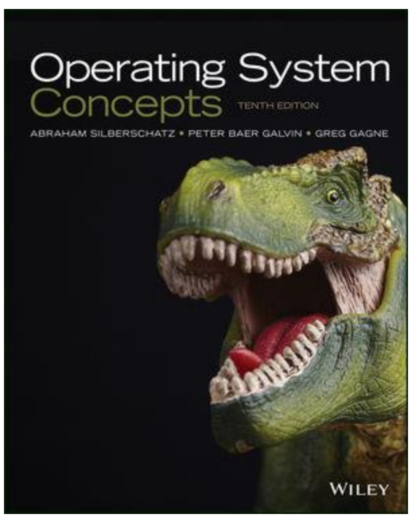
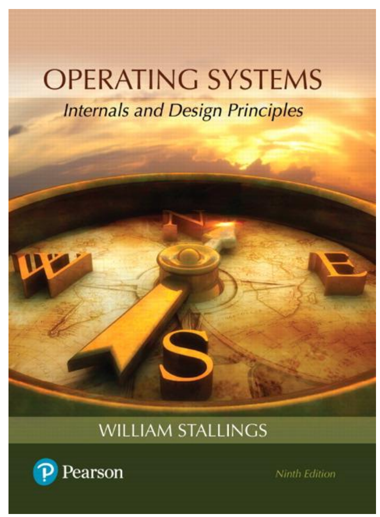
Sistemas Operativos

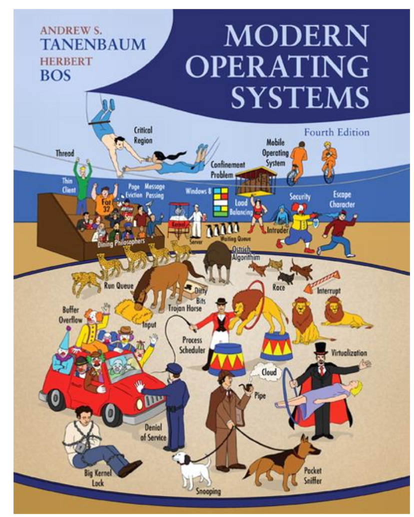
Processes

Disclaimer

- These slides and notes are based on the contents of the books:
  - Abraham Silberschatz, ” Operating System Concepts”, 10th Edition, Wiley, 2018;
  - William Stallings, “Operating Systems: Internals and Design Principles”, 9th Edition, Pearson, 2017;
  - Andrew S. Tanenbaum, “Modern Operating Systems”, 4th Edition, Pearson Education, 2014;
- The respective copyrights belong to their owners.







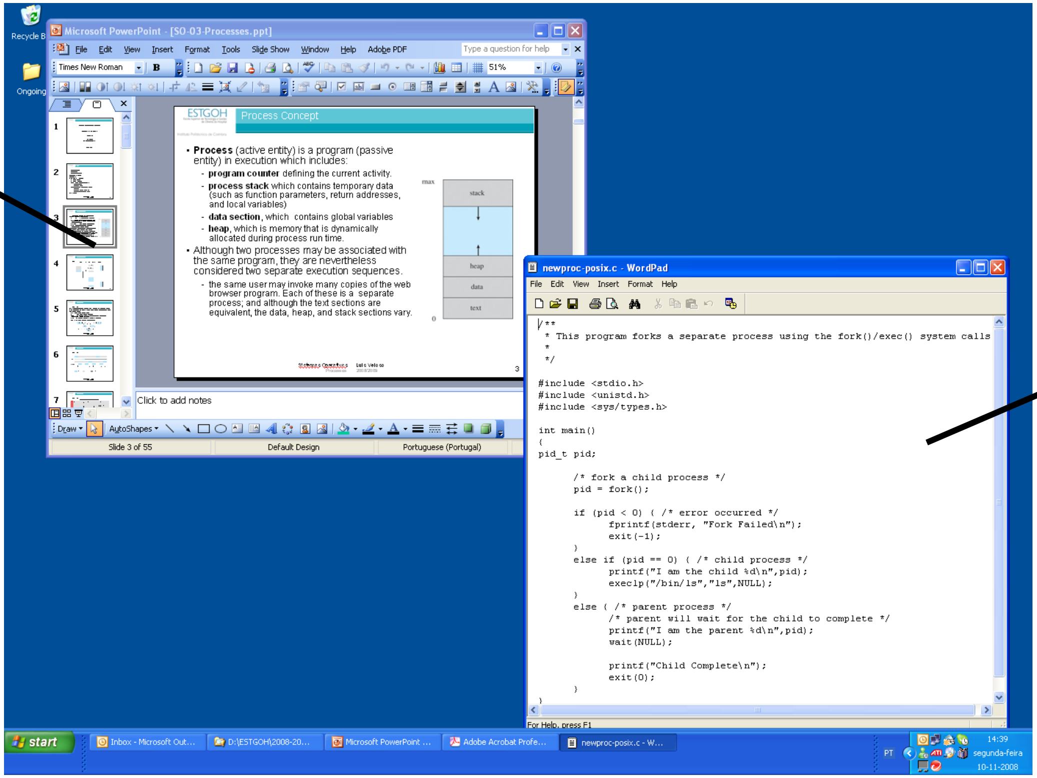
Outline

- **Process Concept**
  - process stack; data section; heap; text
- **Process State**
  - New; Running; Waiting; Ready; Terminated
- **Process Control Block (PCB)**
- **Scheduling Queues**
  - Job Queue
  - Ready Queue
  - Device Queue
- **Schedulers**
  - Long-term scheduler
  - Short-term scheduler
- **Operations in Processes**
  - Process Creation
    - fork(), exec(), wait()
  - Process Termination
    - exit(), wait()
- **References**

Process Concept

- **Process** (<u>active entity</u>) is a program (passive entity) in execution

Powerpoint
(1 process)

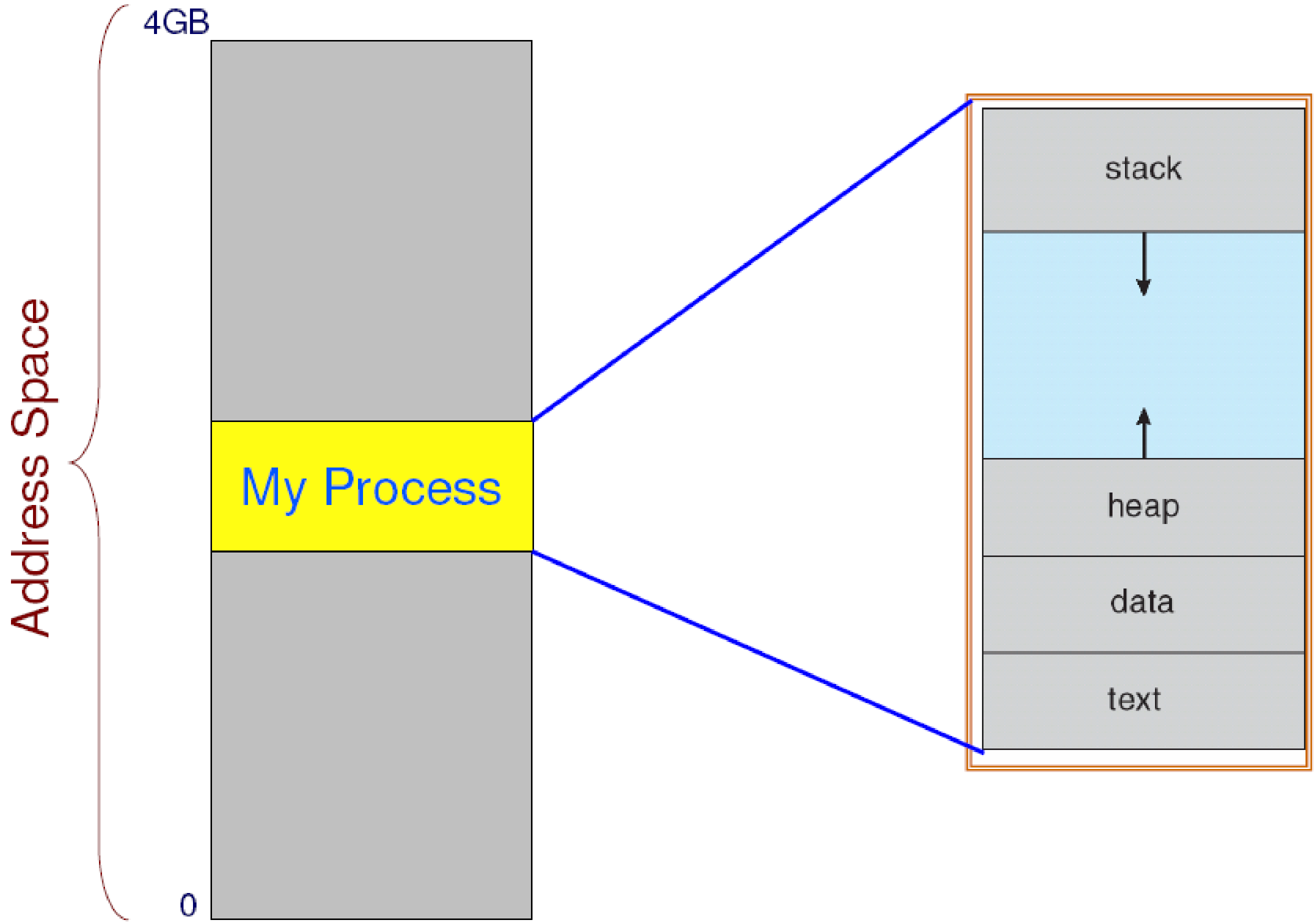


Wordpad
(1 process)

- Program = Binary Image (Executable)
- Process = The living "Image" of a program running

Process Concept

- **Process** (active entity) is a program (passive entity) in execution

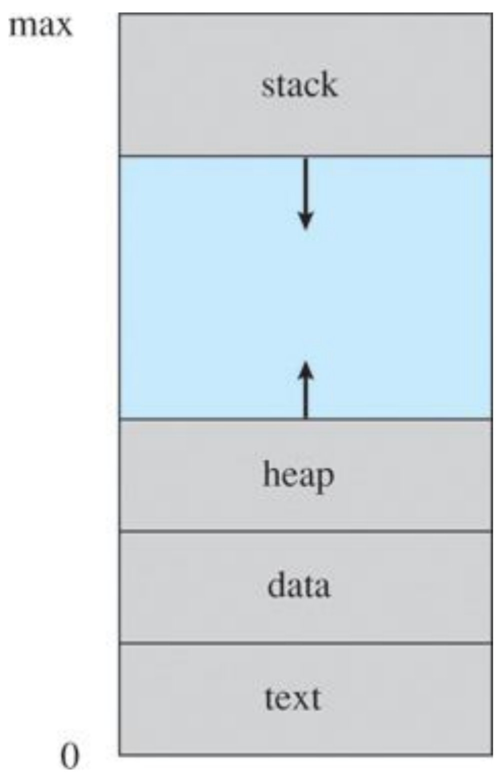


Abraham Silberschatz, ” Operating System Concepts”

Process Concept

- **Process** is a program in execution which includes:
  - **process stack:** contains temporary data (e.g. as function parameters, return addresses, and local variables)
  - **data section**: contains global variables
  - **heap:** memory that is dynamically allocated during process run time (e.g. as a result of *malloc()*).
  - **text:** where the code of the program goes

- Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences.



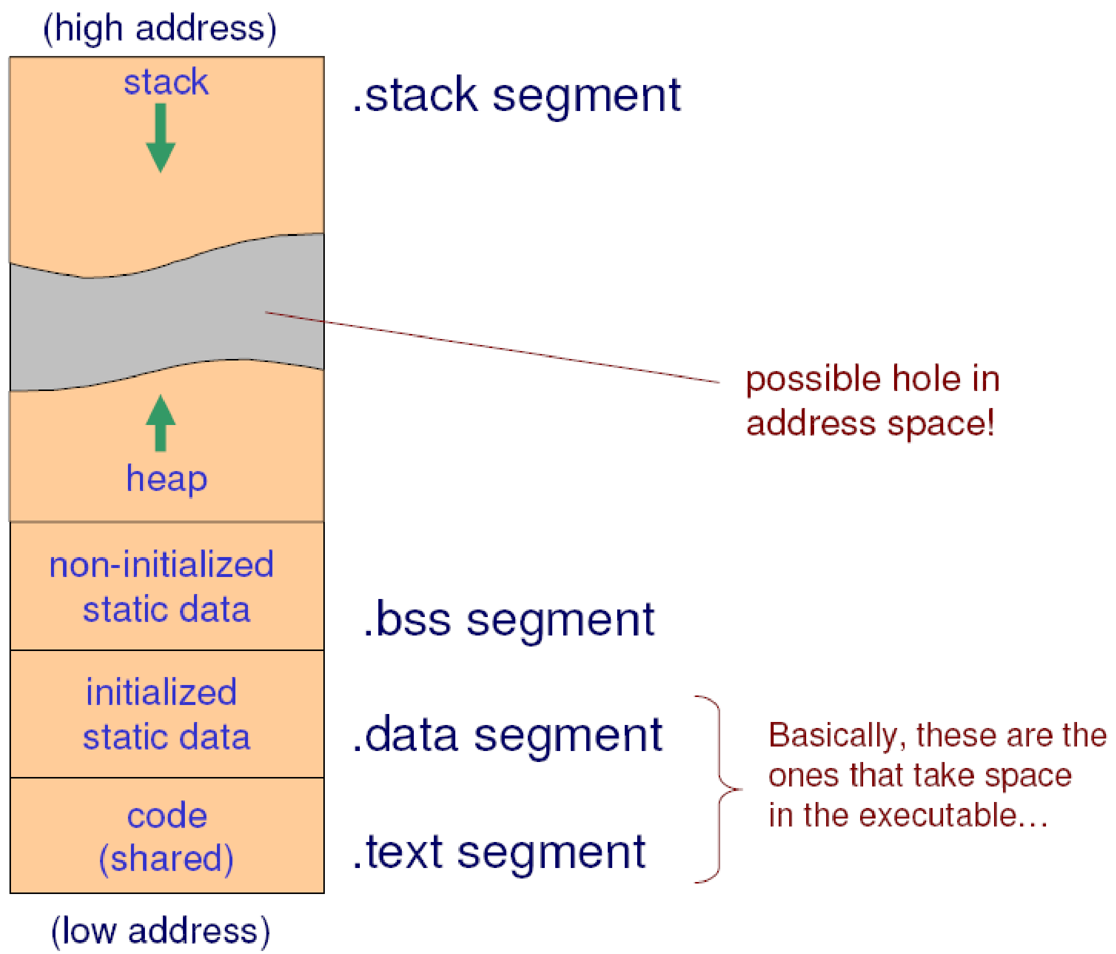
Abraham Silberschatz, " Operating System Concepts"

Process Concept

- **Process**
  - **"Process Memory Image" in Unix:**
    - **Text**: where the code of the program goes
      - called ***.text*** section
    - **Data**: where global and static variables are:
      - ***.data*** **section:**
        - global and static initialized variables to non-zero
      - ***.bss*** **section:**
        - global and static non-initialized variables (or initialized to 0)
    - **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
      - it grows down

Process Concept

- **Process**
  - **"Process Memory Image" in Unix:**



Process Concept

- **Process**
  - **“Process Memory Image” in Unix:**

```
#define KB (1024)
#define MB (1024*1024)

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

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

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

int main() {
        int x = 5;                        .stack

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

Text: where the code of the program goes
Data: where global and static variables are:
- ***.data* section**: global and static initialized variables to non-zero
- ***.bss* section**: global and static non-initialized variables (or initialized to 0)

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

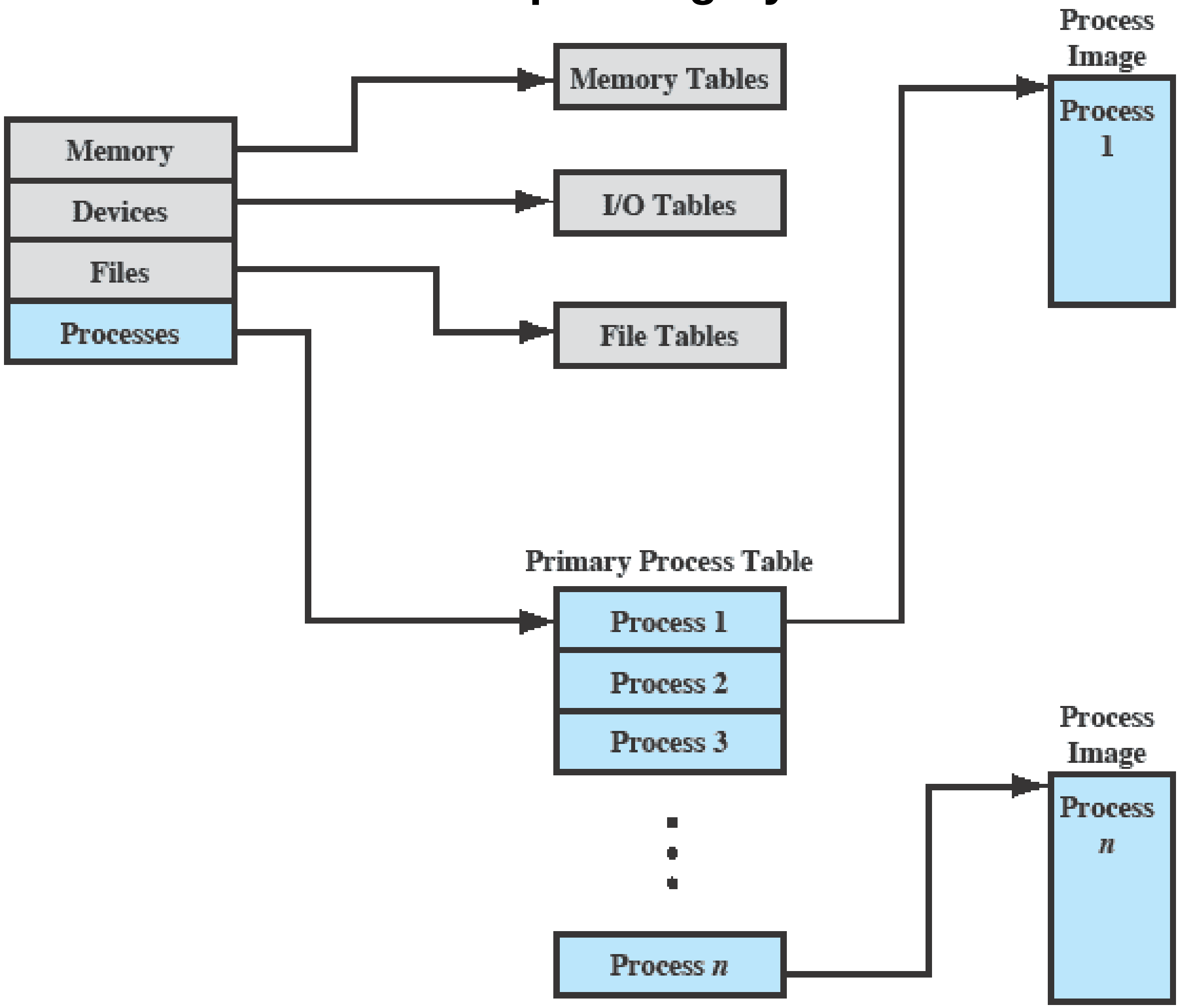
What about the globals:

```
char buf[10*MB] = {0};
char buf[10*MB] = {1};
```

Process Concept

- **Process**

General Structure of Operating System Control Tables

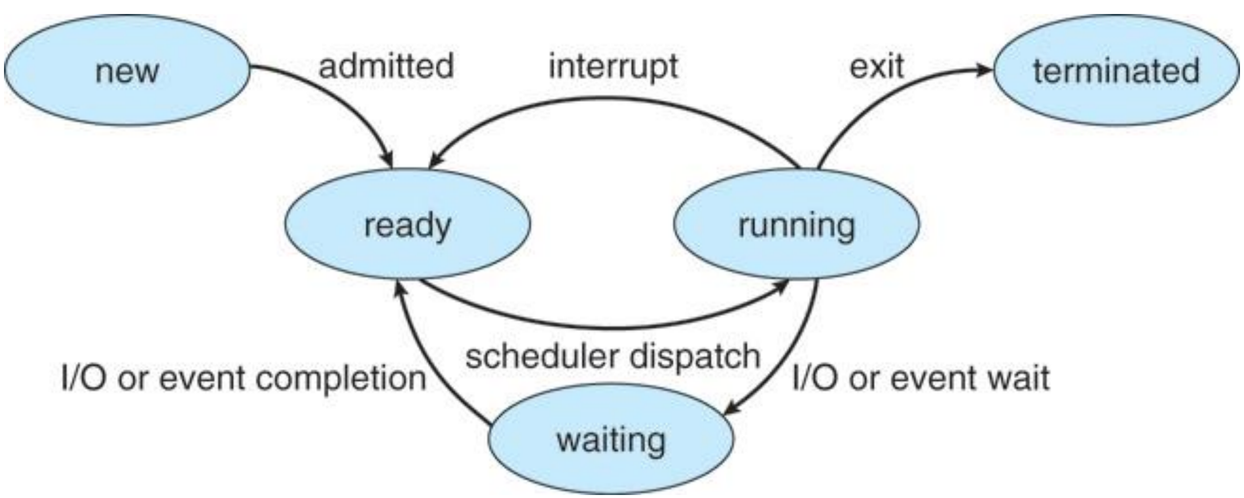


William Stallings, "Operating Systems: Internals and Design Principles"

Outline

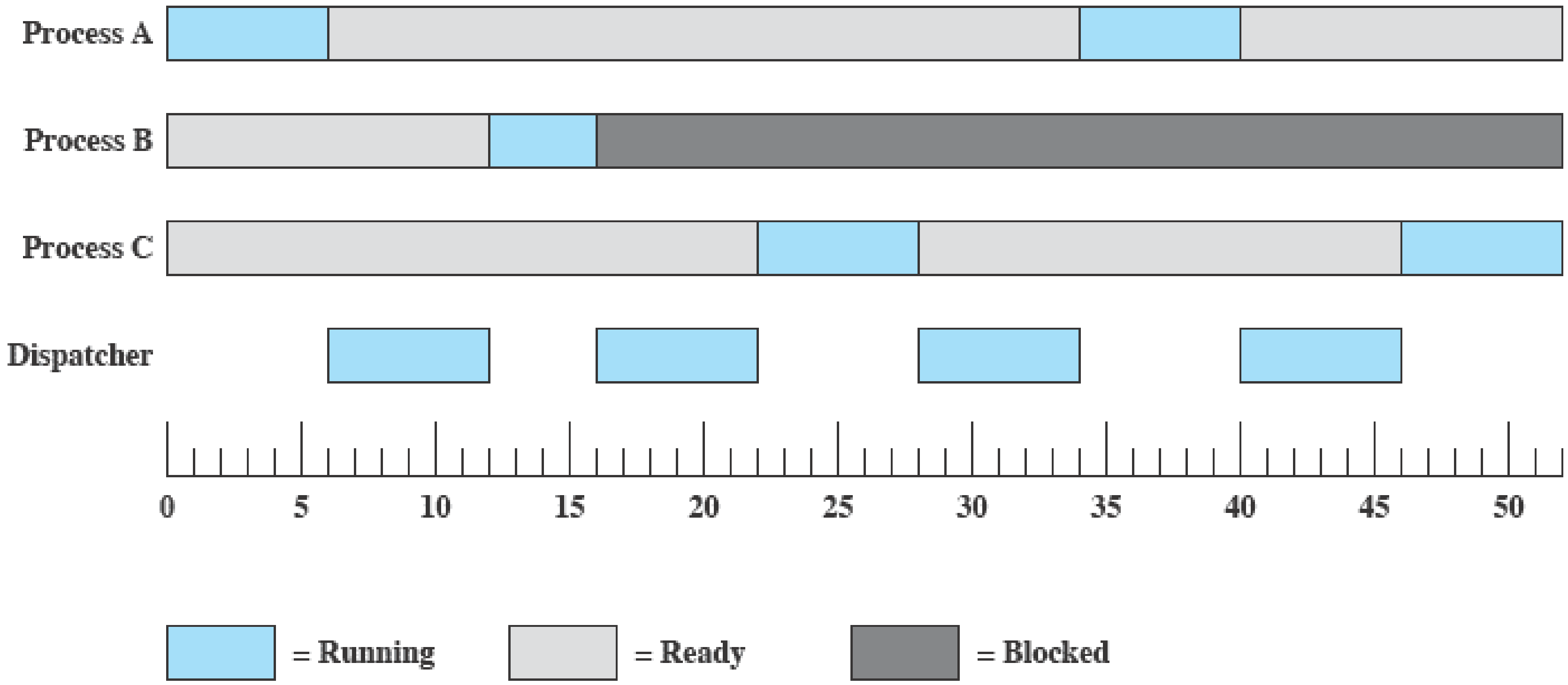
- **Process Concept**
  - process stack; data section; heap; text
- **Process State**
  - New; Running; Waiting; Ready; Terminated
- **Process Control Block (PCB)**
- **Scheduling Queues**
  - Job Queue
  - Ready Queue
  - Device Queue
- **Schedulers**
  - Long-term scheduler
  - Short-term scheduler
- **Operations in Processes**
  - Process Creation
    - fork(), exec(), wait()
  - Process Termination
    - exit(), wait()
- **References**

- **Process State**
- The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states:
  - **New**: the process is being created.
  - **Running**: instructions are being executed.
  - **Waiting**: the process is waiting for some event to occur (such as an I/O completion or reception of a signal).
  - **Ready**: the process is waiting to be assigned to a processor.
  - **Terminated**: the process has finished execution.



Abraham Silberschatz, ” Operating System Concepts”

- **Process State**



William Stallings, “Operating Systems: Internals and Design Principles”

Process State

- **Process State**

To show the state of a process, use the ps command.

```
$ ps -el
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S     0     1     0  0  75   0 -   155 schedu ?        00:00:04 init
0 S     0     2     1  0  75   0 -     0 contex ?        00:00:00 keventd
0 S     0     3     1  0  94  19 -     0 ksofti ?        00:00:00 ksoftirqd_CPU0
0 S     0     4     1  0  85   0 -     0 kswapd ?        00:00:00 kswapd
0 S     0     5     1  0  85   0 -     0 bdflus ?        00:00:00 bdflush
0 S     0     6     1  0  75   0 -     0 schedu ?        00:00:00 kupdated
0 S     0     7     1  0  85   0 -     0 kinode ?        00:00:00 kinoded
0 S     0     8     1  0  85   0 -     0 md_thr ?        00:00:00 mdrecoveryd
0 S     0    11     1  0  75   0 -     0 schedu ?        00:00:00 kreiserfsd
0 S     0   386     1  0  60 -20 -     0 down_i ?        00:00:00 lvm-mpd
0 S     0   899     1  0  75   0 -   390 schedu ?        00:00:00 syslogd
```

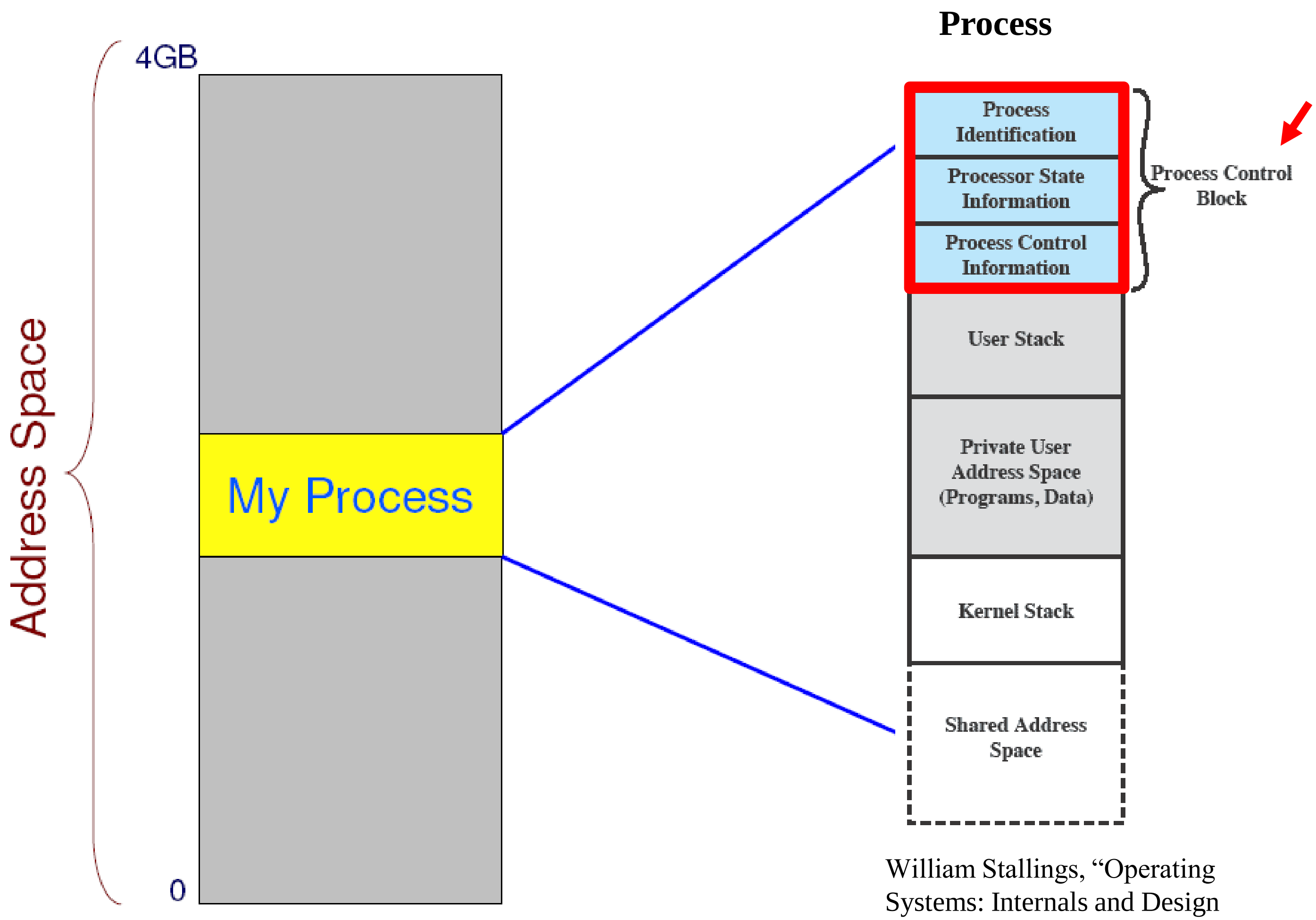
symbol	meaning
S	sleeping
R	running
D	waiting (usually for IO)
T	stopped (suspended) or traced
Z	zombie (defunct)

Outline

- **Process Concept**
  - process stack; data section; heap; text
- **Process State**
  - New; Running; Waiting; Ready; Terminated
- **Process Control Block (PCB)**
- **Scheduling Queues**
  - Job Queue
  - Ready Queue
  - Device Queue
- **Schedulers**
  - Long-term scheduler
  - Short-term scheduler
- **Operations in Processes**
  - Process Creation
    - fork(), exec(), wait()
  - Process Termination
    - exit(), wait()
- **References**

Process Control Block

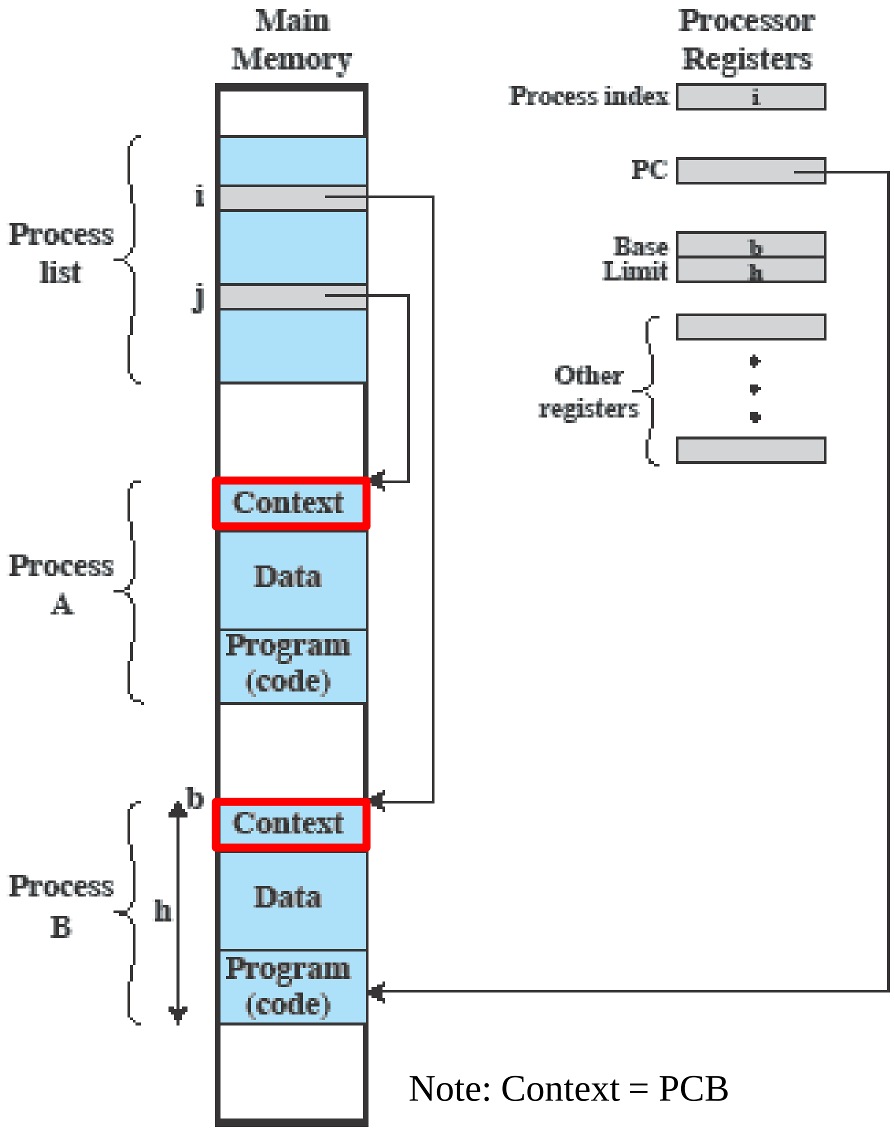
- **Process Control Block (PBC)**



William Stallings, "Operating Systems: Internals and Design Principles"

Process Control Block

- **Process Control Block (PBC)**



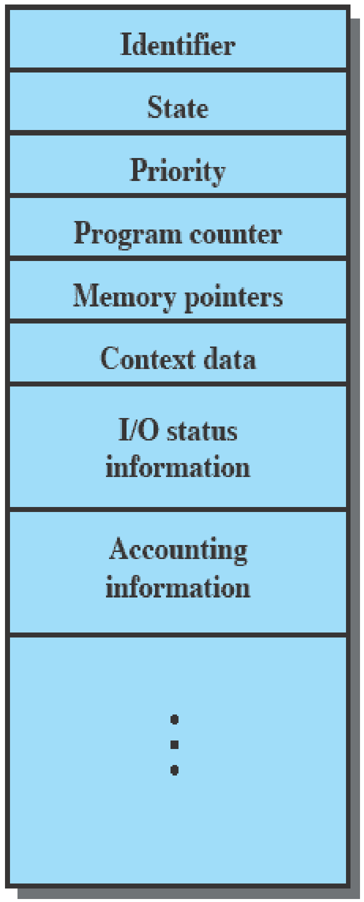
Note: Context = PCB

William Stallings, "Operating Systems: Internals and Design Principles"

Process Control Block

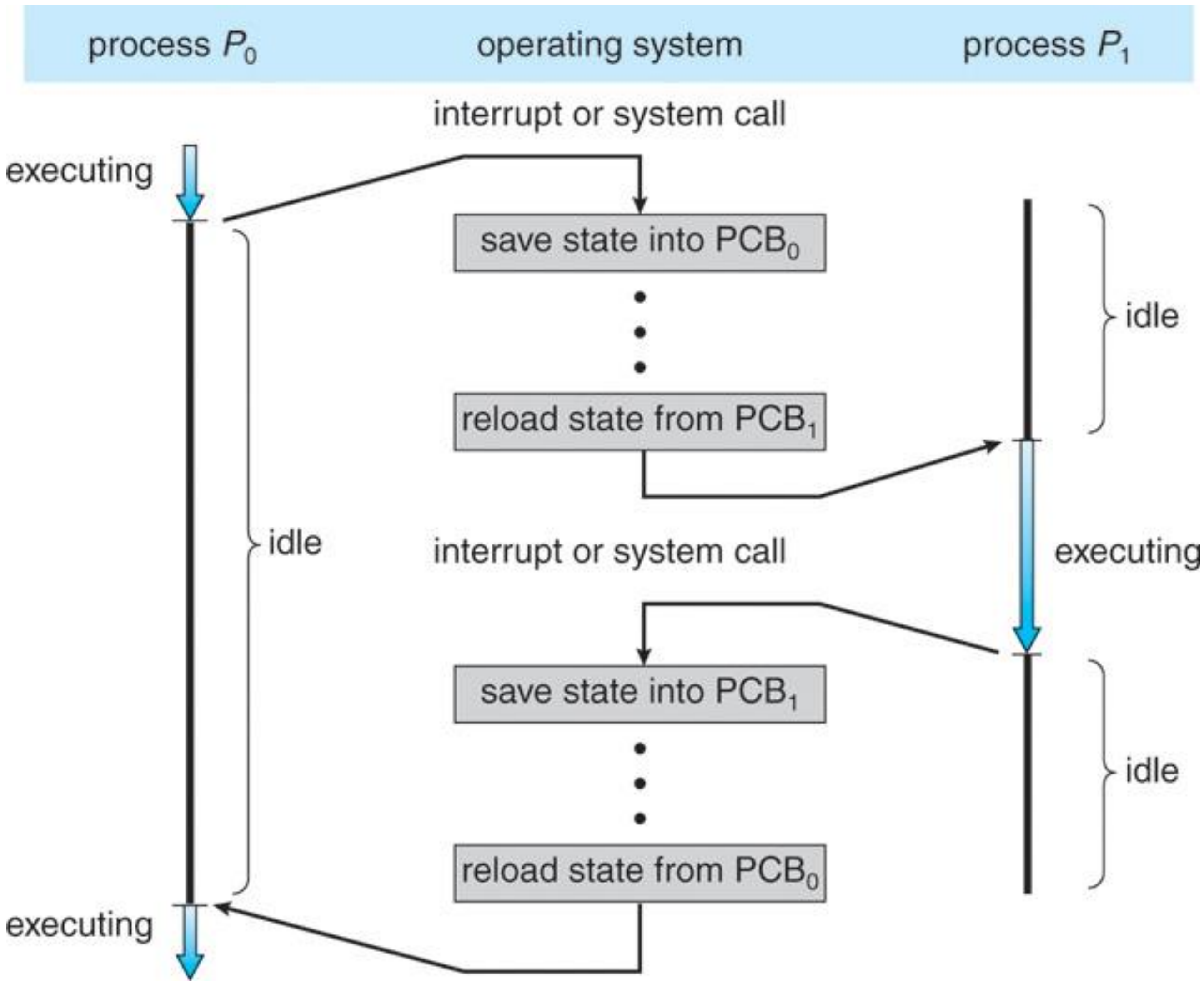
- **Process Control Block (PBC)**
- Each process is represented in the operating system by a *Process Control Block* (PCB) - also called a task control block.
  - **Process state**: the state may be new, ready, running, waiting
  - **CPU-scheduling information:** includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
  - **Program counter**: The counter indicates the address of the next instruction to be executed for this process.
  - **Memory-management information**: includes the value of the base and limit registers, the page tables, or the segment tables,
  - **CPU registers**: Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterwards
  - **I/O status information:** includes the list of I/O devices allocated to the process, a list of open files, and so on.
  - **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

PCB



William Stallings, “Operating Systems: Internals and Design Principles”

- **Process Control Block (PBC)**

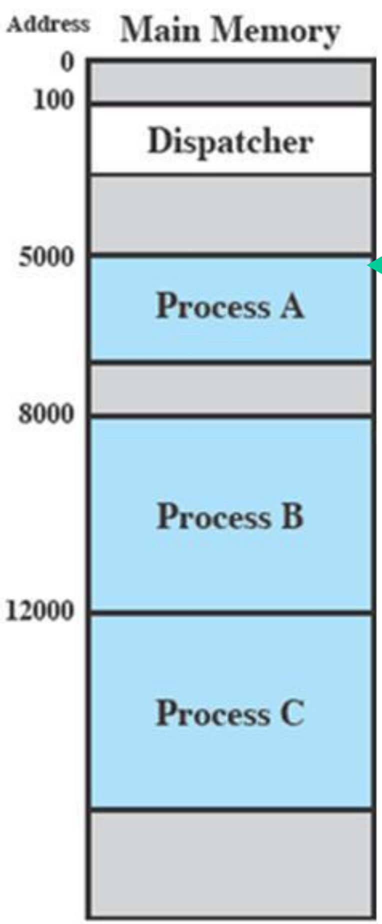


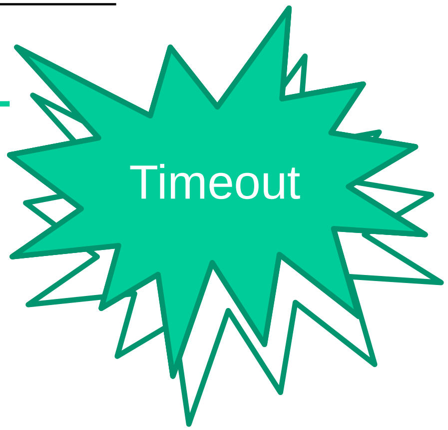
Abraham Silberschatz, ” Operating System Concepts”

Process Concept

- **Example**

Dispatcher is a small program which <u>switches the processor from one process to another</u>





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

100 = Starting address of dispatcher program

Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

William Stallings, "Operating Systems: Internals and Design Principles"

Process Concept

- **Example**

(a) Trace of Process A	(b) Trace of Process B	(c) Trace of Process C
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

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

100 = Starting address of dispatcher program

Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

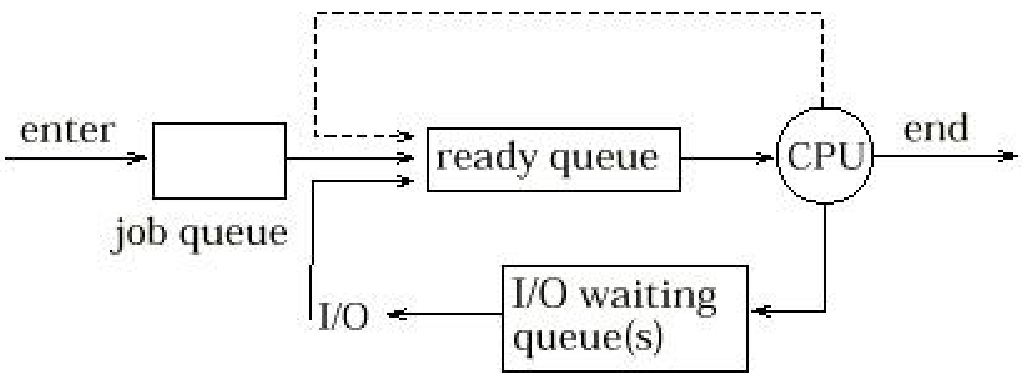
William Stallings, "Operating Systems: Internals and Design Principles"

Outline

- **Process Concept**
  - process stack; data section; heap; text
- **Process State**
  - New; Running; Waiting; Ready; Terminated
- **Process Control Block (PCB)**
- **Scheduling Queues**
  - Job Queue
  - Ready Queue
  - Device Queue
- **Schedulers**
  - Long-term scheduler
  - Short-term scheduler
- **Operations in Processes**
  - Process Creation
    - fork(), exec(), wait()
  - Process Termination
    - exit(), wait()
- **References**

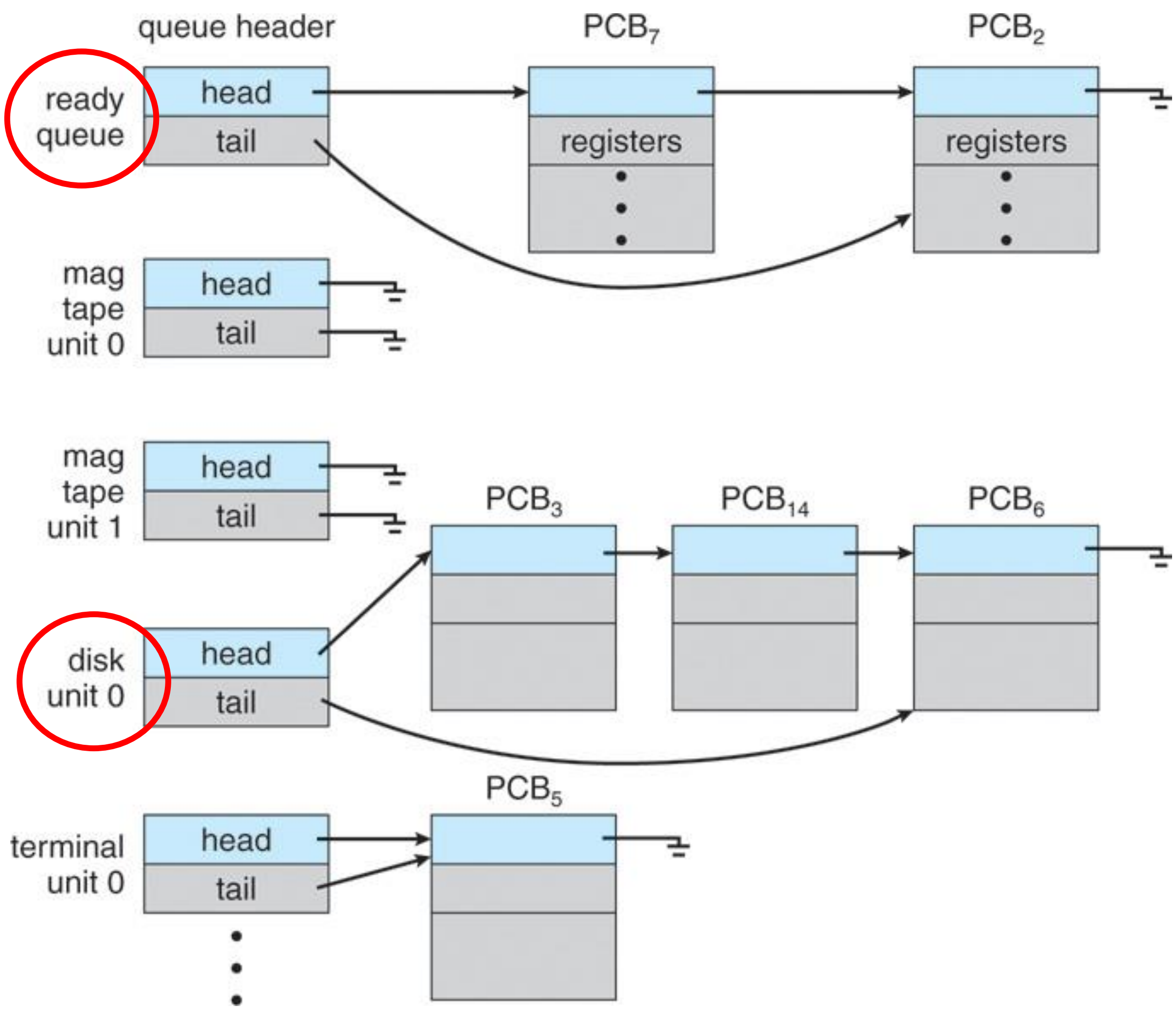
Scheduling Queues

- **Scheduling Queues**
  - **Job Queue**
    - As processes enter the system, they are put into a **job queue**, which consists of all processes in the system.
  - **Ready Queue**
    - The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**. This queue is generally stored as a linked list.
    - A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.
  - **Device Queue**
    - The list of processes waiting for a particular I/O device is called a **device queue**. Each device has its own device queue.



- **Scheduling Queues**

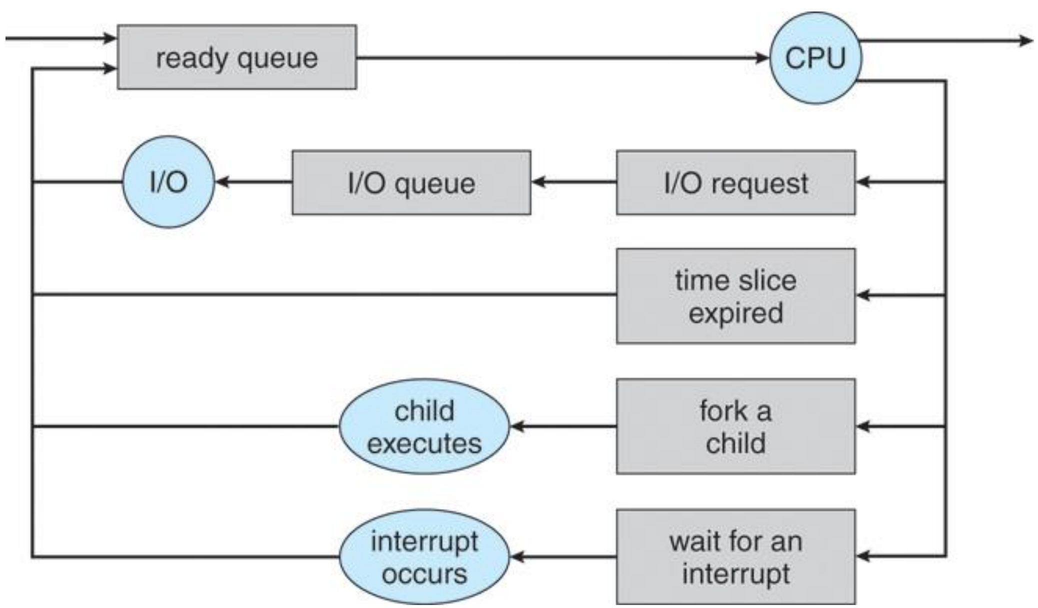
The ready queue and various I/O device queues.



Abraham Silberschatz, " Operating System Concepts"

Scheduling Queues

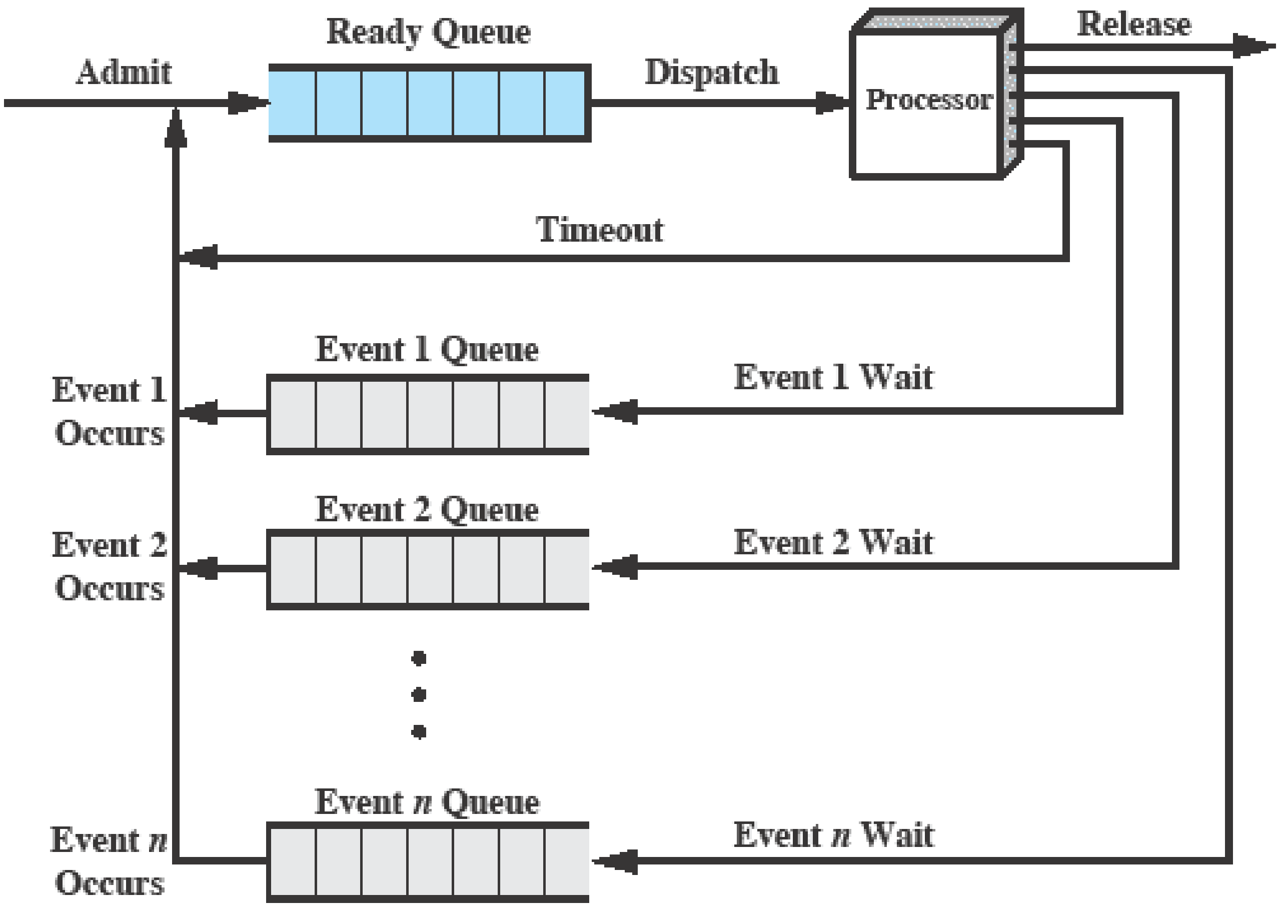
- **Queueing diagram:**
  - A new process is initially put in the ready queue. It waits there until it is selected for execution, or is dispatched.
  - Once the process is allocated the CPU and is executing, one of several events could occur:
    - could issue an I/O request and then be placed in an I/O queue.
    - could create a new subprocess and wait for the subprocess's termination.
    - could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.



Abraham Silberschatz, " Operating System Concepts"

Scheduling Queues

- **Queueing diagram:**

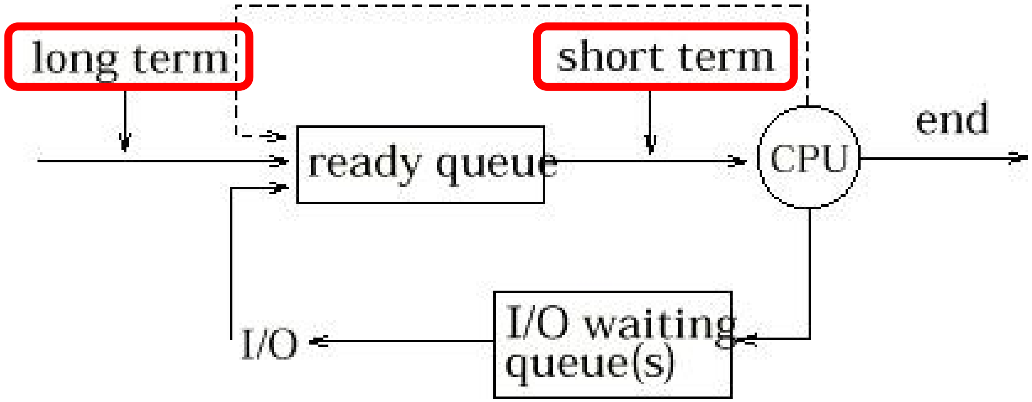


William Stallings, "Operating Systems: Internals and Design Principles"

Outline

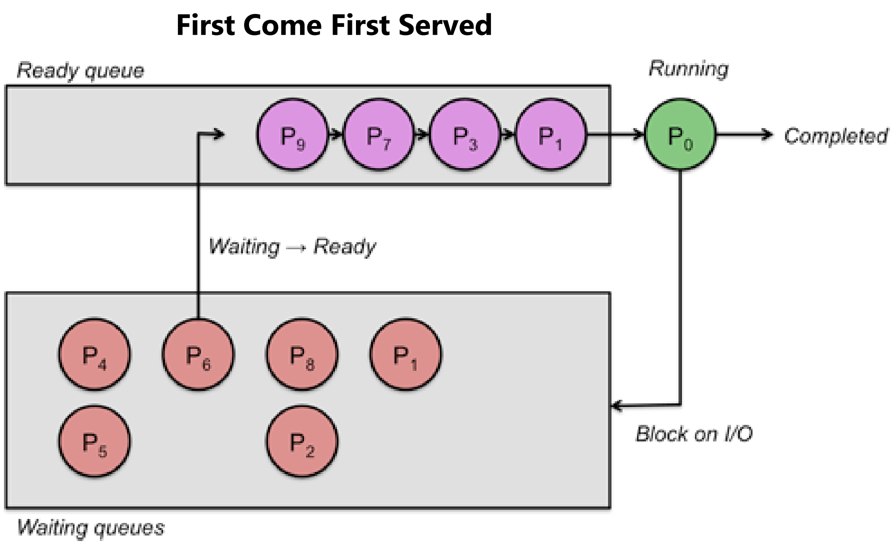
- **Process Concept**
  - process stack; data section; heap; text
- **Process State**
  - New; Running; Waiting; Ready; Terminated
- **Process Control Block (PCB)**
- **Scheduling Queues**
  - Job Queue
  - Ready Queue
  - Device Queue
- $\boxed{\textbf{Schedulers}}$
  - Long-term scheduler
  - Short-term scheduler
- **Operations in Processes**
  - Process Creation
    - fork(), exec(), wait()
  - Process Termination
    - exit(), wait()
- **References**

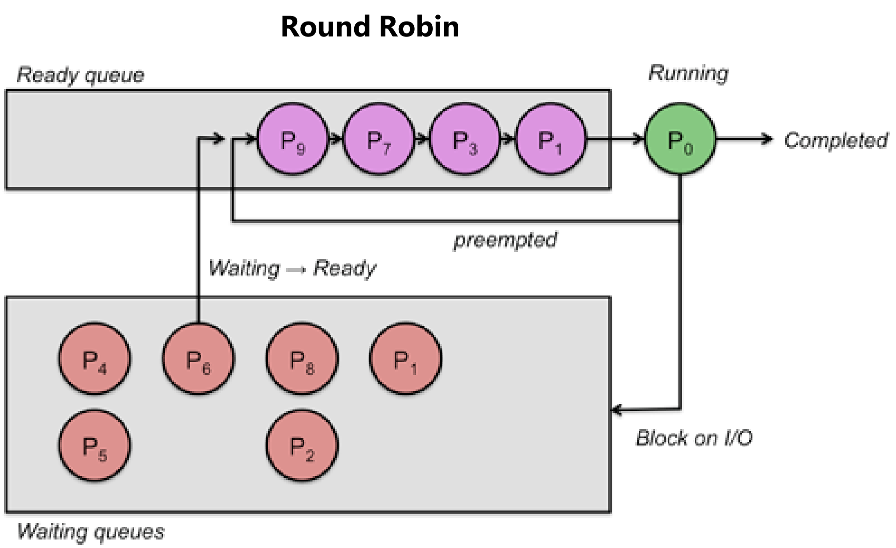
- **Schedulers**
  - A process migrates among the various scheduling queues throughout its lifetime. The selection from these queues is carried out by the appropriate scheduler.
  - **long-term scheduler**:
    - processes are <u>spooled to a mass-storage device (typically a disk), where they are kept for later execution</u>. The long-term scheduler will select processes from this pool and loads them into memory for execution.
  - **short-term scheduler**, or CPU scheduler,
    - selects from among the <u>processes that are ready to execute and allocates the CPU to one of them</u>.
  - <u>It is important that the long-term scheduler select a good process mix of I/O-bound and CPU-bound processes</u>



Schedulers

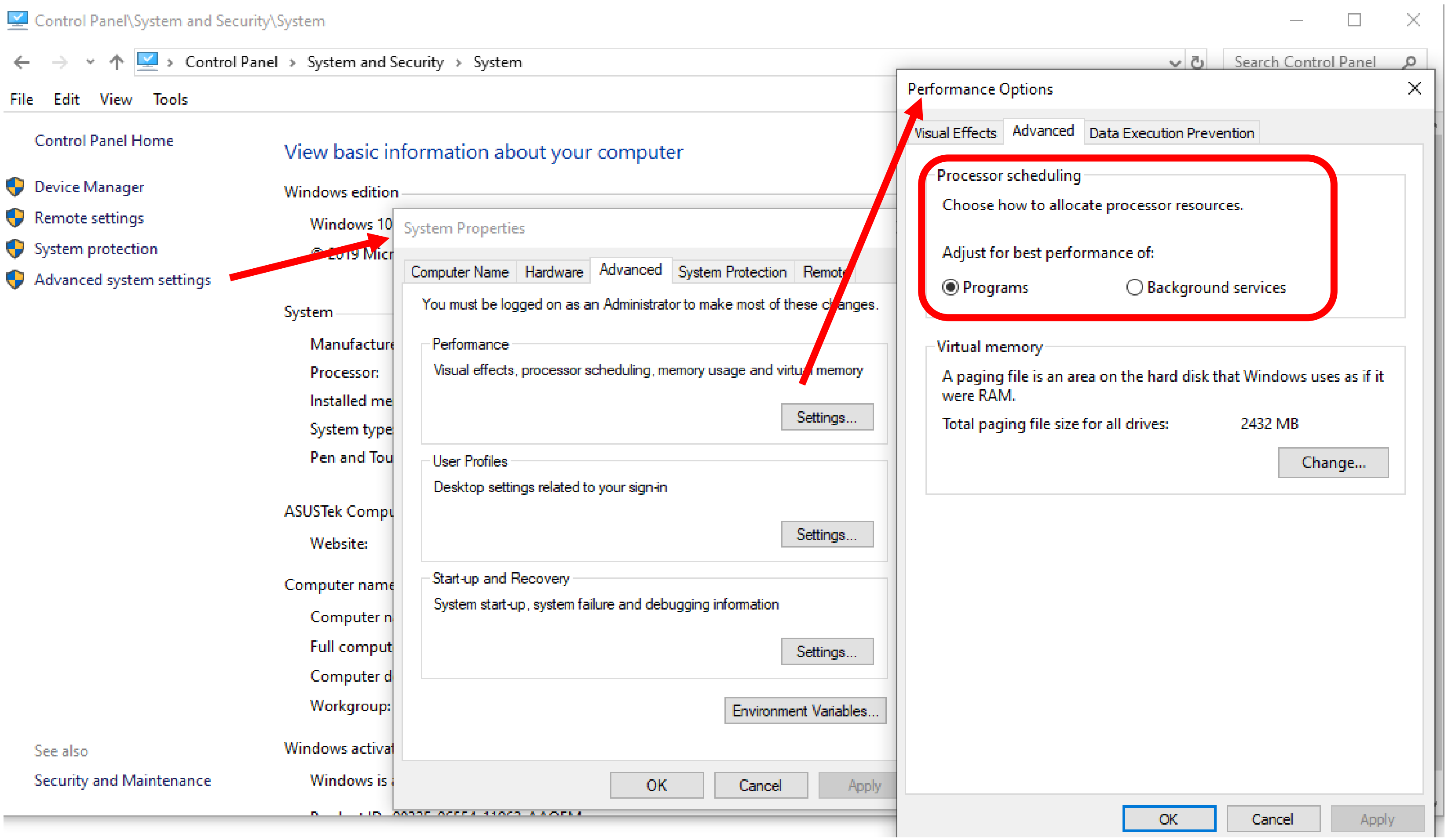
- **Schedulers**
  - **What´s the best scheduler? It depends:**
    - Server Operating Systems
    - User Operating Systems
    - Real-time Operating Systems
    - Embedded Operating Systems
    - Multimedia Operating Systems
    - ….





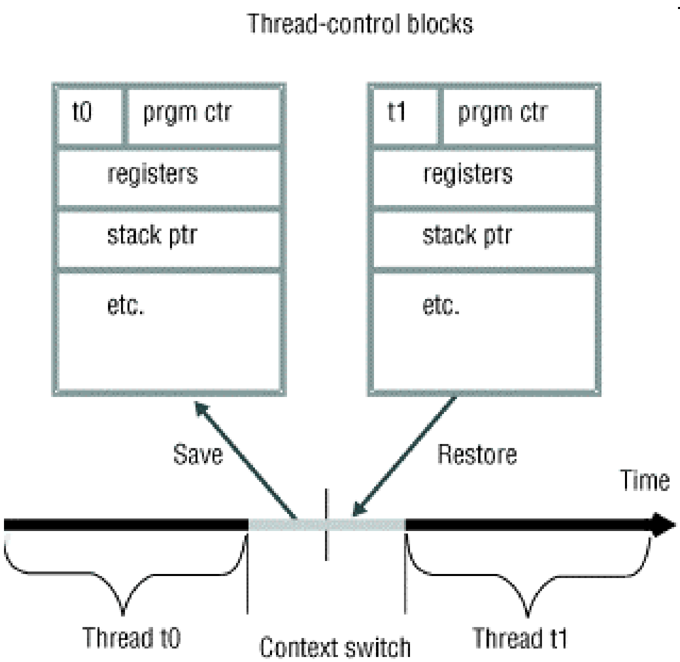
https://www.cs.rutgers.edu/~pxk/416/notes/07-scheduling.html

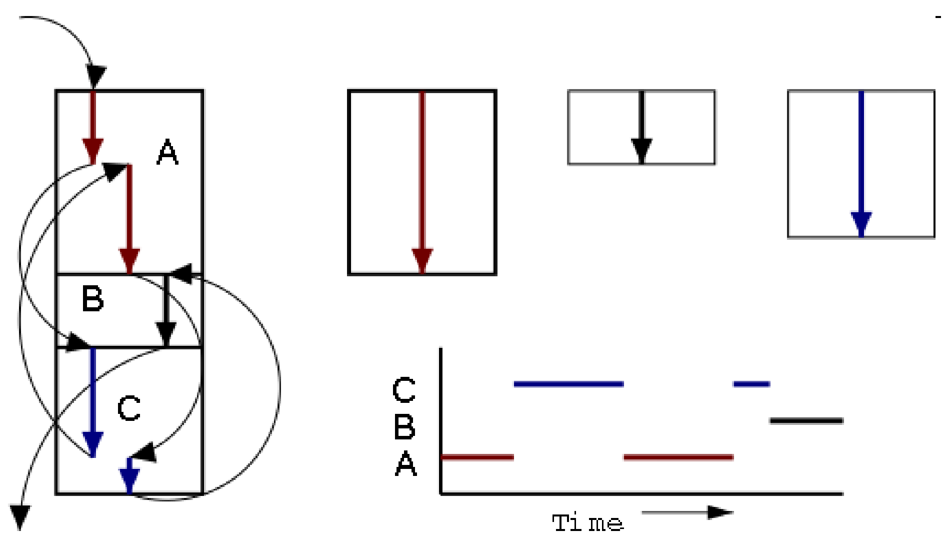
- **Schedulers**
  - **Even in Windows!**



Control Panel » System Security » System » Advanced System Settings » Advanced » Performance » Settings » Advanced

- **Schedulers**
- **Context switch**
  - when changing of running process, the system needs to save the current context of the process currently running on the CPU so that it can restore that context when resuming it
  - The **context** is represented in the PCB of the process, and it includes the value of the CPU registers, the process state, and memory-management information.





Schedulers

- **Schedulers**
- **Context switch**
  - A process switch also occurs when the OS gains control from the currently running process. Possible events giving OS control are:

Mechanism	Cause	Use
Interrupt	External to the execution of the current instruction	Reaction to an asynchronous external event
Trap	Associated with the execution of the current instruction	Handling of an error or an exception condition
Supervisor call	Explicit request	Call to an operating system function

William Stallings, “Operating Systems: Internals and Design Principles”

Schedulers

- **Schedulers**
- **Why Event Driven… "A Human perspective"**

Characteristic			Scaled to Human Time	
CPU frequency	*2GHz*			
Processor Cycle Time	0.5	ns	1	s
L2 cache access	10	ns	20	s
Memory access	80	ns	160	s (2.6 mins)
Thread context switch	5000	ns (5us)	10000	s (2.7 hours)
Disk access	8000000	ns (8ms)	16000000	s (185 days)
Process quantum	100000000	ns (100ms)	200000000	s (6.3 years)
In blue ► Things improving very fast				
In orange ► Things improving to a degree				
In red ► Things not really improving				

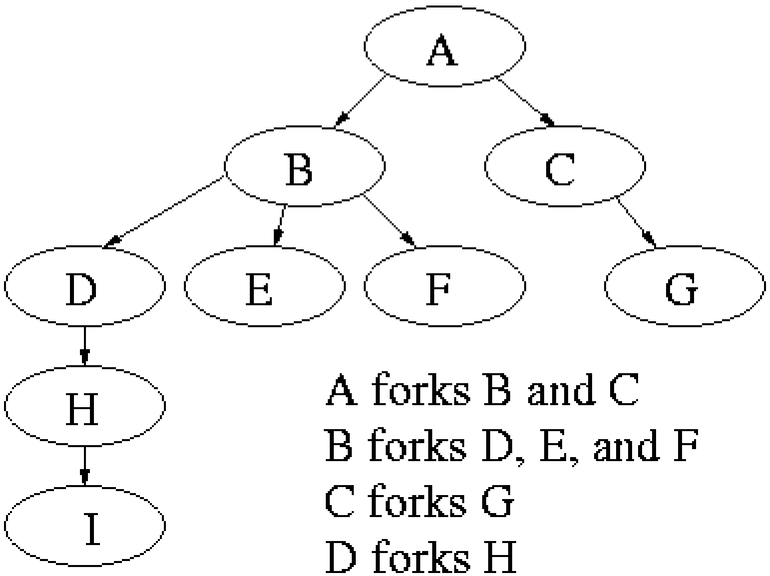
Outline

- **Process Concept**
  - process stack; data section; heap; text
- **Process State**
  - New; Running; Waiting; Ready; Terminated
- **Process Control Block (PCB)**
- **Scheduling Queues**
  - Job Queue
  - Ready Queue
  - Device Queue
- **Schedulers**
  - Long-term scheduler
  - Short-term scheduler
- **Operations in Processes**
  - Process Creation
    - fork(), exec(), wait()
  - Process Termination
    - exit(), wait()
- **References**

Operations in Processes

- **Process Creation**
  - A process may create several new processes, during the course of execution.
  - The creating process is called a **parent process**, and the new processes are called the **children** of that process.
  - Each of these new processes may in turn create other processes, forming a **tree** of processes.

Process Hierarchies



A forks B and C
B forks D, E, and F
C forks G
D forks H
H forks I

In Unix the **fork** system call creates a child process, and the **exit** system call terminates the current process.

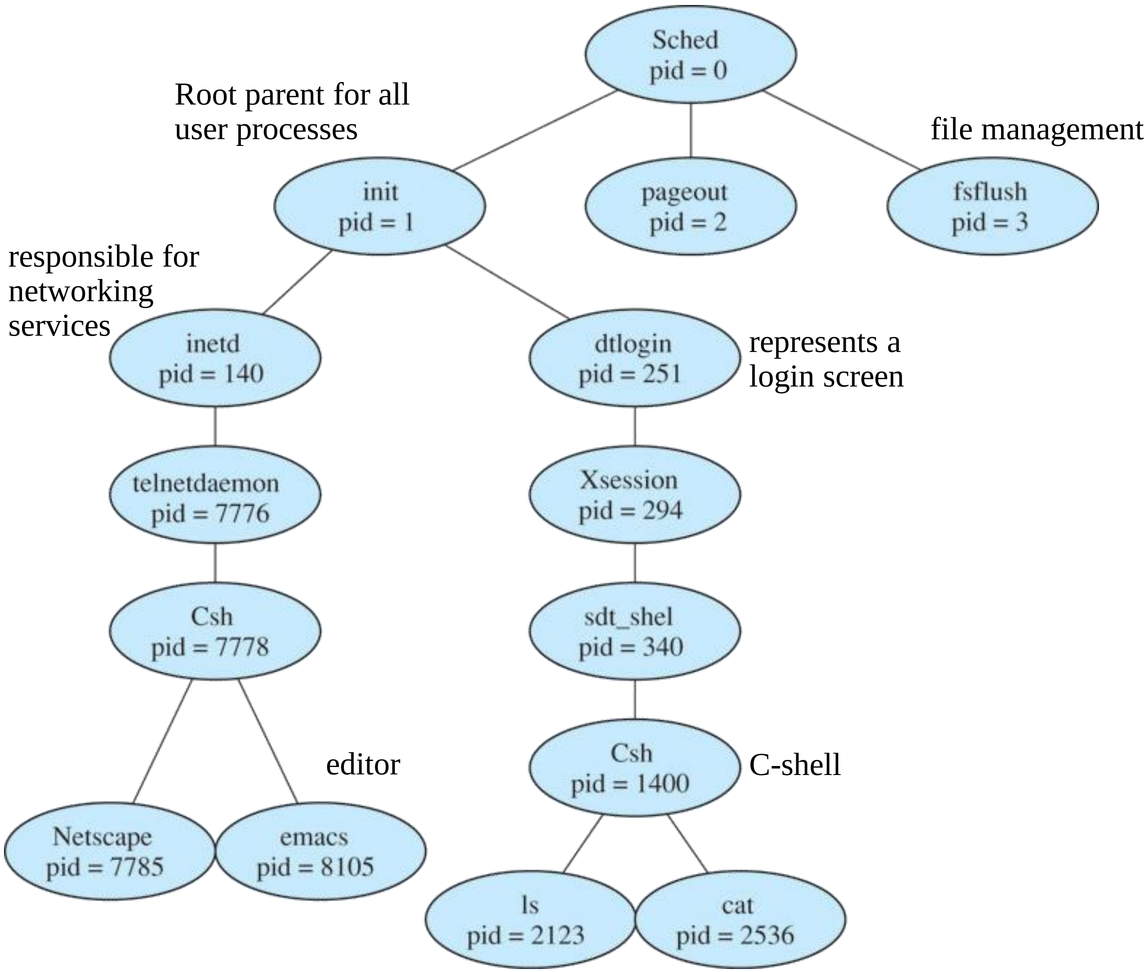
1) After a `fork` both parent and child keep running (indeed they have the same program text) and each can fork off other processes.

2) A process can choose to **wait** for children to terminate. For example, if C issued a `wait()` system call it would block until G finished.

Operations in Processes

- **Process Creation**

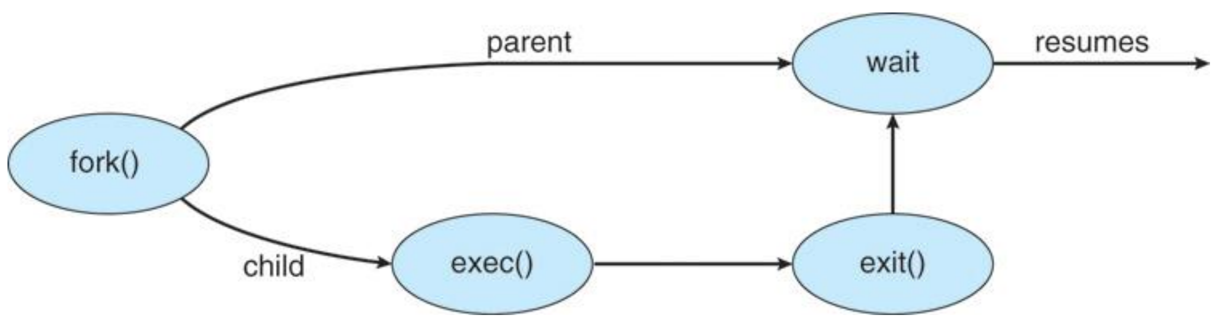
A tree of processes on a typical Solaris system.



Abraham Silberschatz, " Operating System Concepts"

Operations in Processes

- **Process Creation**
  - Most operating systems (including UNIX and the Windows family of operating systems) identify processes according to a unique **Process Identifier** (or **PID**), which is typically an integer number.
  - When a process creates a new process, **two possibilities exist in terms of execution:**
    - 1. The <u>parent continues to execute concurrently</u> with its children.
    - 2. The <u>parent waits until some or all of its children have terminated</u>.



Abraham Silberschatz, " Operating System Concepts"

- **Process Creation**

Fork: this system call creates a new process.
The return code for the fork is:
- **zero** for the new (child) process;
- **nonzero process identifier** of the child for the parent process.

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

int main(){
   pid_t pid;
   // fork a child process
   pid = fork();
   if (pid < 0) { // error ocurred
        fprintf(stderr, "Fork Failed");
        exit(-1);
   }
   else if (pid == 0) { //child process
        execlp("/bin/ls", "ls", NULL);
   }
   else { //parent process
        //parent will wait for the child to complete
        wait(NULL);
        printf("Child Complete");
        exit(0);
   }
}
```

Operations in Processes

- **Process Creation**
  - **Fork:** this system call creates a new process which consists of a copy of the address space of the original process. Both processes (the parent and the child) continue execution at the instruction after the fork, with a difference. The return code for the fork is:
    - **zero** for the new (child) process;
    - **nonzero process identifier** of the child for the parent process.
  - **Exec:** this system call can be used to replace the process's memory space with a new program:
    - loads a binary file into memory (destroying the memory image of the program containing the exec system call) and starts its execution.
  - **Wait:** if it has nothing else to do while the child runs, the parent process can issue a wait system call to move itself off the ready queue until the termination of the child.

Operations in Processes

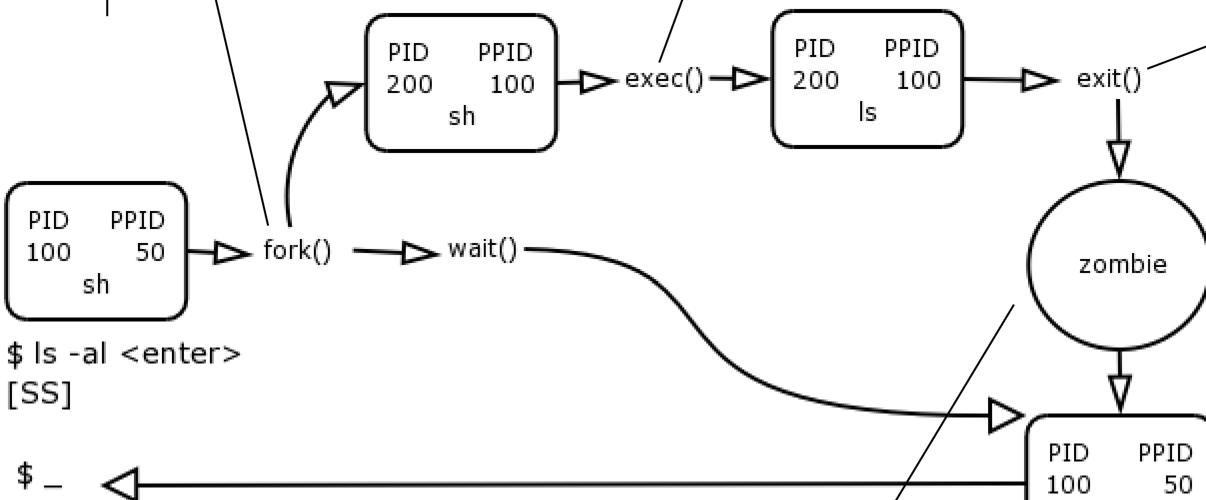
- **Process Creation**

What it does first is to issue a fork() system call. Which performs a type of cloning operation. This gets the kernel to copy an existing process table entry to a next empty slot in the process table. This effectively creates a template for a new process.

The kernel now assigns a new unique PID to this forked process and updates its PPID to reflect the value of the process that forked it.

The exec() system call gets the kernel to read the ls program off the filesystem on the hard disk and place it into memory, overwriting the calling process, in this case the child shell template.

As you hit enter after the "ls -al" command, the shell determines that ls is an external program on the filesystem, namely /bin/ls. It needs to exec() this.

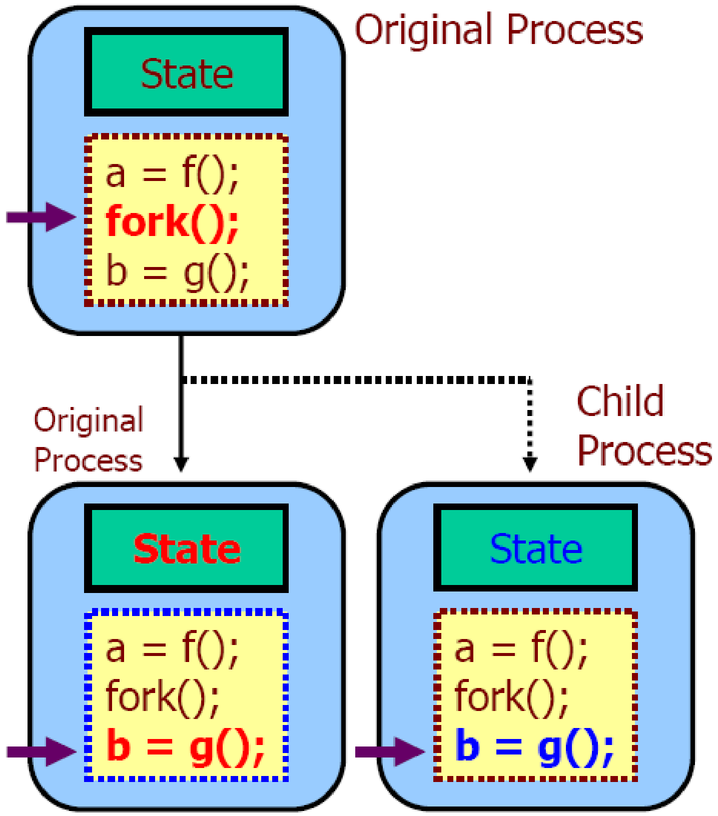


The exit() of the child actually causes the return of the wait() system call, which ends the pausing of the parent process, so that it can now continue running.

The kernel has de-allocated the process memory, however its process table entry still exists. It is the job of the parent to inform the kernel that it has finished working with the child, and that the kernel can now remove the process table entry from the child

Operations in Processes

- **Process Creation**
  - **Process Model Unix**
  - Process creation in Unix is based on <u>spawning child processes which inherit all the characteristics of their fathers:</u>
    - 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.



Operations in Processes

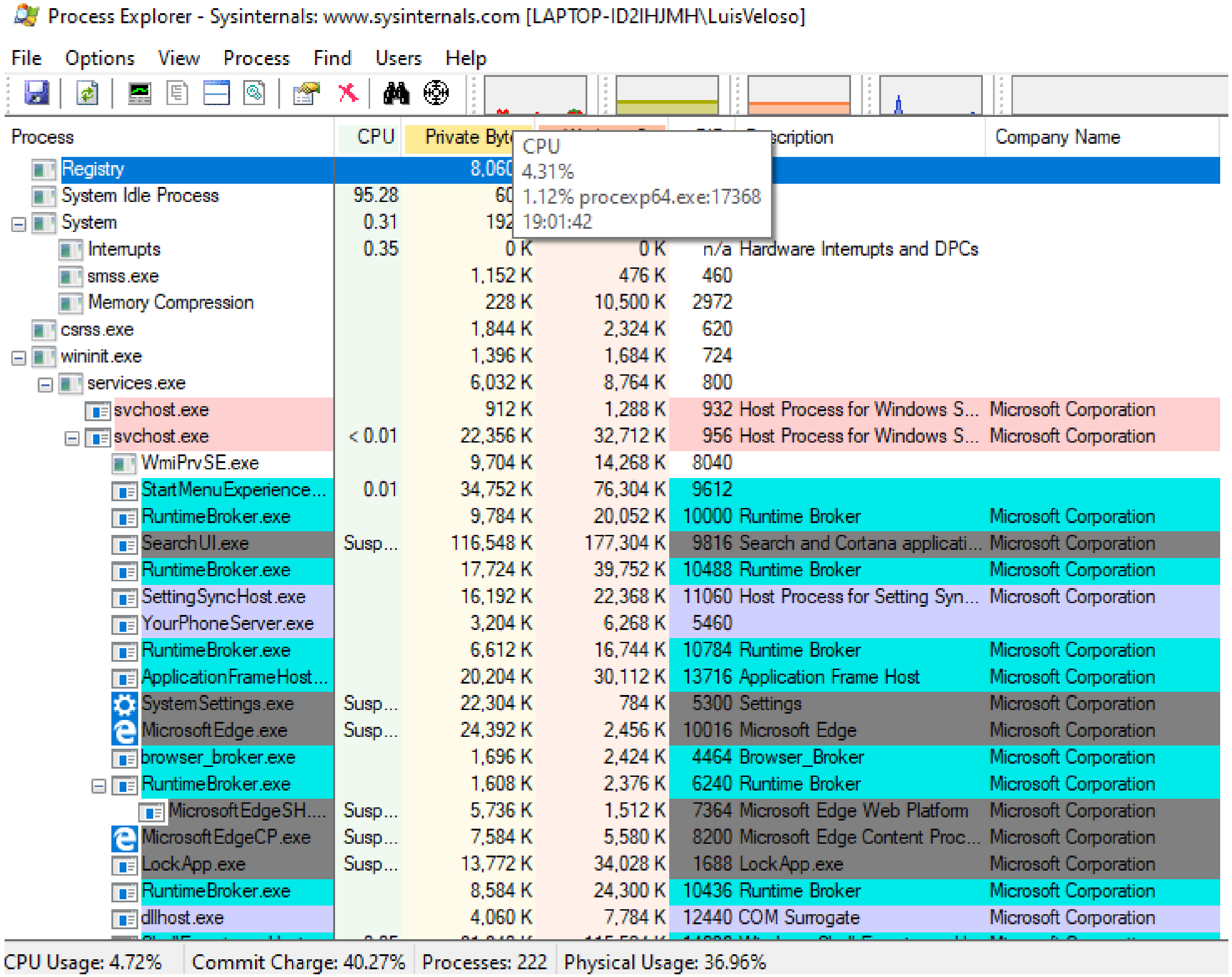
- **Process Creation**

"**init**" is the first process run by the kernel at bootup: this behavior is hard-coded in the kernel. It is "init's" job to start up various child processes to get the system to a usable state

```
$ ps -ef | less
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY        TIME CMD
0 S     0     1     0  0  75   0 -   155 schedu ?      00:00:04 init
0 S     0     2     1  0  75   0 -     0 contex ?      00:00:00 keventd
0 S     0     3     1  0  94  19 -     0 ksofti ?      00:00:00 ksoftirqd_CPU0
0 S     0     4     1  0  85   0 -     0 kswapd ?      00:00:00 kswapd
0 S     0     5     1  0  85   0 -     0 bdflus ?      00:00:00 bdflush
0 S     0     6     1  0  75   0 -     0 schedu ?      00:00:00 kupdated
0 S     0     7     1  0  85   0 -     0 kinode ?      00:00:00 kinoded
0 S     0     8     1  0  85   0 -     0 md_thr ?      00:00:00 mdrecoveryd
0 S     0    11     1  0  75   0 -     0 schedu ?      00:00:00 kreiserfsd
0 S     0   386     1  0  60 -20 -     0 down_i ?      00:00:00 lvm-mpd
0 S     0   899     1  0  75   0 -   390 schedu ?      00:00:00 syslogd
0 S     0   902     1  0  75   0 -   593 syslog ?      00:00:00 klogd
```

the PPID column shows the parent process of the program.

- **Process Creation**



Process Explorer from SysInternals for Windows
https://docs.microsoft.com/pt-pt/sysinternals/

Operations in Processes

- **Process Termination**
  - A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the ***`exit()`*** system call.
    - the process may <u>return a status value (typically an integer) to its parent process via the ***`wait()`*** system call</u>.
    - all the resources of the process (including physical and virtual memory, open files, and I/O buffers) are deallocated by the operating system.
  - <u>A process can cause the termination of another process</u> via an appropriate system call:
    - usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs.

Operations in Processes

- **System Calls**

Process Management		
Posix	**Win32**	**Description**
fork	CreateProcess	Clone current process
exec		Replace current process
wait	WaitForSingleObject	Wait for a child to terminate.
exit	ExitProcess	Terminate current process & return status
File Management		
Posix	**Win32**	**Description**
open	CreateFile	Open a file & return descriptor
close	CloseHandle	Close an open file
read	ReadFile	Read from file to buffer
write	WriteFile	Write from buffer to file
lseek	SetFilePointer	Move file pointer
stat	GetFileAttributesEx	Get status info

References

- Abraham Silberschatz, ” Operating System Concepts”, 10th Edition, Wiley, 2018
- William Stallings, “Operating Systems: Internals and Design Principles”, 9th Edition, Pearson, 2017
- Andrew S. Tanenbaum, “Modern Operating Systems”, 4th Edition, Pearson Education, 2014