Operating Systems 2024/2025

T Class 03 – Processes and Threads

Vasco Pereira (vasco@dei.uc.pt)
Dep. Eng. Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra

operating system
noun
the collection of software that directs a computer's operations, controlling and scheduling the execution of other programs, and managing storage, input/output, and communication resources.

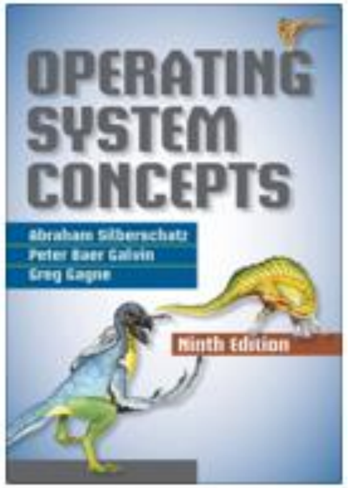
Abbreviation: OS

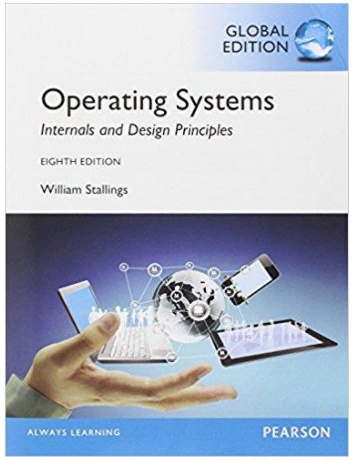
Source: Dictionary.com

Updated on
20 February 2025

Disclaimer

- This slides and notes are based on the companion material [Silberschatz13]. The original material can be found at:
  - http://codex.cs.yale.edu/avi/os-book/OS9/slide-dir/
- In some cases, material from [Stallings15] may also be used. The original material can be found at:
  - http://williamstallings.com/OS/OS5e.html
  - http://williamstallings.com/OperatingSystems/
- The respective copyrights belong to their owners.





Note: Some slides are also based on previous versions from Bruno Cabral, Paulo Marques and Luis Silva (Operating Systems classes of DEI-FCTUC).

Processes and Threads

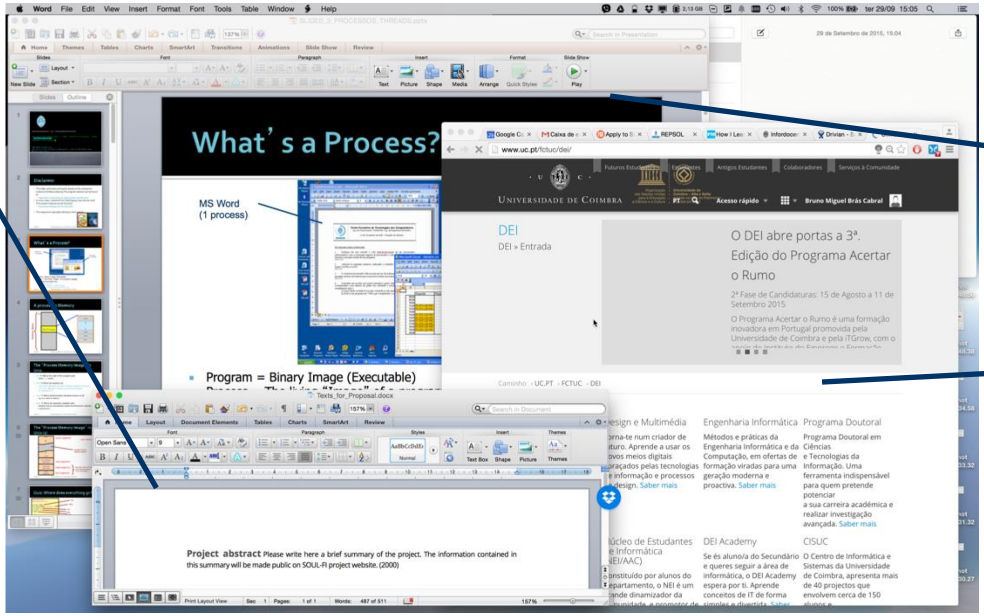
Outlines

- Processes
  - What is a process
  - Process status
  - Process creation/termination
- Threads
  - What is a thread
  - Multithreading

PROCESSES

What is a Process?

MS Word
(1 process)

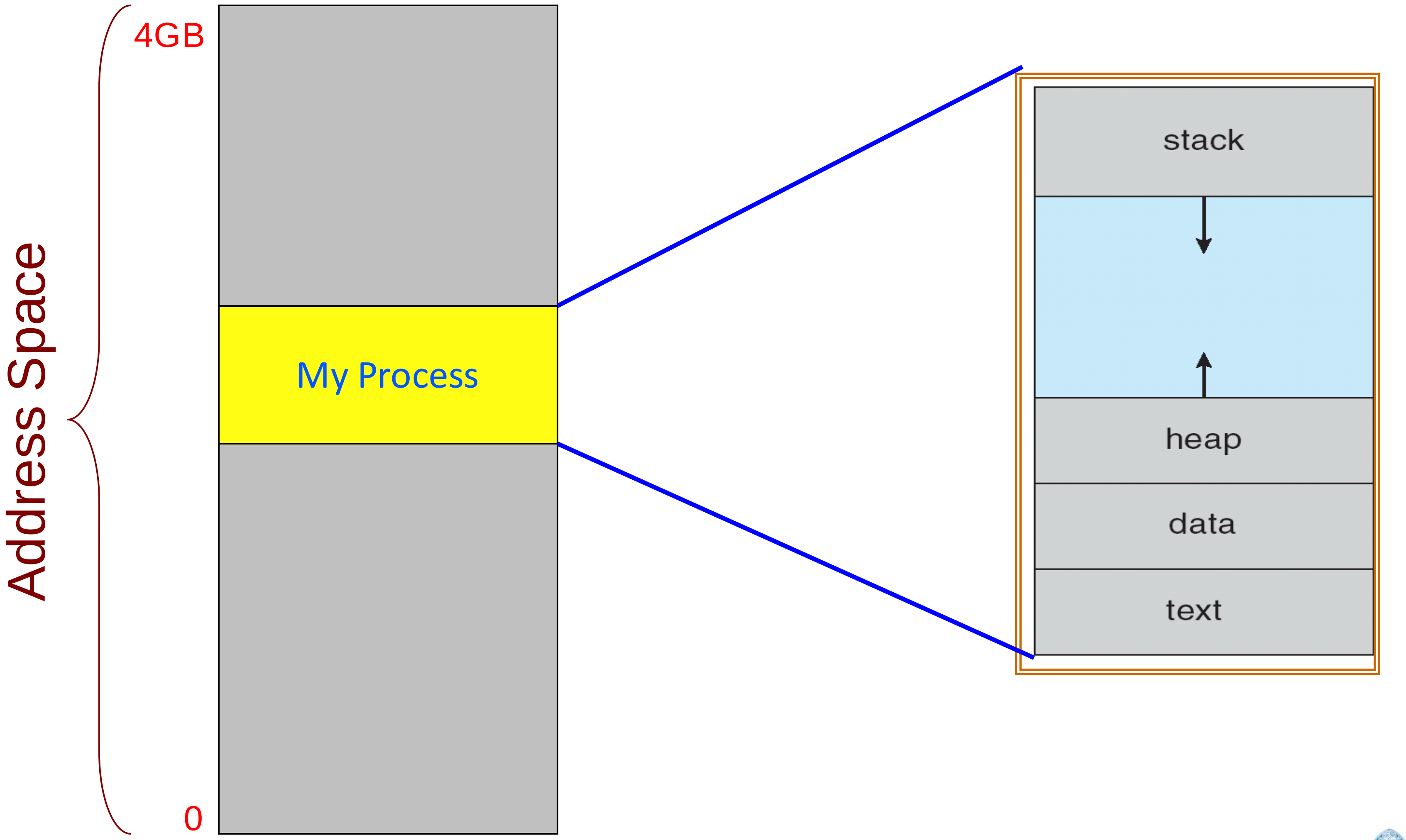


MS Powerpoint
(1 process)

Chrome
(1 process)

- Program = Binary Image (Executable) (passive entity)
- Process = The living "image" of a program running (active entity)
  - It includes information such as :
    - A unique identifier; Program Counter; Owner; Security attributes;
    - I/O status information (e.g., open files)
    - Allocated Memory / Address Space

A process in Memory



The "Process Memory Image" in Unix

- text → Where the code of the program goes
  - Consists of the machine instructions that CPU executes
  - Usually shareable (so that only a copy needs to be in memory, for frequently executed programs)
  - Often read-only, to prevent modifications to instructions
  - Called .text segment

- data → Where all variables are
  - .data → global and static variables, initialized to non-zero
    - This is the initialized data segment
    - Contains global and static variables specifically initialized by the program
  - .bss → global and static variables, non-initialized or initialized to 0
    - .bss aka Uninitialized data segment
    - Data in this segment is initialized by the kernel, to 0 or null before the program starts executing

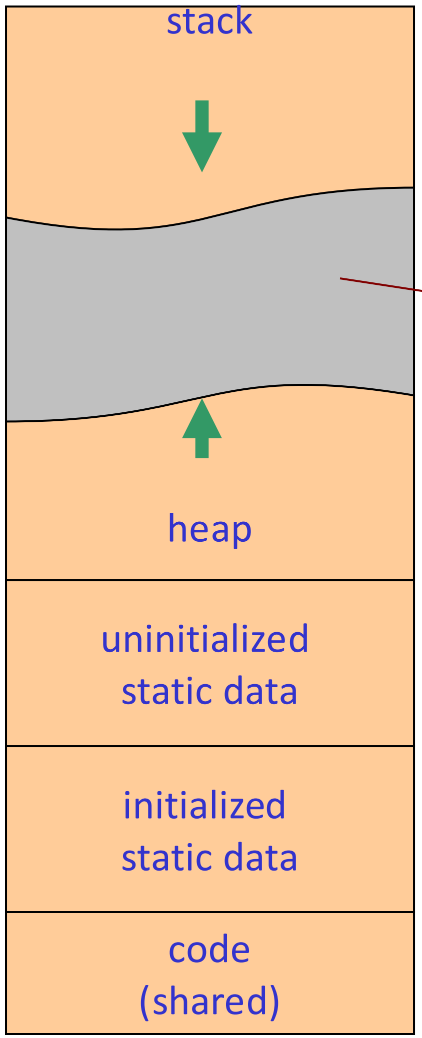
The “Process Memory Image” in Unix (2)

- heap → Where all dynamically allocated memory is set
  - e.g., as a result of malloc()

- stack → Where all automatic variables exist
  - Variables that are automatically created and destroyed in methods
    - Including return address of functions, machine registers, etc.
  - It grows down (higher to lower addresses)

- **Note:** Contents of uninitialized data segment are not stored in the program file on disk, because the kernel sets them to 0 before the program starts running. Only text segment and the initialized data need to be saved in the program file.

The “Process Memory Image” in Unix (3)



.stack segment

(higher address)

possible hole in
address space!

.bss segment

.data segment

.text segment

Basically, these are the
ones that take space
in the executable…

Quiz: Where does everything go?

```c
#include <stdio.h>
#define KB (1024)
#define MB (1024*1024)

char buf[10*MB];
char command[KB] = "command?";
int n_lines = 0;
int n_tries = 20;
int total;

int f(int n) {
     int result;
     static int number_calls = 0;

     ++number_calls;
     result = n*n;
     return result;
}

int main() {
     int x = 5;

     printf("f(%d)=%d\n", x, f(x));
     return 0;
}
```

.bss

.data

.bss

.data

.bss

.stack

.bss

.stack

What about the globals:

char buf[10*MB] = {0};

char buf[10*MB] = {1};

Finding things out in Linux

- Compile and store temporary intermediate files (flag *save-temps*);

```
$ gcc -Wall -save-temps my_prog.c -o my_prog
```

  - my_prog.i – preprocessed file
  - my_prog.s – assembly file
  - my_prog.o – object file
  - my_prog – executable file

- List section sizes (in bytes) and total sizes of executable

```
$ size my_prog
 text    data    bss         dec         hex         filename
 1259    1548    10485824    10488631    a00b37      my_prog
```

- Display assembly code

```
$ less my_prog.s
```

- Displays information from the object file

```
$ objdump -t my_prog | egrep '\.data'
$ objdump -t my_prog | egrep '\.bss'
$ objdump -t my_prog | egrep '\.text'
```

Exercise @home

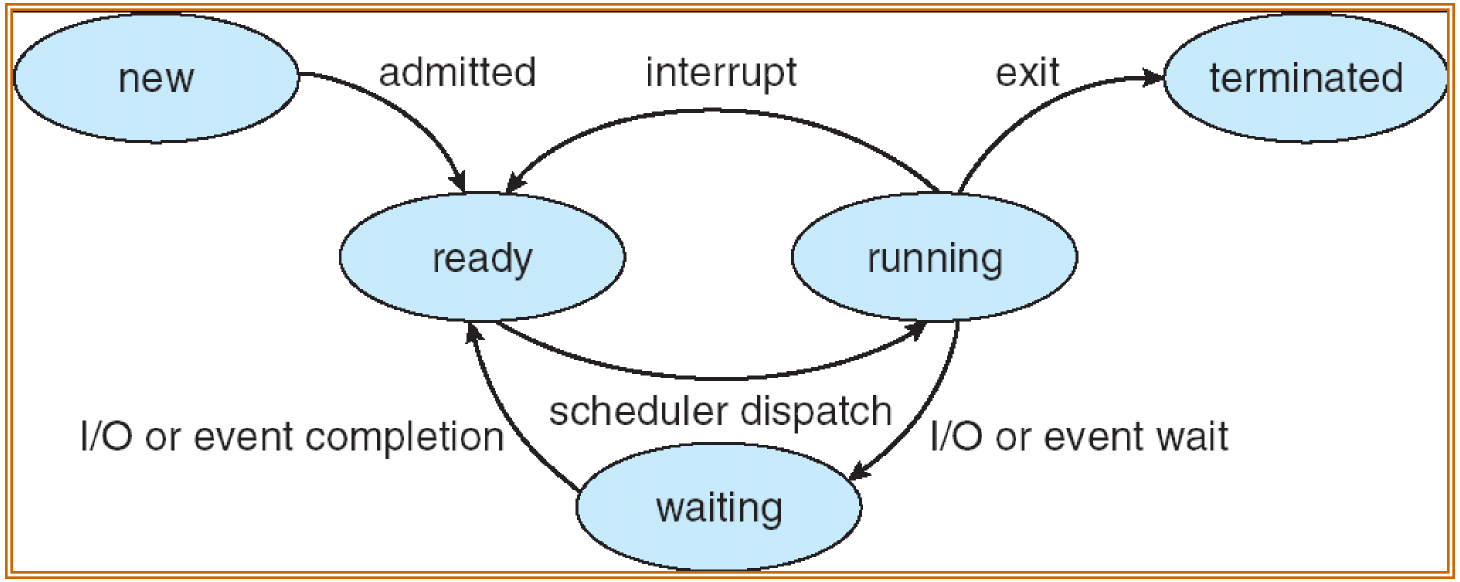
- Start with a simple C program

```
#include <stdio.h>
int main(void){
    return 0;
}
```

Compile and use command `size` at each step, to see check the segments

- 1 - Add one global variable after "`#include`" to the program and check segments
  `int global_var;`
- 2 - Add one variable before "`return 0;`" and check the segments
  `static int i;`
- 3 – Change the last line by initializing the variable, and check the segments
  `static int i = 10;`
- 4 - Change the global variable line by initializing it, and check the segments
  `int global_var =1;`

Process States



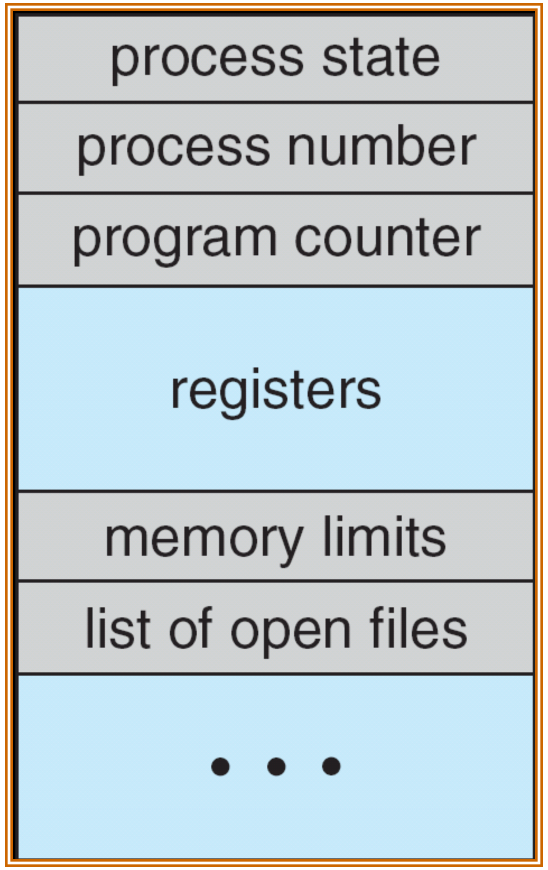
Notes:

- "**Waiting**" is also known as "**Blocked**"
- Processes in the Ready and Blocked states do not consume CPU!
  This is very important!

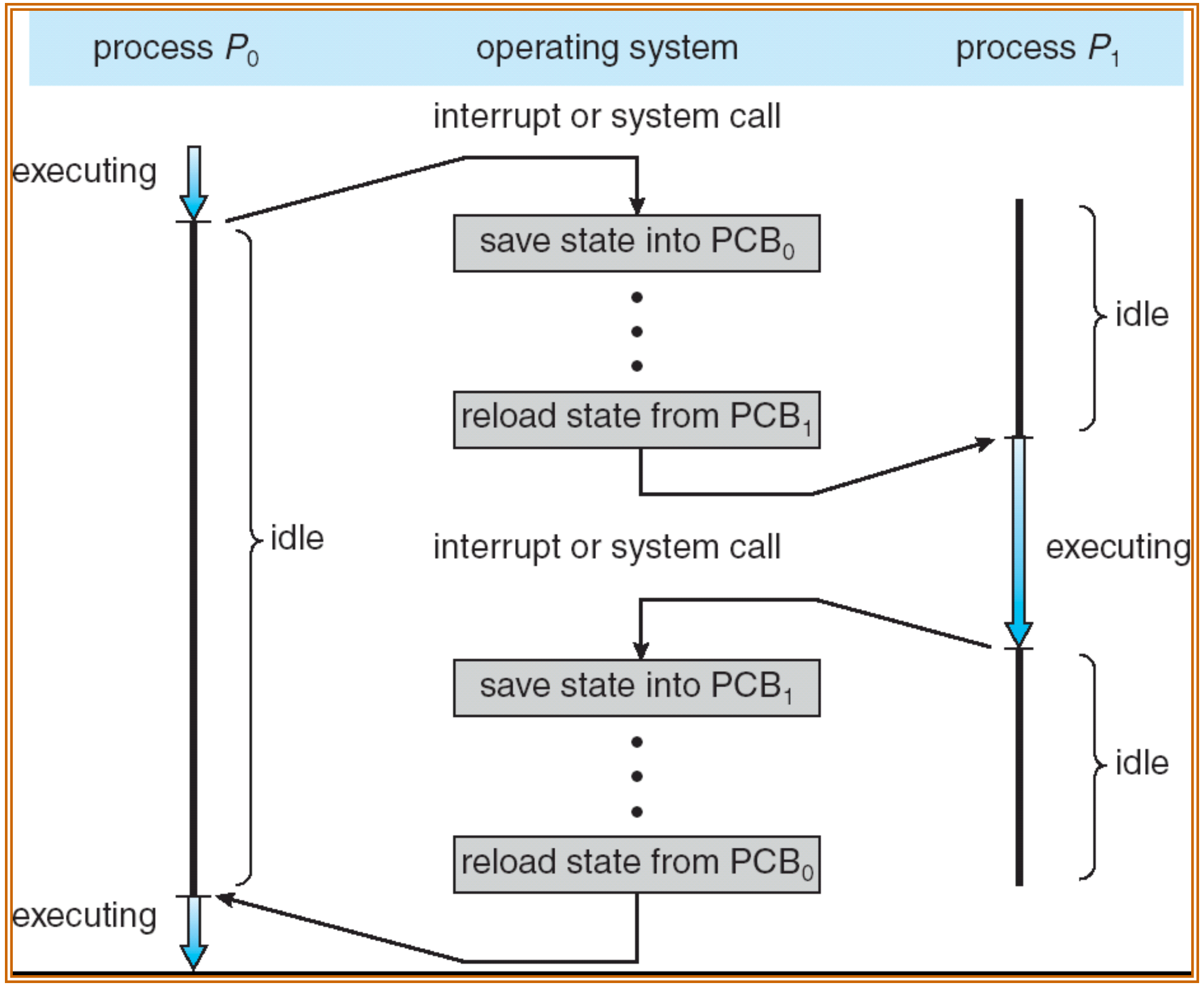
Process Control Block (PCB)

- One of the ***most*** important data structures of the OS
  - Represents a process in the Operating System

- The **PCB** includes:
  - Process state
  - Process identifier
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information
  - List of open files
  - …

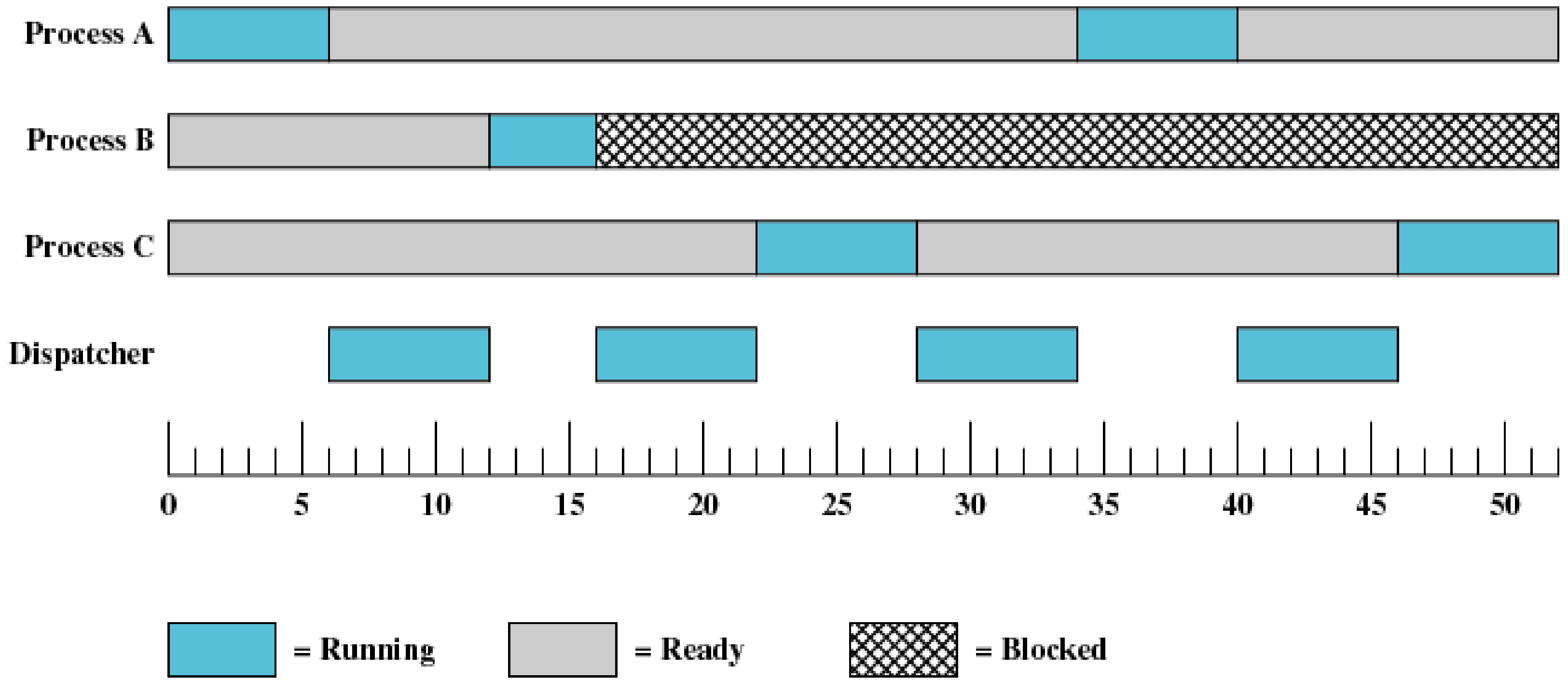


Context Switch from Process to Process



A **process context** is constituted by all the data that represents the current state of a process, which is saved on the PCB. It represents the context within which the process executes.
When the context switch occurs the entire system context is replaced.

Execution of three processes over time…



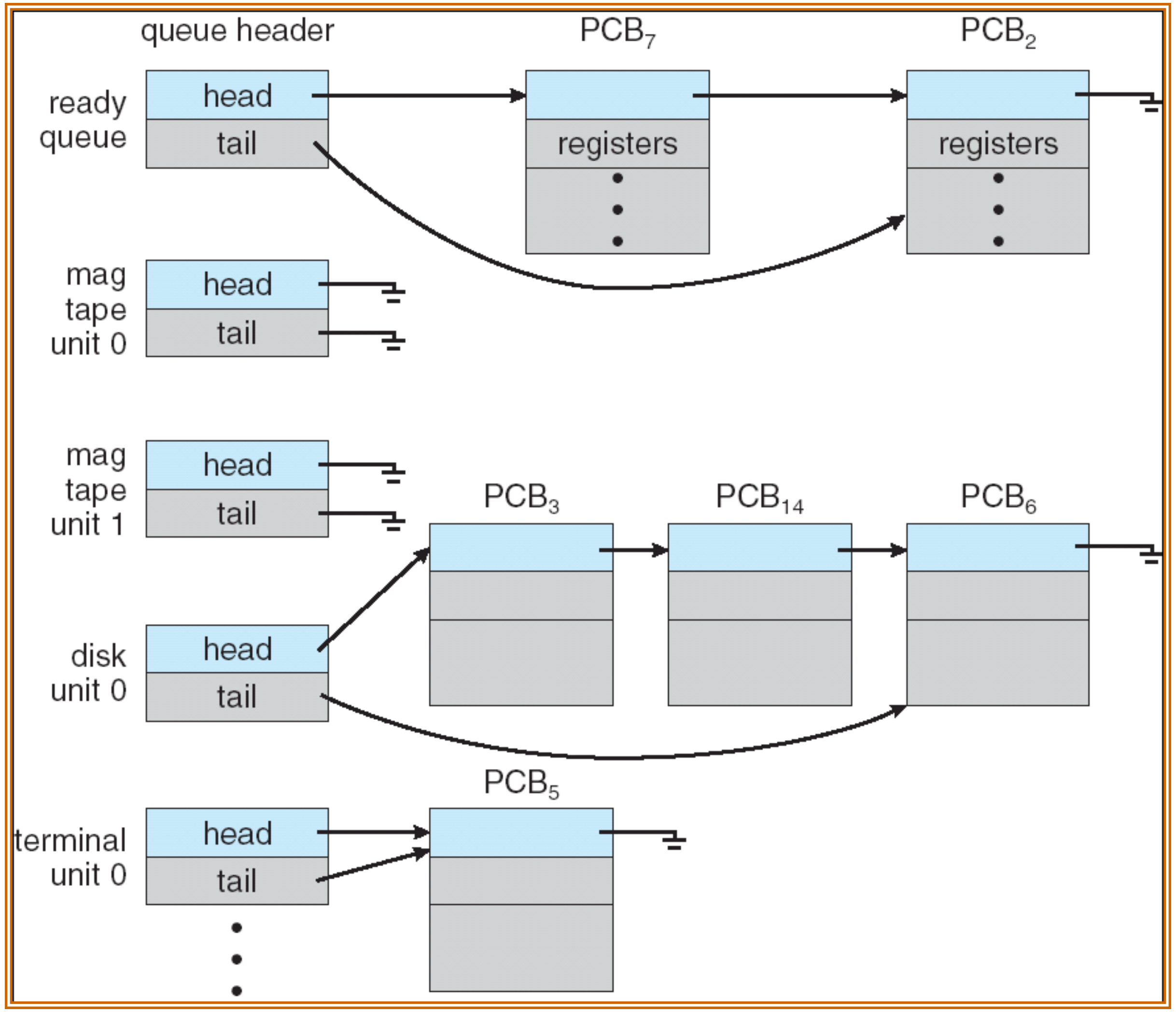
The **dispatcher** is a module that gives CPU to a process previously selected by a scheduler; it takes the process from the ready queue and moves it into the running state

Process Queues

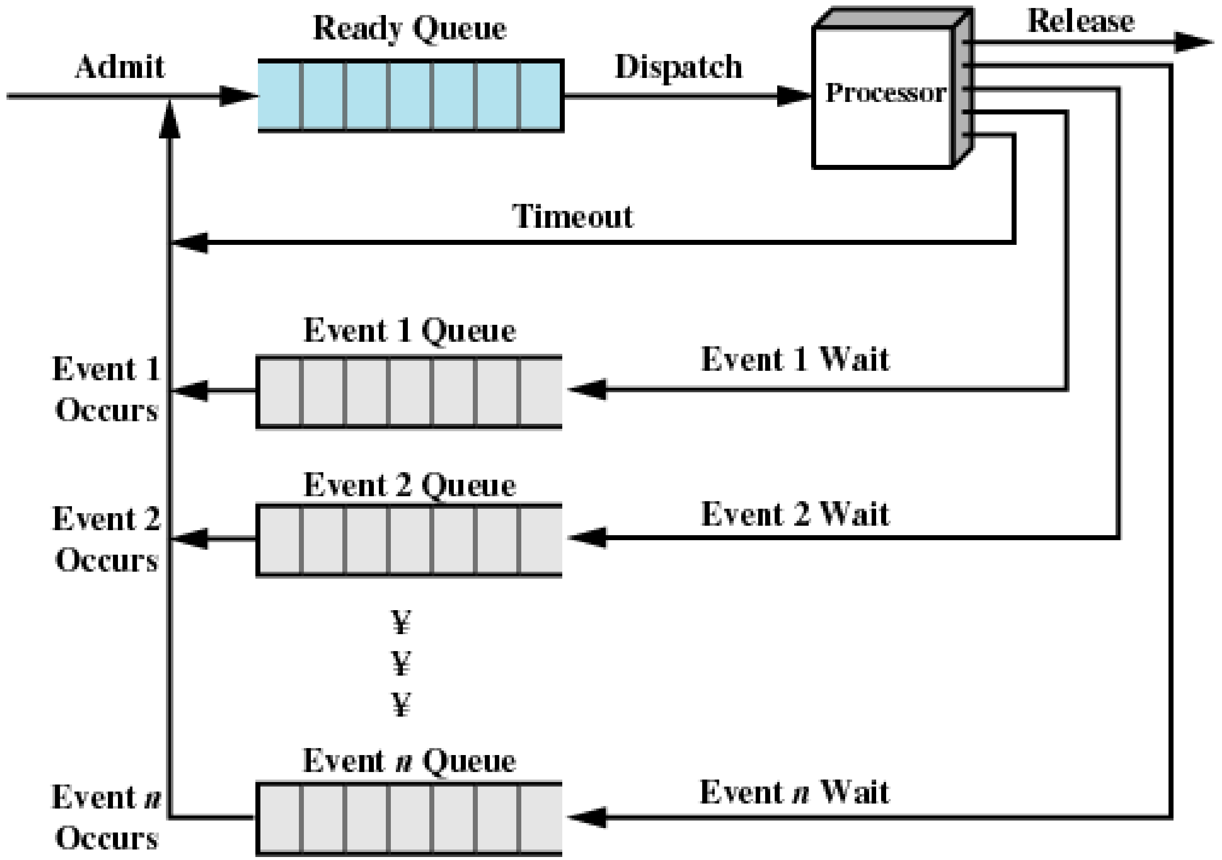
- So, where are all the PCBs saved?
  - Processes migrate among the various queues…
  - **Job queue**
    - Set of all processes in the system
  - **Ready queue**
    - Set of all processes residing in main memory, **ready and waiting to execute**
  - **Device queues**
    - Set of processes **waiting for an I/O device** (also known as blocked queues)

Several Queues...

Device Queues
(i.e. works like a global blocked queue)



PCBs Flow Among Queues



Note that events do not necessarily correspond to I/O… (more on this later)

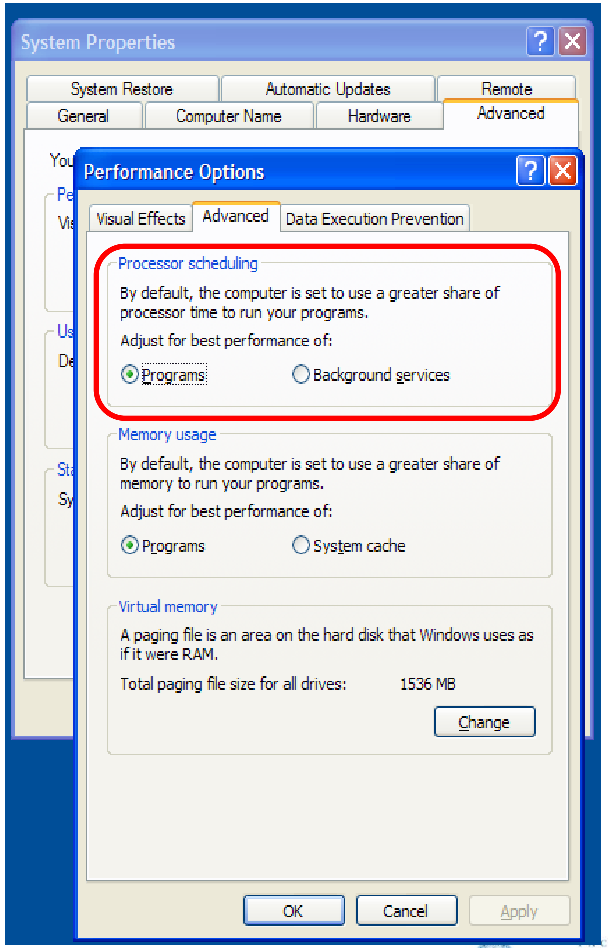
Process Scheduler

What's the Best Scheduler?

It depends...

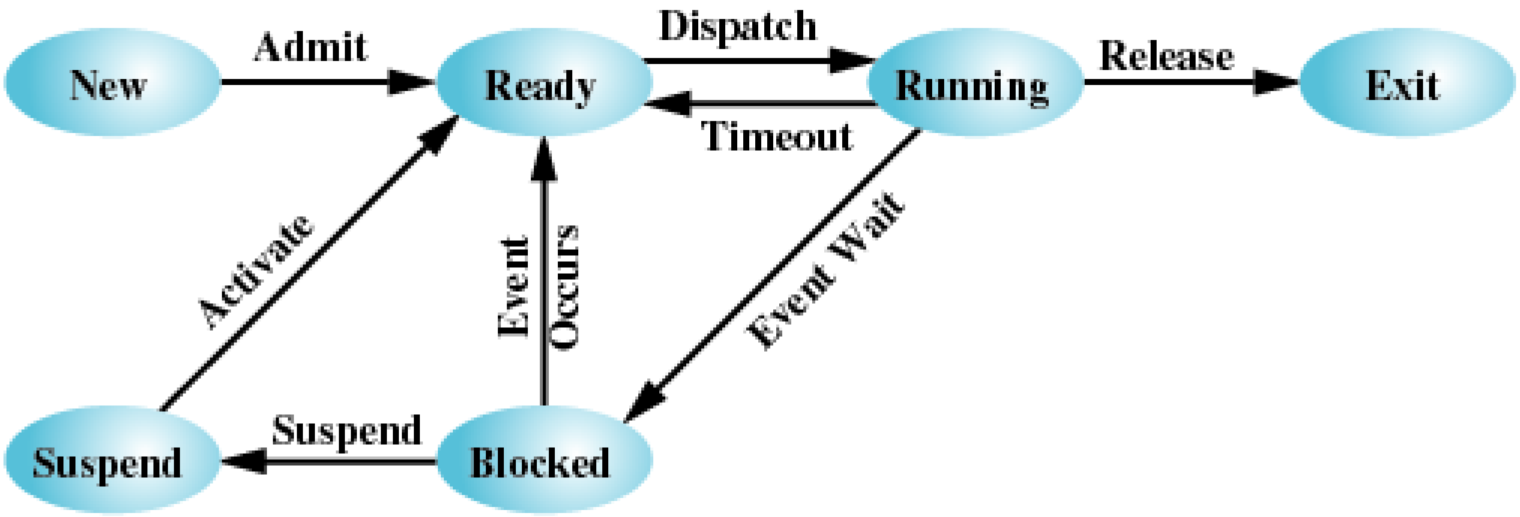
- Server Multitasking Operating Systems
- User Multitasking Operating Systems
- Batch Operating Systems
- Real-time Operating Systems
- Embedded Operating Systems
- Multimedia Operating Systems
- ...

Even in Windows



Suspended States

- In some operating systems, additional states are introduced for describing processes that have been **swapped** to disk

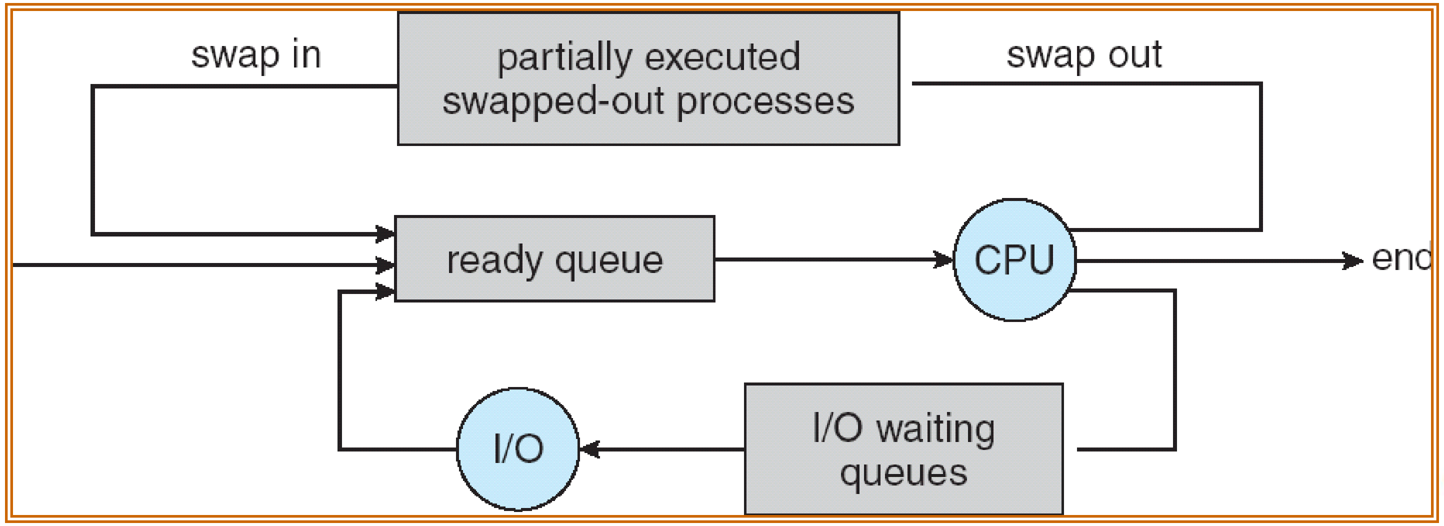


(a) With One Suspend State

Scheduling with Suspended States

- In systems with suspended states it is common to have:
  - A long term (or medium term) scheduler

    … selects which processes should be brought from disk into the ready queue
  - A short-term (or CPU) scheduler
    - … selects which process should be executed next (from the ones in the ready-queue) and allocates CPU



Scheduling with Suspended States

- The schedulers' names suggest the relative frequency with which their functions are executed
  - The **long-term**, or admission scheduler, decides which jobs or processes are to be admitted to the ready queue
    - Sets the degree of concurrency to be supported at any one time
  - The **mid-term** scheduler temporarily moves processes from main memory and places them on secondary memory and vice versa
  - The **short-term** scheduler decides which of the ready, in-memory processes is to be executed

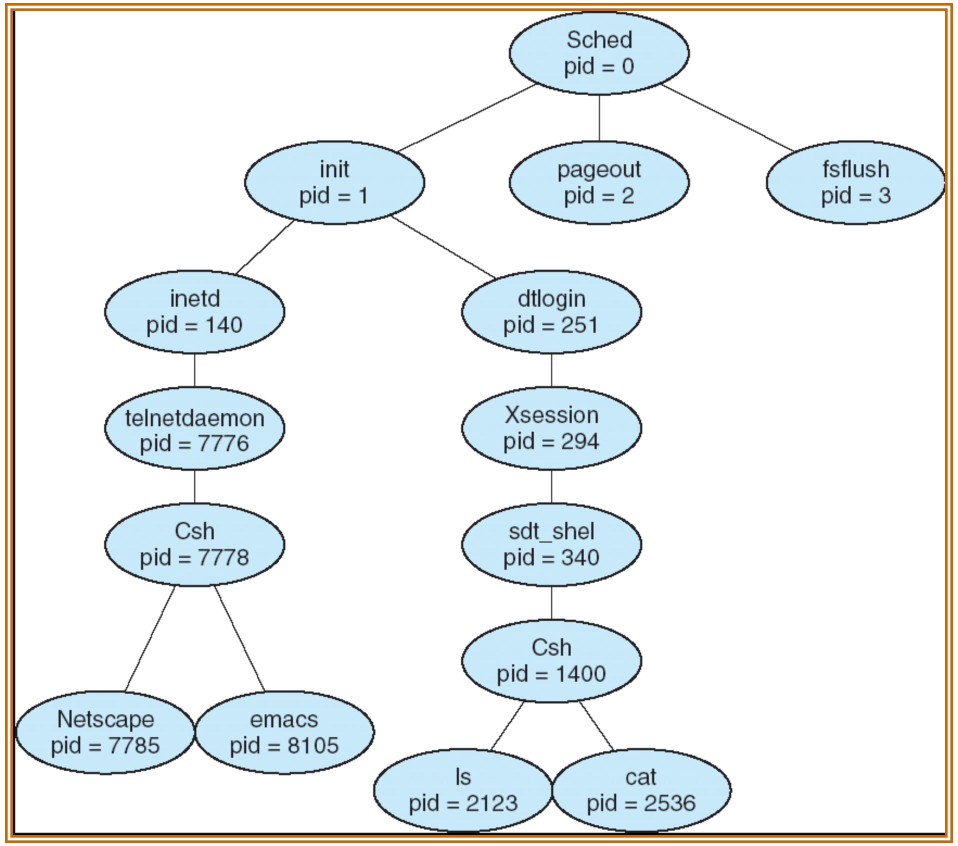
Types of Processes

- **I/O Bound**
  - Spend more time doing I/O than computation (Lives mostly in the blocked queue)
- **CPU Bound**
  - Spend more time doing computation than I/O (Live mostly in the ready queue)

- For a good utilization of computer resources, it is important to have a good mix of I/O bounded and CPU bounded processes: <u>the long-term scheduler is responsible for this</u>

Process Creation

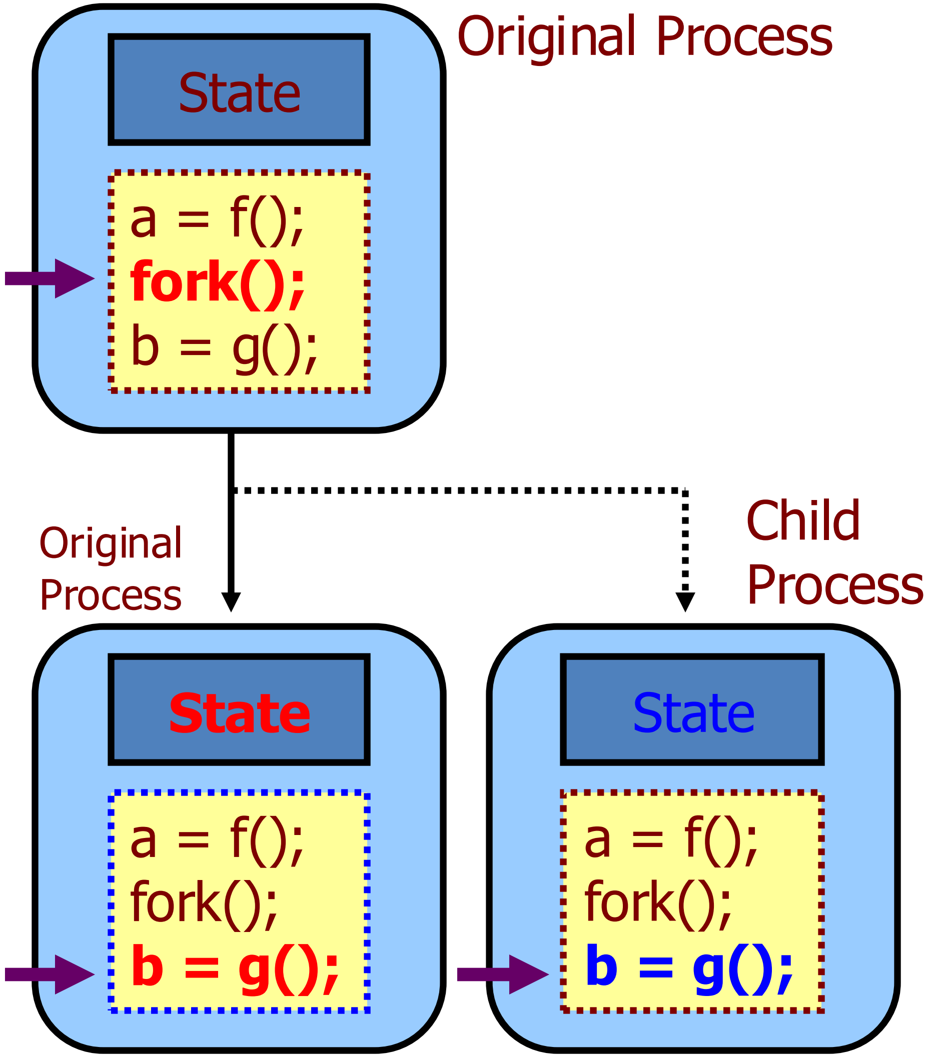
- Parent process create children processes, which in turn create other processes, forming a tree of processes



Process Tree in Solaris

Process Model in UNIX

- Process creation in Unix is based on spawning child processes which inherit all the characteristics of their fathers
  - Variables, program counter, open files, etc.
  - **Spawning a process is done using the fork() system call**
- After forking, each process will be executing having different variables and different state.
  - The Program Counter will be pointing to the next instruction
  - Changing a variable in the child program does not affect its father (and vice-versa)



Different Heritance Models…

- The way resources are shared between parent and child processes varies widely among operating systems…

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

- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate
  - Children terminates if parent terminates (cascading termination)

Copy-on-Write

- Nowadays, the memory of a process is organized in **pages** of typically 4KB
- After a **fork(),** the child process and the parent process share the same pages
- The operating system detects when a page is being written either by the parent process or the child process. When that happens, a copy is made on demand, before the write is allowed to proceed.
- This is called **copy-on-write**
- Thus, changing a variable on a child process does not affect its parent and vice-versa

Shared Address Space with the Kernel

- For making system calls and traps fast, it is possible to jump directly to the kernel handler routine without remapping any memory
  - The address space of each process is divided into two parts:
    - One that is specific of that process
    - One that corresponds to the kernel and is shared by all processes

- How does it work?
  - The user process does not have direct access to the kernel memory; it cannot read nor write that part of the address space
  - Whenever a trap occurs, it enters in "kernel mode" and thus has access to the already mapped memory

Process Termination in UNIX

- A process is only truly **eliminated** by the operating system when its father calls **wait()/waitpid()** on it.
  - This allows the parent to check things like the exit code of its sons

- **Zombie Process**: One that has died, and its parent has not acknowledged its death (by calling wait())
  - Be careful with this if you are designing servers. They are eating up resources!!

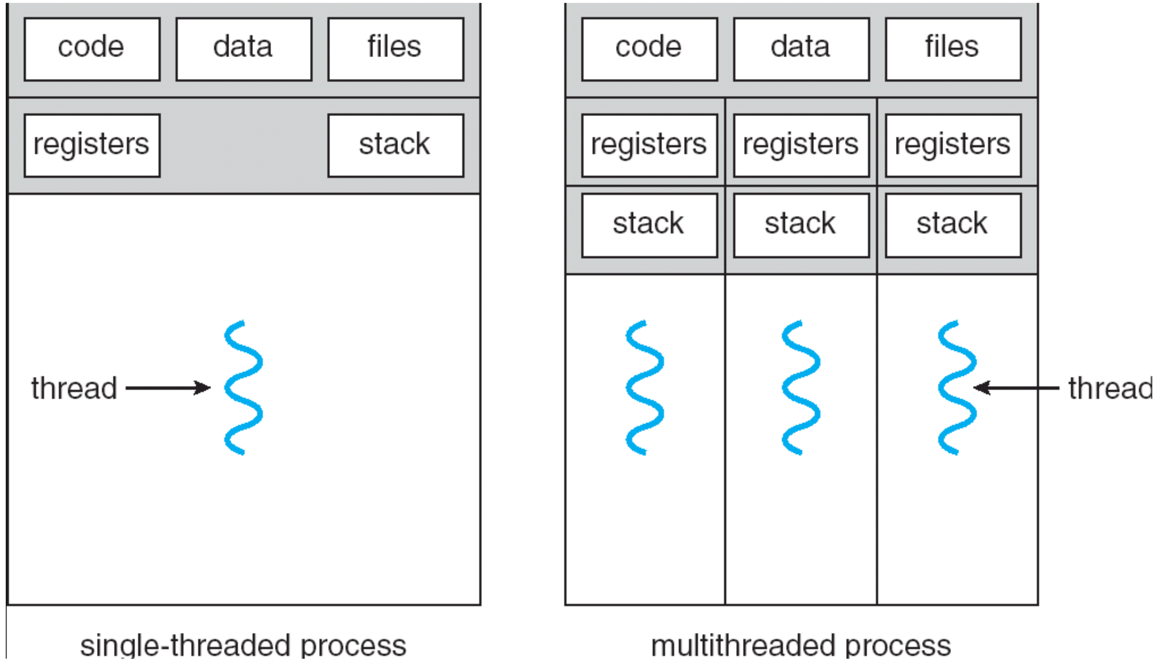
- **Orphan Process**: One whose original parent has died. In that case, its parent becomes init (process 1).
  - Linux follows the same procedure. However, in current versions of the kernel not all orphans are adopted by init.

Reasons for Process Termination

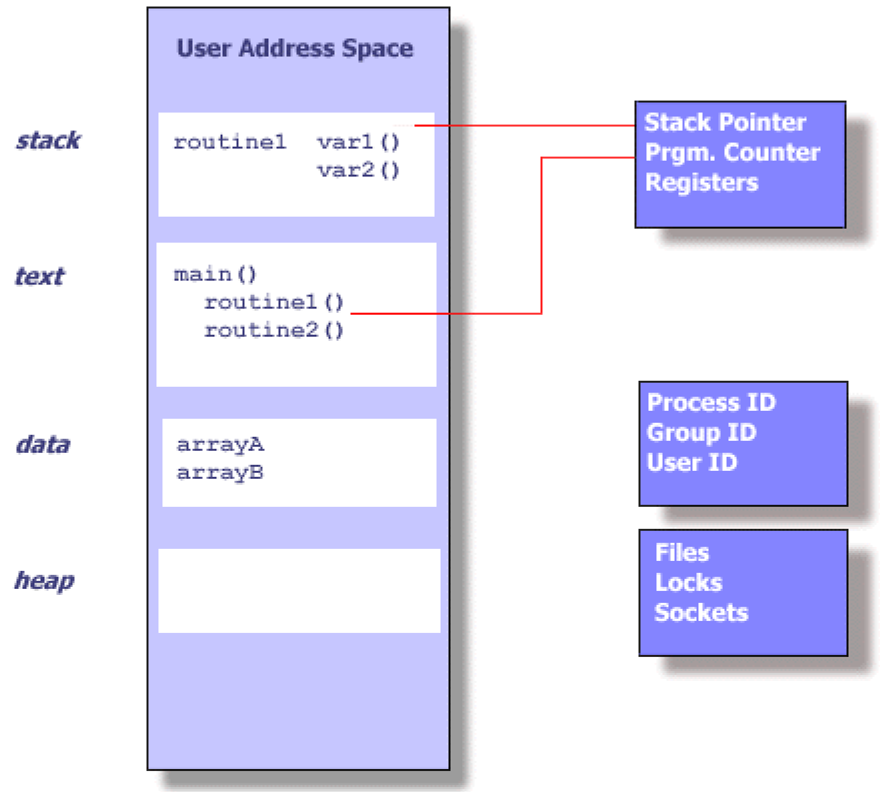
- Normal completion
- Time limit exceeded
- Memory unavailable
- Bounds violation
- Protection error
- I/O failure
- Invalid instruction
- Operating system intervention (e.g. on deadlock)
- Parent terminates so child processes terminate
- Parent request
- …

THREADS
“LIGHTWEIGHT PROCESSES”

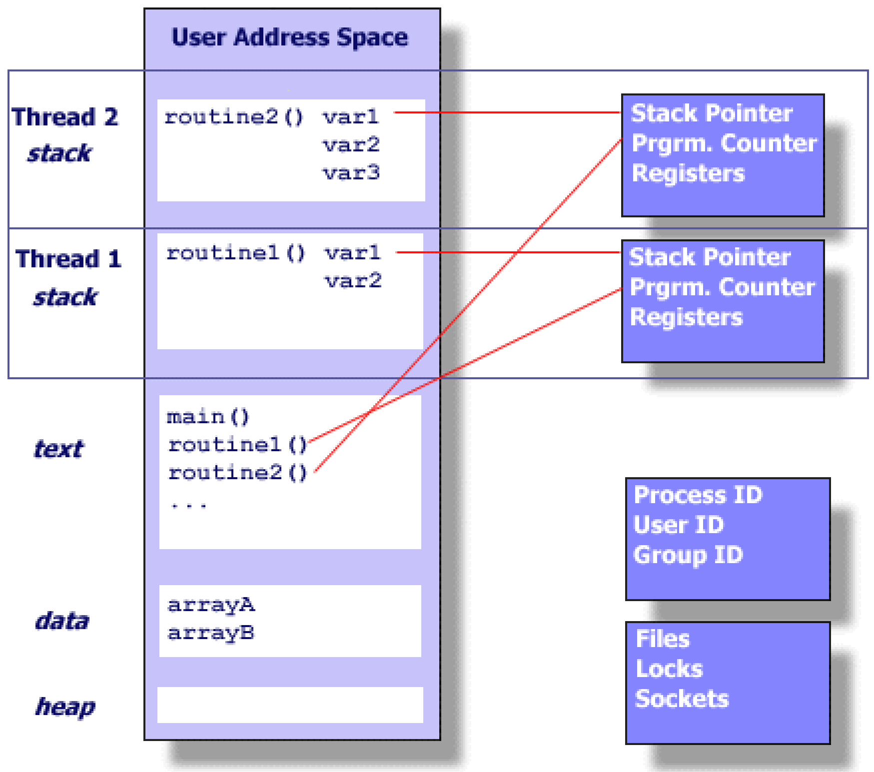
What is a thread?



What is a thread?



One thread



Multiple threads

Motivation

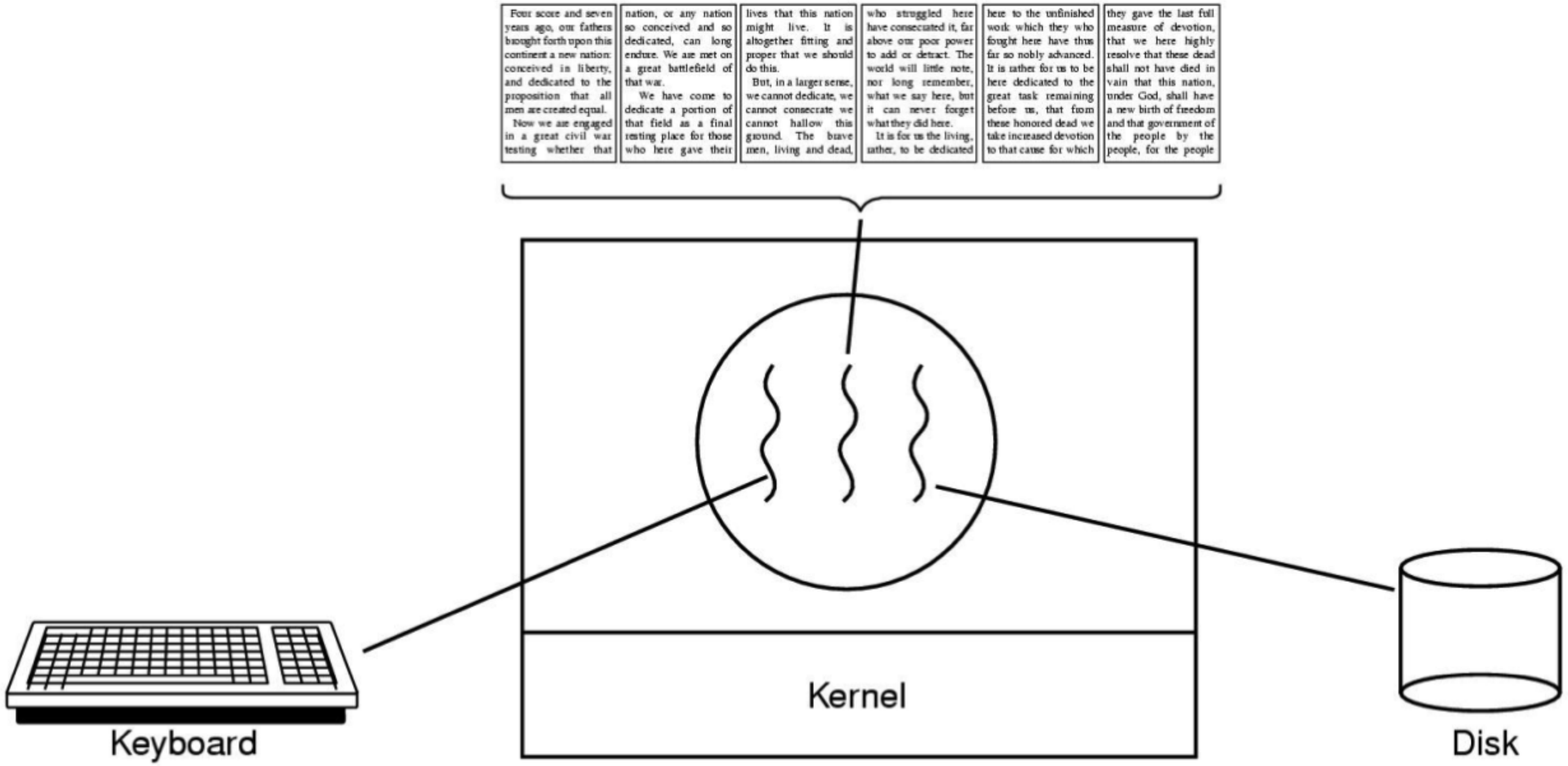
- Process switching is an extremely heavy operation
  - It’s necessary to remap address spaces
  - It’s necessary to establish new security contexts
  - It’s necessary to manage all information associated to processes

- Thread
  - It’s a flow of execution inside a program
  - The address space is the same
    (…changing a global variable in a thread affects all others)
  - Lighter context switching between threads
  - Commutation among threads is very fast
  - Communication among threads is easy to do and fast

Why use threads? (2)

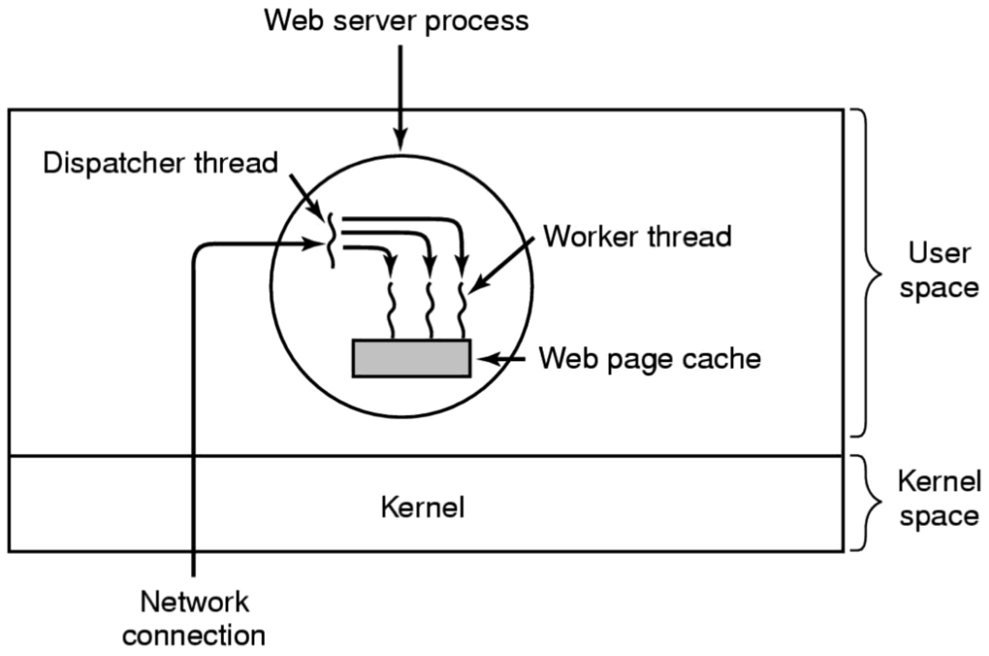
Multithreaded application

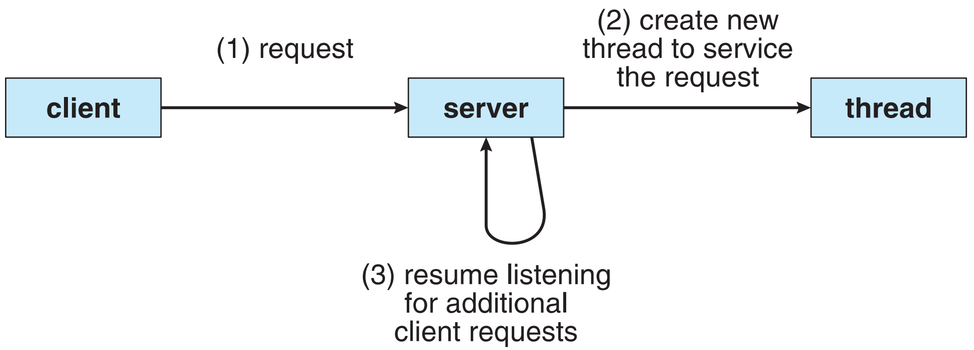
- Example of a word processor with multiple tasks at the sae time



Why use threads? (3)
Multithreaded server

- Web server example





Why use threads? (4)

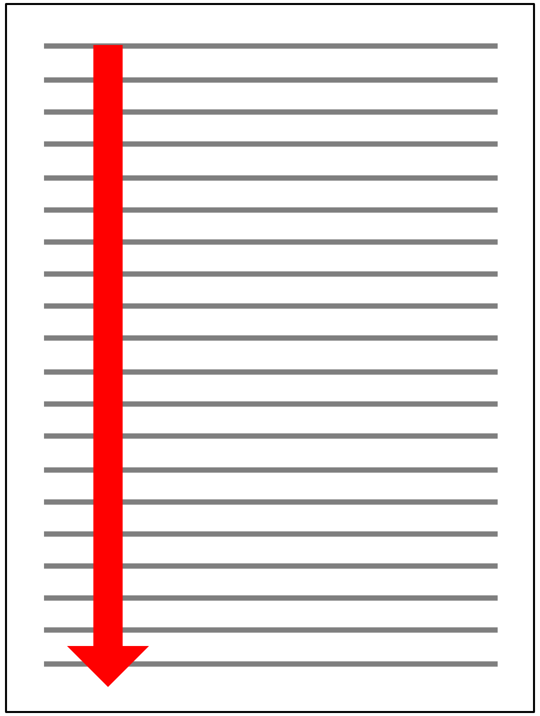
- They are very lightweight compared to processes
  - Light context switches
  - Fast to create and terminate
  - Fast to synchronize
  - Fast to communicate among threads
  - Very useful in multiprocessor/multi-core architectures
  - Economy or resources
- Much easier to program than shared memory!
  - Everything is already shared
  - Be careful to synchronize accesses!

Important

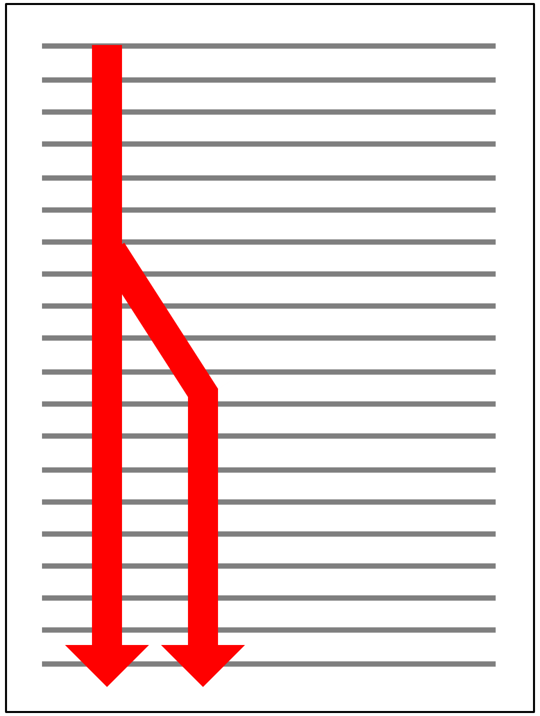
- Threads share the same address space and resources! Changing one thing in one thread affects all others!
  - It is the responsibility of the programmer to assure the correctness in the concurrent access to data and resources (more on this later…)

Single-threaded vs multithreaded process

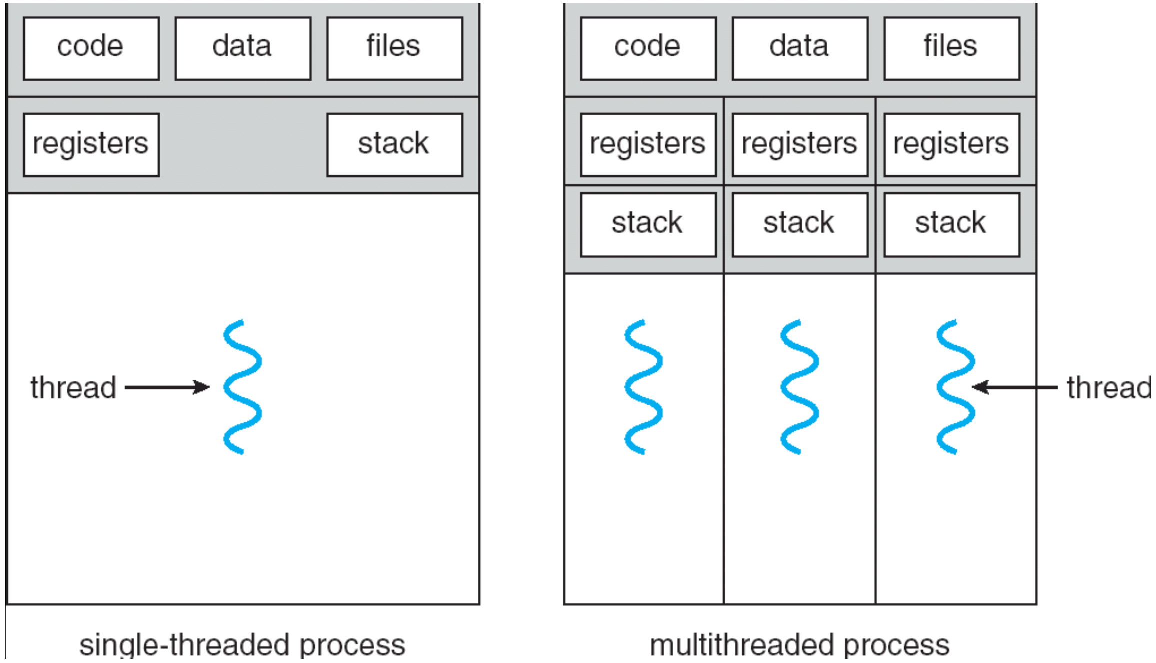
- Single-threaded
  - One thread of execution = one execution flow



- Multithreaded
  - More than one thread of execution = more than one execution flow

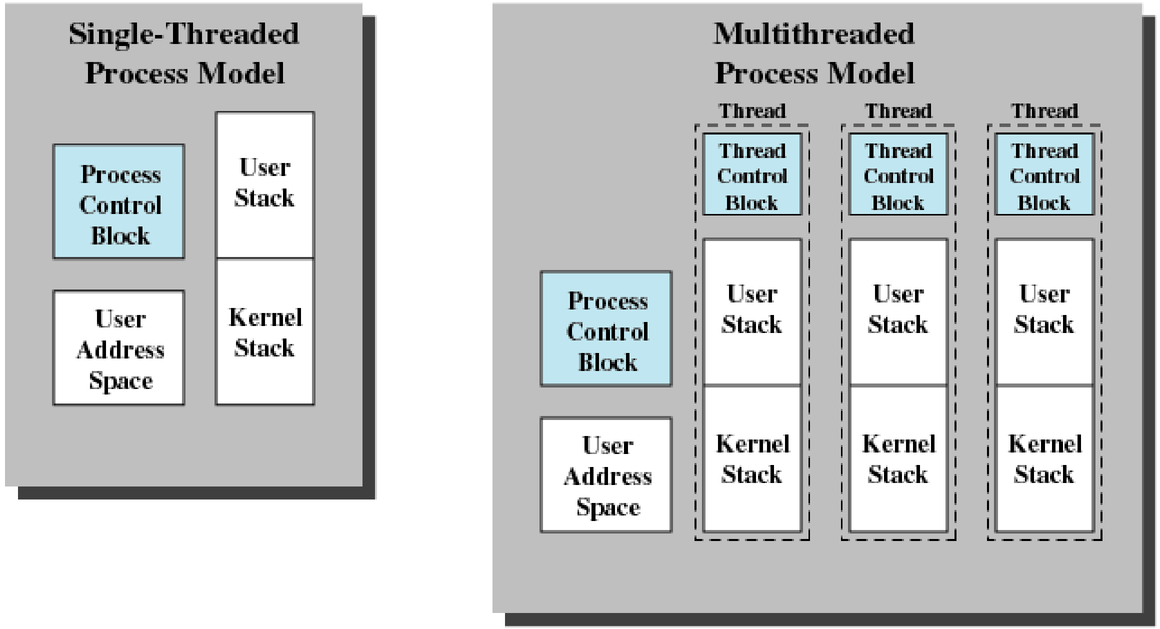


Threads – PCB vs Thread Control Block



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

- Single-threaded process
  - One thread of execution = one execution flow
- Multithreaded process
  - More than one thread of execution = more than one execution flow



How to implement threads

- Support for threads may be provided at user-level, for user threads, or by the kernel, for kernel threads.

- Kernel Level Threads
  - Supported by the OS kernel and handled by the system scheduler
  - Have direct access to system-level features

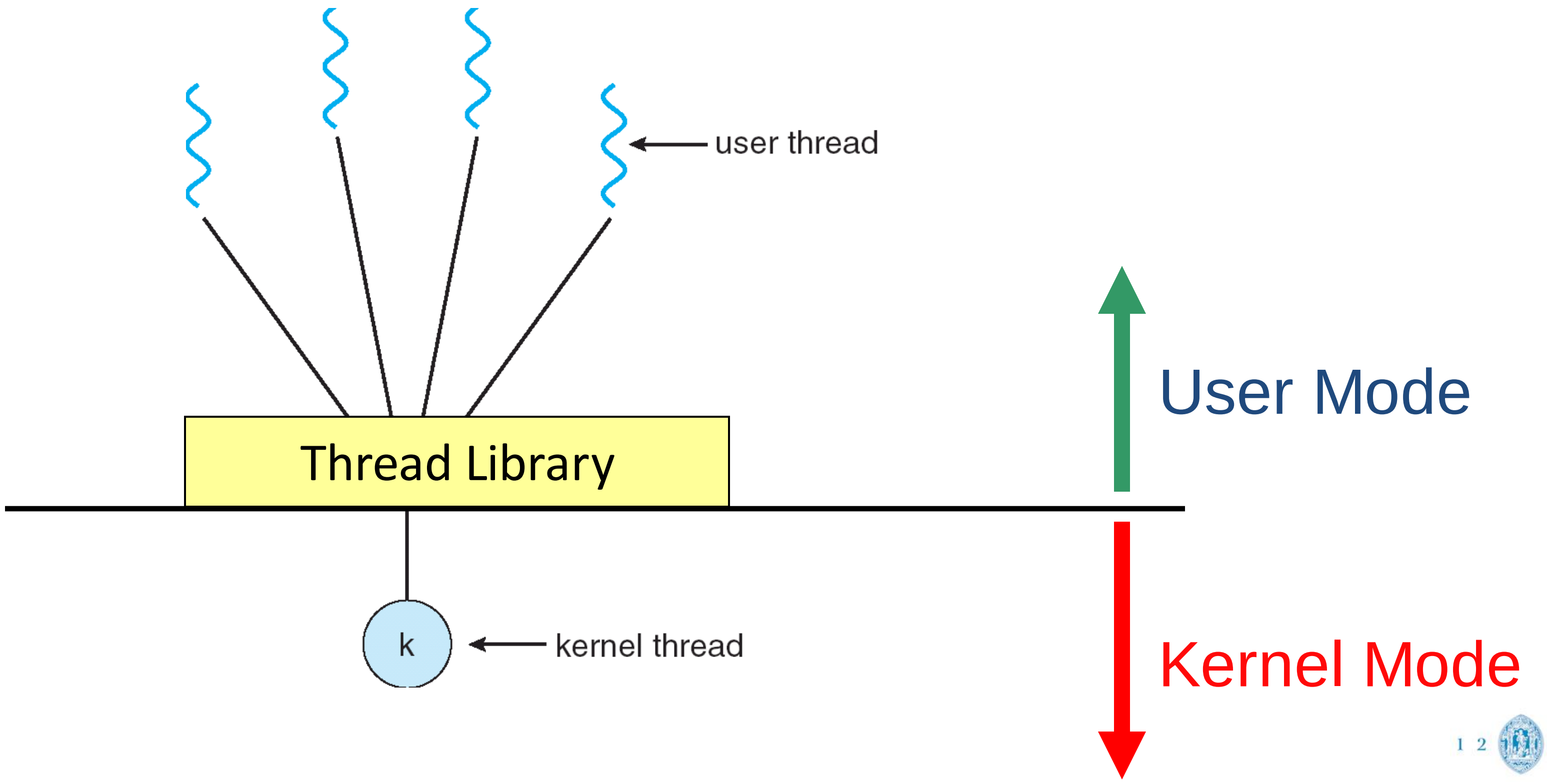
- User Level Threads
  - Created and managed by a user-level thread library, not relying on kernel support
  - Kernel is unaware of their existence
  - Useful in OSs without multithreading support
  - This libraries may be implemented and used across various OSs

How to implement threads

- A relationship must exist between user threads and kernel threads:
  - Many-to-one model
  - One-to-one model
  - Many-to-many model
- OBS: There are also hardware threads, but those are a different subject; more on this later

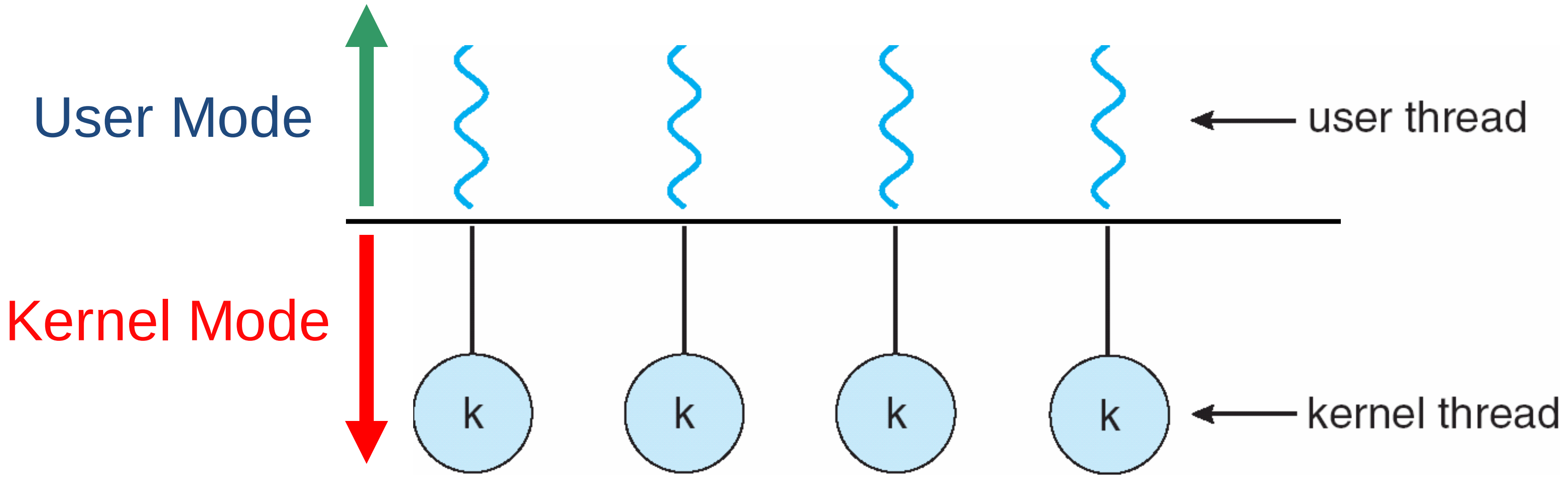
Multithreading Models (1)

- **User threads (N-to-1 model)**
  - Threads are implemented in a library at user space.
  - The kernel is completely unaware of the existence of threads



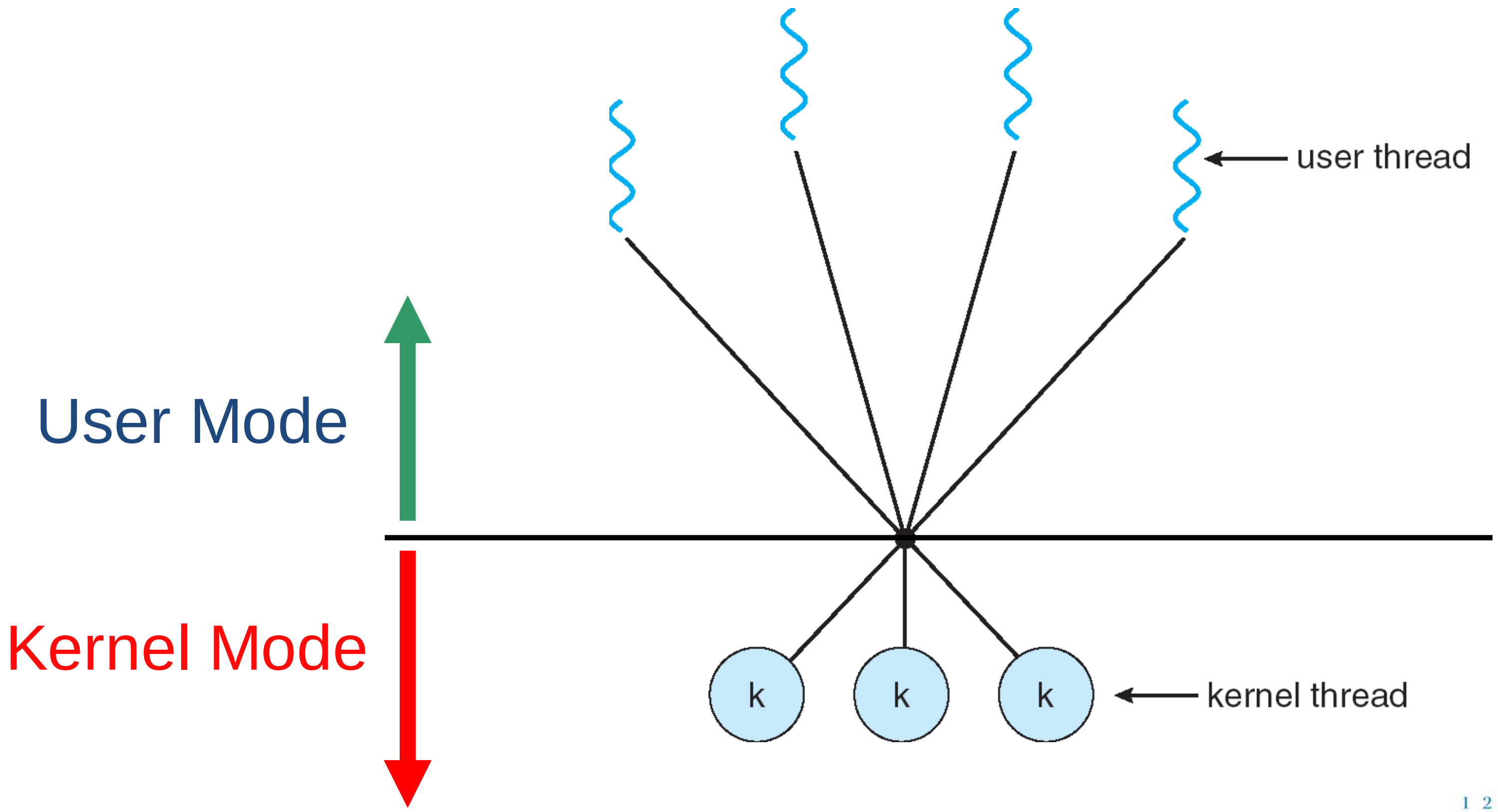
Multithreading Models (2)

- **Kernel threads (1-to-1 model)**
  - Threads are implemented exclusively in kernel space
  - The kernel does all the scheduling of threads



Multithreading Models (3)

- **The Many-to-Many Model**
  - A number of kernel threads can map to a different number of user threads



Advantages and Disadvantages

User Level Threads

- ☺ Commutation among threads is fast since it does not imply a mode switch
- ☺ Thread creation and termination are fast
- ☺ Does not require any special support by the kernel, thus being available in any OS

- ☹ If one thread blocks all threads block (e.g. on I/O)
- ☹ Only one system call can happen at a time
- ☹ Scheduling is not fair
- ☹ One thread terminates all threads terminate

Kernel Threads

- ☹ Commutation among threads is slower since it implies going to the kernel
- ☹ Thread creation and termination is slower
- ☹ Requires that the kernel supports user visible threads

- ☺ If one thread blocks the others can continue
- ☺ Simultaneous system calls

- ☺ Fair scheduling
- ☺ Somewhat independent thread termination

Current Operating Systems

- User-level Threads
  - E.g., Solaris before version 9 (*Green Threads*)
  - E.g., Windows NT/2K with *Fibers* package

- Kernel-level Threads
  - Virtually all modern operating systems:
    LINUX, Solaris, Windows, Mac OS X, Tru64 Unix

Thread Libraries

→ Threads must be implemented by an API (library).
→ Libraries can be user-level or kernel-level
→ Using a library does not mean that it is user-level threading
→ Main libraries in current use: Java, PThreads, Win32

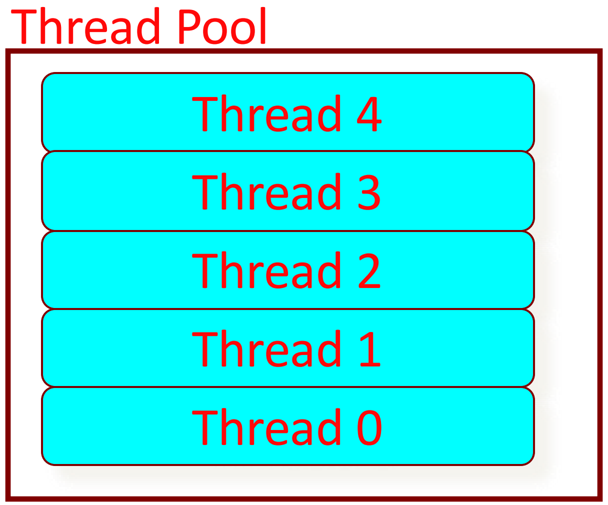
Some technical hurdles…

- What happens if a thread calls fork(), exec() or exit()?
- How to perform thread cancellation (a.k.a. kill-a-thread)?
- What happens if a signal is sent to a multi-threaded process?
- How can a thread have its own private data?

Depends on the specific OS or library implementation!

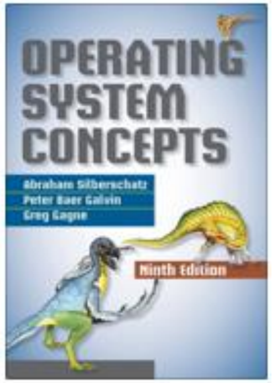
Thread Pools

- Since most operating systems use a 1-to-1 model, it is important not to waste much time creating and destroying threads
  - E.g., if you have a web server concurrently accepting requests from clients, it makes no sense to: 1) create a thread to serve a client; 2) the thread actually serves the client; 3) the thread dies; 4) start all over again…

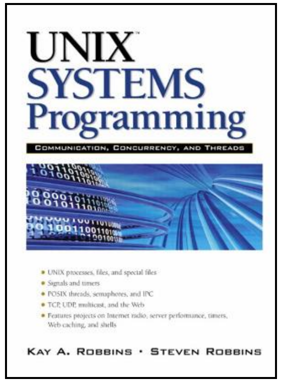


- Some threads are previously created and BLOCKED waiting for work
- Whenever work appears, a managing thread wakes one thread in the pool and assigns it the work to be done
- After the thread completes the work, it returns to the pool waiting for more work

References



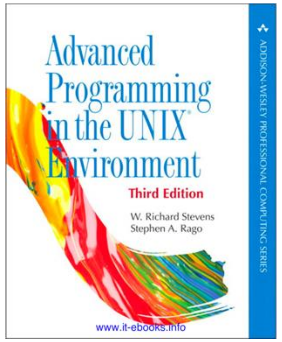
- [Silberschatz13]
  - Chapter 3: Processes
    - 3.1, 3.2, 3.3
  - Chapter 4: Threads
    - All chapter 4!



- [Robbins03]
  - Chapter 2: Programs, Processes and Threads
  - Chapter 3: Processes in Unix

- Linux threading models compared: LinuxThreads and NPTL, by Vikram Shukla, IBM developerWorks

References (2)



- [Stevens13]
  - Chapter 7: Process Environment
    - 7.6

Where to learn more?

- Some interesting videos in YouTube:
  - A PROGRAM is not a PROCESS
  - The Most Successful Idea in Computer Science
  - Why Are Threads Needed On Single Core Processors
  - Threads On Multicore Systems

Thank you! Questions?

I keep six honest serving men. They taught me all I knew. Their names are What and Why and When and How and Where and Who.
—Rudyard Kipling