chapter 3

* Process : a Program in execution → must Progress sequentially

in time sharing systems: unit of work, tasks

all Processes are executed concurrently

in batch systems : job

~~

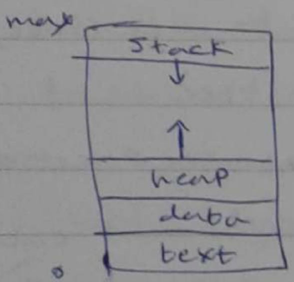
Program code → text section

current activity → represented by Program Counter and contents of registers

Process stack → contains temporary data (function Params ...)

data section → contains global variables

heap → contains dynamically allocated memory & during run time



$\text{Program(Passive)} \xrightarrow{\text{load to memory}} \text{progress (active)}$

exe file

a Program can have several Processes

⇒ { example : web browser

same text section

JVM

a Process can be an execution environment for other code

* Proces State

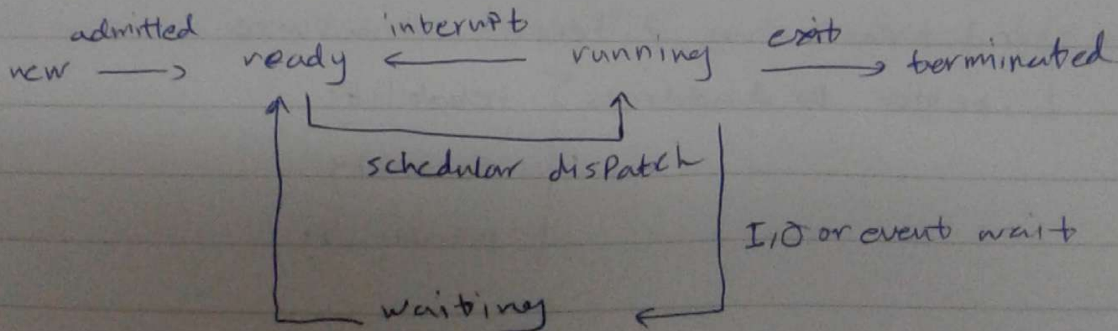
new: Process is being created

running: instructions are being executed → only one Process can be running

waiting: waiting for some event to occur

ready: waiting to be assigned to a Processor

terminated: has finished execution



* Process Control block (PCB) (also called task Control Block)

each Process is represented by a PCB

PCB ⟶ Process state

└⟶ Program counter

└⟶ CPU registers ⟶ along with PC must be saved during an interupt

└⟶ CPU scheduling info = Process Priority + Pointer to scheduling queues + ...

└⟶ mem management info = base and limit registers + ...

└⟶ accounting info = CPU and real time usage + time limits + ...

└⟶ I/O status info = list of I/O devices, open files and ...

* Process scheduling

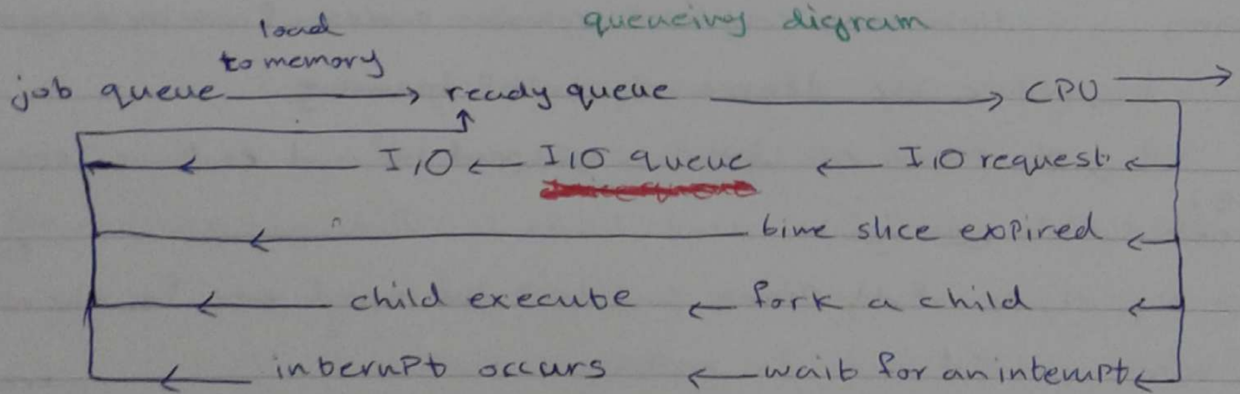
multiProgramming: have some Process running at all times

⟹ maximize CPU utilization ~~time~~

time sharing: switch the CPU among processes frequently

⟹ scheduling

queueing digram



ready queue: stored as a linked list

header contains Pointers to first and last PCB

each device has it's own device queue

Process termination ⟶ Process is removed from all queus and has it's PCB and resources deallocated

* Schedulers

- long-term scheduler (job scheduler) ↳ Job queue to ready queue
  - ↳ is invoked infrequently (seconds, mins) ⟹ may be slow
  - ↳ controls the degree of multiprogramming (# of processes in memory)
- short-time scheduler (CPU scheduler) ↳ ready queue to CPU
  - ↳ is invoked frequently ⟹ must be fast (ms)
  - ↳ sometimes the only scheduler in a system

* if the degree of multiprogramming is stable, the long-term scheduler may need to be invoked only when a process leaves the system.

* if there's no long-term schedular, the stability of the system depends on ⟶ physical limitations (number of terminals and ...)
  ↳ human self-adjusting nature

Processes ⟶ I/O bound ⟶ more time doing I/O (device queue)
↳ CPU bound ⟶ more time doing computations (ready queue)
⟹ long-term schedular must balance these processes

- swapping ⟶ medium-term schedular removes process from memory and CPU
  - ↳ reduce the degree of multiprogramming
  - ↳ the process can later be swapped in and continue execution

(PCB)
* Context switch (تعويض عن)
- ↳ save the state of old process and load the saved state for the new process
- ↳ is pure overhead
- ↳ the more complex the system (O.S., hardware) and PCB ≡ the longer the context switch
- ↳ time dependent on hardware (multiple set of registers ≡ multiple context)

* Process Creation

Parents + children ⟶ Process tree   Parent has children's Pid

Process identifier (Pid) ⟶ integr

resource sharing option {
- children share subset of parent's resources
  ⟹ prevents processes from overloading the sys
- children and parent share all resources
- children and parent share no resources

Parent can initialize data to child

execution option → Parent and child execute concurrently

↳ Parent waits until children terminate

address space possibilities { child duplicate of Parent → fork()

child has a Program loaded into it → exec()

exec() → loads a new binary file to memory (Process address space)

wait() → moves Process off the ready queue

↳ does not return control until an error occurs

Parent → Pid = fork()

(Pid > 0) Parent → wait → Parent resumes

(Pid = 0) child → exec() → exit() → wait

* Process termination → exit() system call

↳ Process returns an integer value to it's Parent via wait() system call

↳ all Process resources are deallocated

termination → when a Process finishes executing

↳ caused by a system call (TerminateProcess() in windows)

↳ can only be called by Parent using Pid

Parent terminating child → child task in no longer required

↳ Parent is exiting → cascading termination (OS)

↳ child exceeding it's usage of some of the resources

↳ child entry in the Process table remains there 'till Parent calls wait()

wait() → Zombie → exist only briefly

→ orphan → (Linux) assigning $\boxed{\text{init}}$ Process as parent

→ root of the Process tree

→ calls wait()

* interprocess communication

Process → independent → can't affect or be affected → doesn't share data
↳ cooperating → can affect or be affected → shares data

adv ↳ info sharing → files and ...

" ↳ computation speedup → breaking a task into subtasks and running them in parallel

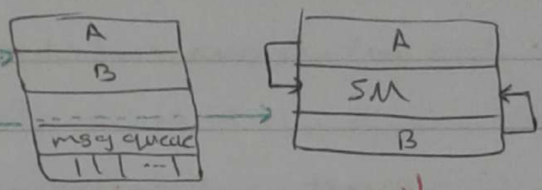
" ↳ modularity

" ↳ convinience

need ↳ interprocess communication (IPC)

↳ message passing

↳ shared memory



useful for small data

no conflict

easy to implement (system calls)

fast

no kernel intervention

* for multicore systems message passing is better *

* shared memory systems

shared memory region resides on the address space of the process creating it ⇒ other processes must add it to their address space

⇒ removing restrictions

Processes must not write to the same location simultaneously

⇒ Producer consumer problem → bounded buffer → consumer waits
↳ unbounded buffer → both wait

* Message passing systems ⇒ Communication link

useful for distributed systems

message can be fixed or variable in size

easy implementation — hard programming — easy programming — hard implementation send() / recieve() operations.

direct or indirect

direct → a link between every two processes (one and only one)

↳ each link associated with only two process

↳ processes must explicitly name each other

↳ symmetry or asymmetry

↳ symmetry → both processes know each other's name

↳ asymmetry → ~~only~~ sender knows receiver's name

↳ receiver knows sender's id

disadv ↳ limited modularity (hard coding)

indirect → messages sent to and recieved from ports or mailboxes

↳ each mailbox has a unique id

↳ two processes can communicate only if they have a shared port or mailbox ⇒ link

↳ a link may be associated with more than two processes

↳ two processes may have several links

↳ when a process that owns a port terminates, the port disappears → permissions

↳ a mailbox can be owned by the O.S.

* Synchronization

Message Passing → blocking (synchronous)

↳ non blocking (asynchronous)

blocking send() and receive() ⇒ rendezvous → producer - consumer problem