Von Neumann Architecture:

CPU connected to memory and bus

- Program on memory
  - Performs any execution -> from memory to CPU
    - Becomes a process

Interrupts help move stuff between the cpu and devices

- Treated on the kernel/OS mode

User mode

- Write a program
  - Asks help from kernel to perform task

Kernel mode

- Moves from user to kernel mode and special bit to tell us where we are
- Learn about IDT table (interrupt descripter table and vectors)
  - Pass through interrupt
    - Polling or interrupt vector
      - Detects typ of interrupt
        - We get address which guides us to function to solve task of interrupt
          - Then it goes down to hardware to perform task
            - When task finishes, return to user mode
            - Next slide -> go to IDT table, indicate we have interrupt (detects) of hardware and return

That is not context switching

- Context switch is between processes (2 different ones)

Processes

- Communicate between processes – exchange info
  - Networking
    - Between 2 computers
    - Internet etc

What is a process?

- Program executes and it becomes a process
- Program is passive
- Process is active execution

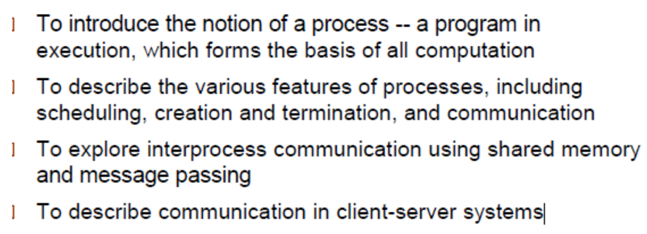
Schedule processes

How to make one

How to end a process

Comms

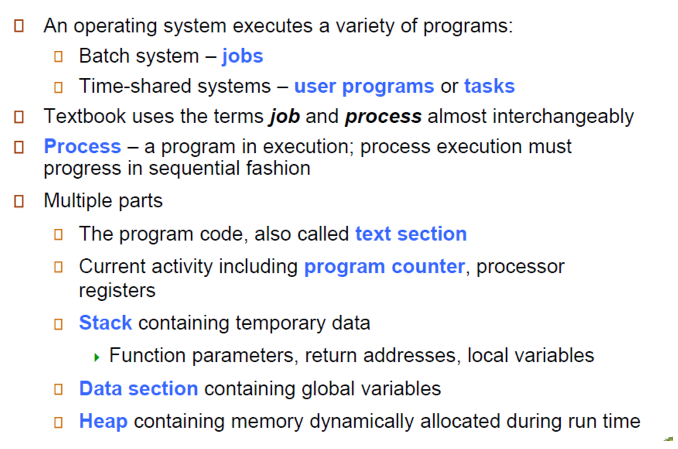
- Process talking to itself
  - Or with different processes



- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explore interprocess communication using shared memory and message passing
- To describe communication in client-server systems

- **Shared memory**
- **Message passing memory**
- Both computers use these 2 approaches

Client-server on Linux/windows



- An operating system executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**
- Textbook uses the terms ***job*** and ***process*** almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

Execute program = process

OS helps perform task

Jobs in unix world

Similar to processes

Batch

- A bunch of statements
- Performed after another
- In a series
- Simple

Time share

- Many tasks or programs run together
- Make the illusion of it running In parallel
- Multitasking
  - Switches program quickly to make it appear like running together
  - 20 years old

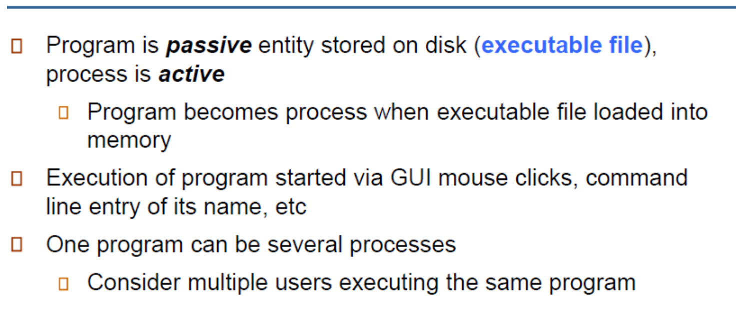
Stack = static

Process in sequential fashion

- Finish one line, go to next

Process components

- Program code
  - Special memory
    - Line by line for program
  - Text section
    - **static**
  - And tools for execution
- Program counter
  - Which line of code to point to
- Processer registers
  - Move program from memory to cpu
    - Bring from memory via bus from memory to register in cpu via cache and bus
- Stack
  - Function params
  - RA
  - Local variable
    - Bring to function
  - Pointer to function
  - **Dynamic memory**
- **Data**
  - Data comes with process
    - Contains global processer

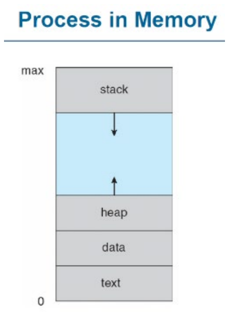


- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

variables
  - **Static**
- **Heap**
  - Also **dynamic allocation**
    - Malloc, new
    - During runtime

Perform switch from program to process

- Give some resources
- File/program has statements and ask for resources
- OS allows us to perform/execute program then becomes a process



Or type name of process or click on gui to start the program

mem space of each process

At least 4 different types

Stack = function pointers

- Call to function – recursive
- Stack increases
- If no stop, stack overflow, fill empty memory – and run out of room
  - Stack overflow

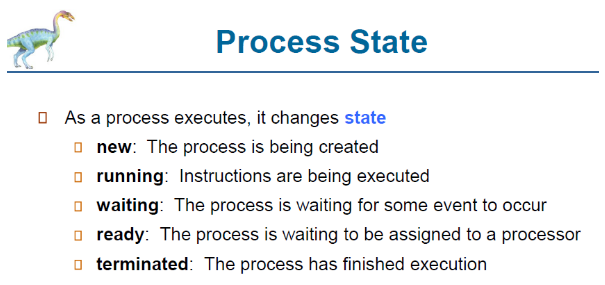
Heap

- Runtime allocation
- Competes with stack for same memory
- Important for writing program, remember to allocate and free memory
- Also dynamic
- Heap overflow
  - Rejecting dynamic allocation

Data = global

Text = static = code we wrote

--



Process State

- As a process executes, it changes **state**
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

Run = execute

Waiting for devices or memory

- Waiting to get resources
  - More memory for example
  - Moved to waiting while we don't have the memory were executed

Ready = all resources but for the processer

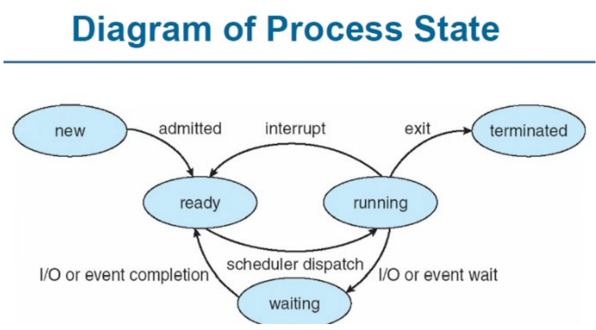
- Mem, files, network, but not cpu
- Still waiting for cpu

Terminated

- give back all resources to the OS
- asked for 1gb of memory but finished
  - give back to OS again to other processes

finished process didn't deallocate resources

- os is still alive but no more benefit of processes
- didn't give back resources
- OS should reallocate to same process as unused
    - **Ghost processes**
        - If many of them, it slows down
        - A lot of allocation but no one used

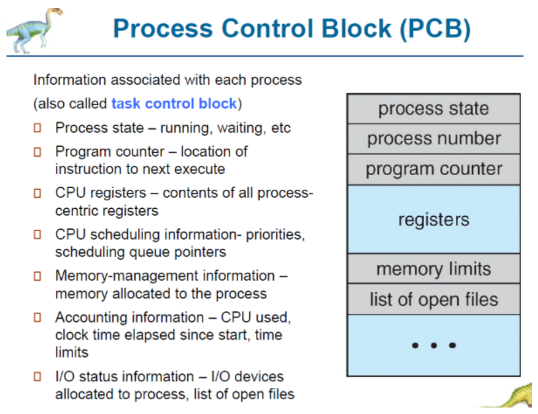


Remember by heart

- **Time sharing vs batch**
- Running can ask for more info or resources
    - Goes to waiting
        - Get resources -> go to ready
            - Then picks up process via scheduler
            - Then moves to running
    - Interrupt can bring back to ready
        - Exception would terminate
        - Time sharing
            - Each process has 10 minutes lets say
                - When time runs out
                    - We lost right to use it
                        - Use interrupt – indicates no more right
                            - Loses cpu
                                - Moves to ready state

Exit system call

- Syscall named exit
- Purpose to close and terminate process
- Bring back resources to OS



Process Control Block (PCB)

Information associated with each process (also called **task control block**)

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files

1 state to another = interrupt which allows to change states

data structure

different fields

1st field of process

PCB – each process has a PCB

PCB of process 1

State – where located? New/ready/waiting. Etc

Id – process #

- 1 in our case
- Another process comes into OS gets next number
- How multi threading works
    - Running processes together
    - And gives us overview
- PC – line to run
- Registers
    - Store info, connected to process
- Mem limit
    - Max and min mem can used

- List of open file
    - Syscall to open to access file from hard disk or usb
    - Important to open and close
        - If its open and not closed, system has an issue

Multiple programs same open files need some synchronization

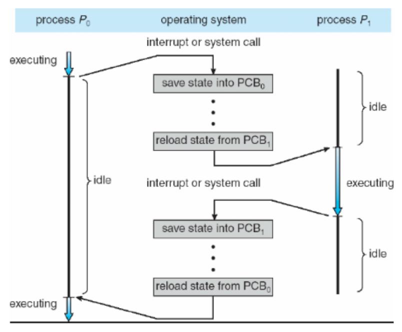
More fields:

- Accounting – stats get put here
- Os can perform better decisions based on this info

I/o status

- Info on devices connected to process

CPU Switch From Process to Process



2 different processes with their each PCBs

P0 is running (using cpu)

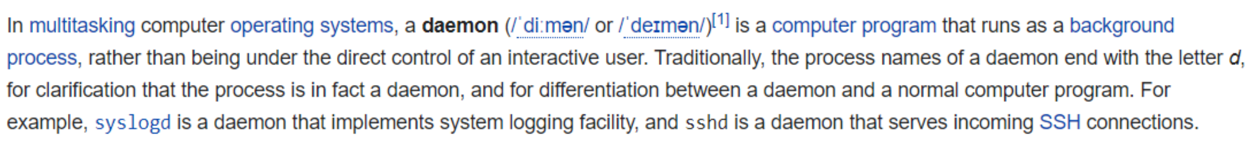
Interrupt to stop0

Now chose a new process assuming p1 is in ready state

Now switch p1 with p0

Catch interrupt

- Polling
- Or interrupt vectors


In multitasking computer operating systems, a **daemon** (/ˈdiːmən/ or /ˈdeɪmən/)[1] is a computer program that runs as a background process, rather than being under the direct control of an interactive user. Traditionally, the process names of a daemon end with the letter *d*, for clarification that the process is in fact a daemon, and for differentiation between a daemon and a normal computer program. For example, `syslogd` is a daemon that implements system logging facility, and `sshd` is a daemon that serves incoming SSH connections.

Pcb 0 is saved, bring pcb 1 to system

- Run on cpu
- Reload op on pcb
    - Then run

Delay between system running

- **Overhead** of OS – takes time
- Different operations that OS performs
- Save PCB and reload PCB1

OS goal is to reduce the overhead of these operations

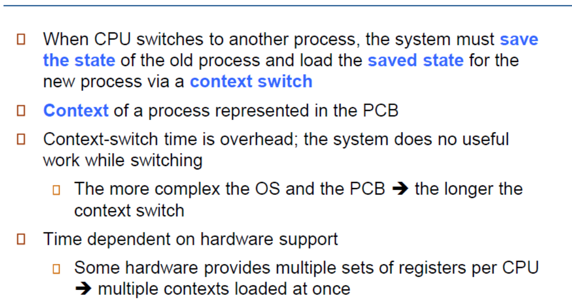
Assuming we have 2 processes

- Save pcb of process 1 and reload pcb of process 0

Optimization = reduce overhead

Context switching

Save and reload operation – minimize time spent


- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
    - The more complex the OS and the PCB ➔ the longer the context switch
- Time dependent on hardware support
    - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once

- Hardware might help us with PCB and better context switching

Program executing a process on a processer

- Passive
- 1 access to CPU (processes)
- **Single threaded**
- 1 huge operation – active on CPU

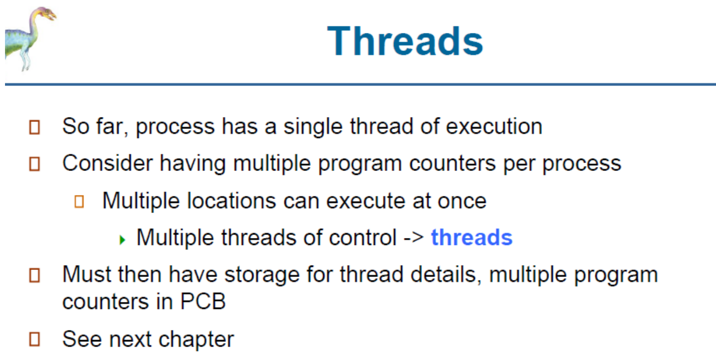
Thread

- More than 1 thread
- 1 unit of operation
- Perform a lot of operations in parallel

Memory = heap, data, text, stack

Multi-thread

- Powerful CPU
- Each thread has the code and PC
- Each thread has its own stack, register
- PC
- All the rest is common to all threads on that cpu
- Each thread is having its own power
- Parallel / concurrent execution



Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB
- See next chapter

Concurrent

0 run in parallel but each thread might have race to access power of each thread

- Parallel but uses time sharing

How to switch between threads?

Lot of processes running at same time

- How to schedule CPU time?

Process scheduler

- Uses notion of queue to help control exec of process

Different types of queues

Job – all processes in system

Ready – store all processes waiting to be from ready to running state

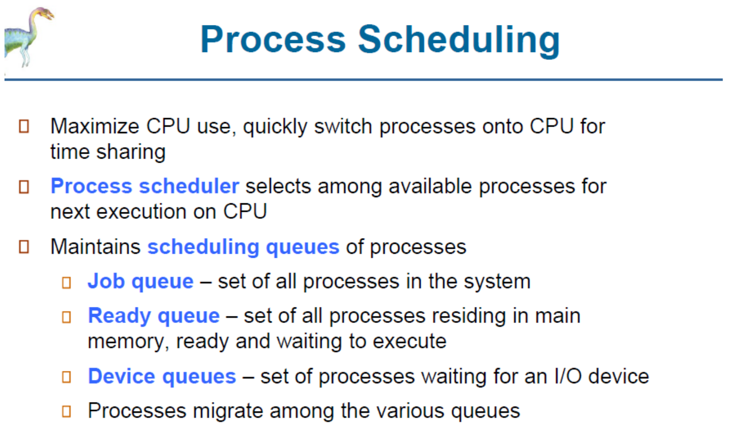
Device queue – where In waiting state?

OS migrates from 1 queue to another

- Via states
- Manipulation

Migration of process

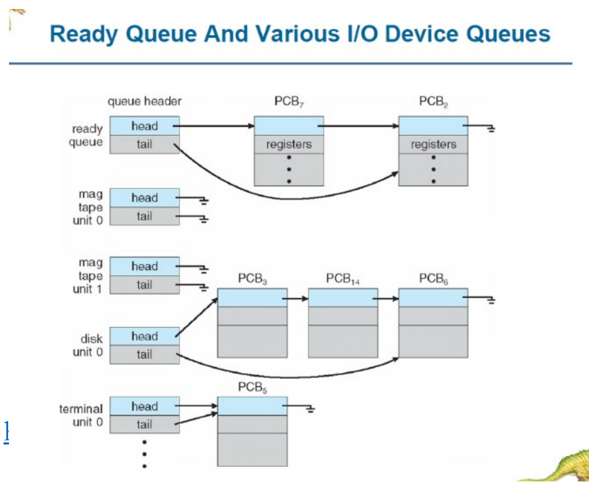
- Moving pcb connected to process to another queue



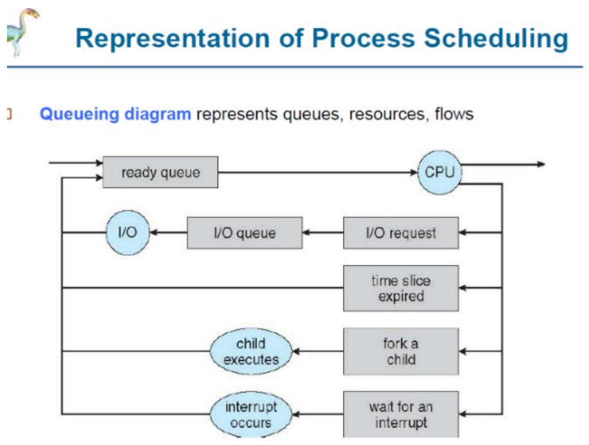
Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **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
  - Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



© 1



Linked list or other algorithms

Or vector

Or tree data structure

- FIFO policy
- Where to put process based on policy to manage queue
- 2 processes
- Pcb 7 and pcb 2
  - Waiting to move to running state
    - Each device on computer
      - Has a queue
        - Disks have queues
          - Coming from waiting state

Disk is one unit, usb, ram etc…

- Hard disk
  - Has 3 processes waiting to access disk
    - Via plicies , select what one we need
    - Pcb access to disk
      - Taken out and to readystate
        - End or start based on OS policy

Its an art of the OS to manage a queue properly and avoid jams

States

- New
  - Automatically go to ready
  - All resources but no cpu
  - Highest priority = get cpu right away

After a while , time slice expired, interrupt now move back to ready queue

- New processes come up, s lets chose from ready queue
  - Controlled by a data struct

i/o

- Process runs and requests specific request
  - To access queue of device
    - Selected by device queue
      - Performs op, then go to ready queue

Then select one of them

Process giving birth to another process = fork a child

- From 1 process to another
- Important and powerful system call
- 2 processes run -> executed
  - Parents and newborn together?
- Child executes – have everything … then goes back to ready queue
  - Skips ready queue

© https://cs.aviparshan.com/os

- Special fn of system call

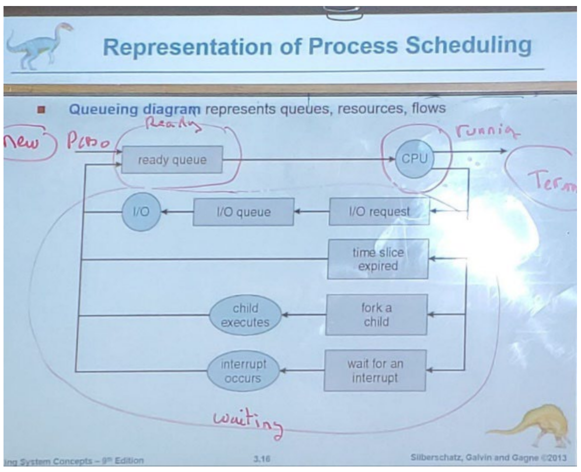
interrupt during execution

- Request for new memory allocation
- Then wait for request till its complete
  - Then go back to ready queue

Waiting for interrupt

- CPU waiting for interrupt to happen
- IDT table

Learn from diagrams:



Representation of Process Scheduling

- Queueing diagram represents queues, resources, flows

Schedules

- Long term
  - Select processes from waiting state to ready queue
  - Takes a lot of time
  - Seconds, minutes, or more
  - Ask to download a huge file
    - Hours or days
  - Optimize schedule to bring as much as possible and as fast as possible to the ready queue
- Short term
  - Right away execute cpu process
  - Frequent ops based on OS, algos, etc
  - Milliseconds or faster

This defines degree of multiprogramming

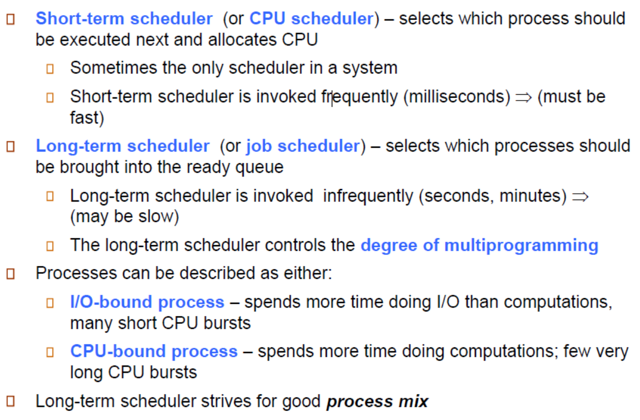
- Multiprogramming
  - Power to run a bunch of programs together
  - Bring as many processes as possible to ready state
  - Who is involved?
    - Short term schedule via time slice
    - Or takes a lot of time from I/O queue
      - Device can be busy, lots of processes requesting device
- Can take a long time
- More stuff in ready queue = better degree of multiprogramming
  - # of processes can run

2 different process types in any computers

- i/o bound
  - memory
  - in/out ops for processer
  - short cpu bursts
    - cpu not used so frequently
    - lots of IO requests
- cpu bound
  - need a lot of computation
  - don't access io devices so much

we want to mix as much as possible to optimize long term scheduler

IO bound example



- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***

- downloading pictures

CPU bound process

- editing images

cpu bound but to gpu reduces the operation

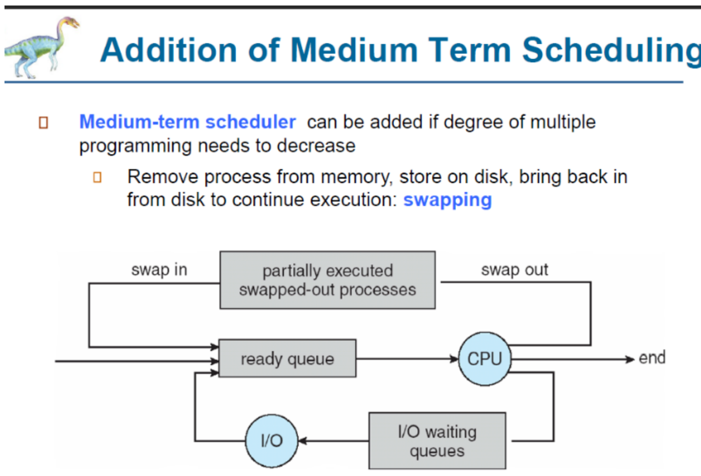
- and moves it to gpu
    - demanding use of cpu/gpu

matrices, calculations

- alu unit operations
- cpu bound

some OS have medium term schedule

- time slice expired
- still have resources, no long term scheduler needed
- use a swap in/out to bring process from running to ready state fast



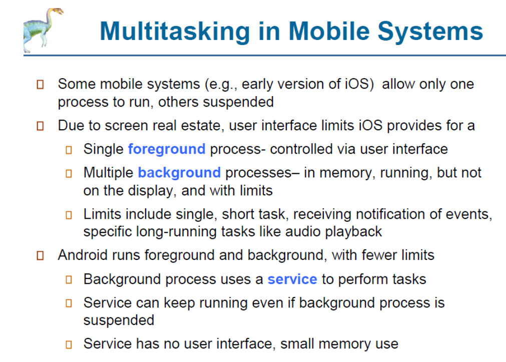
Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
    - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**

write a code which optimizes to perform the swap operation in 2 statements with no additional variables

```
var1 = 10

   var2 = 5

   # swap in 2 lines

   var1 = var1 + var2

   var2 = var1 - var2

   var1 = var1 - var2
```

swap is now efficient



Multitasking in Mobile Systems

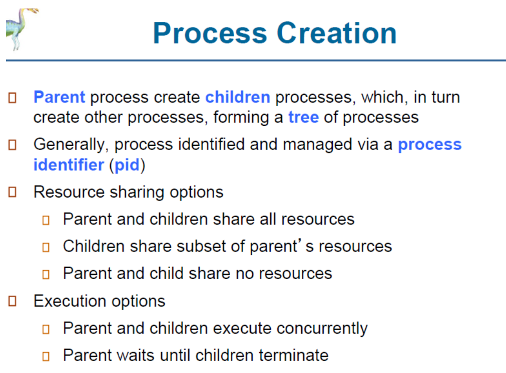
- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
    - Single **foreground** process- controlled via user interface
    - Multiple **background** processes– in memory, running, but not on the display, and with limits
    - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
    - Background process uses a **service** to perform tasks
    - Service can keep running even if background process is suspended
    - Service has no user interface, small memory use

To manage windows, kernel needs processes to do context switches etc

System must provide mechanisms for:
process creation,

process termination,

and so on as detailed next



Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
    - Parent and children share all resources
    - Children share subset of parent's resources
    - Parent and child share no resources
- Execution options
    - Parent and children execute concurrently
    - Parent waits until children terminate

process create = fork

process terminate

Creation from existing process – parent process

- who made a child
  - child can me another process =
  - then we can have a grandparent process

PID – also managed by PCB connected to process

1 process can have many children – not a binary tree

New process characteristics depend on which OS we use

- option

execution options

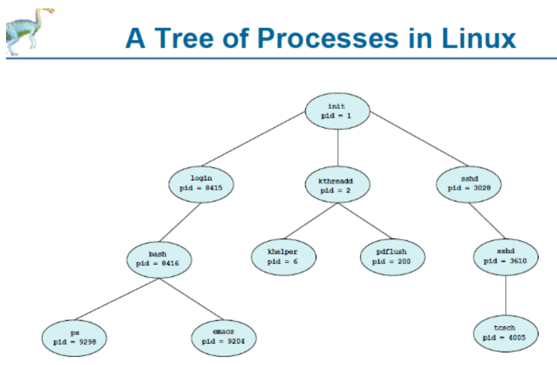
- parent and child concurrently
  - together
- or parent waits till children terminate
  - client server
  - master-slave
    - parent = master
    - children = slave and serve the parents

example of parent and process examples:

- build a building
  - construction manager tells class to take stones and glue and some people bring stones and some put glue to build a wall fast
  - babel tower
- or work together (same task)
  - lanes on highway
  - on toll road, all people collect money at same time

a + 1 same time in parallel

- parent is a and children is the 1



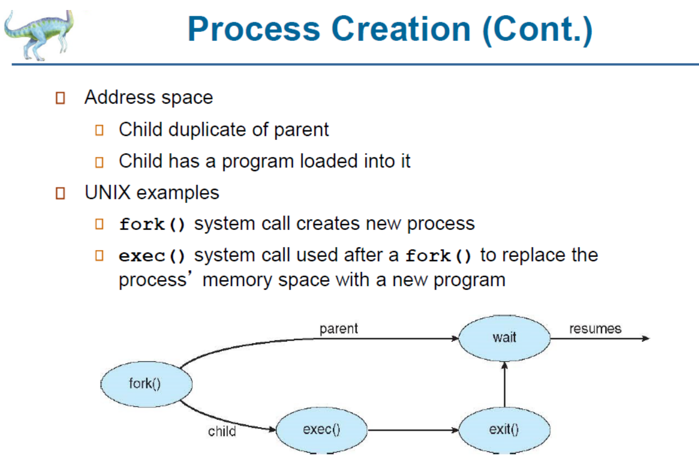
A Tree of Processes in Linux

Init makes different processes which create other task processes

- tree of processes for linux OS

windows is same

each proceed has PID # and pcb to follow process



Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - `fork()` system call creates new process
  - `exec()` system call used after a `fork()` to replace the process' memory space with a new program

parent
wait
resumes
fork()
child
exec()
exit()

- syscall on unix
  fork makes a new process
- master/slave approach
  - child performs op and parent does nothing and waits for child

parent and child run concurrently

parent has flag and waits for child to continue operation

child sends interrupt to parent (exit) then parent continues

- both were in running

© https://cs.aviparshan.com/os

Writing realtime or parallel code

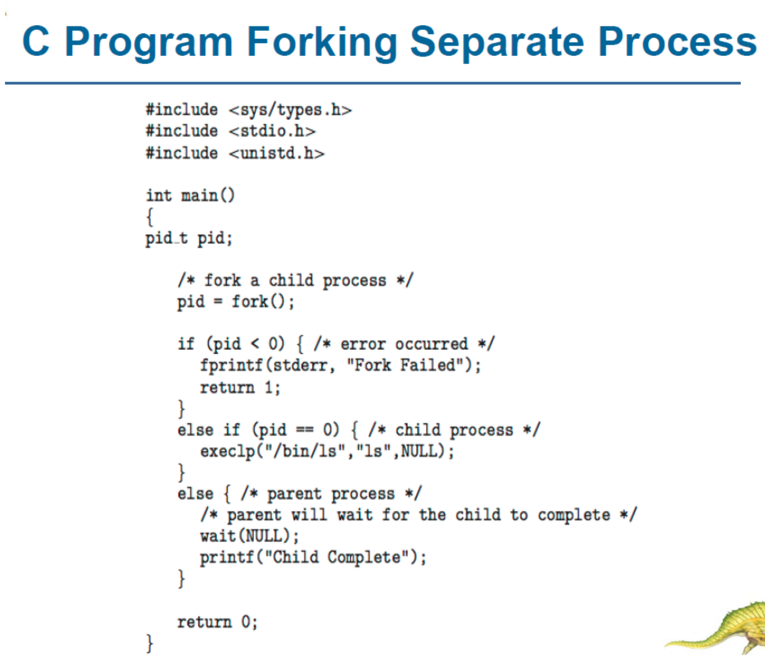
Draw diagram and code via that

# include <iostream> now to do it in c++

Main function

- defines beginning of process
- when we run code
    - lines are in text memory
- program runs , PC comes at main start

main fn – indicate what type of allocation, how much to allocate to process and if needed a special operation



C Program Forking Separate Process

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

int main()
{
pid_t pid;

   /* fork a child process */
   pid = fork();

   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 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");
   }

   return 0;
}
```

main = beginning of process

- memory and pc
- allocate local variable PID
    - pid.t pid
        - different type
        - special type , like an integer (long one)
        - name of the type is pid.t
        - typedef – define a new type based on a definition of old type
        - allocation performed
            - stack – temporary variable
            - in main, go to stack, pointer with fn name and then {} = scope of main function
- fork() syscall
    - create a new process
        - special fn, in same time 2 processes running, parent and child
            - assuming it was successful

pid return depends on if running child or parent

- can have 2 different values , if cpu running child or parent process

each process return same value, saving pid, but id is different for each person

fork(0 returns based on which process running

- pid parent if parent is running
- if fork is running the child
    - hes returning value of child

2 possible values

- <0 = error
    - – negative value, stop program

Purpose = make child and run child code

Pid =0

- Meaning the child is running, only me and my children running
- Does child know ID when created?
    - Only parent knows id of child

Parent running, child – value of pid fork is child

- Positive value
    - In the else
    - Parent is just waiting for child to finish
    - Wait(NULL)
        - Uses OS signal and allows parent to continue

But if pid is child

- Bc child doesn't know ID
- It performs execute
- Syscall
- Ls – list file in directory

This code style uses the option

- Unix/linux
    - **Parent and child share all resources**
    - Parent resource is all the memory, file open, everything
        - Child is born and gets access to these too
        - Implication
            - Perform a copy of all resources from parent
                - Text (code)
                    - Duplication

Child – duplicates the code

- Running by parent, and also the child at same time

PC is another resource

- Child and parent
    - Share same PC and next instruction
        - Next after fork() then pid assign is the next instruction
        - Pid = fork();
            - One returned by child and one by parent
            - Same code
            - Pc isn't going on the same code at same rate

How to specify task of parent or child, via control statement

- Guide same code to perform code via parent or child
    - In parallel

Perform different tasks

When child is done , sends to interrupt to say im the child and I finished

- Pcb has field with resources
- And new process connected to pc of process
- To terminate or finish process, return all resources

Base of parrale execution

Write cout for pid

Shalom

- Child first

- Then parent

Operation cout << shalom run by both processes

2 of these share the same terminal

How to make a process in windows now

Function of API of windows kernel

- Less powerful and less optimized than syscall
- CreateProcess
    - Makes 2 actions
    - Fork and parent waiting
    - And makes child and execution
    - Does it automatically

Linux needs to specify state by state

is everything a child process of the OS?

- init
- each process has a connection with system

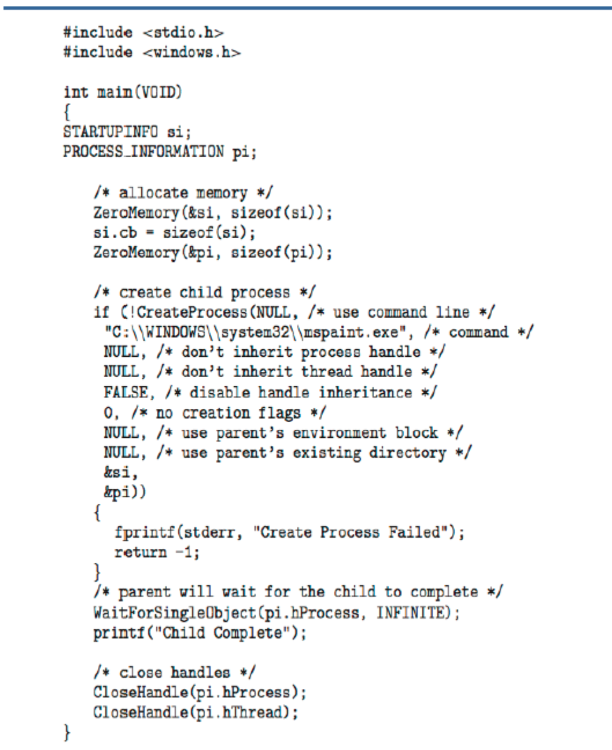
if closes init, everything shut dows and closes

- cascade effect

if one process dies

- child doesn't die
    - but the OS becomes the guardian of the child process
    - the only process forever, is the OS (init)
        - takes care of this process and bring resources to process
        - zombie processes
            - don't belong to anyone

runs ms paint program



```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
      fprintf(stderr, "Create Process Failed");
      return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

© https://cs.aviparshan.com/os