft, &sumt);
    }

    printf("Avg %ld.%06ld\n", sumt.tv_sec, sumt.tv_usec / HOWMANY); // bad coding
}
```

```
overhead 0.000000

1736 1737 0.000083
1737    0 0.000176
1736 1738 0.000055
1738    0 0.000134
1736 1739 0.000086
1739    0 0.000156
1736 1740 0.000061
1740    0 0.000107
1736 1741 0.000087
1741    0 0.000151
Avg 0.000074
```

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[]);
```

`execv()` replaces the "image==executable" of the current process with a new one.

The `execv()` functions return only if an error has occurred.
The return value is -1, and `errno` is set to indicate the error.

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“, strerror(errno));
        return -1;
    }
```

→ Creates new PCB and Address Space

→ Child starts new program image of new process
this will NEVER return but for an error

→ Parent waits for child process to finish

Fork , exec, wait (and their variants) are system calls

pid is the object handle that the operating system returns for a process

clone()

See:

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

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.

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

Signal()

- Means to "signal a process" (i.e. get its attention).
- 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.) 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.) 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 user stack frame"
* kernel "switches IPC to *sig_handler*
* kernel continues process
* Process will continue with *sig_handler*
* Process 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 **stop**") 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 **out**put 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

A few basics to remember

- `fork()`, `exec()`, `wait()` and it's variants are OS system calls to create and manipulate processes
- PIDs are *handles* that the operating system identifies life processes by
- There are user processes and system processes

File Edit Tabs Help

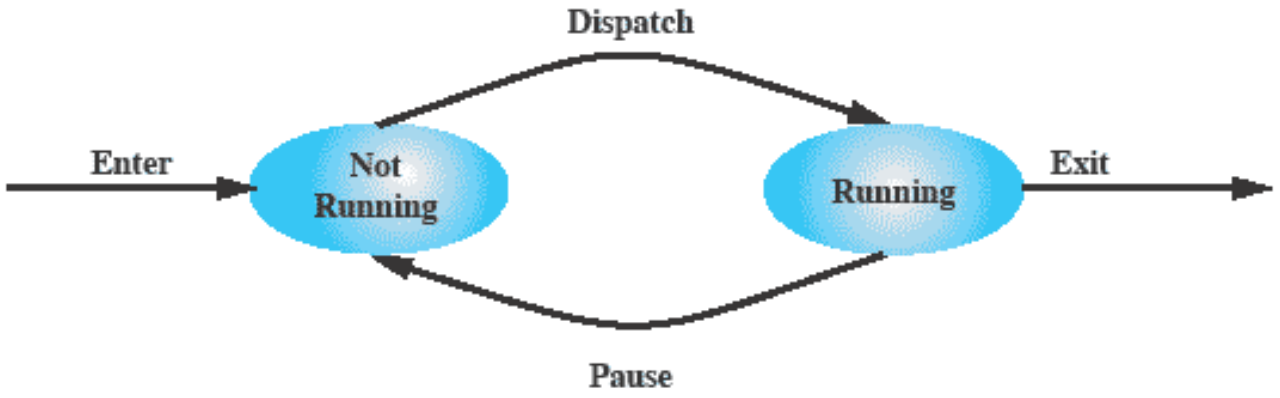
```
top - 07:35:00 up  2:07,  1 user,  load average: 0.18, 0.05, 0.01
Tasks: 160 total,   1 running, 159 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.2 us,  0.2 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  2048316 total,   597732 free,   216968 used,  1233616 buff/cache
KiB Swap:  2096124 total,  2096124 free,        0 used.  1650628 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
  925 root      20   0  281812  75396  28132 S   1.3  3.7   0:36.30 Xorg
 2322 frankeh   20   0  430512  31004  21536 S   1.0  1.5   0:17.76 x-terminal-emul
 1865 frankeh   20   0  116204   2160   1792 S   0.3  0.1   0:17.16 VBoxClient
 2064 frankeh   20   0  379044  19440  15280 S   0.3  0.9   0:01.44 openbox
    1 root      20   0  120004   6148   3968 S   0.0  0.3   0:02.46 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   0:00.12 ksoftirqd/0
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
    7 root      20   0       0      0      0 S   0.0  0.0   0:00.89 rcu_sched
    8 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
    9 root      rt   0       0      0      0 S   0.0  0.0   0:00.07 migration/0
   10 root      rt   0       0      0      0 S   0.0  0.0   0:00.06 watchdog/0
   11 root      rt   0       0      0      0 S   0.0  0.0   0:00.07 watchdog/1
   12 root      rt   0       0      0      0 S   0.0  0.0   0:00.07 migration/1
   13 root      20   0       0      0      0 S   0.0  0.0   0:00.41 ksoftirqd/1
   15 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/1:0H
   16 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kdevtmpfs
   17 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 netns
   18 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 perf
```

Process State Model

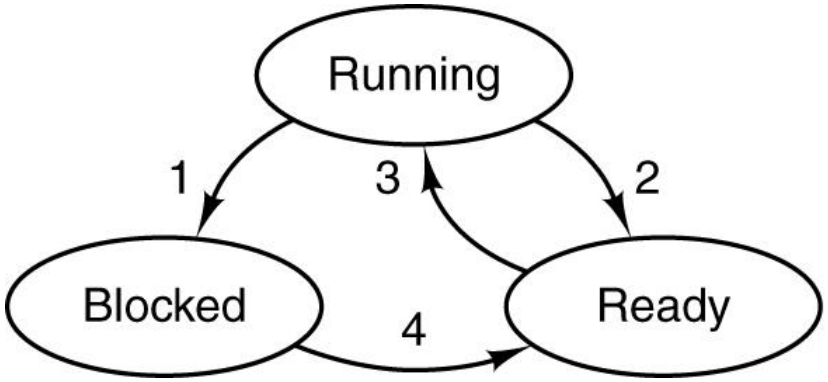
- Depending on the implementation, there can be several possible state models.

- The Simplest one: Two-state diagram

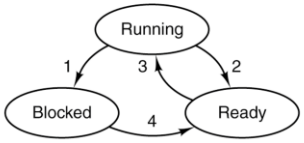


(a) State transition diagram

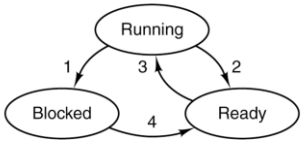
Process State Model: Three-State Model



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

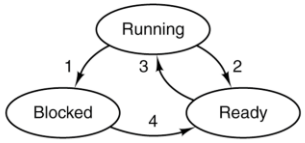


Process_1



Process_2

. . .



Process_N

Scheduler (picks process and drives state transition)

Process State Model: Five-State Model

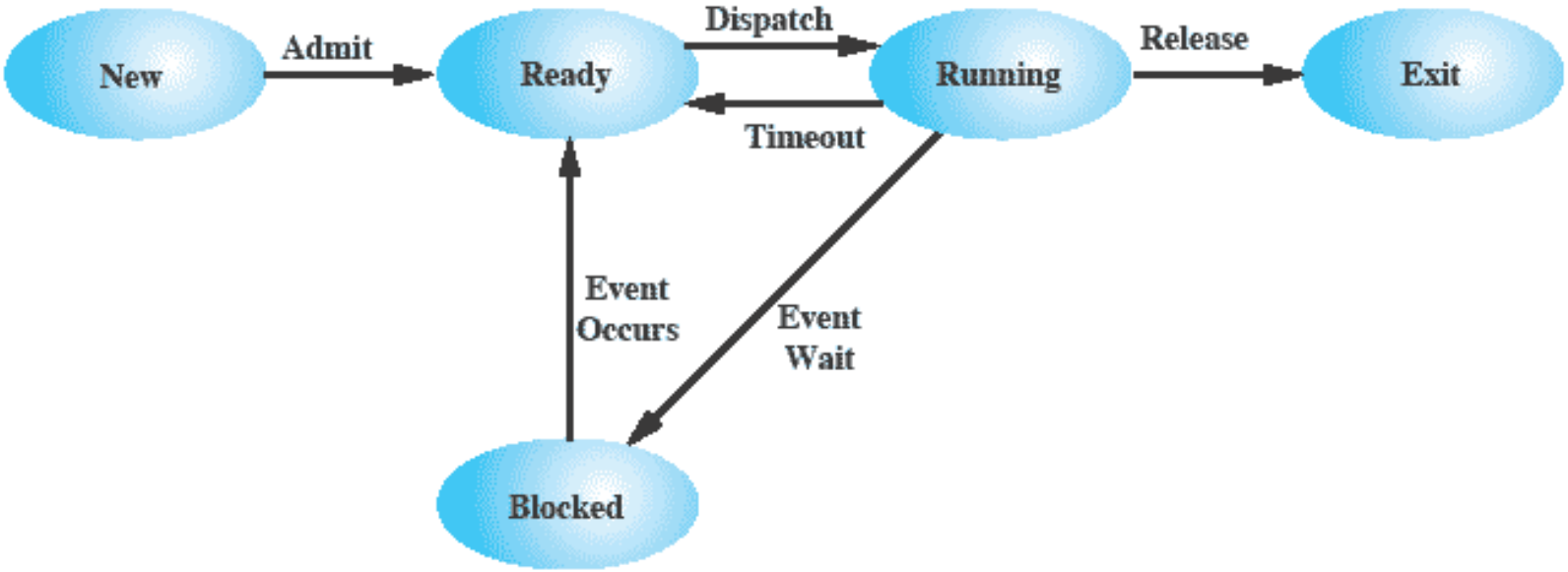
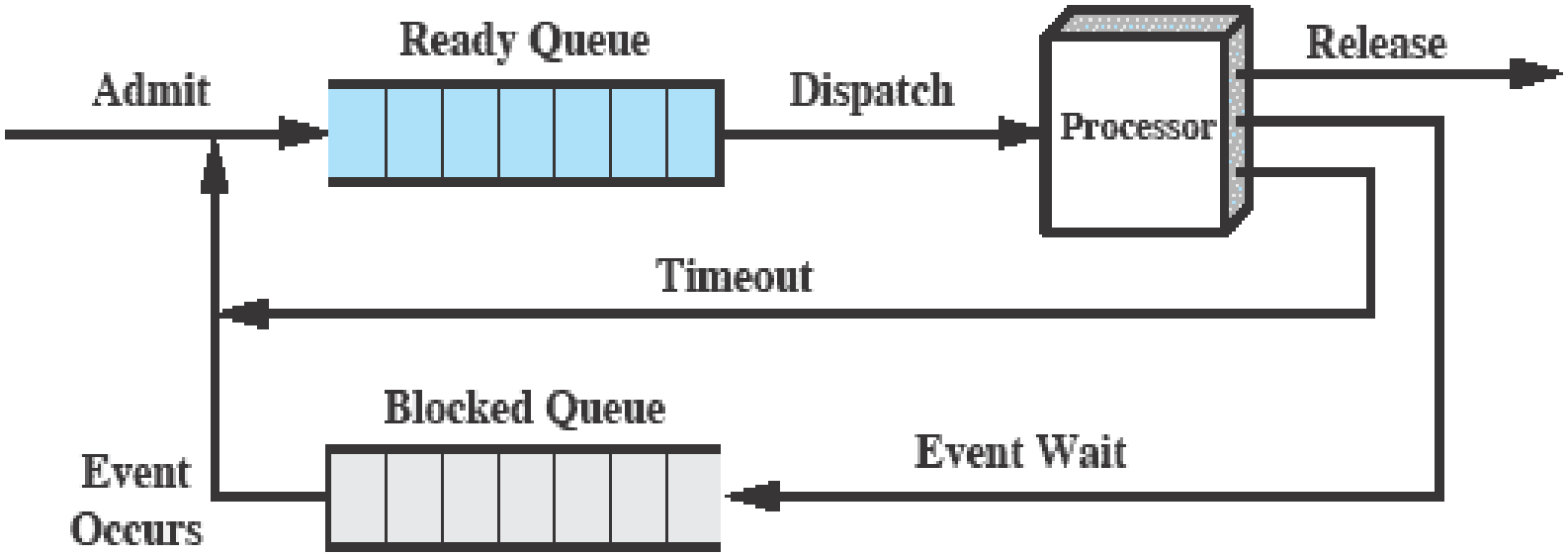


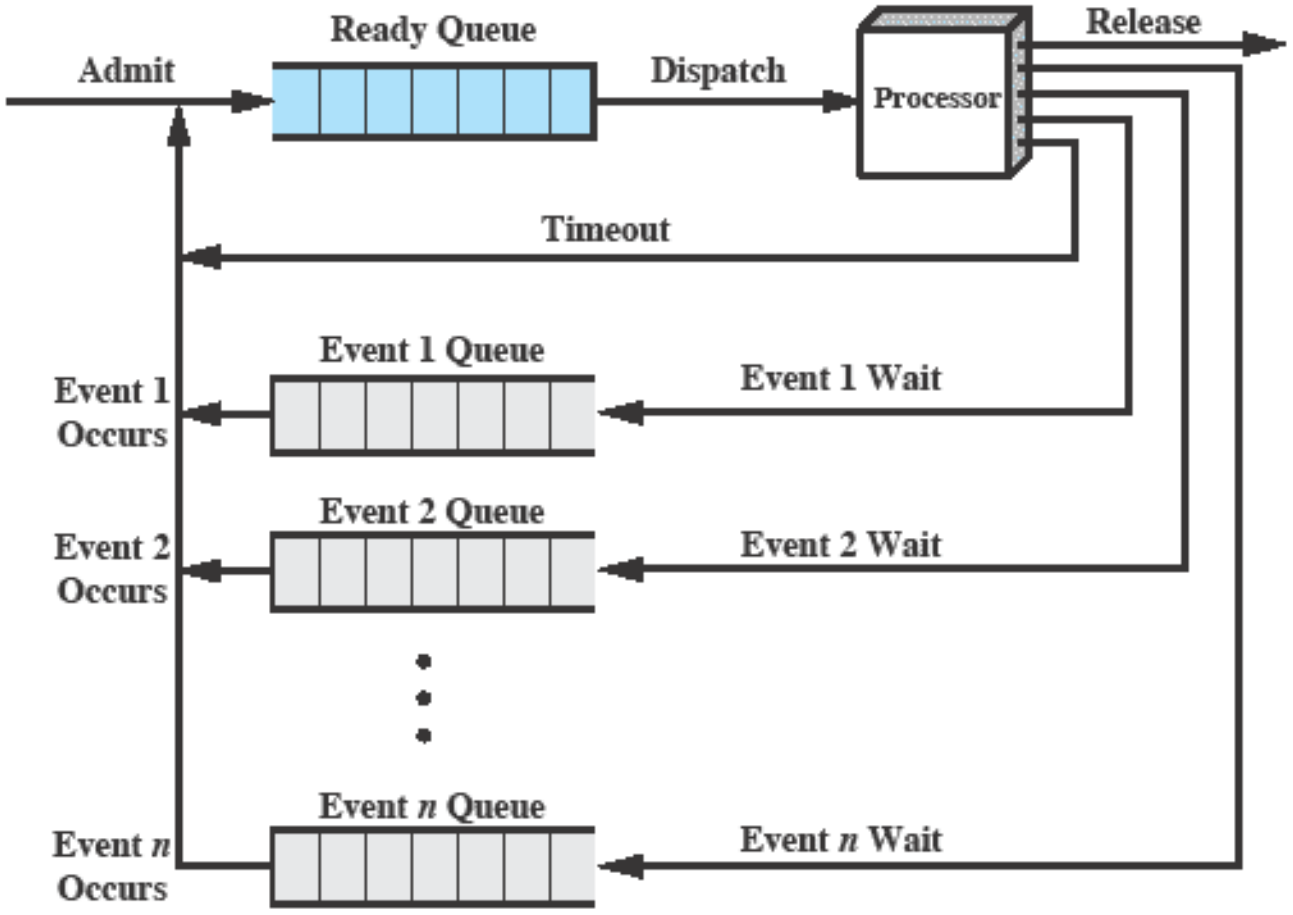
Figure 3.6 Five-State Process Model

Using Queues to Manage Processes



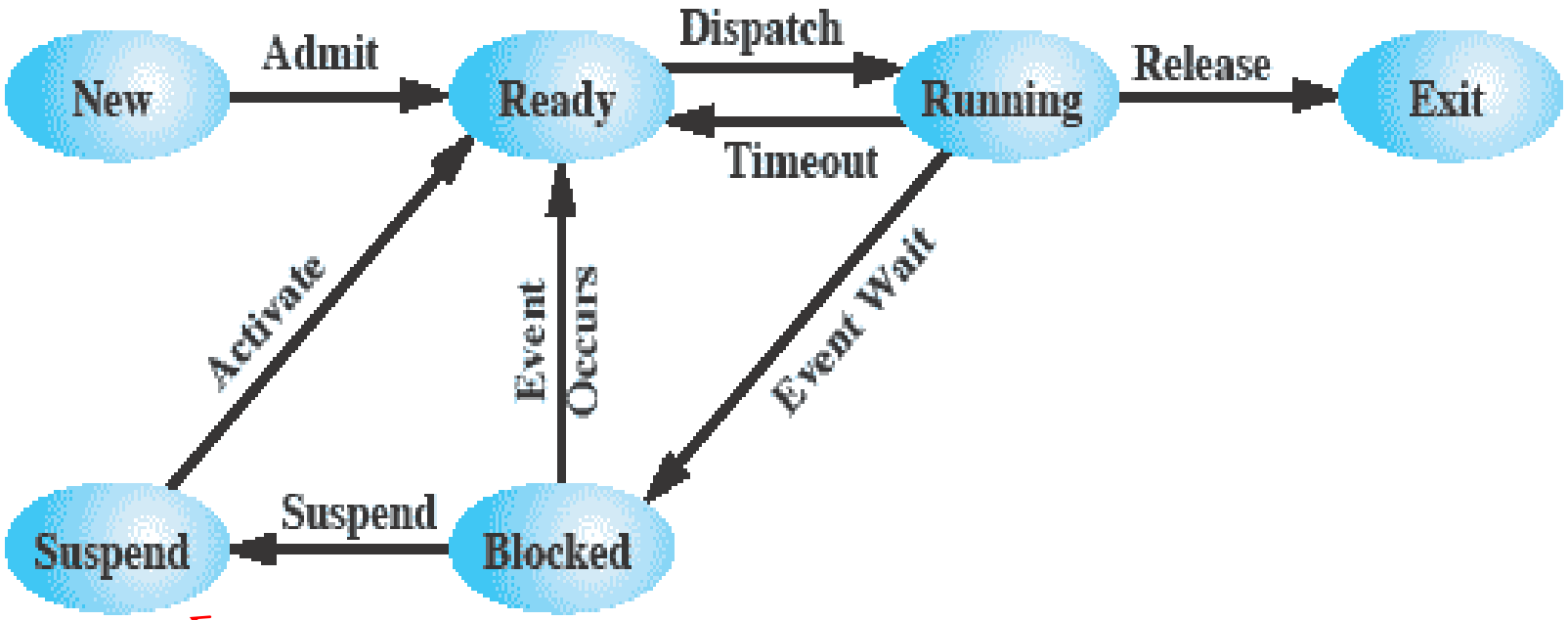
(a) Single blocked queue

Using Queues to Manage Processes



(b) Multiple blocked queues

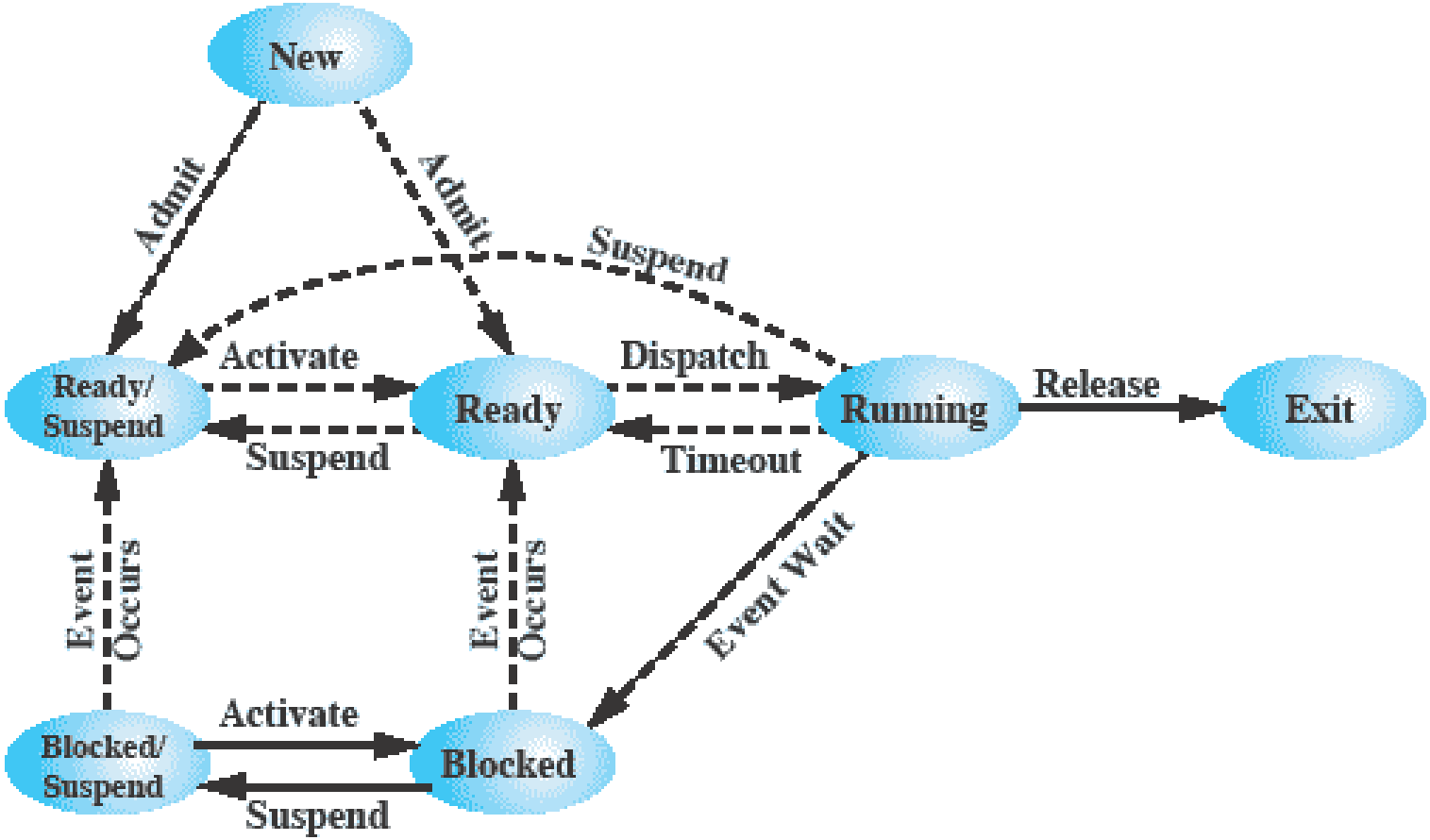
One Extra State!



Swapped to disk

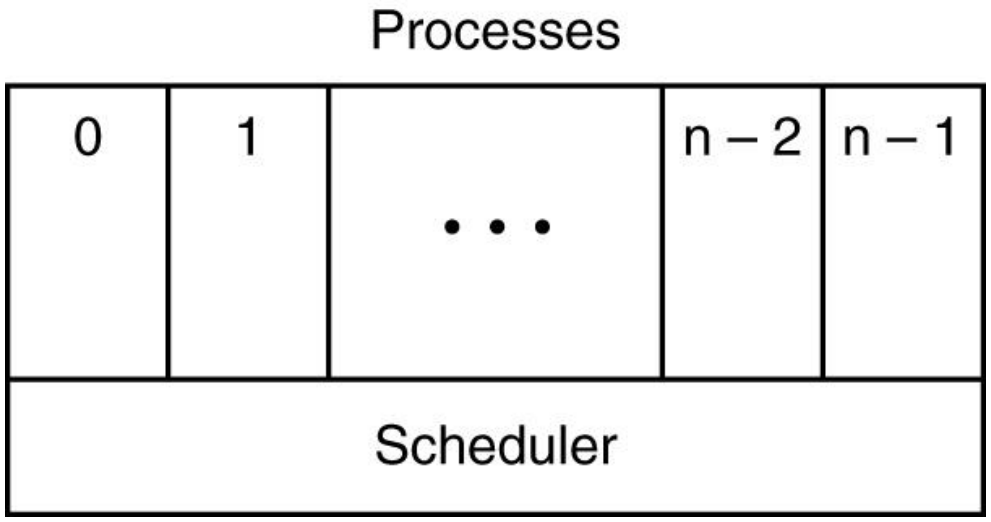
Whole process and its “address space” is moved to disk

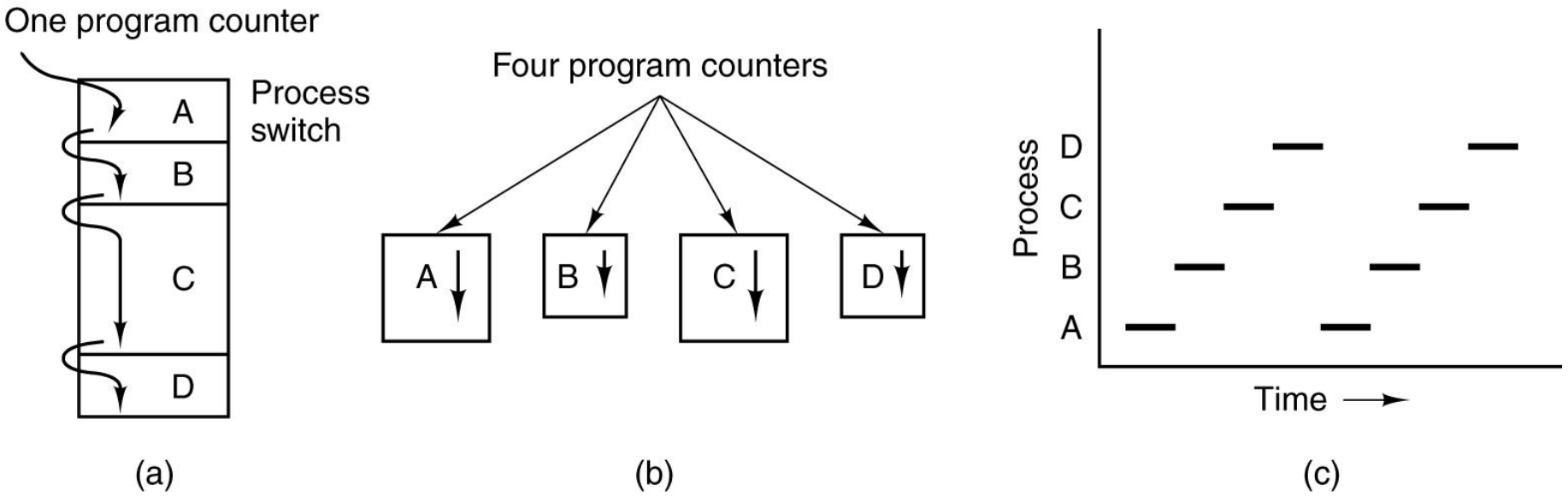
One Extra State!



Multiprogramming

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





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

Running the same program several times will not result in the same execution times due to:

- interrupts
- multi-programming

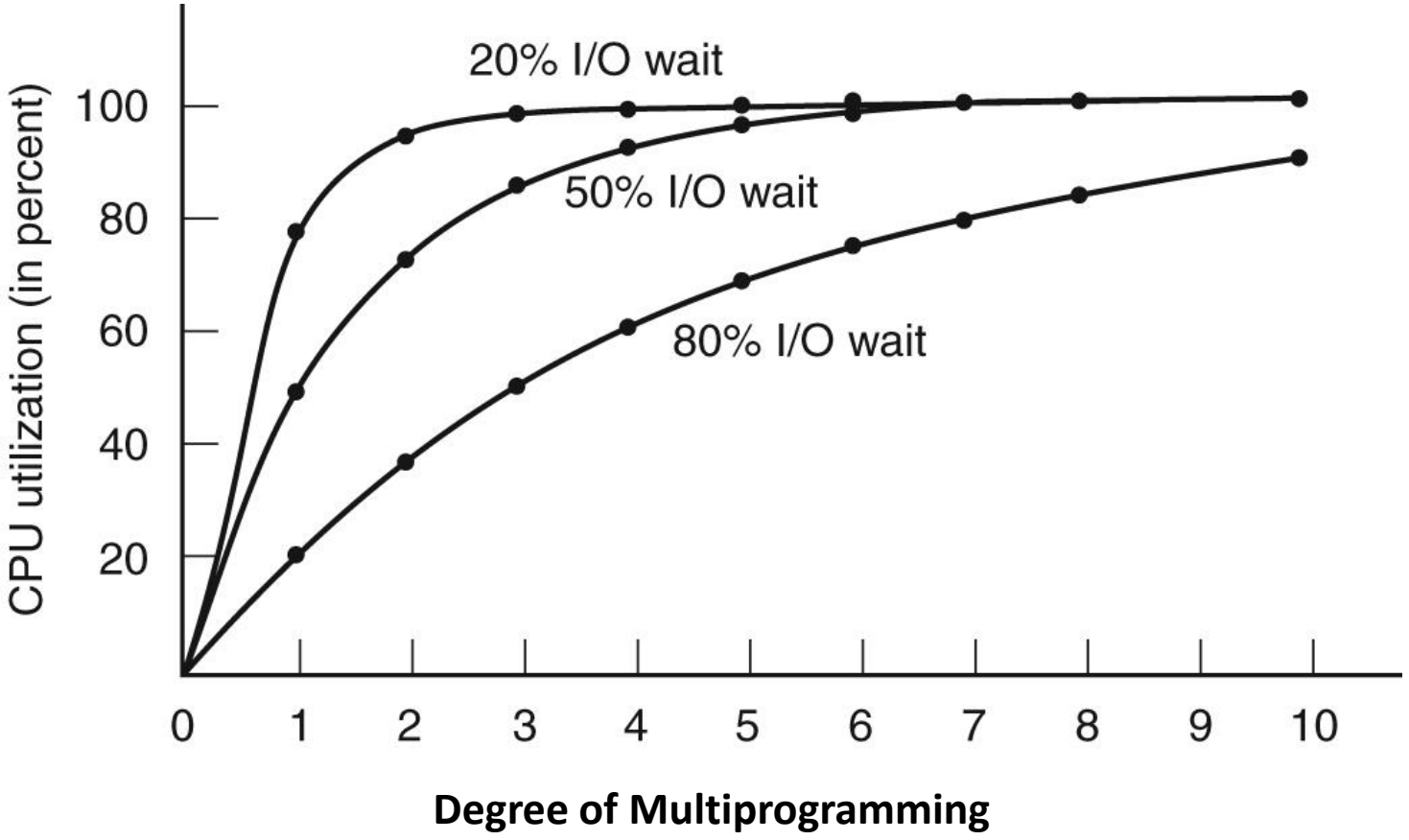
Concurrency vs. Parallelism

- **Concurrency** is when two or more tasks can start, run, and complete in overlapping time **periods**. It doesn't necessarily mean they'll ever both be running **at the same instant**. For example, *multitasking* on a single-core machine.

- **Parallelism** is when tasks *literally* run at the same time, e.g., on a multicore processor.

Simple Modeling of Multiprogramming

- A process spends fraction p waiting for I/O
- Assume n processes 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 do multiprogramming

- Really a question of how to increase concurrency (e.g. multi-core system) and overlap I/O with computation.

- 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
    - Each process has its own address space (code, stack, heap, files, …..)
  - Who owns perceived single resource:
    - E.g. webserver port 80 / 1080 / 8080 ????

Threads

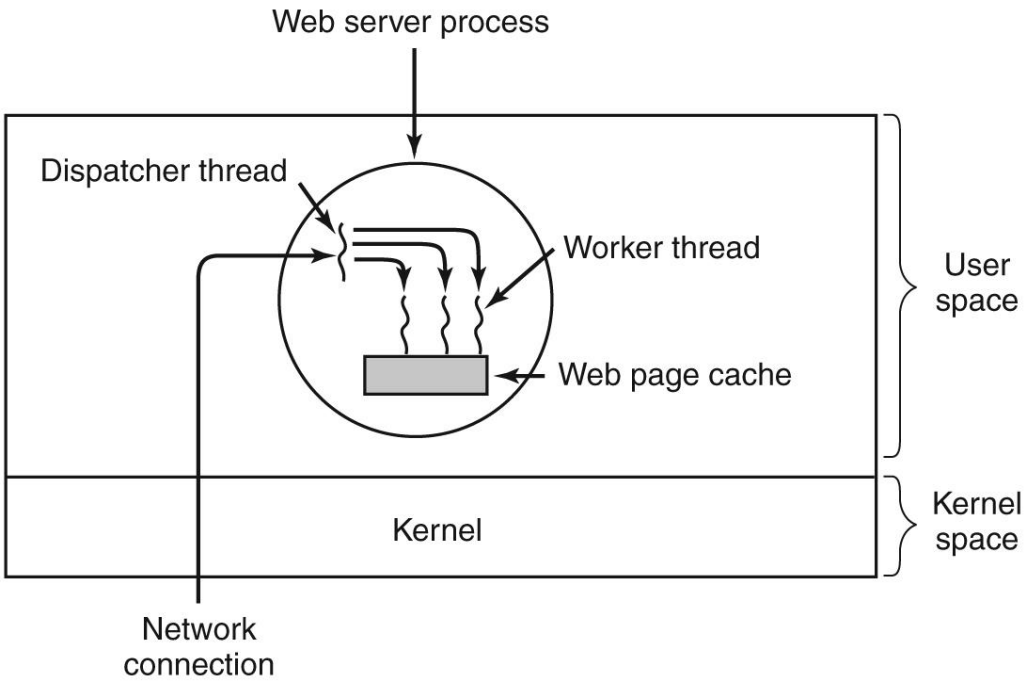
- Multiple threads of control within a process
  - unique execution

- All threads of a process share the same address space and resources (with exception of stack)

Why Threads?

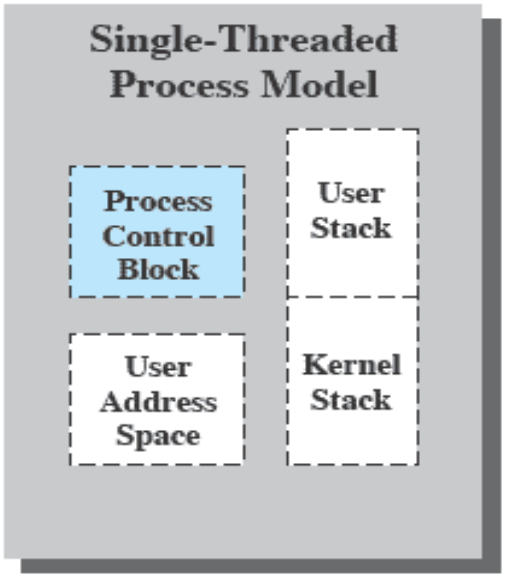
- For some applications many activities can happen at once
  - With threads, programming becomes easier
    - Otherwise application needs to actively manage different logical executions in the process
    - This requires significant state management
  - Benefit applications with I/O and processing that can overlap
- Lighter weight than processes
  - Faster to create and restore
    (we just really need a stack and an execution unit, but don't have create new address space etc.)

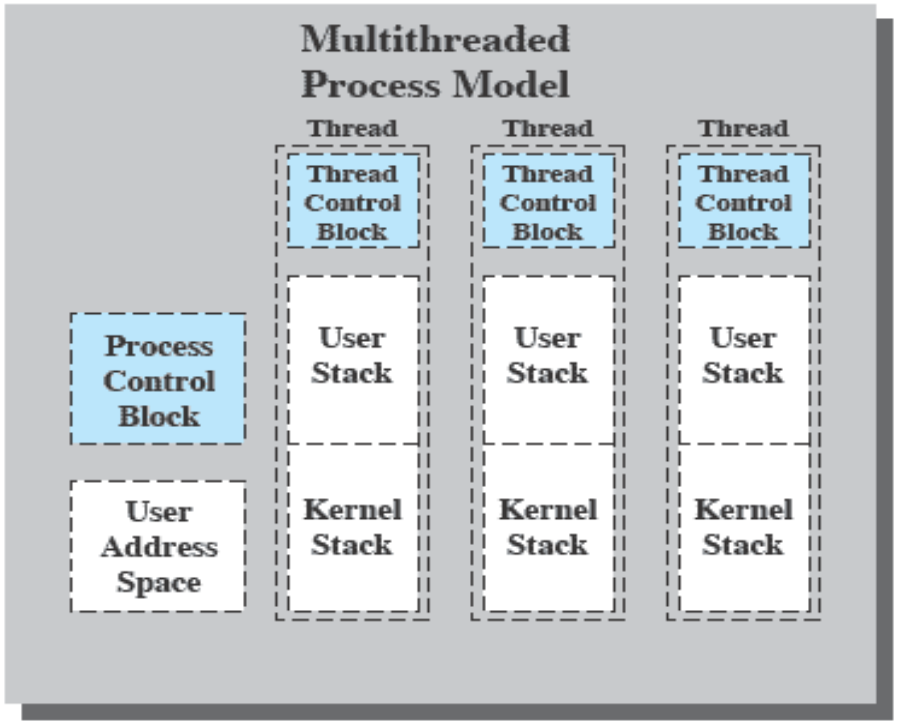
Example : Multithreaded Web Server



Processes vs. Threads

OS internal Data Structure implications:





Processes vs Threads

- Process groups resources
  - (Address Space, files)
- Threads are entities scheduled for execution on CPU
- Threads can be in any of several states: *running, blocked, ready*, and *terminated*
  (remember the process state model ?)
- No protections among threads (unlike processes) [Why?] → this is important

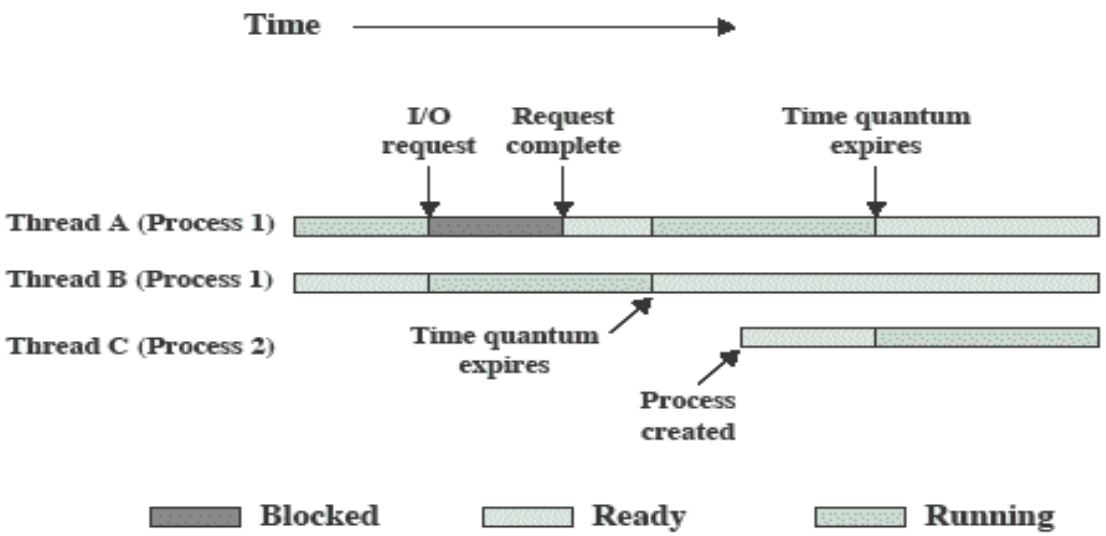
Processes vs Threads

- The unit of dispatching is referred to as a ***thread*** or ***lightweight process (lwp)***
- The unit of resource ownership is referred to as a ***process*** or ***task***
  (unfortunately in linux struct task represents both a process and thread)
- ***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 (one) 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)

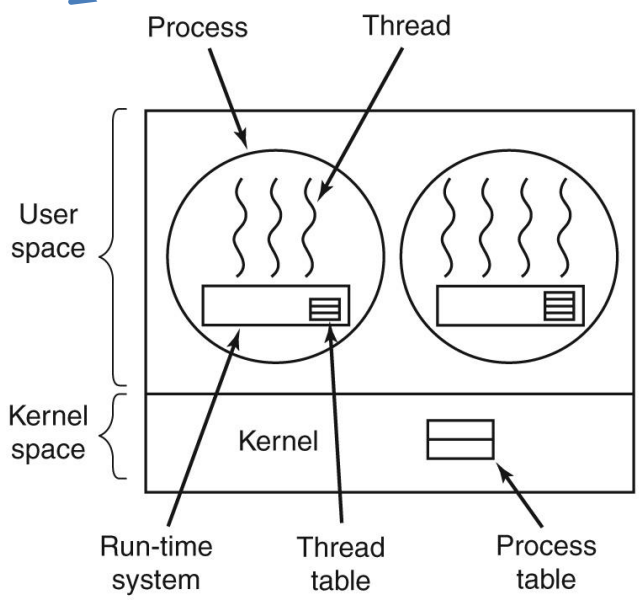
Multithreading on Uniprocessor System

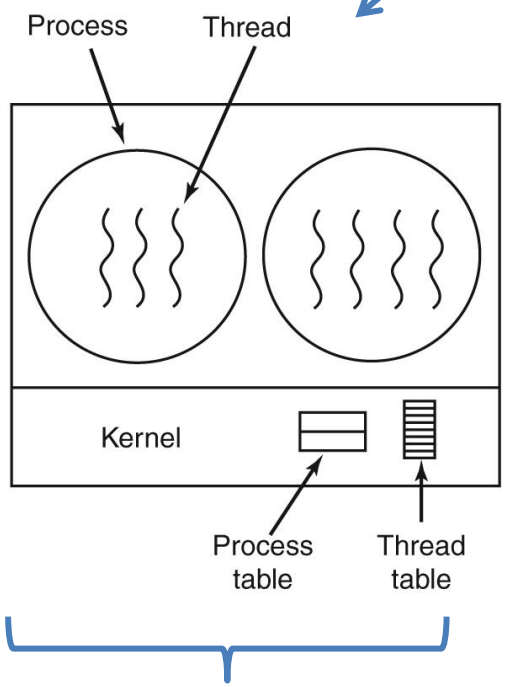


Where to Put The Thread Implementation / Package?

User space

Kernel space

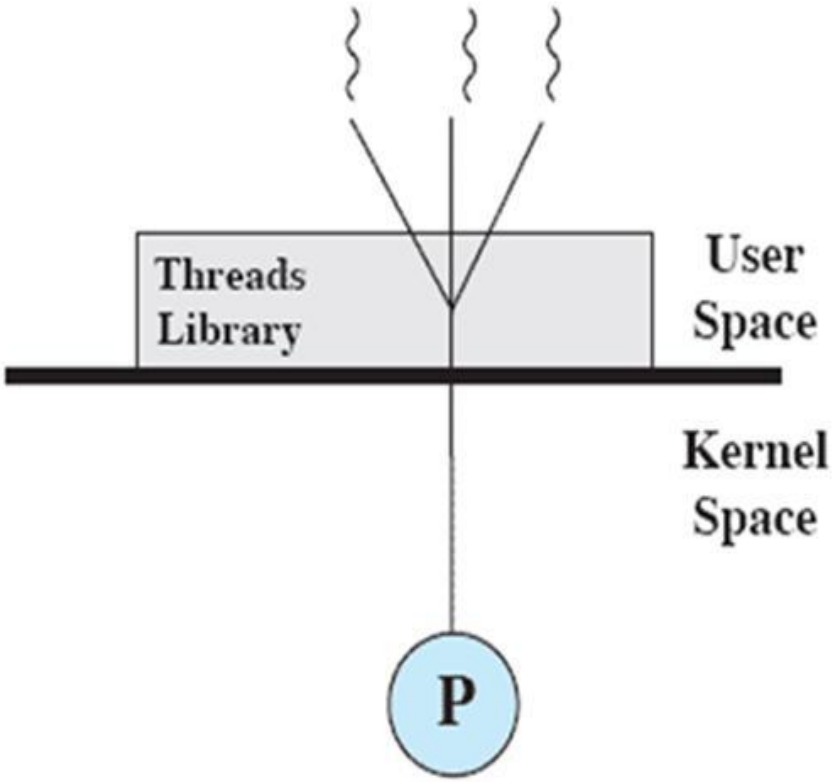




Discussed in previous slides

User-Level Threads (ULT)

- All thread management is done by the application
- Initially developed to run on kernels that are not multithreading capable.
- The kernel is not aware of the existence of threads

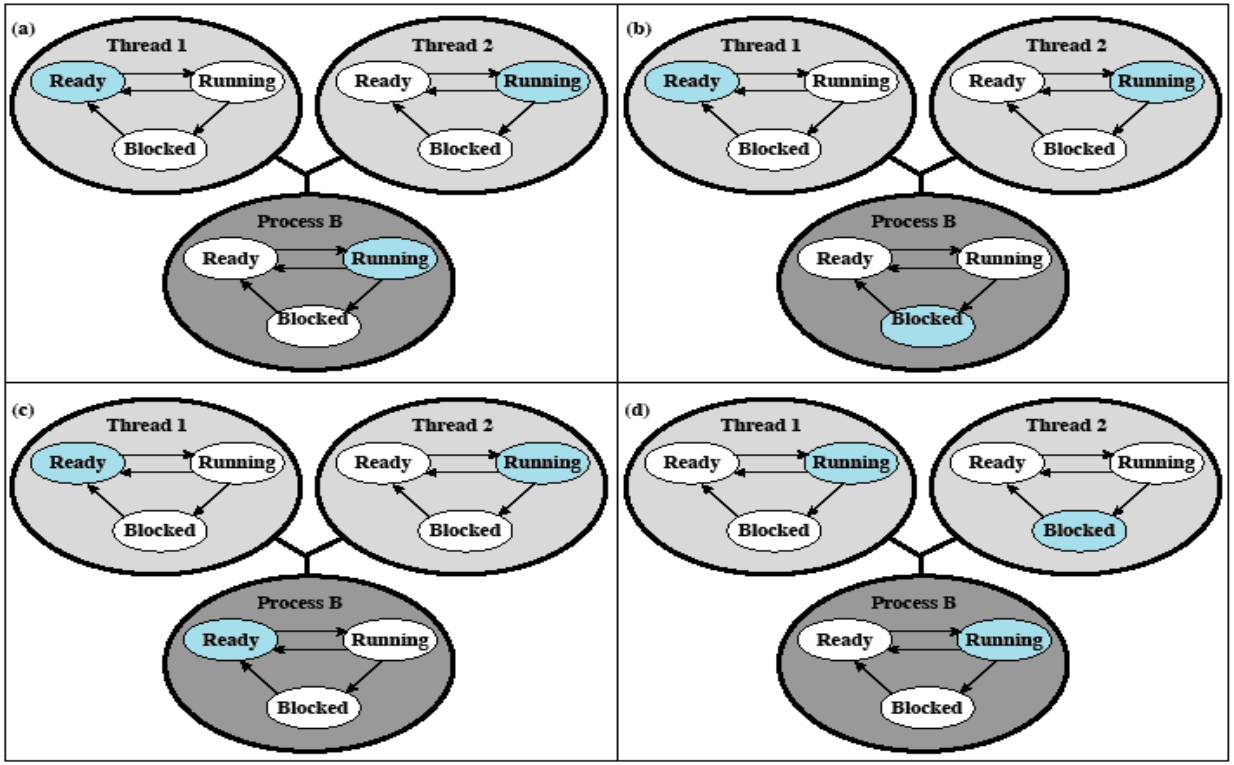


Implementing Threads in User Space

- Threads are implemented by a library
- Kernel knows nothing about threads, only processes and there is one "execution" associated with that process.
- Only one thread of the process can technically be executing at a given time.
- Each process needs its own private thread table in userspace
- Thread table is managed by the runtime system

User-Level Threads (ULTs)

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



Colored state
is current state

User-Level Threads (ULTs)

Advantages

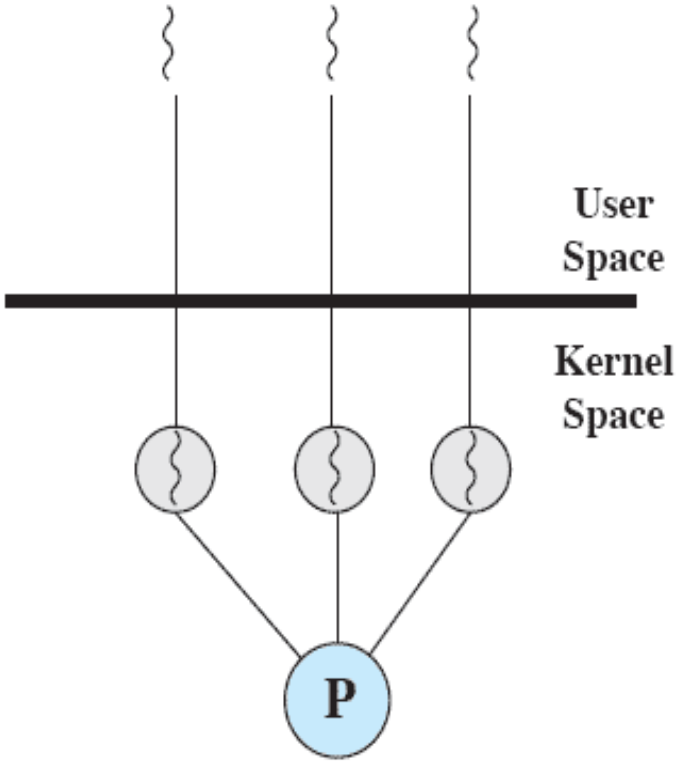
- Thread switch does not require kernel-mode (still similar expense)
- Scheduling (of threads) can be application specific.
- Can run on any OS.
- Scales better.

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



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

- In linux a `task_struct` and stacks are allocated and linked to the current process and its address space

Kernel-Level Threads (KLTs)

Advantages

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

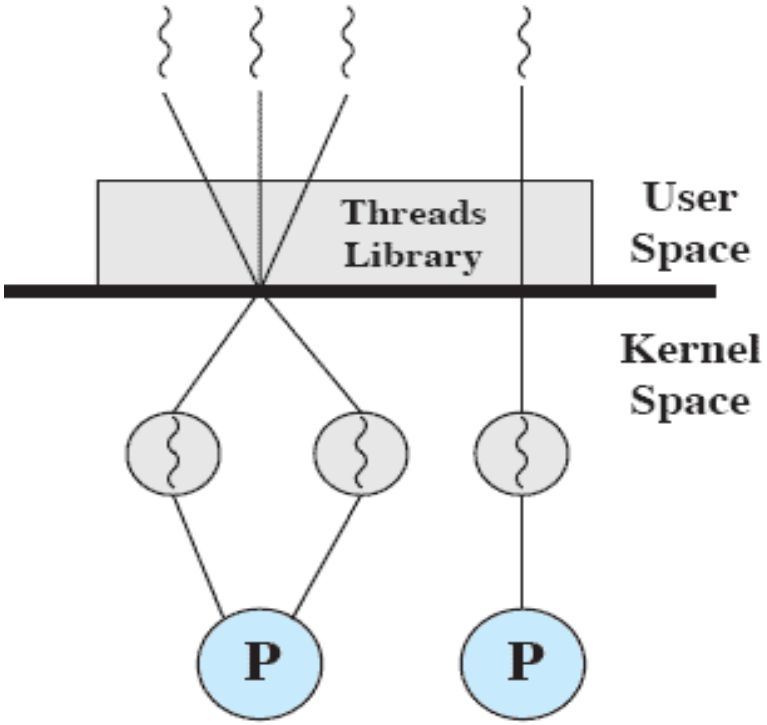
Disadvantages

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

But note, if you want to implement interruption for user level thread scheduling, you have to issue and deliver a "signal(SIGVTALARM)" which is much more expensive.

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



Different Naming Conventions

- Thread Models are also referred to by their 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

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
```

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 for the user level

b) Calls clone() to create a new schedulable thread inside current process
(in linux it's a `struct task` obj)

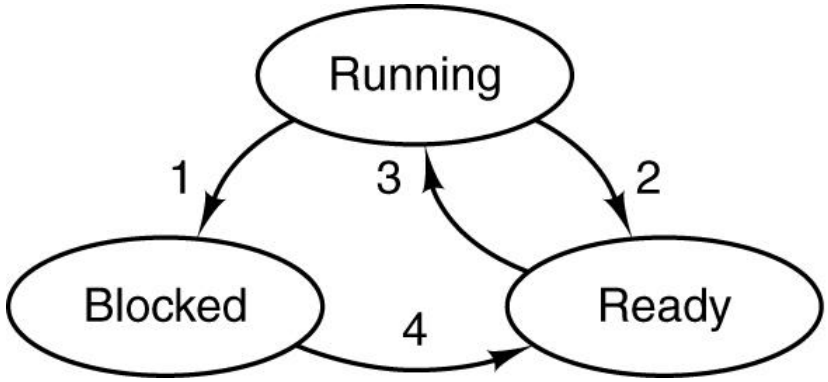
c) Sets the thread's stack pointer to (a)

d) Sets the thread's instruction pointer to `(*start_routine)(arg)`

e) makes thread scheduable

Thread State Model:

- What changes to the state model in a kernel based thread model (1:1 or M:N) ?
- 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"

Threads + fork()

- Linux threads are not replicated with a fork() only forking thread remains

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

void*  thfunc(void* arg)
{
    sleep(2);
    while (1) {
        printf("%4d %lx\n",getpid(), pthread_self());
        sleep(1);
    }
    return NULL;
}

int main()
{
    pthread_t  th1, th2;
    pthread_create(&th1,NULL,thfunc,NULL);
    pthread_create(&th2,NULL,thfunc,NULL);
    sleep(1);
    int rc = fork();
    printf("%4d %lx %d\n",getpid(),pthread_self(), rc);
    sleep(10000);
    return 0;
}
```

```
1835 7f690ae14740 1838
1838 7f690ae14740 0
1835 7f690ae13700
1835 7f690a612700
1835 7f690ae13700
1835 7f690a612700
1835 7f690ae13700
1835 7f690a612700
1835 7f690ae13700
1835 7f690a612700
```

$ ps -edfmT

```
frankeh     1835     -    1798  0 09:06 pts/0    00:00:00 ./mthread
frankeh        -  1835       -  0 09:06 -        00:00:00 -
frankeh        -  1836       -  0 09:06 -        00:00:00 -
frankeh        -  1837       -  0 09:06 -        00:00:00 -
frankeh     1838     -    1835  0 09:06 pts/0    00:00:00 ./mthread
frankeh        -  1838       -  0 09:06 -        00:00:00 -
frankeh     1839     -    1824  0 09:06 pts/1    00:00:00 ps -edfmT
```

Threads + fork()

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

void*  thfunc(void* arg)
{
    int rc = fork();
    printf("%4d rc=%4d pts=%lx\n",getpid(),rc, pthread_self());
    sleep(2);
    while (1) {
        printf("%4d %lx\n",getpid(), pthread_self());
        sleep(1);
    }
    return NULL;
}

int main()
{
    pthread_t  th1, th2;
    printf("main %4d %lx\n",getpid(), pthread_self());
    pthread_create(&th1,NULL,thfunc,(void*)0);
    pthread_create(&th2,NULL,thfunc,(void*)1);
    sleep(10000);
    return 0;
}
```

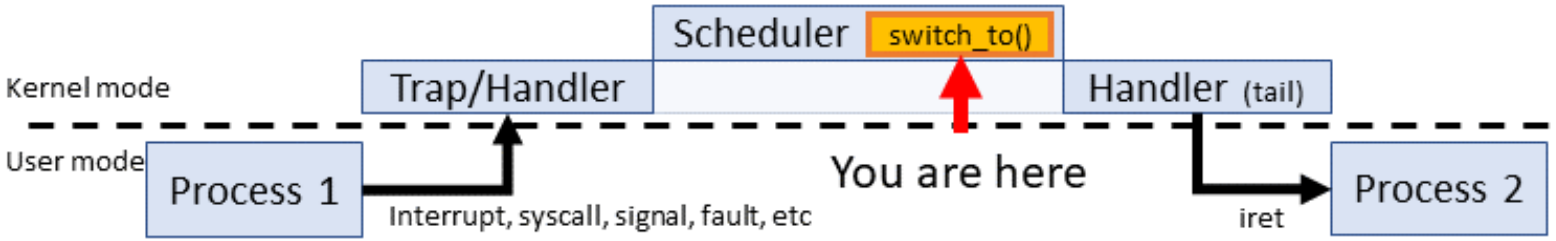
```
main 3481 7fe413831740
3481 rc=3484 pts=7fe413830700
3484 rc=   0 pts=7fe413830700
3481 rc=3485 pts=7fe41302f700
3485 rc=   0 pts=7fe41302f700
3484 7fe413830700
3485 7fe41302f700
3481 7fe413830700
3481 7fe41302f700
3484 7fe413830700
3481 7fe41302f700
3485 7fe41302f700
3481 7fe413830700
3484 7fe413830700
3485 7fe41302f700
3481 7fe413830700
3481 7fe41302f700
```

```
frankeh     3481       -    1972  0 14:39 pts/1    00:00:00 ./mthread1
frankeh        -    3481       -  0 14:39 -        00:00:00 -
frankeh        -    3482       -  0 14:39 -        00:00:00 -
frankeh        -    3483       -  0 14:39 -        00:00:00 -
frankeh     3484       -    3481  0 14:39 pts/1    00:00:00 ./mthread1
frankeh        -    3484       -  0 14:39 -        00:00:00 -
frankeh     3485       -    3481  0 14:39 pts/1    00:00:00 ./mthread1
frankeh        -    3485       -  0 14:39 -        00:00:00 -
```

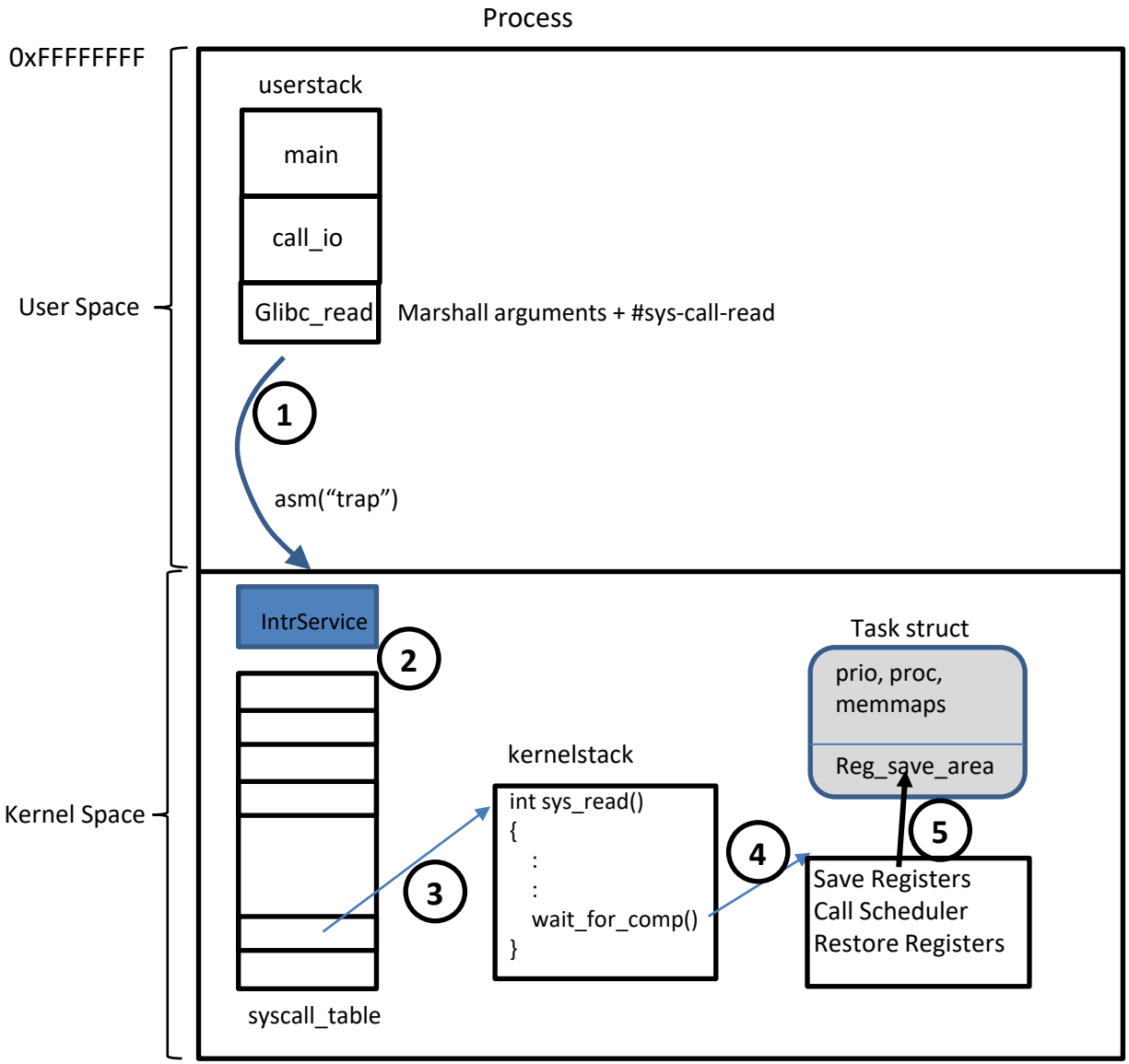
Context Switch

- Scenarios:
  1) Current process (or thread) blocks OR
  2) Preemption via interrupt

- Operation(s) to be done
  - Must release CPU resources (registers)
  - Requires storing "all" non privileged registers to the PCB or TCB save area
  - Tricky as you need registers to do this
  - Typically an architecture has a few privileged registers so the kernel can accomplish this
  - All written in assembler



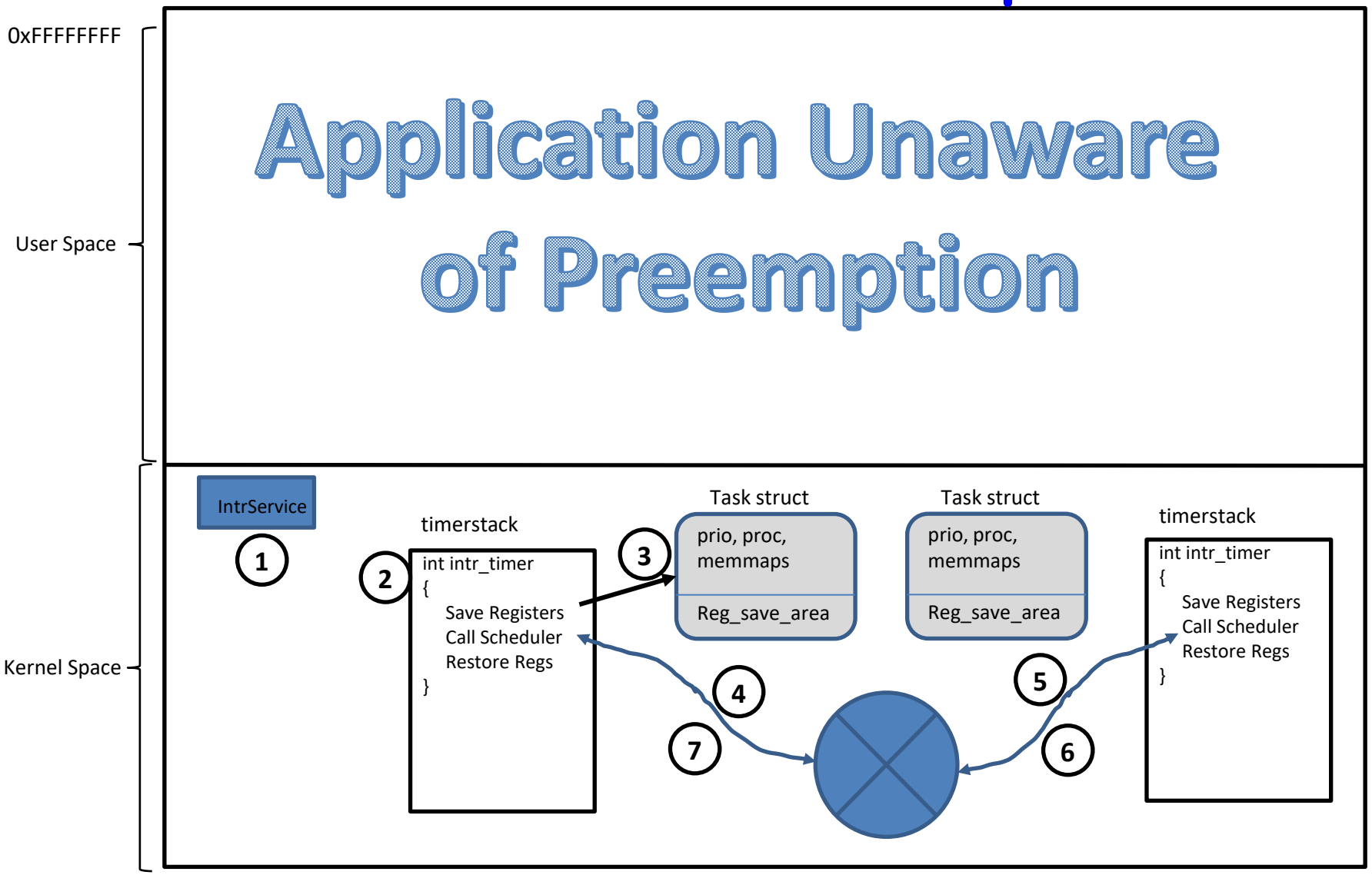
CtxSwitch: Process Blocks




```
int call_io()
{
   char buffer[128];
   return read(0,buffer,128);
}

main() {
   call_io();
}
```

CtxSwitch: Preemption



Examples of low-level context switch routine for x86

Details not important, appreciate the complexity of the ASM code

```
/** include/asm-x86_64/system.h */

#define SAVE_CONTEXT    "pushfq ; pushq %%rbp ; movq %%rsi,%%rbp\n\t"
#define RESTORE_CONTEXT "movq %%rbp,%%rsi ; popq %%rbp ; popfq\n\t"
#define __EXTRA_CLOBBER
    ,"rcx","rbx","rdx","r8","r9","r10","r11","r12","r13","r14","r15"

#define switch_to(prev,next,last)
    asm volatile(SAVE_CONTEXT
         "movq %%rsp,%P[threadrsp](%[prev])\n\t" /* save RSP */
         "movq %P[threadrsp](%[next]),%%rsp\n\t" /* restore RSP */
         "call __switch_to\n\t"
         ".globl thread_return\n"
         "thread_return:\n\t"
         "movq %%gs:%P[pda_pcurrent],%%rsi\n\t"
         "movq %P[thread_info](%%rsi),%%r8\n\t"
         "btr  %[tif_fork],%P[ti_flags](%%r8)\n\t"
         "movq %%rax,%%rdi\n\t"
         "jc   ret_from_fork\n\t"
         RESTORE_CONTEXT
         : "=a" (last)
         : [next] "S" (next), [prev] "D" (prev),
           [threadrsp] "i" (offsetof(struct task_struct, thread.rsp)),
           [ti_flags] "i" (offsetof(struct thread_info, flags)),
           [tif_fork] "i" (TIF_FORK),
           [thread_info] "i" (offsetof(struct task_struct, thread_info)),
           [pda_pcurrent] "i" (offsetof(struct x8664_pda, pcurrent))
         : "memory", "cc" __EXTRA_CLOBBER)
```

```
/** arch/x86_64/kernel/process.c */

struct task_struct *__switch_to(struct task_struct *prev_p, struct task_struct *next_p)
{
    struct thread_struct *prev = &prev_p->thread,
                 *next = &next_p->thread;
    int cpu = smp_processor_id();
    struct tss_struct *tss = init_tss + cpu;

    unlazy_fpu(prev_p);

    tss->rsp0 = next->rsp0;

    asm volatile("movl %%es,%0" : "=m" (prev->es));
    if (unlikely(next->es | prev->es))
        loadsegment(es, next->es);

    asm volatile ("movl %%ds,%0" : "=m" (prev->ds));
    if (unlikely(next->ds | prev->ds))
        loadsegment(ds, next->ds);

    load_TLS(next, cpu);

    /* Switch FS and GS. */
    {
        unsigned fsindex;
        asm volatile("movl %%fs,%0" : "=g" (fsindex));

        if (unlikely(fsindex | next->fsindex | prev->fs)) {
            loadsegment(fs, next->fsindex);
            if (fsindex)
                prev->fs = 0;
        }
        /* when next process has a 64bit base use it */
        if (next->fs)
            wrmsrl(MSR_FS_BASE, next->fs);
        prev->fsindex = fsindex;
    }
    {
        unsigned gsindex;
        asm volatile("movl %%gs,%0" : "=g" (gsindex));
        if (unlikely(gsindex | next->gsindex | prev->gs)) {
            load_gs_index(next->gsindex);
            if (gsindex)
                prev->gs = 0;
        }
        if (next->gs)
            wrmsrl(MSR_KERNEL_GS_BASE, next->gs);
        prev->gsindex = gsindex;
    }

    /* Switch the PDA context. */
    prev->userrsp = read_pda(oldrsp);
    write_pda(oldrsp, next->userrsp);
    write_pda(pcurrent, next_p);
    write_pda(kernelstack, (unsigned long)next_p->thread_info + THREAD_SIZE - PDA_STACKOFFSET);

    /* Now maybe reload the debug registers */
    if (unlikely(next->debugreg7)) {
        loaddebug(next, 0);
        loaddebug(next, 1);
        loaddebug(next, 2);
        loaddebug(next, 3);
        /* no 4 and 5 */
        loaddebug(next, 6);
        loaddebug(next, 7);
    }

    /* Handle the IO bitmap */
    if (unlikely(prev->io_bitmap_ptr || next->io_bitmap_ptr)) {
        if (next->io_bitmap_ptr) {
            memcpy(tss->io_bitmap, next->io_bitmap_ptr, IO_BITMAP_BYTES);
            tss->io_bitmap_base = IO_BITMAP_OFFSET;
        } else {
            tss->io_bitmap_base = INVALID_IO_BITMAP_OFFSET;
        }
    }

    return prev_p;
}
```

Task switching in ARM

```
extern struct task_struct *__switch_to(struct task_struct *, struct thread_info *, struct thread_info *);

#define switch_to(prev,next,last)                                       \
do {                                                                    \
        __complete_pending_tlbi();                                      \
        if (IS_ENABLED(CONFIG_CURRENT_POINTER_IN_TPIDRURO))             \
                __this_cpu_write(__entry_task, next);                   \
        last = __switch_to(prev,task_thread_info(prev), task_thread_info(next));        \
} while (0)

#endif /* __ASM_ARM_SWITCH_TO_H */
```

```
ENTRY(__switch_to)
        .fnstart
        .cantunwind
        add     ip, r1, #TI_CPU_SAVE
        stmia   ip!, {r4 - r11}         @ Store most regs on stack
        str     sp, [ip], #4
        str     lr, [ip], #4
        mov     r5, r0
        add     r4, r2, #TI_CPU_SAVE
        ldr     r0, =thread_notify_head
        mov     r1, #THREAD_NOTIFY_SWITCH
        bl      atomic_notifier_call_chain
        mov     ip, r4
        mov     r0, r5
        ldmia   ip!, {r4 - r11}         @ Load all regs saved previously
        ldr     sp, [ip]
        ldr     pc, [ip, #4]!
        .fnend
ENDPROC(__switch_to)
```

Conclusions

- Process/Threads are one the most central concepts in Operating Systems
- 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