process contact

project scheduling

operation on process

interprocess communication

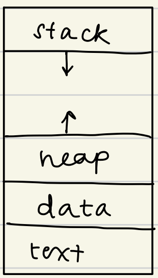
Processes and Threads

thread

Process concept

- process = job ⇒ program in execution
  (don't confuse with program which is a static concept)
- batch OS → jobs
  time sharing OS → user programs & commands
- processes must progress in a sequential fashion.
  ↳ dynamic processes

- **process in memory**



← parameters, local variables (grows top down)
eg normal & recursive calls.

← dynamically allocated variables (grows upwards)
eg. obj created using new.

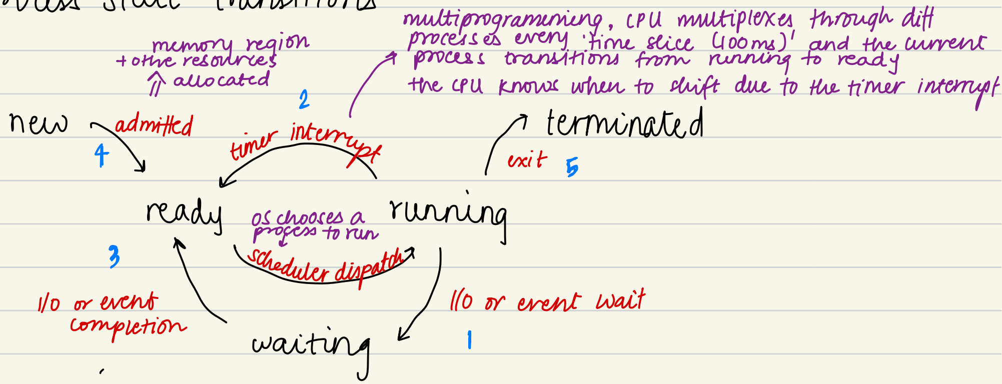
← global var

← code

- **process state**

1. new ← creation
2. running ← execution of instructions
3. waiting ← process is waiting for I/O orevents
4. ready ← ready to run, waiting for CPU
5. terminated ← done.

· process state transitions



· Process Control Block (PCB)

↓

· a data structure to maintain the state of the process

↓

· supports OS f^n of switching the process in the CPU acc to timer interrupts or I/O events wait.

base+limit ←

· PCB

Process state | Process number/ID

- Program counter
- CPU reg
- Process priority (used in scheduling)
- memory management info (base+limit reg val)
- info regarding files (list of open files)

base

→ PCB →

all PCB's are stored in a queue

· PCB's are stored in main memory and have hardware protection enabled

- PCBs are stored in kernel space, but processes are in the user space.
∴ every change to the PCB occurs in kernel mode

Process Scheduling

① Context Switch

- time-sharing, every 100 ms, timer interrupt + process change
- the OS comes into play in the kernel mode to update the PCBs of the switching jobs
  ↳ "saving context"
  + "load context"
- the context switch time is overhead ⇒ system does no useful work while switching b/w processes

② Process Scheduling Queues

- job queue : a queue of all processes with the same state
  eg. ready queue
  device queue (waiting on I/O) ← by device

all stored in memory in kernel space
- processes migrate b/w queues as state changes
- running process is stored as the first element in the queue

③ diff types of schedulers (all OS controlled)

- **long-term scheduler (or job scheduler)**: selects processes from disks and loads them into main memory for execution
  → basically "creates" the project / "admit"s the project

infrequent
initially controls degree of multiprog.

- **short-term scheduler (or CPU scheduler)**: selects from among the processes that are ready to execute + allocates the CPU to one of them
  → chooses those that transition from 'ready' to 'running'

invoked frequently

- **medium-term scheduler**:
  → when system load is heavy, swaps out partially executed process from memory to hard disk
  → when system load is light, such processes are swapped back to main memory from disk
  → responsible for adjusting the degree of multiprogramming
  → VM allows this

multiple processes, single processor.

takes over long-term scheduler for multiprog.

degree of multiprogramming is the number of processes in the main memory that can be supported by a multi-proggrammed system

Operation on Processes

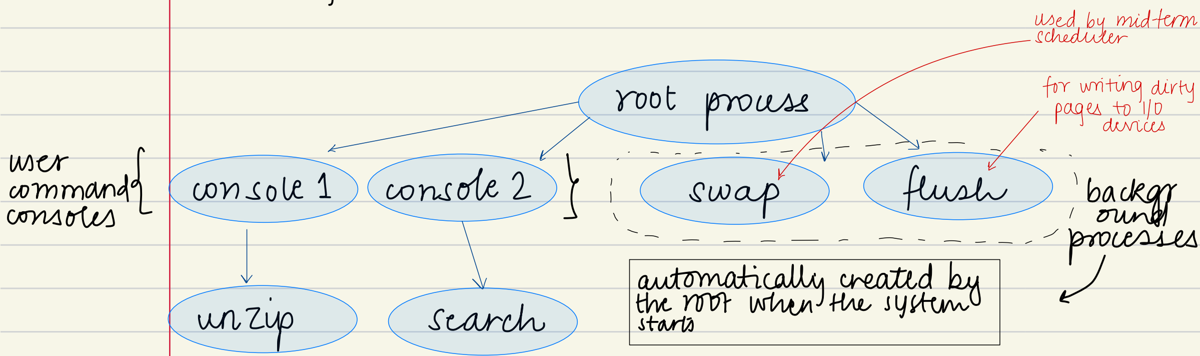
creation

execution

termination

① Process Creation

- parent process creates children processes and so on, resulting in a tree of processes (called a fork



- the creation occurs through system calls

② Process Execution

- 2 orders : $\xrightarrow{1}$ parent & child execute concurrently & independently

  $\xrightarrow{2}$ parent waits until children terminate
  - ↳ calls system call 'wait()' (join() in nachOS) and the parent process state becomes waiting
  - ↳ state will change to ready when the child process finishes execution

③ Process Termination

exit   abort

- **exit** :: process executes last statement and asks the OS to delete it
  - child may output return data to its parent
  - process resources are de-allocated by the OS

- **abort** :: parent may terminate execution of children processes anytime where:
  - child has exceeded allocated resources
  - task assigned to child is no longer required
  - parent is exiting

    OS kills child process if parent process terminate, this is called cascaded termination

Interprocess communication

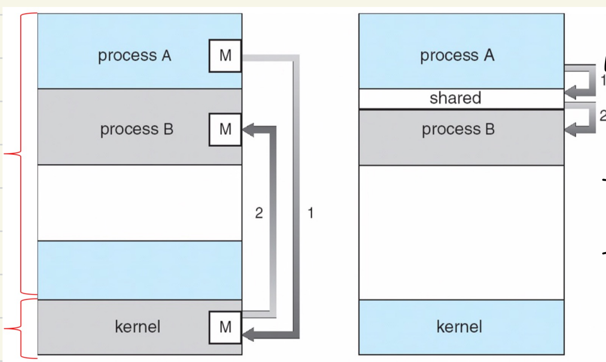
about processes that work together

- independent process : cannot affect or be affected by the execution of other processes
- cooperating process : can affect / be affected by the execution of other processes
  - communicates to share data
  - 2 models of Inter-Process communication (IPC)
    - → message passing
    - → shared memory

Inter-Process communication

M : mailbox

- message is placed in the mailbox in the kernel space
- process B takes the message from the kernel to itself



→ updates data in memory buffer

→ reads data from memory buffer

→ communication through read / write operations

→ address of shared memory space must be known to both processes

∴ it is in the user space

message passing	shared memory
·	- only 1 system call to est. shared space.
· small data.	· large data

does direct messaging also require mailbox? yes

direct ← **message passing** → indirect

- processes communicate and synchronize their actions without resorting to shared variables

- 2 operations (system calls) are required
  → send () ⟶ message size is fixed or variable
  → recieve ()

- for communication:
  ① est. a communication link between them
  ② exchange messages via send/receive

- Direct : processes must name eachother explicitly :
  send (P, message) → send to P
  receive (Q, message) → receive from Q

- Indirect : uses mailbox, no name necessary
  - obj where messages are placed or removed
  - indentified by ID
  - can be implemented as a queue

same as mailbox? yes

- buffer is used to store messages or is it data??? nope
  - ↳ unbounded - buffer (no practical limit)
  - ↳ bounded - buffer (fixed size)

- process sending messages does so until mailbox is full, then waits
- process receiving messages does so unil mailbox is empty,

Threads

- same as processes, just nachOS calls it threads

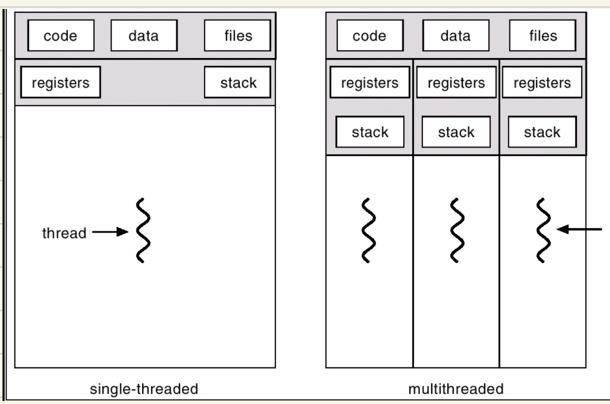
Overview

- thread : basic unit of CPU, <u>light-weight process</u>
  → thread id
  → program counter
  → register set
  → stack space

  called so because resources for code, data, files need not be allocated again

- thread shares w/ peer threads in the same process
  → code & data sections
  → OS resources (open files)

- traditional process ≡ executing prog w/ a single thread of control

- 

  if a particular thread is awaiting input / is blocked, the other threads can run

  thread

  ⇓

  higher throughput

∴ resources need not be re-allocated

does so and is deallocated

doesn't take much time

· multithread server

client → server → thread

① request

② create new thread to service request

③ resume listening for more request

∴ server can serve multiple clients

Thread Implementation Models

· we want unlimited thread creation possible by the user but the OS Kernel can only support a limited number of threads due to resource constraints

→ **solution**

→ user threads (logical), in user space

→ kernel threads (physical), in kernel space
  - slower to make + manage
  - resources are eventually allocated
  - can be executed on diff processors

→ OS maps user threads to kernel threads
  - → many to one : disad : unable to run in parallel, if a thread is stuck (system call / I/O), the rest are too
  - → one to one : more concurrency, but too many
  - → many to many : disad : not easy to decide mapping