toc:

Process Description and Control

Housekeeping

No cheatsheet for midterm and final b/c it's multisection and the other section profs don't want to do cheatsheets.

What is a Process

Computers used to only execute 1 program at a time. This program had complete control of the system and access to all it's resources.

Now they can have multiple programs loaded into memory and executed at the same time.

With that evolution we needed more control and compartmentalization of running programs.

Thus came the birth of the process.

A process is:

- a program in execution
- the unit of owrk in a modern computing system
- an entity that can be assigned to a processor and executed on a processor
- a unit of activity characterized by
    - the execution of a sequence of instructions
    - having a current state
    - having an associated set of system resources

2 essential components of processes:

- program code
    - text section
- set of data associated with that code
    - data section

We may be more precise later

A process can be uniquely characterized by anumber of elements:

- identifier
    - unique identifier associated w/ process to distinguish from all other processes
    - pid - process ID unique to a process
    - no 2 processes have the same pid
- state
    - if a prcess is currently executing it is in the `running` state
- priority
    - priority relative to other processes
    - what processes can it interrupt and what processes can interrupt it
- program counter
    - the address of the next instruction in the program to be executed
- memory pointers

    - includes pointers to the program code and data associated witht he process plus any memory blocks shared with other processes
- context data
    - data that are present in the registers during execution
- i/o status information
    - outstanding i/o requests
    - assigned devices
    - list of files used in the process
- accounting information
    - may include
        - amount of processor time
        - clock time used
        - time limits
    - helps us log activities
    - find out who/what accessed things, when and why

This is all very general, every OS has their own implementations but most every OS is going to have these things just under different names.

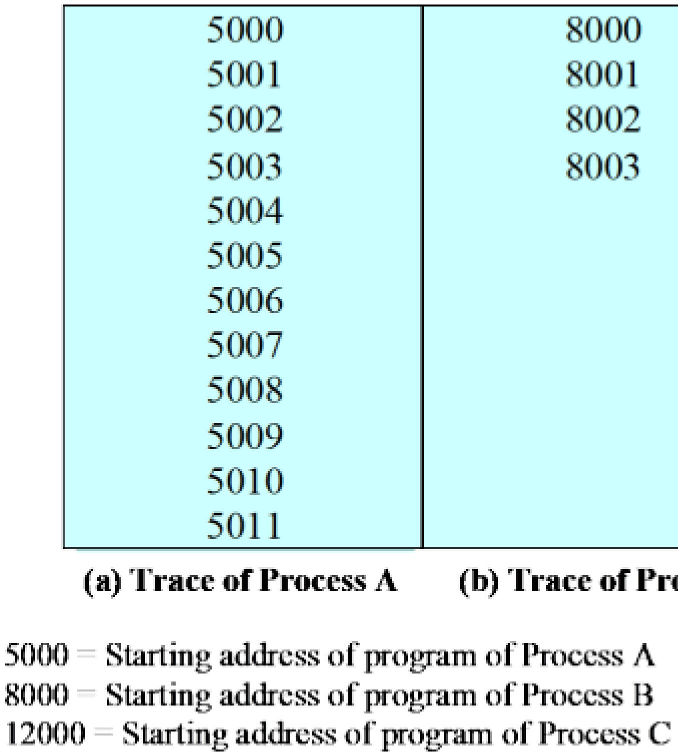
All of the above is stored in the process control block (PCB):

- created and managed by the operating system
- makes it possible to interrupt a running process and later resume execution as if the interruption had not occured (context-switching)
    - context-switching is what this course is all about
        - trying to keep the processor busy and feed it new processes in order to increase the performance and allat
        - the speed of the processor is the speed limit of the computer, no one is faster than the processor. less of a road speed limit and more of a physical speed limit, like the speed of light

Each process has only one program counter and thread for now. We talk about multithreading and parallelism later on.

Process States

Process Trace



(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

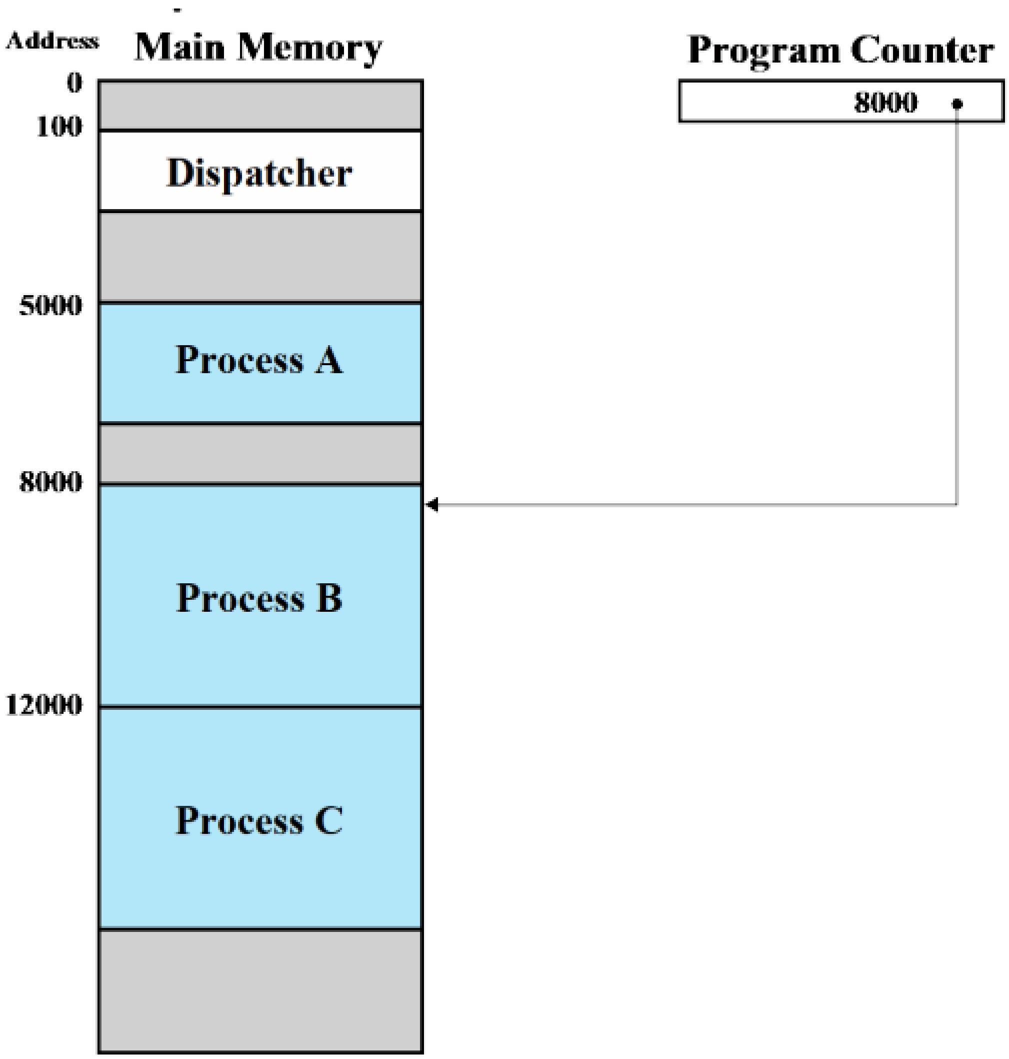
Traces of Processes

Trace:

- Characterize the behavior of an individual process by listing the sequence of instructions that execute for that process
- *set of instructions for a process*
- show how the traces of the various processes are interleaved

---

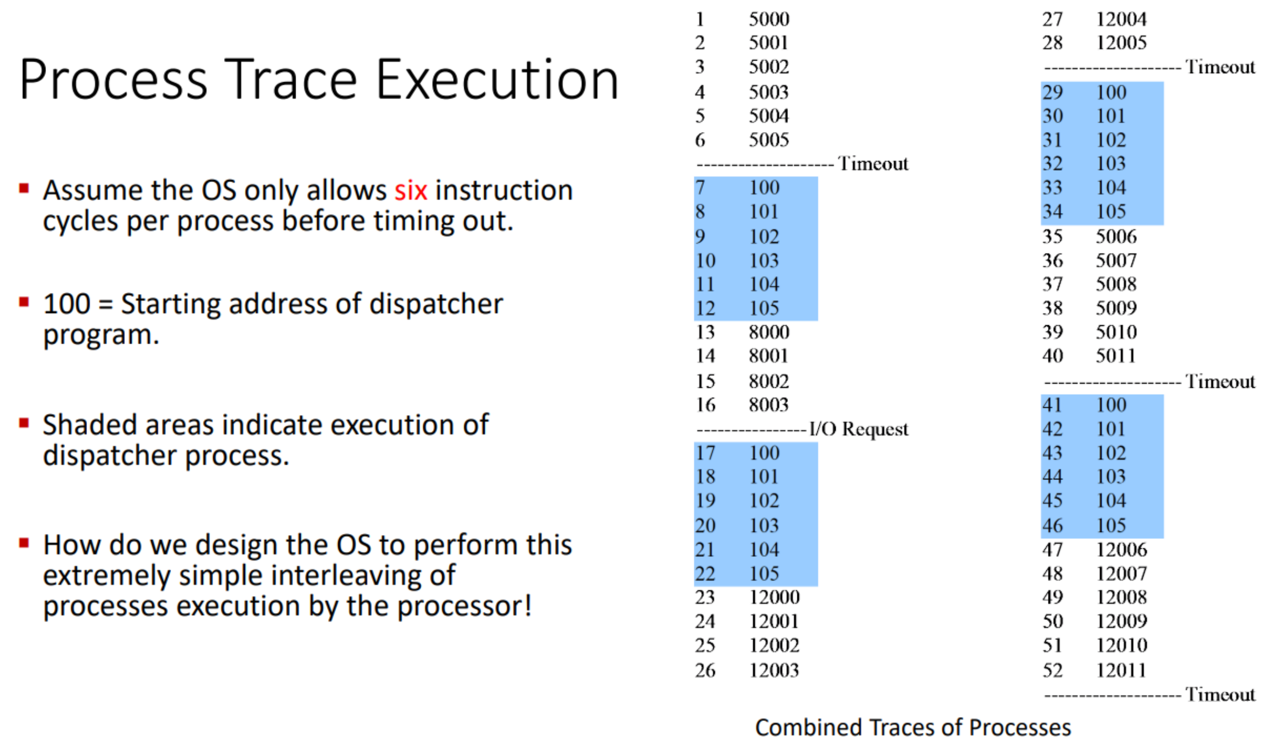
Example



Snapshot of Execution at Instruction Cycle - Example

We have a small dispatcher process that switches the process from one process to another.

Process Trace Execution



Process Trace Execution

- Assume the OS only allows six instruction cycles per process before timing out.
- 100 = Starting address of dispatcher program.
- Shaded areas indicate execution of dispatcher process.
- How do we design the OS to perform this extremely simple interleaving of processes execution by the processor!

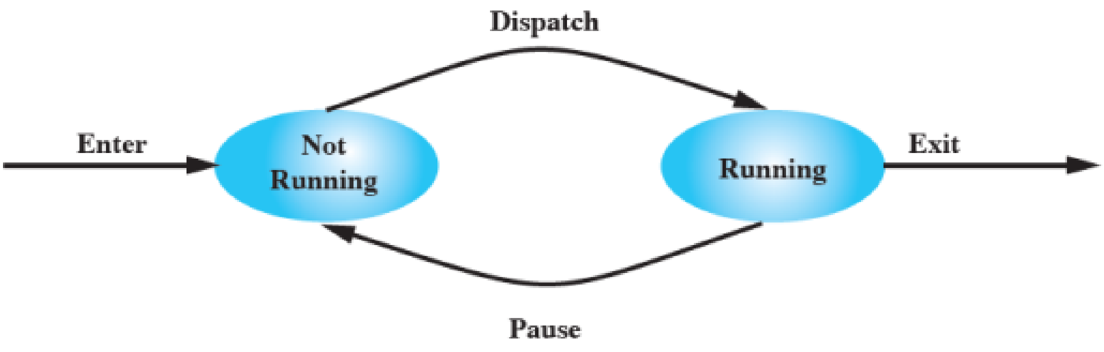
```
1    5000                         27   12004
2    5001                         28   12005
3    5002                         -------------------- Timeout
4    5003                         29   100
5    5004                         30   101
6    5005                         31   102
-------------------- 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
```

Combined Traces of Processes

There's a limit to how long a process can stay in the processor.

The above image shows how the dispatcher switches between the different processes in the previous image to the one above.

Simple Two-State Process Model



(a) State transition diagram

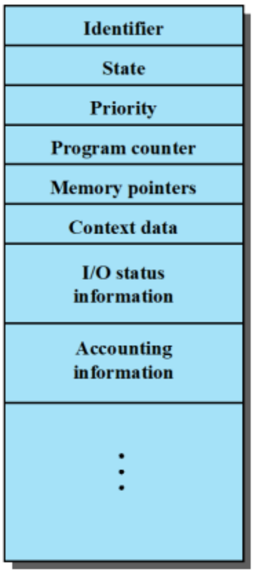
Two-State Process Model

Processes change state as they execute.

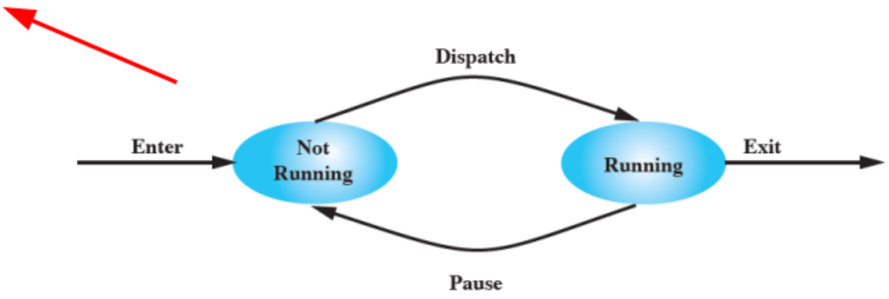
With a 2 state model we have 2 states:

- running
  - the process is being executed

- not running
    - not being executed by processor

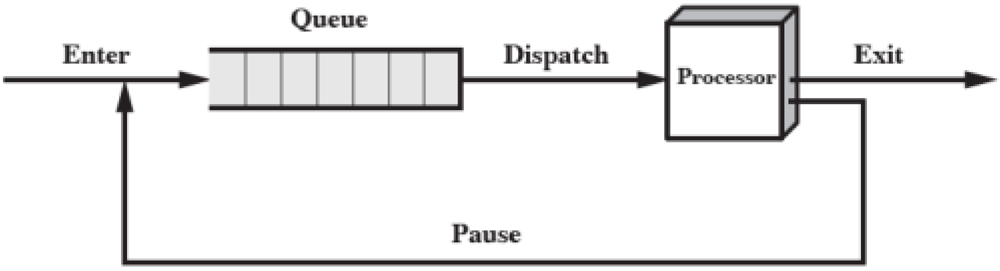


Process Control Block (PCB)



(a) State transition diagram

Two-State Process Model



(b) Queuing diagram

Two-State Process Model

We can have the processes in a queue stored in the ram and we cycle through them, kicking a process out of the processor and back into the queue. When we do that we set it's state to the `Not Running` state.

When we start executing the new process we get to set that process' state to `Running`.

Note: the entries in the queue are links to the process' PCB as opposed to the whole process itself.

The queue is a first-in-first-out list and the processor operates in round-robin fashion on the available processes.

Round-Robin: each process in the queue is given a certain amount of time, in turn, to execute and then returned to the queue, unless blocked.

```
I think every prof seems to have a different definition of "round robin"
```

Operations on Processes

We need to be able to _____ processes:

- create
- terminate

Creation

4 reasons to create a new process:

- new batch job
  - OS is provided with a batch job control system
  - operating system is given a ton of jobs and has to execute them all in the proper priority
- interactive logon
  - system d, inet, etc
  - tons of processes start up to serve the user
- created by os to provide a service
  - when the user does stuff they will often start processes, withuot the user having to wait
  - send information to printers
- spawned by existing process
  - opportunity to exploit parallelism or for the purposes of modularity
  - modular design will allow any user to take advantage of the work that the process is doing
  - processes made to support other processes

On Spawning:

- parent process creates child processes forming a tree of processes
- they may or may not be communicating with one another and/or sharing resources
- all, some, or none of the resources are shared between the parent and child process
- the parent and child may execute concurrently or the parent may wait until the child terminates
- it all depends on what the purpose for the spawning was and what the processes were designed to do

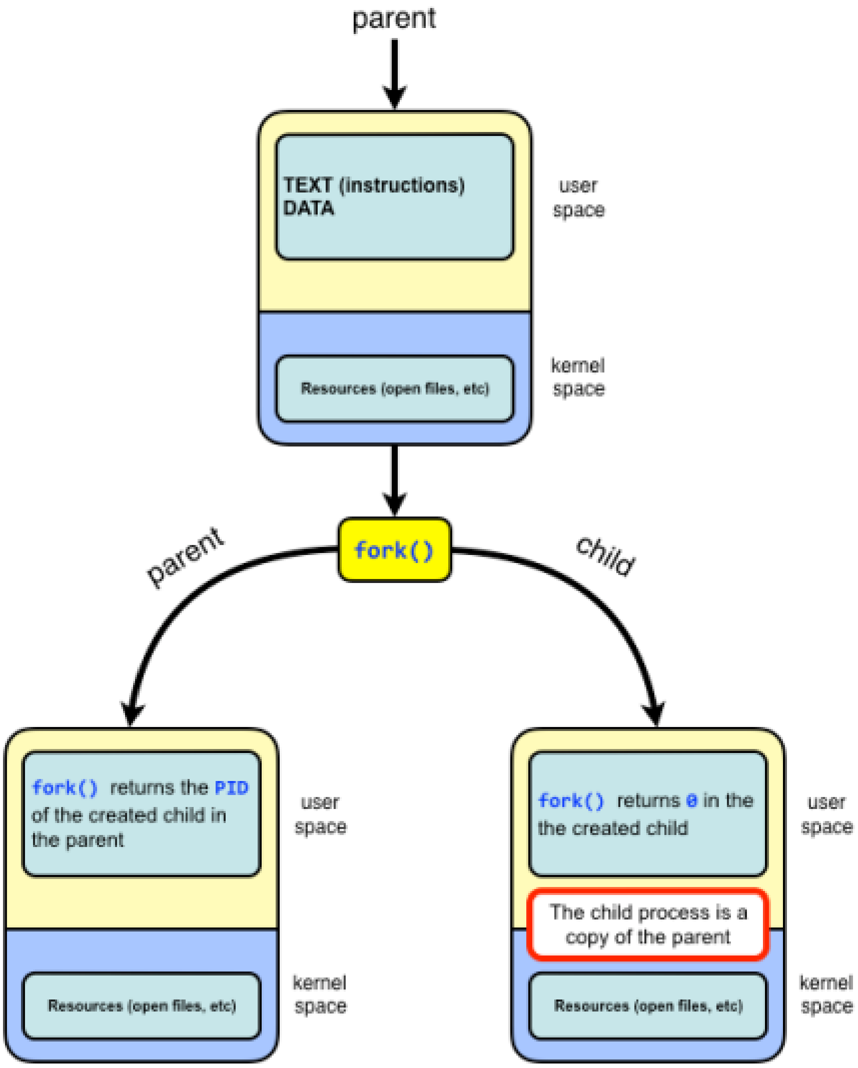
fork

In unix-like OSes, the creation of a child process is established through a system called `fork`

`fork` system call creates a copy by duplicating the parent process

- returns the child process ID to the parent and returns 0 to the child process

- the child and parent processes know each others PIDs


parent
TEXT (instructions)
DATA
user space
Resources (open files, etc)
kernel space
fork()
parent
child
fork() returns the PID of the created child in the parent
user space
Resources (open files, etc)
kernel space
fork() returns 0 in the the created child
user space
The child process is a copy of the parent
Resources (open files, etc)
kernel space

child process can execute a different program using `exec` system call

child can't terminate until the parent collects the output of the child. parent gives a `wait` command to the child and does the collection.

While the child waits then the child is known as a zombie.

If the parent dies then the child becomes an orphan.

---

fork can be used in a program written by the user.

There are a variety of different reasons why you would want to fork so there's many things that the child and parent processes can do together or separate. We could see that they duplicate and continue to execute the same thing or they'll duplicate and start doing different things and receiving commands from the parent program.

Mircrosoft doesn't like `fork()`

- fork is insecure
- fork gives the child more access than it needs since it copies everything by default
- defies the least privilege principle

    - give the least privilege possible to get a job done in order to protect the user and system
- programs that fork but don’t exec render address-space layout randomisation ineffective, since each process has the same memory layout

fork is fast and ha 0 parameters while windows’ CreateProcess has many parameters.

We also don’t run into the error of running out of memory.

`prof really likes this paper and found it very interesting.`

CreateProcess

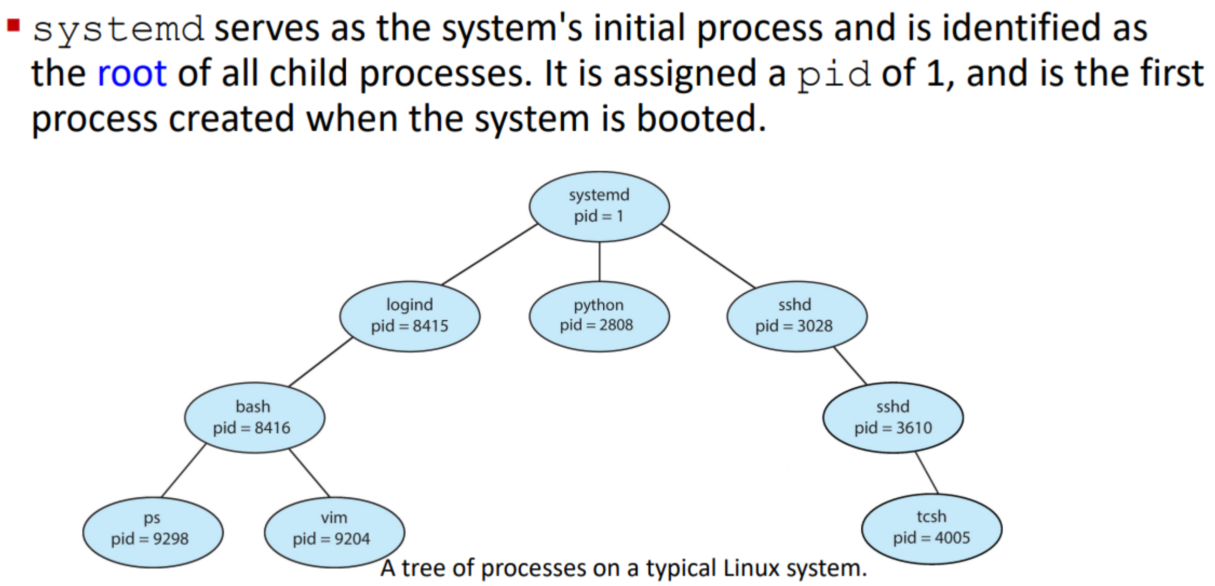
Windows doesn’t have a direct equivalent to fork.

`CreateProcess` is used by Windows API

Spawn vs Fork

- fork
    - existing process creates identical copy of itself
- spawn
    - more general term for creating new process
    - implemented in various different ways depending on OS and programing env
    - doesn’t necessarily involve duplicating the existing process
    - spawned process may or may not have a relationship with the parent process
        - spawned process can be a different program altogether

Tree of Processes in Linux

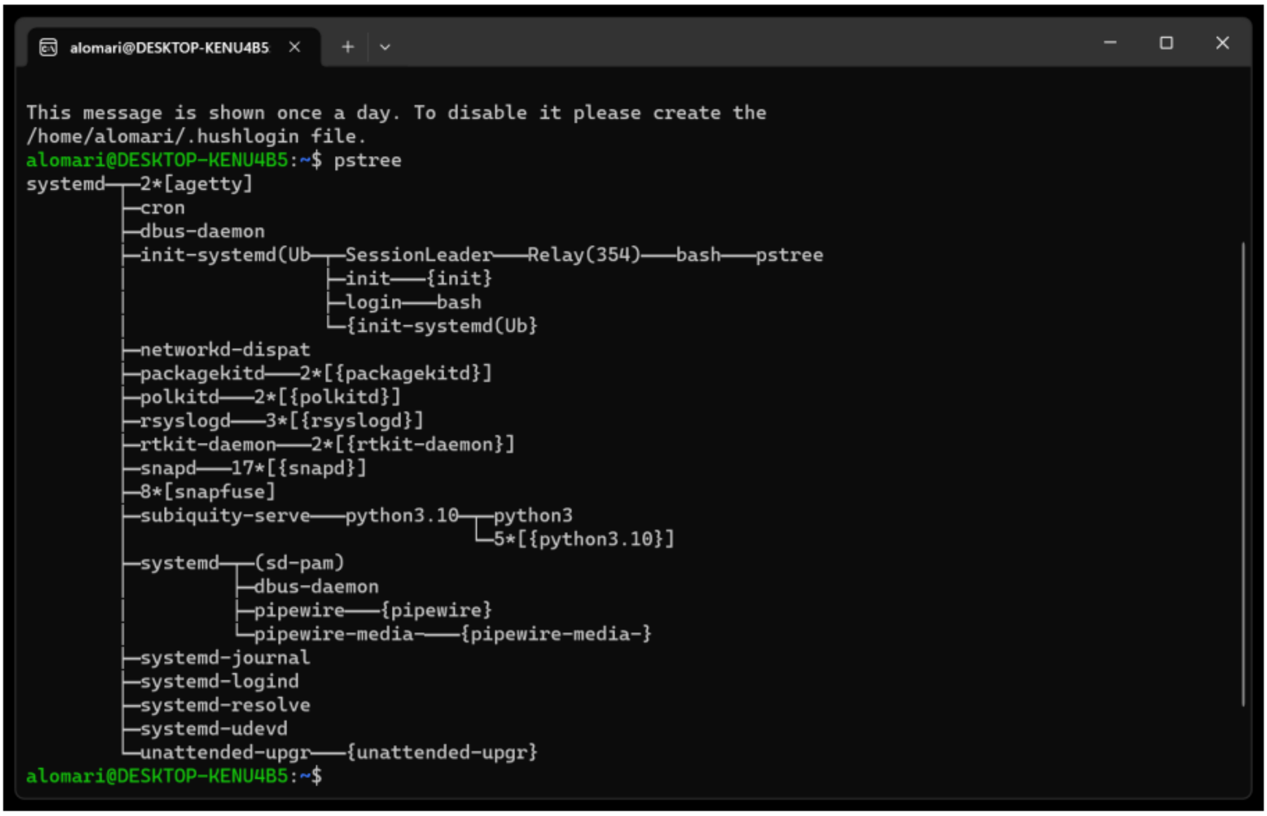


- `systemd` serves as the system's initial process and is identified as the root of all child processes. It is assigned a `pid` of 1, and is the first process created when the system is booted.

A tree of processes on a typical Linux system.

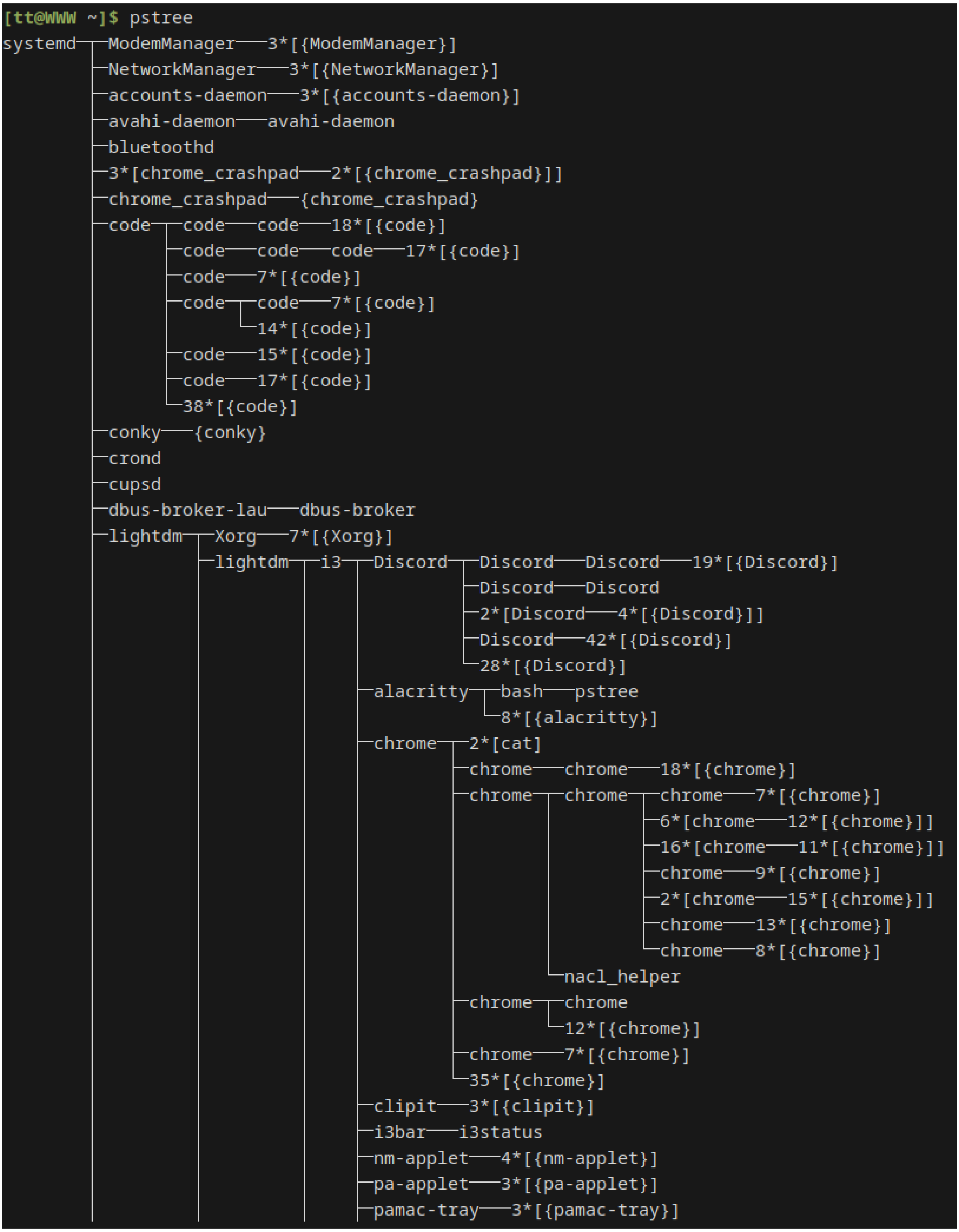
`pstree`

prof:



```
This message is shown once a day. To disable it please create the
/home/alomari/.hushlogin file.
alomari@DESKTOP-KENU4B5:~$ pstree
systemd─┬─2*[agetty]
        ├─cron
        ├─dbus-daemon
        ├─init-systemd(Ub─┬─SessionLeader───Relay(354)───bash───pstree
        │                 ├─init───{init}
        │                 ├─login───bash
        │                 └─{init-systemd(Ub}
        ├─networkd-dispat
        ├─packagekitd───2*[{packagekitd}]
        ├─polkitd───2*[{polkitd}]
        ├─rsyslogd───3*[{rsyslogd}]
        ├─rtkit-daemon───2*[{rtkit-daemon}]
        ├─snapd───17*[{snapd}]
        ├─8*[snapfuse]
        ├─subiquity-serve───python3.10─┬─python3
        │                              └─5*[{python3.10}]
        ├─systemd─┬─(sd-pam)
        │         ├─dbus-daemon
        │         ├─pipewire───{pipewire}
        │         └─pipewire-media-───{pipewire-media-}
        ├─systemd-journal
        ├─systemd-logind
        ├─systemd-resolve
        ├─systemd-udevd
        └─unattended-upgr───{unattended-upgr}
alomari@DESKTOP-KENU4B5:~$
```

mine:



```
[tt@WWW ~]$ pstree
systemd─┬─ModemManager───3*[{ModemManager}]
        ├─NetworkManager───3*[{NetworkManager}]
        ├─accounts-daemon───3*[{accounts-daemon}]
        ├─avahi-daemon───avahi-daemon
        ├─bluetoothd
        ├─3*[chrome_crashpad───2*[{chrome_crashpad}]]
        ├─chrome_crashpad───{chrome_crashpad}
        ├─code─┬─code───code───18*[{code}]
        │      ├─code───code───code───17*[{code}]
        │      ├─code───7*[{code}]
        │      ├─code─┬─code───7*[{code}]
        │      │      └─14*[{code}]
        │      ├─code───15*[{code}]
        │      ├─code───17*[{code}]
        │      └─38*[{code}]
        ├─conky───{conky}
        ├─crond
        ├─cupsd
        ├─dbus-broker-lau───dbus-broker
        ├─lightdm─┬─Xorg───7*[{Xorg}]
        │         ├─lightdm─┬─i3─┬─Discord─┬─Discord───Discord───19*[{Discord}]
        │         │         │    │         ├─Discord───Discord
        │         │         │    │         ├─2*[Discord───4*[{Discord}]]
        │         │         │    │         ├─Discord───42*[{Discord}]
        │         │         │    │         └─28*[{Discord}]
        │         │         │    ├─alacritty─┬─bash───pstree
        │         │         │    │           └─8*[{alacritty}]
        │         │         │    ├─chrome─┬─2*[cat]
        │         │         │    │        ├─chrome───chrome───18*[{chrome}]
        │         │         │    │        ├─chrome─┬─chrome─┬─chrome───7*[{chrome}]
        │         │         │    │        │        │        ├─6*[chrome───12*[{chrome}]]
        │         │         │    │        │        │        ├─16*[chrome───11*[{chrome}]]
        │         │         │    │        │        │        ├─chrome───9*[{chrome}]
        │         │         │    │        │        │        ├─2*[chrome───15*[{chrome}]]
        │         │         │    │        │        │        ├─chrome───13*[{chrome}]
        │         │         │    │        │        │        └─chrome───8*[{chrome}]
        │         │         │    │        │        └─nacl_helper
        │         │         │    │        ├─chrome─┬─chrome
        │         │         │    │        │        └─12*[{chrome}]
        │         │         │    │        ├─chrome───7*[{chrome}]
        │         │         │    │        └─35*[{chrome}]
        │         │         │    ├─clipit───3*[{clipit}]
        │         │         │    ├─i3bar───i3status
        │         │         │    ├─nm-applet───4*[{nm-applet}]
        │         │         │    ├─pa-applet───3*[{pa-applet}]
        │         │         │    ├─pamac-tray───3*[{pamac-tray}]
```

Termination

A process terminates when it finishes executing its final statement and askss the operating system to delete it by using the `exit()` system call

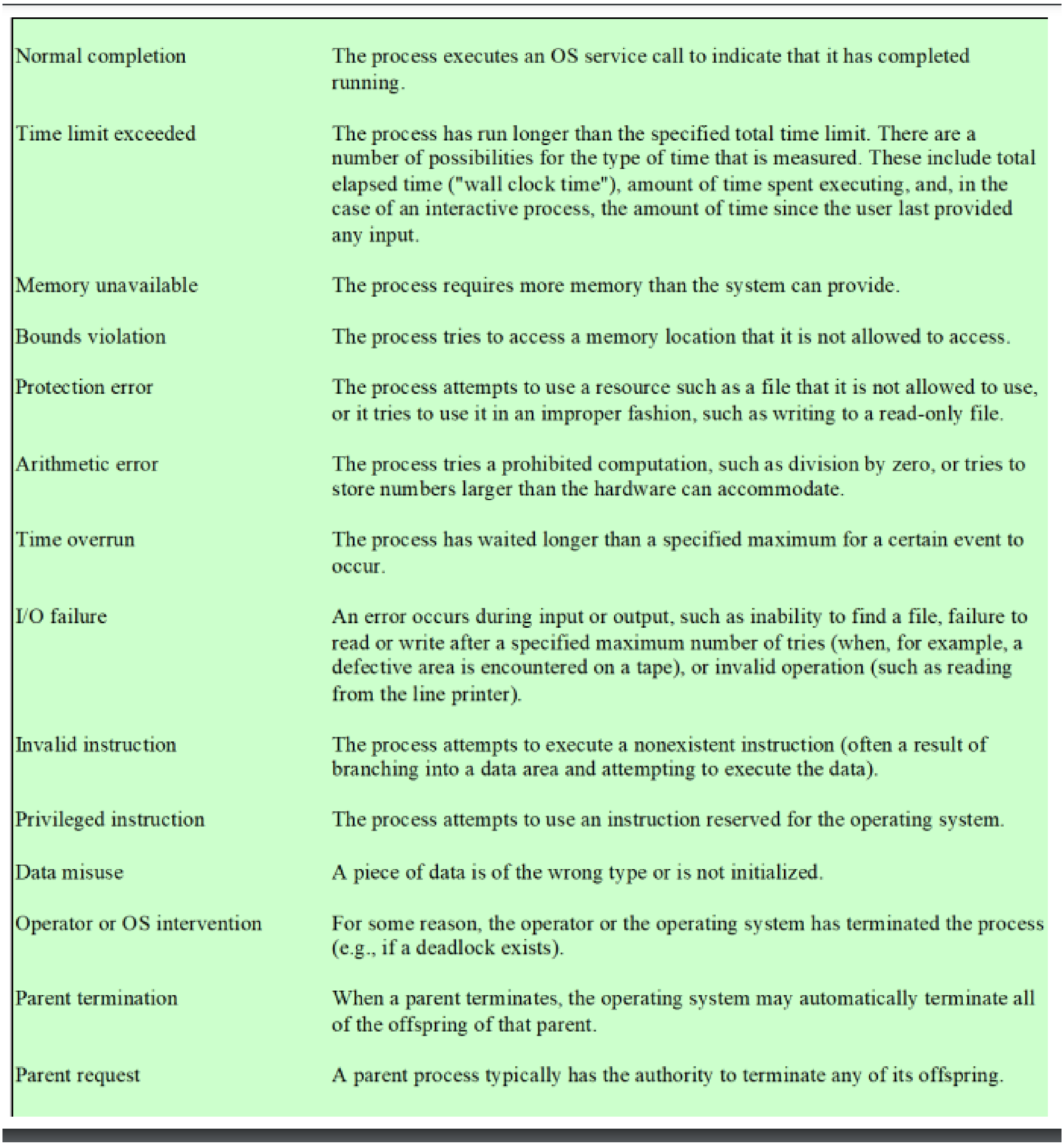
all the resources of the process (physical and virtual) are deallocated and reclaimed by the operating system.

Batch job should include a halt instruction or an explicit os service call for termination. Halt will geernate an interrupt to alert the os that a process has completed.

Interactive applications (browsers and shit used by user) are terminated when the user decides typically.

parents can terminate children

WOW:



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.

not reading allat but pretty cool :thumbsup:

Five-State Process Model

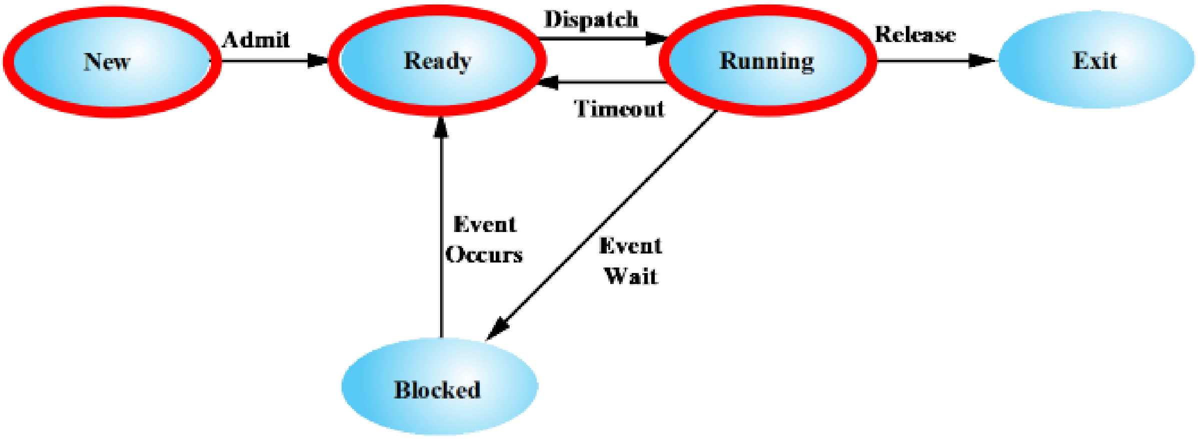
Note: THis is on the assignemnt (Thanks Cate)

USing a single queue the dispatcher could not just select the process at the oldest end.

Dispatcher needs to find a process that is not blocked and that has been in the list for the queue for the longest.

Turn not `Not Running` into `Ready` and `Blocked`

State process diagram



Five-State Process Model

States:

- new
    - newly created process not yet admitted to the pool of executable processes by the OS
- ready
    - prepared to execute when given opportunity
- running
    - process that is currently being executed
- blocked
    - unix-based: waiting
    - process cannot execute until some event occurs such as completeion of an I/O operation
- exit
    - unix-based: terminated
    - process released from pool of executable processes by the OS
    - process has been halted or aborted for whatever reason

Transitions:

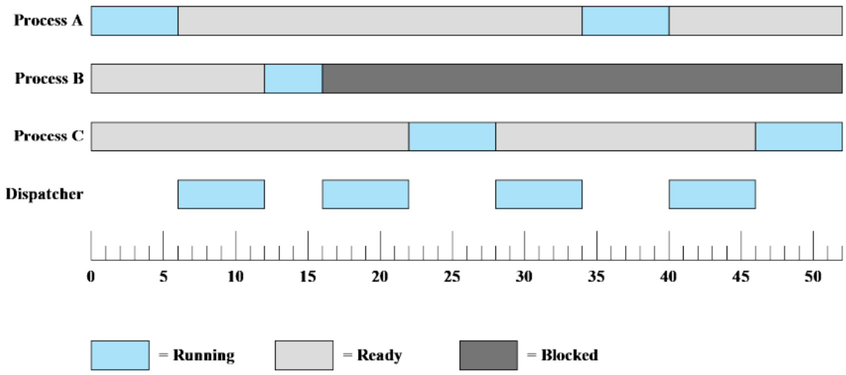
- null -> new
    - new process is created to exec program

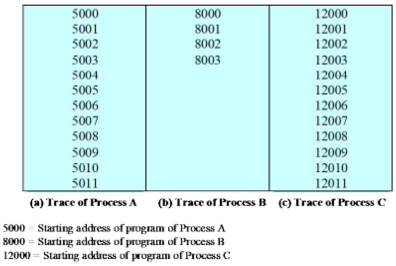
- new -> ready
  - os is prepared to take on additional process
- ready -> running
  - dispatcher dispatches the process
  - OS chooses ready process to run
- running -> exit
  - process is terminated by the OS if the process indicates it has completed or if it aborts
- running →ready
  - running process has reached teh maximum allowable time for uninterrupted execution
  - no longer your turn on the xbox
- running →blocked
  - process requrests something for which it must wait
- blocked →ready
  - process in blocked state is moved to ready state when the event for which is has been waiting occurs

remember that we never have blocked to running.

still not perfect but more later

Example



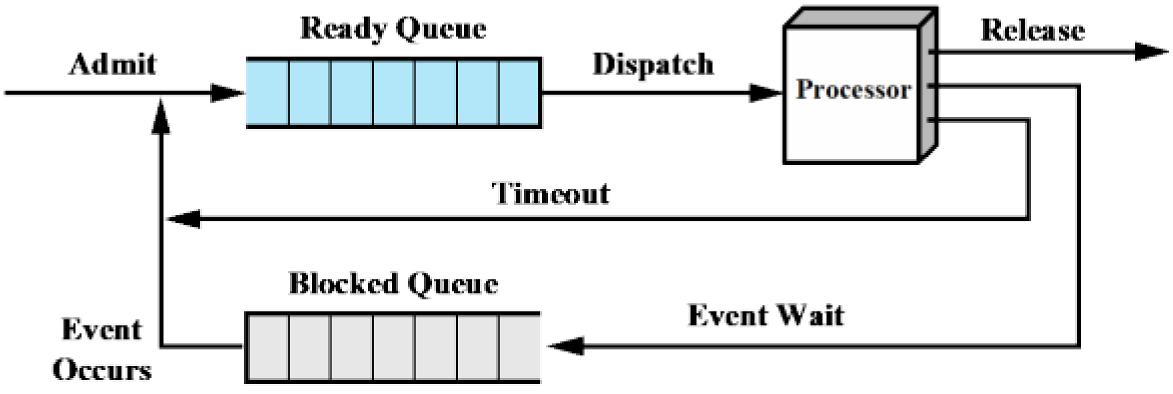


(a) Trace of Process A	(b) Trace of Process B	(c) Trace of Process C
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

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

Process States for the Trace Shown on the Right

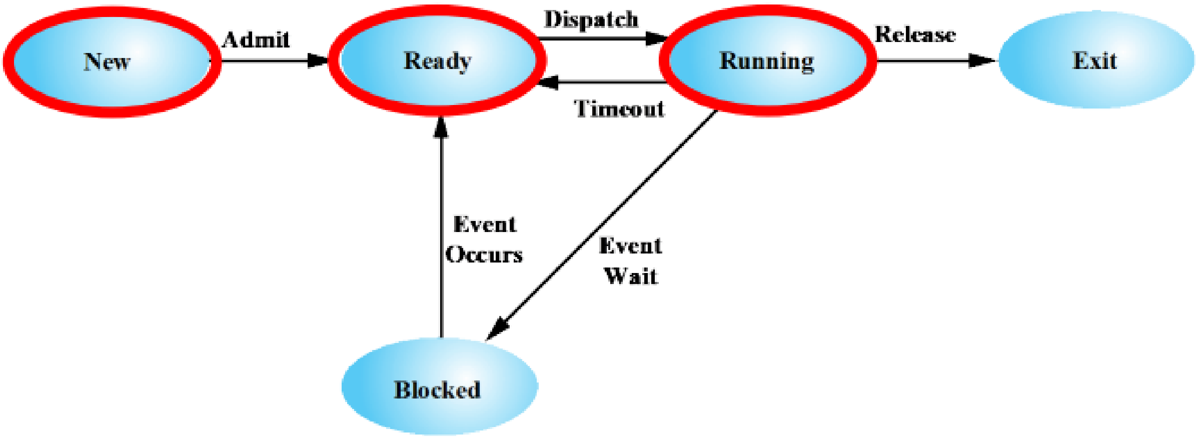
while the processses are simply ready but not running, the dispatcher is running in order to go to dispatch them



(a) Single blocked queue

There are 2 queues:

1. ready queue
2. blocked queue



Five-State Process Model

So when we ready something up it actually goes into a queue and when something gets hit with an event wait it gets put into a blocked queue.

When a process

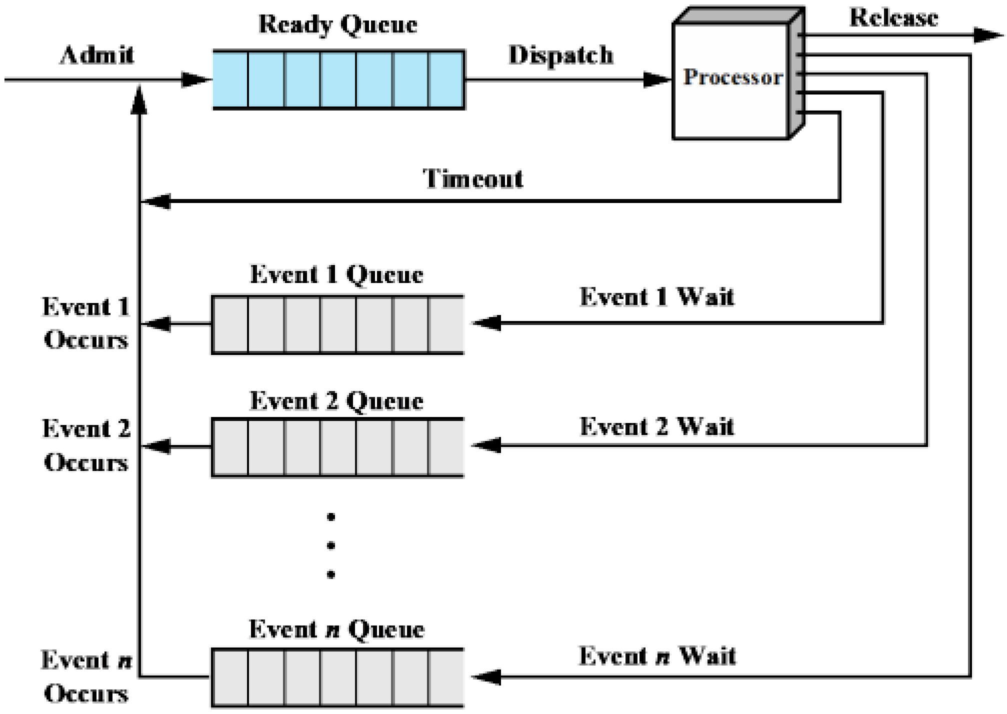
- is admitted to the system ⟶ admit to requdy queue
- is chosen to run ⟶ select from ready queue
- has to wait ⟶ send to blocked queue
- times out ⟶ send to ready queue again
- finishes ⟶ release

In the absence of any priority scheme this can be a simple FIFO queue (round-robin). Otherwise we could use something like a priority queue.

When an event occurs, any process in the blocked queue that has been waiting on that event only is moved to the ready queue.

The problem is that we have to scan the entire blocked queue which could have 100s or 1000s of blocked processes.

We can remedy this with the use of multiple blocked queues



(b) Multiple blocked queues

For each event that will cause an interrupt we will make a queue for it that way we don't have to scan through the whole queue to find a process that corresponds to the event.

Suspended Processes

Problem - it will become common for all of th eprocesses in memory to be waiting for I/O, and for the processor to become idle.

- there's only so much memory space to store so many processes
- we can pretty easily fill up the memory with all the processes and leave the processor idle since it only really works with shit in the memory.

Solution - swapping

- move part or all of a process from mai memory to disk
- it's a very specific part of the disk that's allocated for the memory to work with
- "there is a specific page file dedicated by the operating system"
- this is like the linux swap

The only problem is how do we do that?

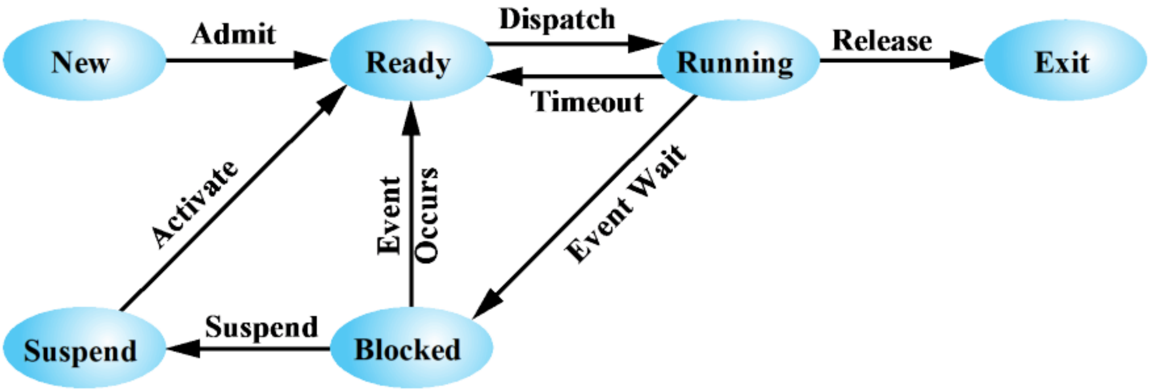
We add another state to our process model

When the OS swaps one of the `blocked` processes out on to the disk we put it into a `suspend`ed queue:

- a queue of existing processes that have been temporarily kicked out of th emain memory or suspended

the os then brings in another process from the suspend queue or it honors a new-process request

execution then continues withthe newly arrived process



(a) With One Suspend State

So now when we block a process from there it can either go back to the ready queue or get sent to the suspend queue.

When it gets sent to the suspend queue then it gets sent to the disk to make more space in the memory.

Once it’s in the suspend queue, we can activate it again.

So there are now 4 ways to get a process into the ready queue

1. a newly admitted process
2. a proces times out
3. a blocked process is unblocked thanks to the event it was waiting on occuring
4. a suspended process is selected to become active once more

If there is nothing in the ready queue and we’re waiting on blocked processes then we will start suspending processes to swap out of main memory so that we can admit new processes or activate one from the suspend.

It’s always preferrable to activate a suspended process.

The swap is always available - it is built into the architecture - and there’s always labels for each state.

When the OS performs a swapping-out operation, it has 2 choices for selecting a process to bring into main meomry:

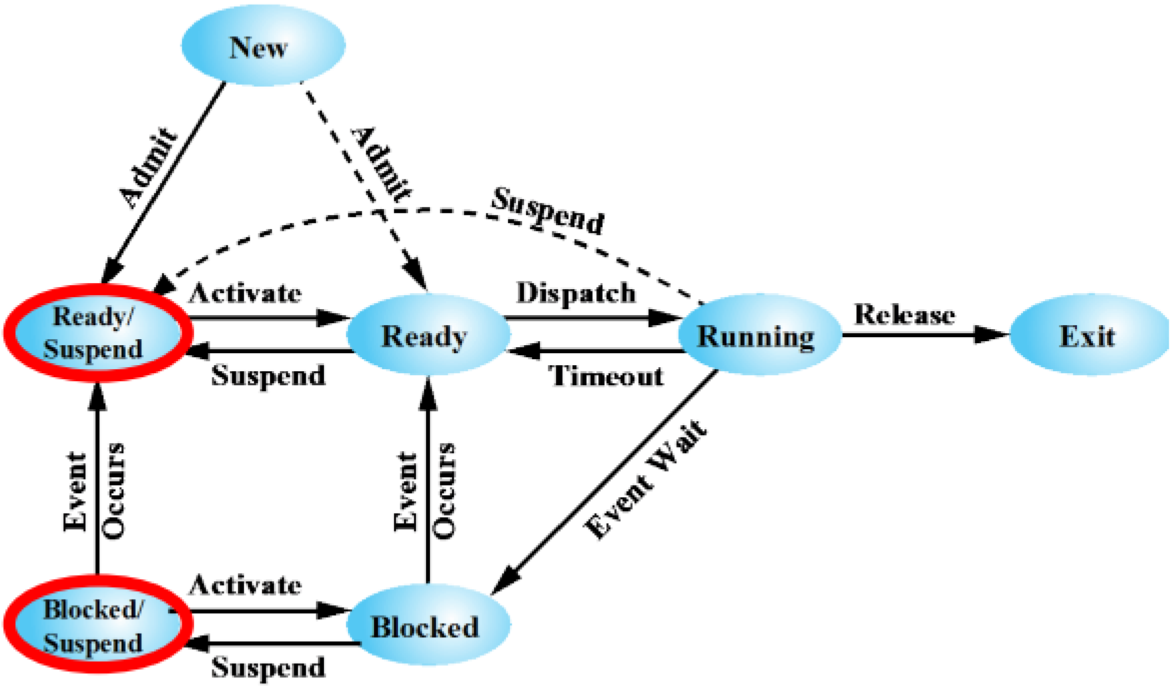
1. admit a newly created process
2. bring a previously suspended process(preferred)

problem:

- all suspended processes were blocked when they were suspended
- which one do we bring back?

Each process in the suspend state was originally blocked on a particular event, something tha twe accounted for with the use of multiple queues. We also know that once that event is over we know that the processes in its corresponding queue are no longer blocked.

We want to make another state for suspended processes



(b) With Two Suspend States

Dashed lines in the figure indicate possible but not necessary transitions.

For the processes in the secondary memory we're able to have a ready and blocked queue for the suspended.

When we hear/read "suspend" then we think of the disk.

`Blocked` → `Blocked+Suspend`:

- if there are no `Ready` processes then at least one blocked process is swapped out to make room for another process that is not blocked.

`Blocked+Suspend` → `Ready+Suspend`:

- a process was waiting for an event

- the event finishes
- the process is no longer blocked but it’s still suspended so it remains in the disk space but gets sent to a suspended ready queue

`Ready+Suspend` ⟶ `Ready`:

- when there are no `Ready` processes the OS will need to bring oen in to continue execution

`Ready` ⟶ `Ready+Suspend`:

- to free up a sufficiently large block of main memory
- we have a ton of processes that are ready.

`Running` ⟶ `Ready+Suspend`:

- the OS is preempting the process because a higher-priority process on the `Blocked+Suspend` queue has just become unblocked
- something higher priority just got unblocked and we need to execute on that one now instead
- the os could move the running process directly to the `Read+Suspend` queue and fre some main memory
- process preemption occurs when an execution process i interrupted by the processor so that another process can be executed

`New` ⟶ `Ready+Suspend`:

`New` ⟶ `Ready`:

We could add more states but then there’s more work involved in changing between states. There are more states in different designs however.

Q: How do we decide wether a `New` process goes straight to `Ready+Suspend` or `Ready`?

A: we prefer to send a new process to the suspended queue. THe OS will prefer grabbing from ready suspend. Very rare that we will send something straight to ready since that would mean that the suspend is empty as well so there would be no point in sending it there first and wasting compute time on switching

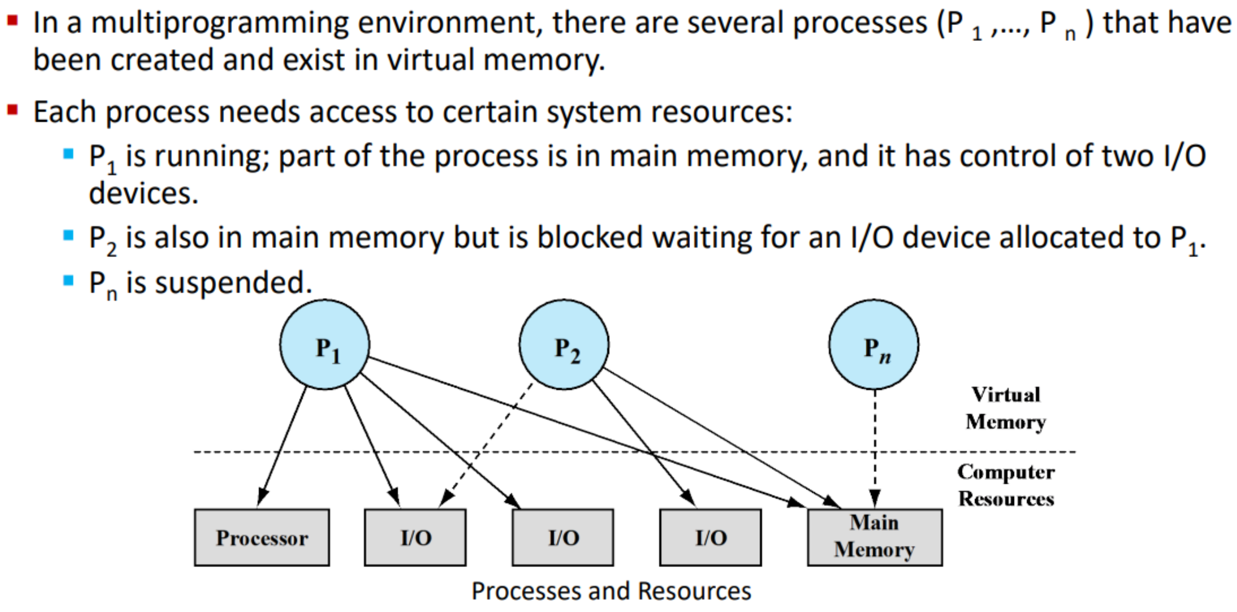
A suspended process

- non immediately available for execution
  - all info sitting on secondary memory instead of main
- process was placed in a suspended state by
  - itself
  - a parent process
  - OS
- suspended in order to prevent execution
- may or may not be waiting on an event
- process may not be removed from this state until the agent orders the removal
- reasons for suspensions

    - swapping
    - other OS reason
        - illegal memory access
            - usually terminates in that case
        - buffer overflow
    - interactive user request
        - debugging
        - want to use resource
        - minimizing applications or switching off of tabs so we don’t really need the program working
    - timing
    - parent process request

Process Description and Control

In a multiprogramming env there are several processes that have been created and exist in virtual memory.



- In a multiprogramming environment, there are several processes (P_1 ,..., P_n) that have been created and exist in virtual memory.
- Each process needs access to certain system resources:
    - P_1 is running; part of the process is in main memory, and it has control of two I/O devices.
    - P_2 is also in main memory but is blocked waiting for an I/O device allocated to P_1.
    - P_n is suspended.

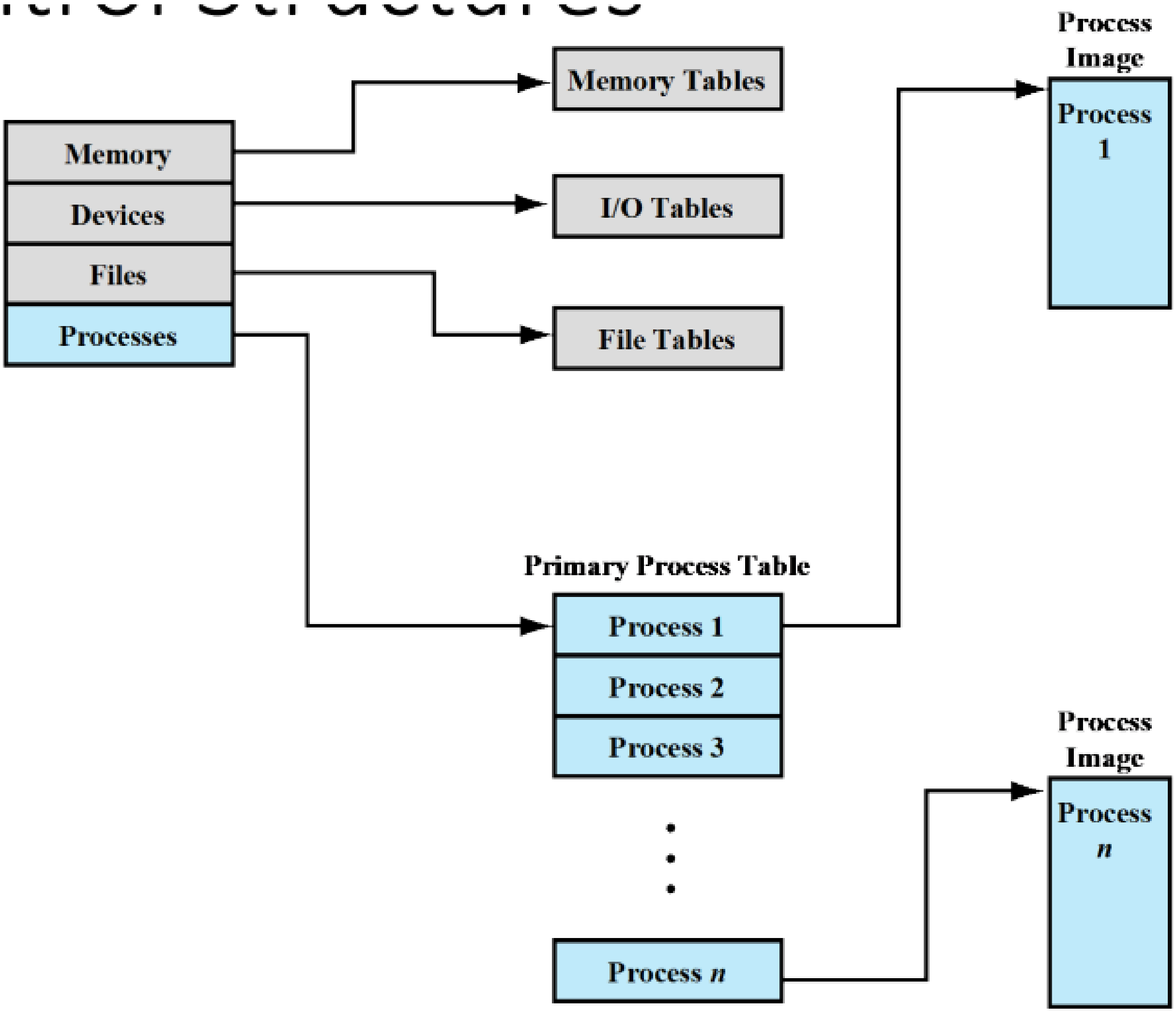
Processes and Resources

What info doe sthe OS need to control processes and manage resources for them?

OS constructs and maintains tables of information about each entity that it is managing:

- memory tables
- I/O tables
- file tables
- process tables

ntrol Structures



General Structure of Operating System Control Tables

Memory Tables

main = real

secondary = virtual

we use memory tables to do:

- allocation of main meory to processes
- allocation of secondary memory to processes
- any protection attributes of blocks of main or virtual memory
  - which processes may access certain shared memory regions
  - prevent programs from accessing main or virtual memory that it’s not allowed to
- any information needed to manage virtual memory

I/O Tables

used by the OS to manage the I/O devices and channels of the computer system

at any given time, an I/O device may be available or assigned toa particular process

OS needs to know the status of the I/O operation and the location in main memory being used as the source or destination fo the I/O transfer

File Tables

Provide information about the:

- existence of files
- location on secondary memory
- current statwus
- other attribs

may be maintined by a file management system (windows NTFS, linux/unix FAT) or the OS based on different operating system

Process Tables

Must be maintained to manage processes

References memory, I/O, and files directly or indirectly in the process tables to show what the process needs/uses

when the OS is initialized it must have some access to some configuartion data that define the basic environment

- all the info required to build out the tables
- everything you can see and view in the bios
- I think anyways

Process Control

to manage and control a process, the OS must know

- where the process is located
- the attribs of the process that are necessary for its management
- basically everything in the PCB

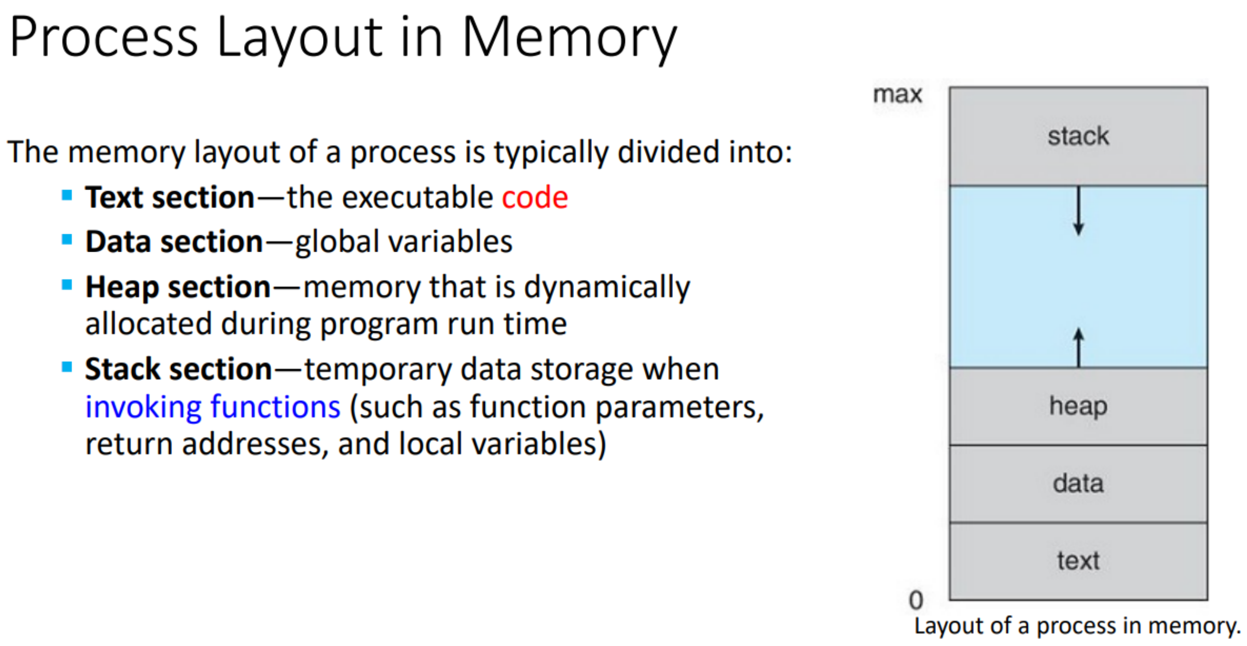
What is the physical manifestation of a process?

a process must inclue:

- a program or set of programs to be executed
- a set of data location for local and global variables
- stack that is used to keep track of
  - procedure calls
  - parameter passing between procedures
- a number of attributes that are used by the OS for process control (Process control block)

We refer to this as the `process image`

Process Layout in Memory



Process Layout in Memory

The memory layout of a process is typically divided into:

- **Text section**—the executable code
- **Data section**—global variables
- **Heap section**—memory that is dynamically allocated during program run time
- **Stack section**—temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)

Layout of a process in memory.

The stack goes from higher to lower addresses while the heap grows from lower addresses to higher addresses.

Process Location

location of a process image will depend on the memory managemnt scheem being used

At any given time parts of the process image can be both in the main and secondary memory

so we need process tables maintained by the OS to show the location of each page of each process image

Process Attributes

info resides in a process control block

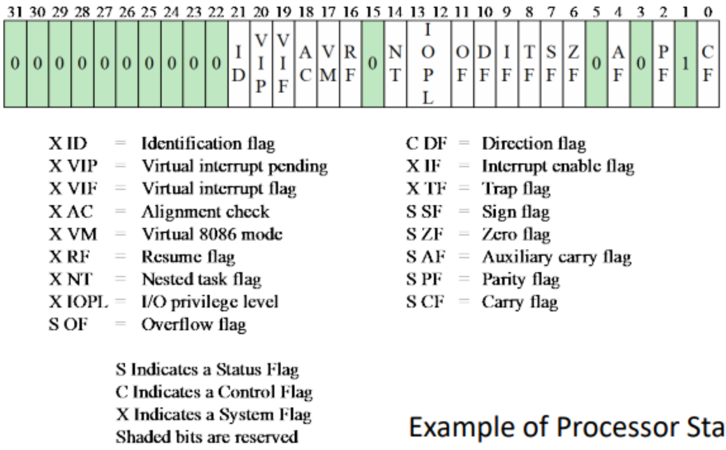
there are 3 general categories

1. process identification
2. processor state information
3. process control information

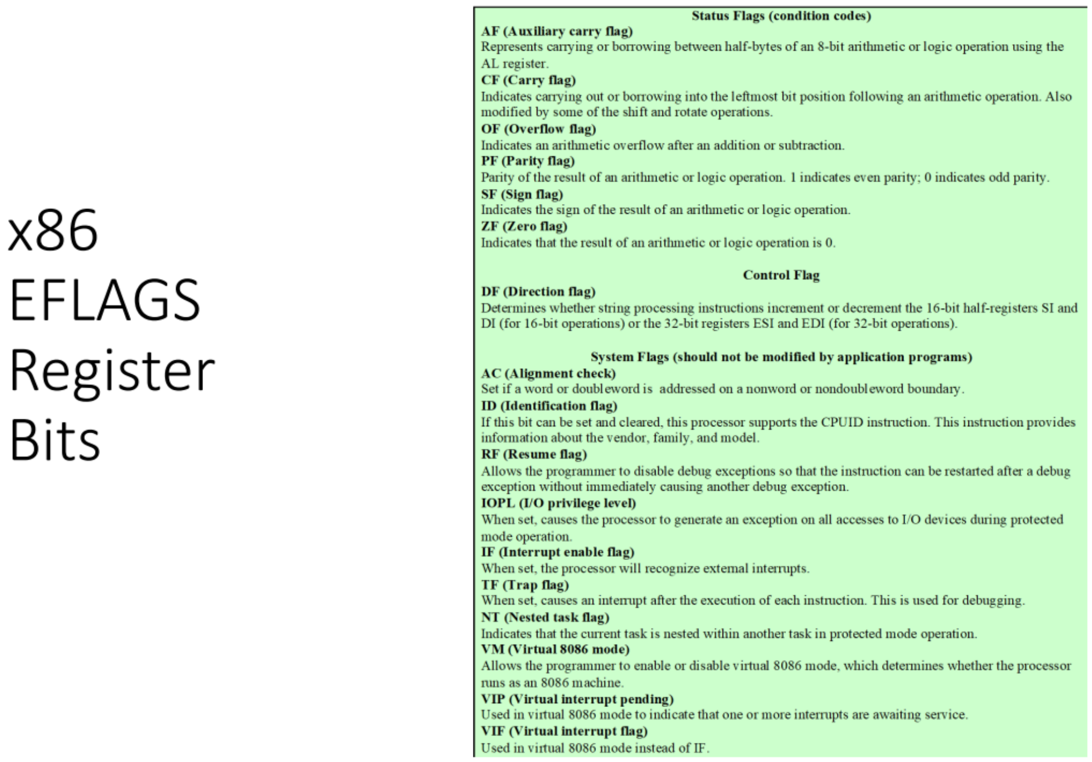
process identification:

- each processor gets a unique numeric identifier
    - PID
- many of the bales controlled by the OS may use process identifiers to cross-reference process tables
- interprocess communication uses PIDs to determine who is who and where
- PIDs let us see who are parent and descendent processes
    - the parents have the child PIDs and the children have the parent PID

Process State Information

- what is the current state of the process
- consists of the contents of processor registers
    - registers on the cpu itself, very fast
- nature of registers involved depend on the design of the processor but typically
    - user-visible registers
    - control and status registers
    - stack pointers
- all processor designs include a register or set of registers often known as the program status word (PSW) that contains condition codes plus status information
    - EFLAGS registers is an example of a PSW used by any OS running on an x86 processor
    - 
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID = Identification flag
X VIP = Virtual interrupt pending
X VIF = Virtual interrupt flag
X AC = Alignment check
X VM = Virtual 8086 mode
X RF = Resume flag
X NT = Nested task flag
X IOPL = I/O privilege level
S OF = Overflow flag
C DF = Direction flag
X IF = Interrupt enable flag
X TF = Trap flag
S SF = Sign flag
S ZF = Zero flag
S AF = Auxiliary carry flag
S PF = Parity flag
S CF = Carry flag
S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
Shaded bits are reserved

Example of Processor Status Word: X86 EFLAGS Register



x86 EFLAGS Register Bits

Status Flags (condition codes)

AF (Auxiliary carry flag)
Represents carrying or borrowing between half-bytes of an 8-bit arithmetic or logic operation using the AL register.

CF (Carry flag)
Indicates carrying out or borrowing into the leftmost bit position following an arithmetic operation. Also modified by some of the shift and rotate operations.

OF (Overflow flag)
Indicates an arithmetic overflow after an addition or subtraction.

PF (Parity flag)
Parity of the result of an arithmetic or logic operation. 1 indicates even parity; 0 indicates odd parity.

SF (Sign flag)
Indicates the sign of the result of an arithmetic or logic operation.

ZF (Zero flag)
Indicates that the result of an arithmetic or logic operation is 0.

Control Flag

DF (Direction flag)
Determines whether string processing instructions increment or decrement the 16-bit half-registers SI and DI (for 16-bit operations) or the 32-bit registers ESI and EDI (for 32-bit operations).

System Flags (should not be modified by application programs)

AC (Alignment check)
Set if a word or doubleword is addressed on a nonword or nondoubleword boundary.

ID (Identification flag)
If this bit can be set and cleared, this processor supports the CPUID instruction. This instruction provides information about the vendor, family, and model.

RF (Resume flag)
Allows the programmer to disable debug exceptions so that the instruction can be restarted after a debug exception without immediately causing another debug exception.

IOPL (I/O privilege level)
When set, causes the processor to generate an exception on all accesses to I/O devices during protected mode operation.

IF (Interrupt enable flag)
When set, the processor will recognize external interrupts.

TF (Trap flag)
When set, causes an interrupt after the execution of each instruction. This is used for debugging.

NT (Nested task flag)
Indicates that the current task is nested within another task in protected mode operation.

VM (Virtual 8086 mode)
Allows the programmer to enable or disable virtual 8086 mode, which determines whether the processor runs as an 8086 machine.

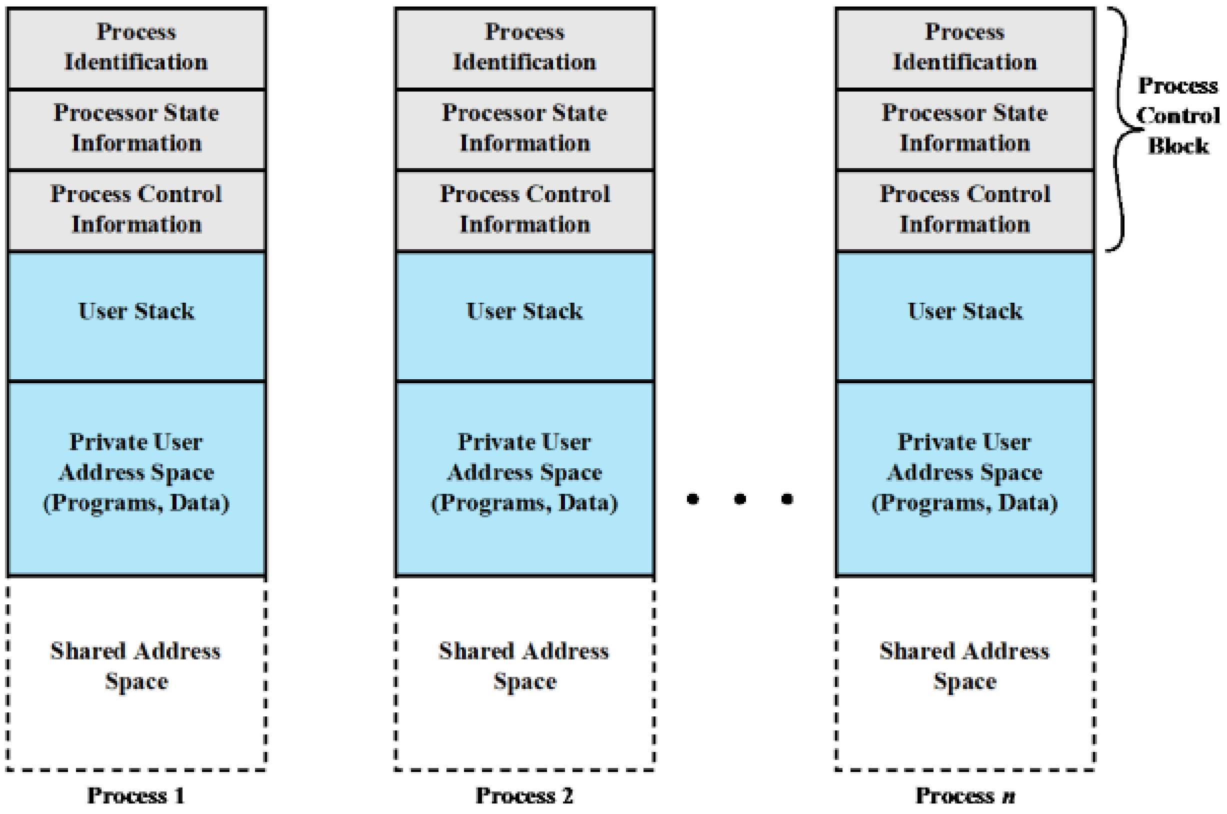
VIP (Virtual interrupt pending)
Used in virtual 8086 mode to indicate that one or more interrupts are awaiting service.

VIF (Virtual interrupt flag)
Used in virtual 8086 mode instead of IF.

○

Process Control Information

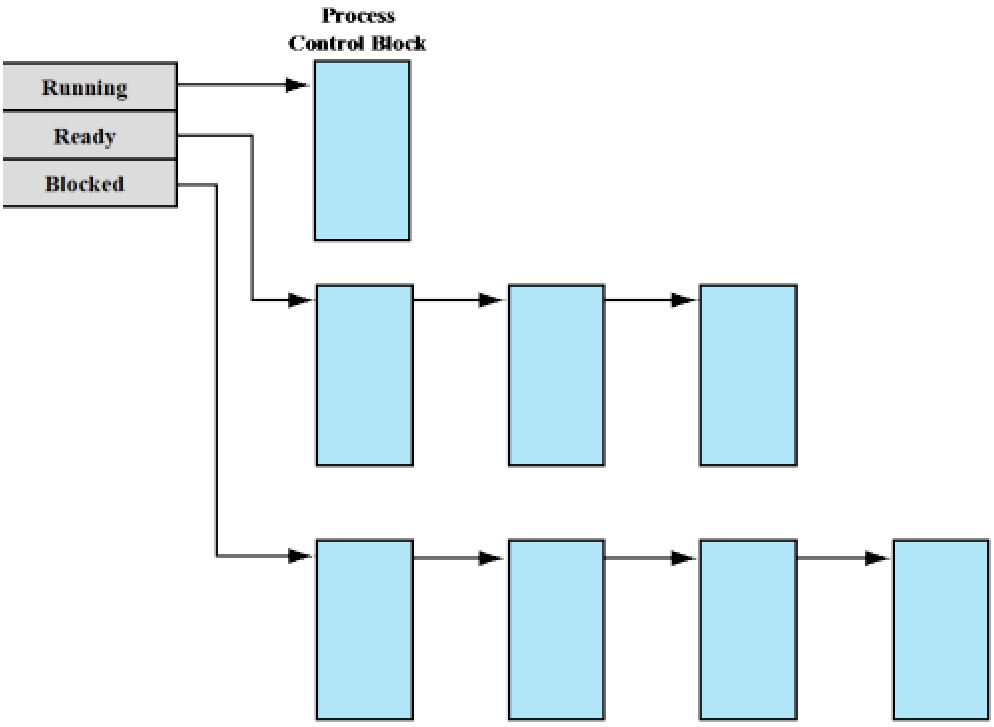
- the additional information needed by the OS to control and coordinate the various active processes



User Processes in Virtual Memory

Role of the Process Control Block

- most important data structure in an OS
    - contains all info needed by OS about process
    - blocks are read and/or modified by virtually every module in the OS
- the set of PCBs define that state of the OS



Process List Structures

Process Control

Typical mangement functions:

- creation and termination
- scheduling and dispatching
- switching
- synchronizationand support for interprocess communication
- mangaement of PCBs

we’re going to be looking at a lot of these.

Creation

OS decides to create a new process

1. assign a unique PID to the new process

2. allocate space for the process
3. initialize the process control block
4. set the appropriate linkages (put it in the queue)
5. creates or expands other data structures

Lect start

Switching

A process switch may occur at any time that the OS has gained control from the currently running process

Some possible events that may give control to the OS

mechanism	**cause**	**use**
interrupt	something external to the execution of the current instruction interrupts	let’s the OS react to that asynchronous external event
trap	associated with the execution of the current instruction	handling an error or exception condition
supervisor call	explicit request	call to an operating system function

Interrupt:

- an event external to and independent of the current process
- examples
  - clock interrupt
  - timeout
  - I/O interrupt
  - memory fault
    - wow so epic

Trap:

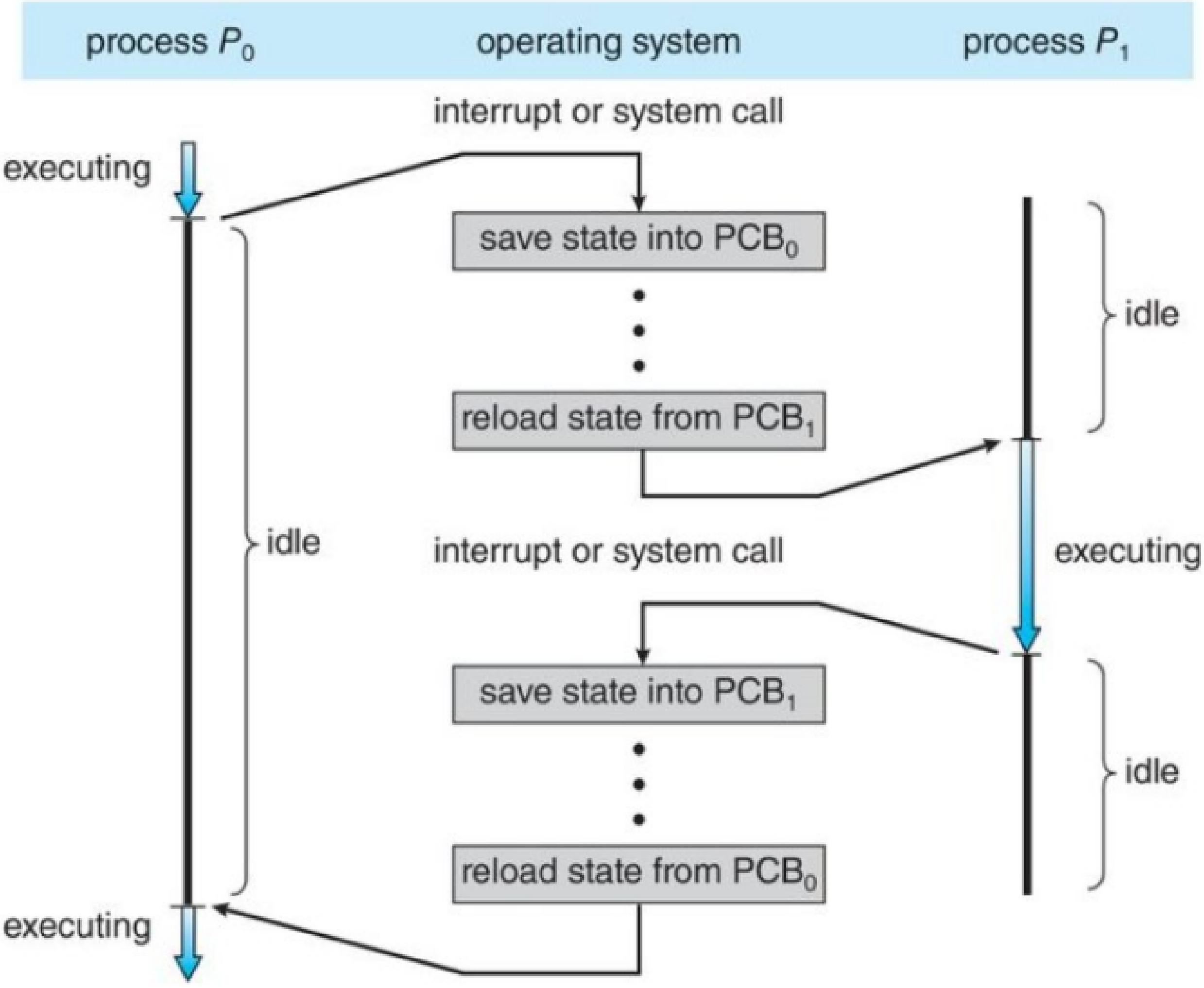
- an error or exception condition generated within the currently running process
- OS determines if the condition is **fatal**
  - fatal
    - move to exit state and a process switch occurs
  - not fatal
    - action will depend on the nature of th eerror and the design of the OS
    - the OS will try to recover

Supervisory Call:

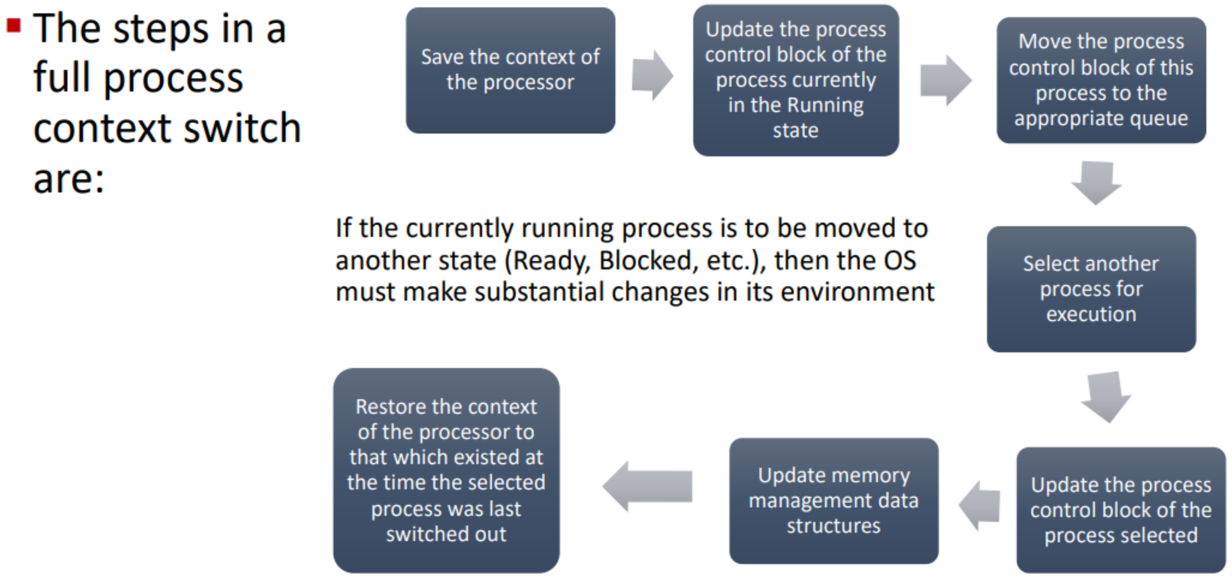
- OS activated by a supervisor call from teh program being executed
- getting the manager involved b/c of smth not on ur level

Context Switch

- description:
    - when the cpu switches to another process, the system must save the state of the old process and load the saved state for the new process
    - process’ context = PCB
- we have to efficiently save our old PCB and start using the new PCB to execute
- context witch time is overhead
    - no useful work is done while switching
    - more complex os and pcb = longer switch time
    - we want to minimize context switches so we don’t waste too much time
- time dependent on hardware support
    - multiple sets of registers per CPU = multiple loaded contexts
        - not common
    - typical speed is several microseconds



notice that there is idle time for the operating system to perform the saving of PCB_0 and subseqent loading of PCB_1



- The steps in a full process context switch are:

Save the context of the processor

Update the process control block of the process currently in the Running state

Move the process control block of this process to the appropriate queue

Select another process for execution

Update the process control block of the process selected

Update memory management data structures

Restore the context of the processor to that which existed at the time the selected process was last switched out

If the currently running process is to be moved to another state (Ready, Blocked, etc.), then the OS must make substantial changes in its environment

`"blah blah blah" - professor`

Execution of the Operating System

recall:

- OS functions in the same was as ordinary computer software
    - OS is a set of programs executed by the processor
- OS frequently relinquishes control and depends on the processor to restore control to the OS

Question

- OS is just a colection of programs
- it's executed by the processor like any other program
- is the OS a process?
- how is it controlled?

OS just throws control back to itself.

Whenever it gives away control there's always the contingency that it'll come back.

the collection of processes control one another.

---

3 approaches:

1. separate kernel
2. execution within user processes
3. process-based operating system

Separate Kernel

Execute the kernel of the OS outside of any process

- the concept of a the proess is considered to apply only to user programs
- OS executed as separate entity operated in privileged mode
- common on older OSes
  - not often used today

User Processes

All OS software is executed in the context of a user process:

- OS a collection of routines that the user calls to perform variousfunctions, executed within the environment of the user’s process
- common on OSes on smaller computers (PCs)
  - commonly used in today’s laptops

Process-Based OS

OS as a collection of system processes

- software that is part of the kernel executes in a kernel mode
- major kernel functgions are organized as separate processes
- imposes a program design discipline that encourages the use of a modular OS with minimal, clean interfaces between the modules.
- used in
  - servers in more modern computing
  - modular design

Summary

The most fundamental concept in a modern OS is the process.

The principal function of the OS is to create, manage, and terminate processes.

While processes are active, the OS must see that each is allocated time for execution by the processor, coordinate their activities, manage conflicting demands, and allocate system resources to processes.

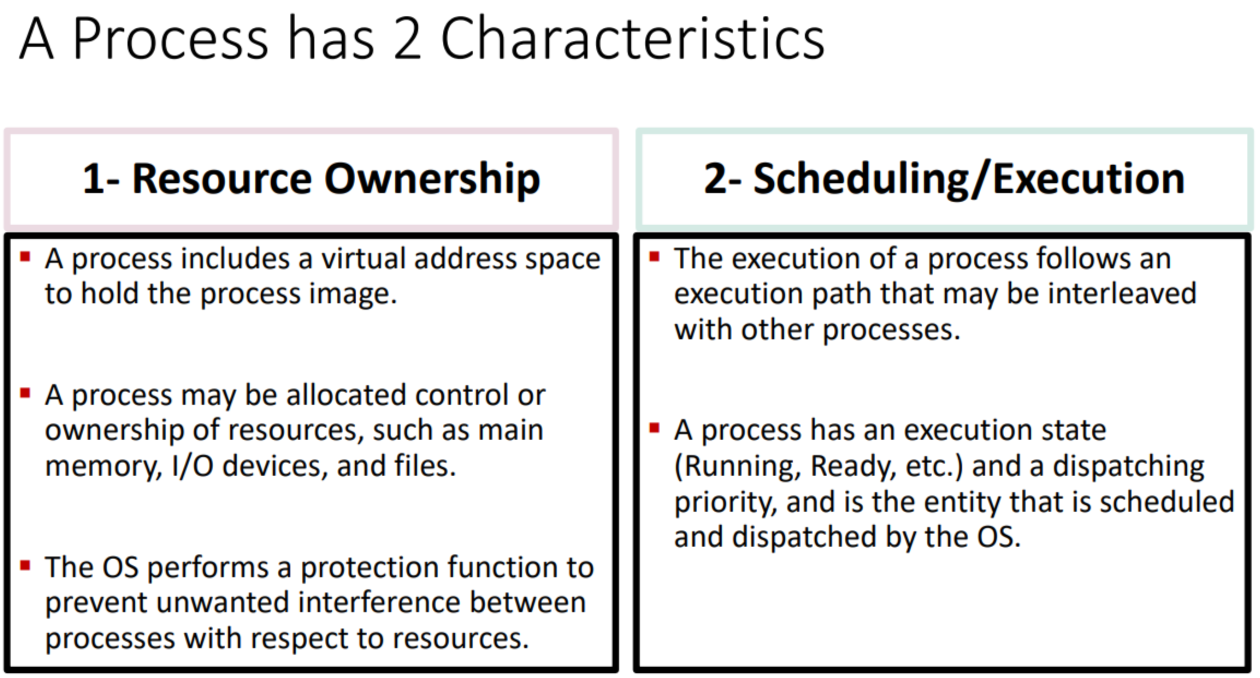
---

time for the threads lecture

windows, linux, and solaris process and thread management she leaves to us

---

Processes and Threads



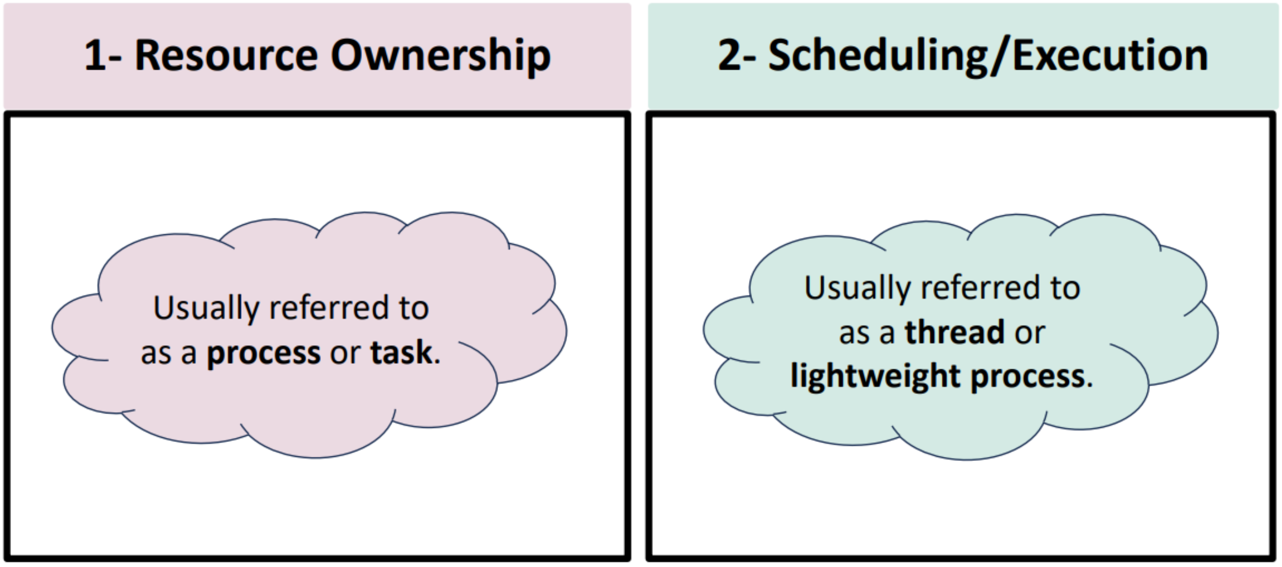
A Process has 2 Characteristics

1- Resource Ownership

- A process includes a virtual address space to hold the process image.
- A process may be allocated control or ownership of resources, such as main memory, I/O devices, and files.
- The OS performs a protection function to prevent unwanted interference between processes with respect to resources.

2- Scheduling/Execution

- The execution of a process follows an execution path that may be interleaved with other processes.
- A process has an execution state (Running, Ready, etc.) and a dispatching priority, and is the entity that is scheduled and dispatched by the OS.



1- Resource Ownership

Usually referred to as a **process** or **task**.

2- Scheduling/Execution

Usually referred to as a **thread** or **lightweight process**.

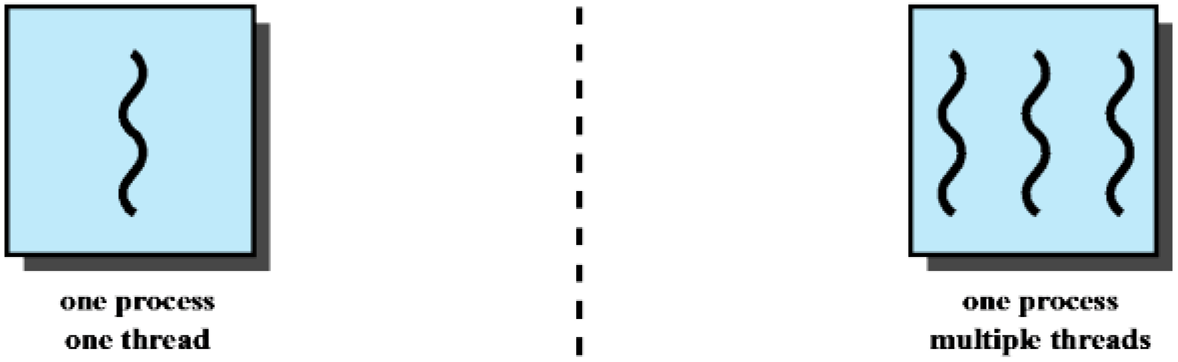
The actual fetching and execution of the instructions is thought of as a thread or lightweight process (LWP).

Most unix based processing will say threads.

Some OSes don’t differentiate between threads and processes, treating them all as processes.

THe process model introduced so far assumed that a process as an executing program with a single thread of control.

Virtually all modern operating sytems, however, provide features enabling aprocess to contain multiple threads of control.

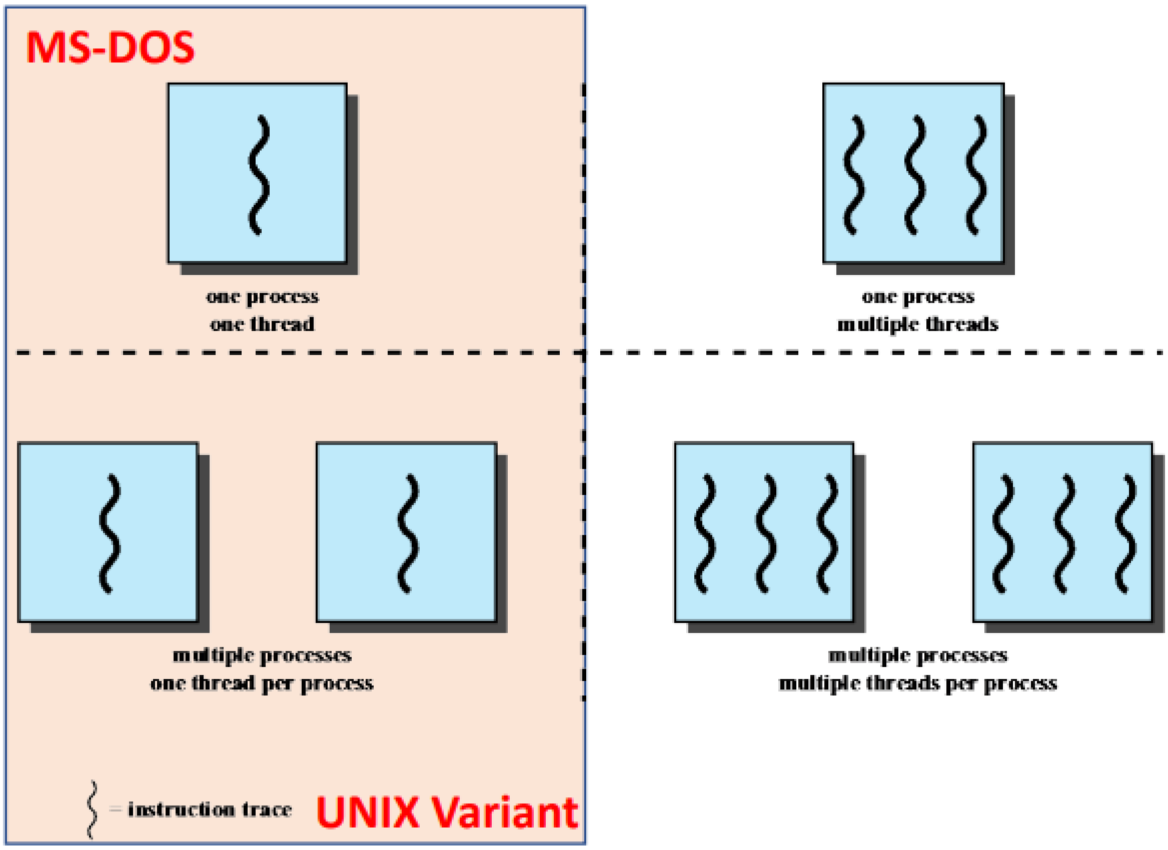


Threaded Approaches

Single threaded approaches

a single thread of execution per process

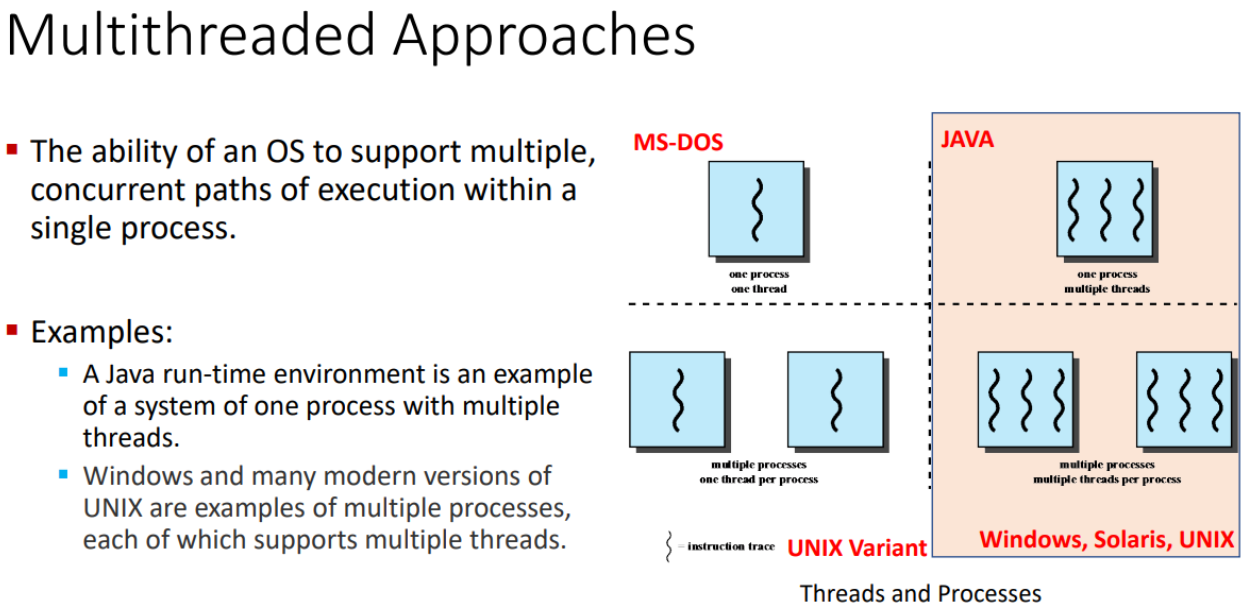
threads are not recognized



Threads and Processes

ex.

- ms-dos
- old linux



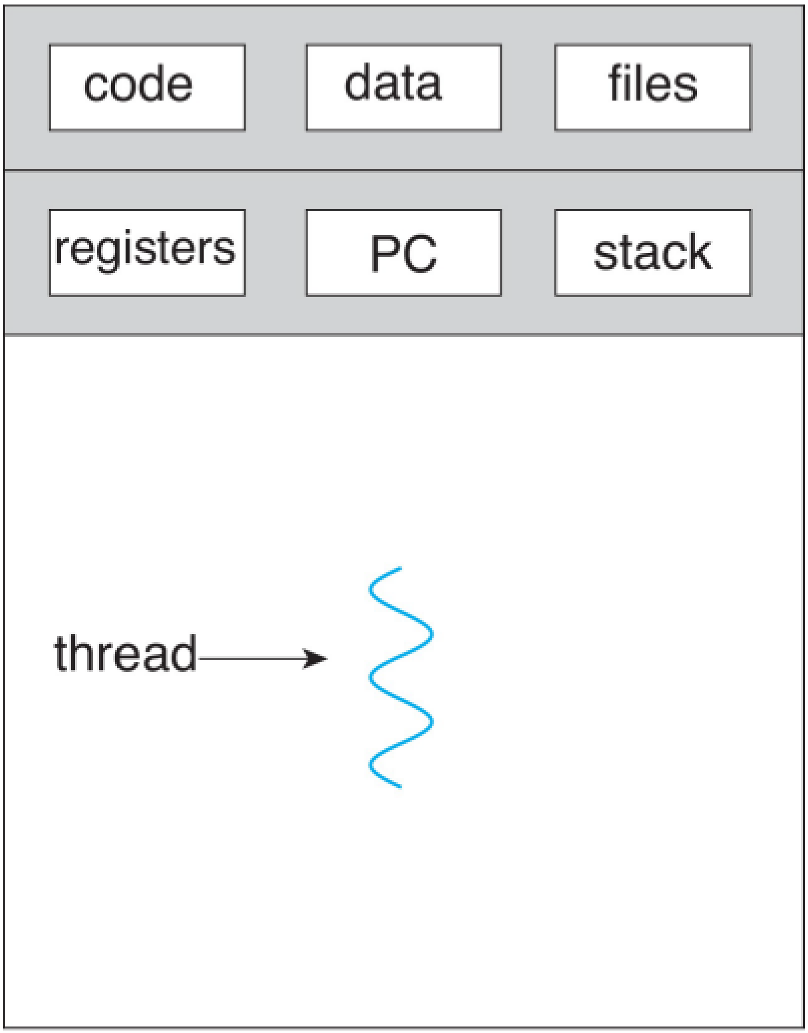
Multithreaded Approaches

- The ability of an OS to support multiple, concurrent paths of execution within a single process.
- Examples:
  - A Java run-time environment is an example of a system of one process with multiple threads.
  - Windows and many modern versions of UNIX are examples of multiple processes, each of which supports multiple threads.

Threads and Processes

we can make it seem as though the processes are all working simultaneously but we’re actually just switching around very quickly

What is a Thread

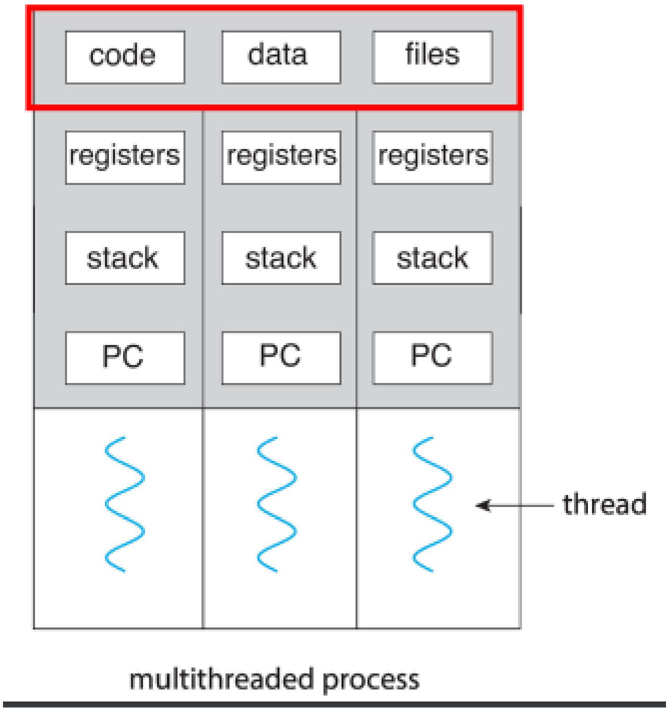


single-threaded process

thread is a basic unit of cpu utilization

- each thread knows its parent process
- thread ID
- program counter
- a register set
- a stack

thread = instruction trace



If a process has multiple threads of control then it can perform more than one task at a time.

Threads belonging to the same process

- share
  - code section
  - data section
  - other operating system resources
    - open files
    - signals
- have unqiue
  - registers
  - stack
  - PC
  - ID

Why Multithread?

Most modern applications are multhreaded

an application is typically implemented as a separate process with several threads of control

multiple tasks with the application can be implemented by separate threads for different reasons

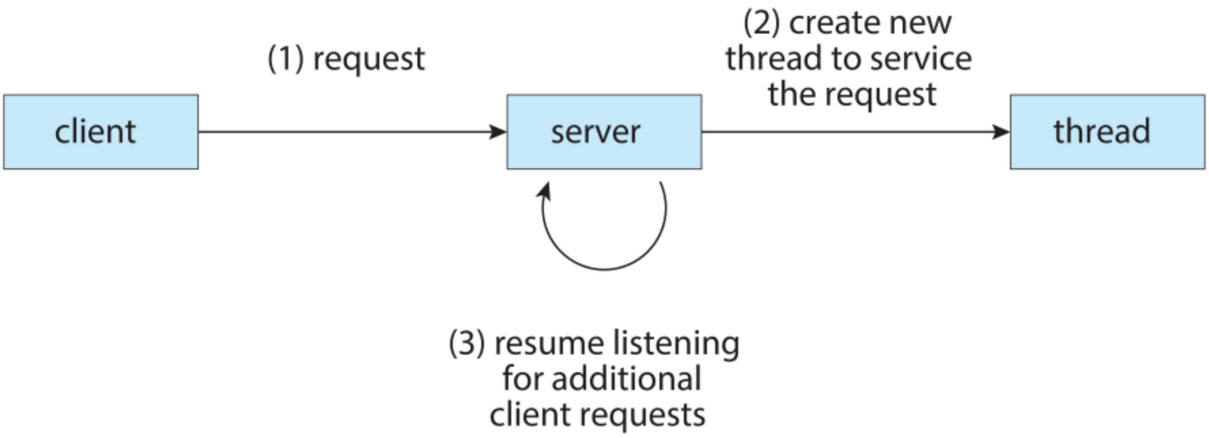
ex:

- making thumbnails of tons of different images at once

    - 1 thread per image
- loading a webpage
    - 1 thread for text
    - 1 thread for images
- word processors
    - 1 thread for responding to keystrokes
    - 1 thread for displaying graphics
    - 1 thread for spell checking

we feed each thread a different task and they don't interfere with one another

multithreading is also good for client/server applications


(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests

Multithreaded Server Architecture

Server will have

- have a port open and a thread to listen for connection requests
- make a thread to do what the request wants
- the open port
    - very vulnerable
    - an open door
    - necessary as otherwise no one can connect

Q: why multithread and not multi proces

A: It's much faster

- process creation is heavywhite while thread creation is lightweight
- creation, termination, and switching of threads is faster than with processes
- threads are ten times faster to create than processes in UNIX

kernels are generally multithreaded

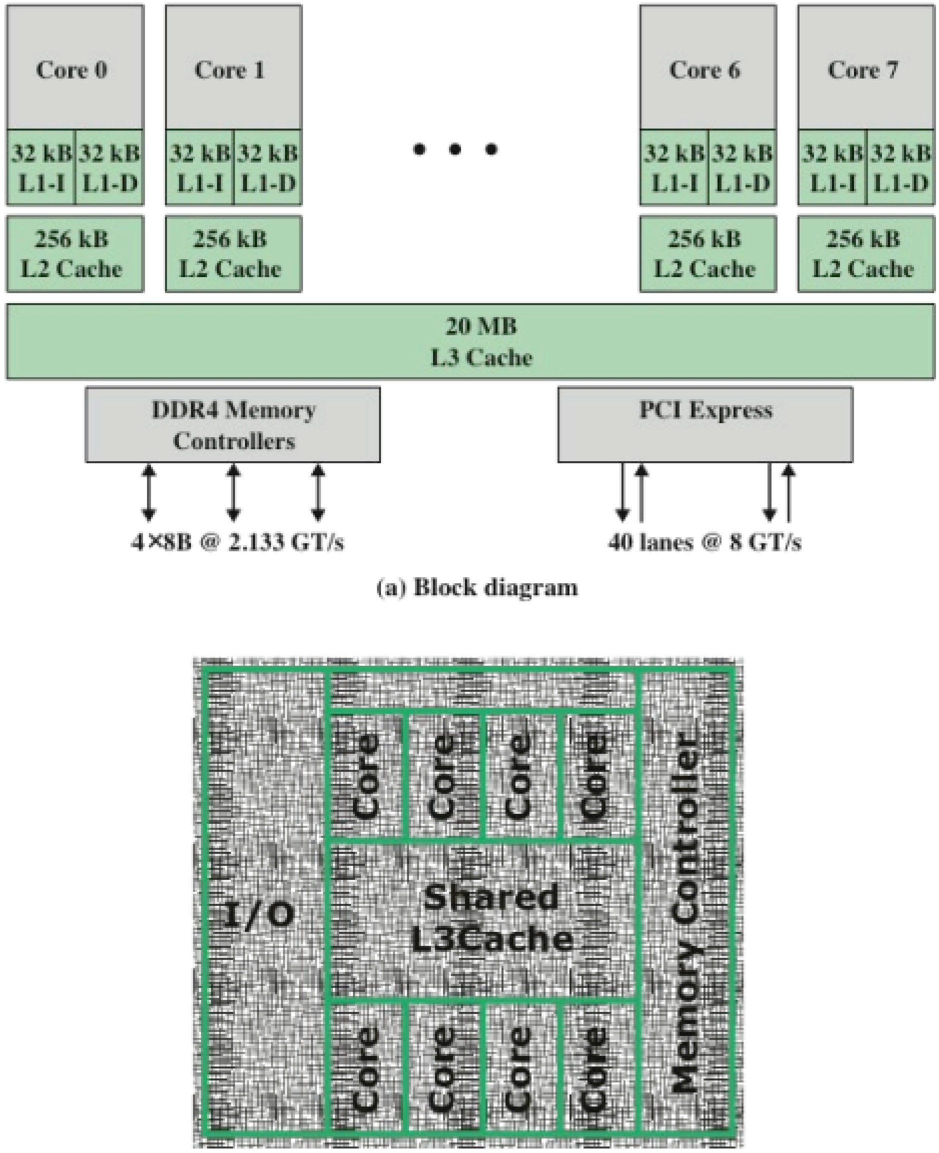
- during system boot time on linux system, several kernel threads are created
- each thread performs a specific task
    - device management
    - memory management

    - interrup handling
    - etc

benefits of multithreading

- responsiveness
    - allow contonued execution if part of th eprocess is blocked
    - good for user interfaces
- resource sharing
    - threads share resources of process
    - easier than shared memory or message passing b/w processes
- economy
    - cheaper than process creation
    - thread switching lower overhead than context switching
- scalability
    - process can take advantage of multicore architectures
    - one process can use at most one core
    - multi-threads could use all available cores or CPUs
    - speed of execution

Multicore Programming



(a) Block diagram

(b) Physical layout on chip

Figure 1.20 Intel Core i7-5960X Block Diagram

multicore:

- multiple computing cores on a single chip
- each core appears as a separate CPU to the operating system
- each core can execute 1 trace at a time

```
word on caches, specifically in this intel architecture

L1 cache has all of the instructions and stuff.
All the highest value things

L2 can allow communication between 2 cores physically next to one another. They look
separate in the diagram but that's just the allocation we do, they can look at each
other's work and talk to one another basically.

L3 is on the chip and is shared by all of the cores
```

```
There are different kinds of architectures out there so the interesing L2 cache
sharing won't be true of every architecture.
```

multithreaded programming

- provides a mechanism for more efficeint use of these multiple computing cores and improved concurrency

we need to program properly in order to take advantage of the presence of multithreading

Concurrency vs Parallelism

Parallelism

- implies a system can perform more than one task simultaneously
- 2 cores doing 2 threads at the same time
- working in parallel

Concurrency

- supports more than one task by allowing all the tasks to make progress

Q: Is it possible to have concurrency without parallelism?

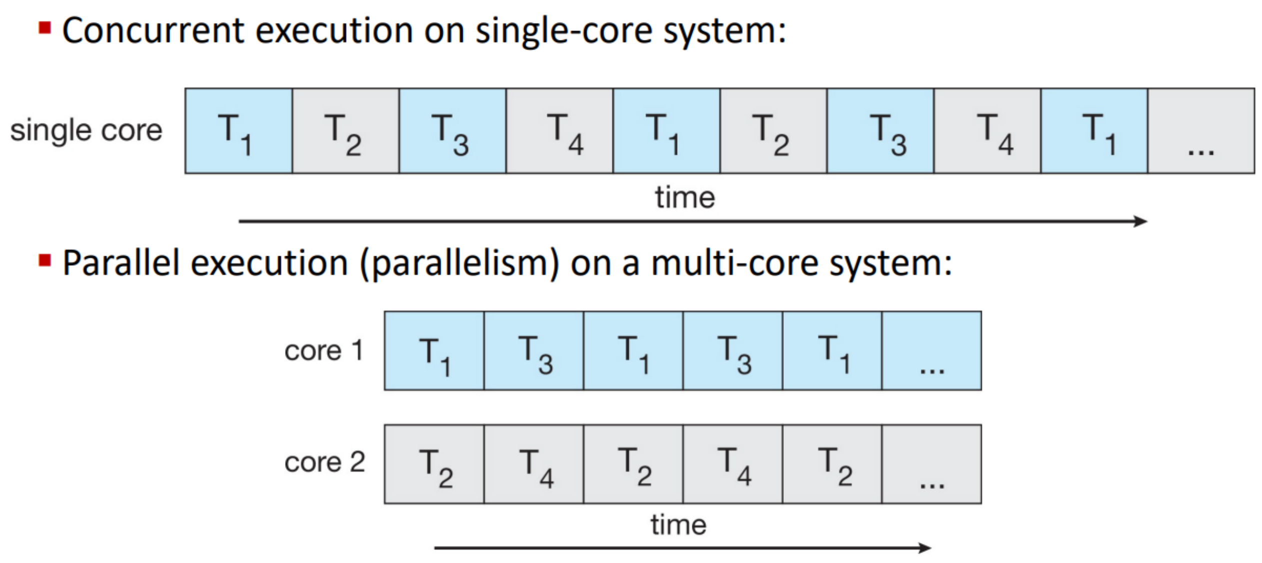
A: yes. utilize thread switching. parallelism requires parallel execution. We can have all the tasks progress together without parallel execution by way of thread switching.

```
parallelism cannot happen on a single core

parallelism implies concurrency

there can be concurrency on a single core

parallelism is faster than concurrency alone
```



- Concurrent execution on single-core system:

- Parallel execution (parallelism) on a multi-core system:

We are either parallelising data or tasks

data parallelism:

- distribute distinctive chunk of data across multiple cores, and perofmr the same operation on each
- ex
  - each core gets a different image
  - each core performs the same operation of creating a thumbnail

task parallelism

- distributing threads across cores
- each thread performs a unique operation
- not necessarily all working on the same data
  - 1 core is summing rows
  - 1 core is averaging rows
  - 1 core is finding the median
  - all may or may not be working on the same rows

---

lect end at ahmdahl’s law

Multithreading Models