Lecture: Processes / Mechanisms ①

Processes

- What are they / why needed?
- How does dispatcher run a process?
- How does system create a process?

"Virtual CPU" — "Activity of some kind" ⇒ Program, I/O, state

Process: def

Execution stream, in the context of a process state

more simply

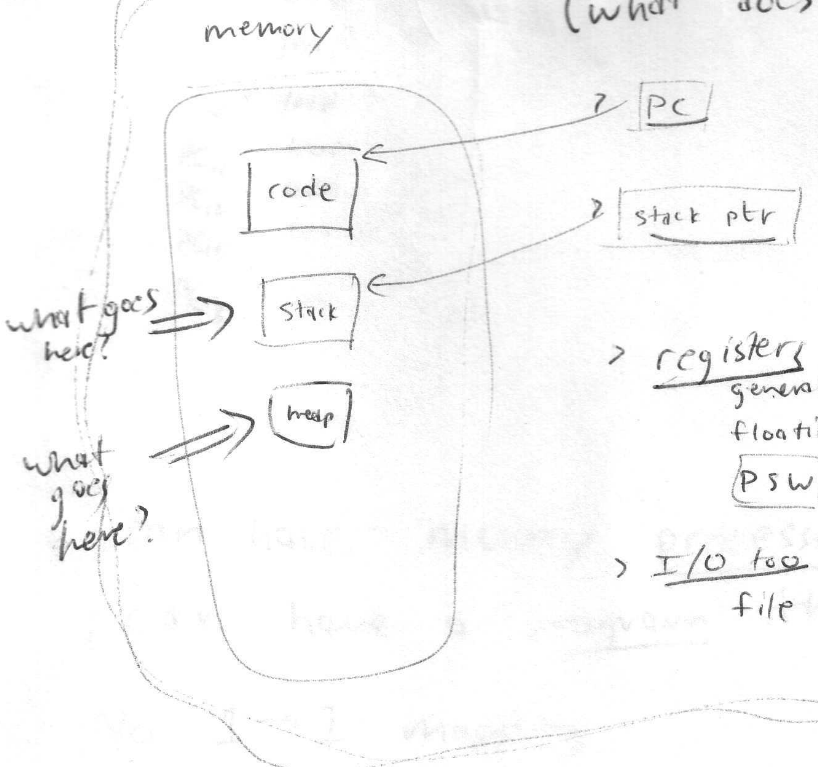
"Running program + stuff it needs/uses"

"thread of control" — can affect, be affected by

What things can code affect / be affected by?

⇒ think about machine architecture

(what H/W does prog. affect while running?)



process state: where we are in current execution

- registers
  - general purpose
  - floating point
  - PSW
- I/O too
  - file handles for open files

Program vs Process

Different!

Program: static code + static data

Program

sequen instructions

```
load x, reg
jmp "f"
...
f.  load
    load
    add
    store
    ret
```

```c
int
main (int argc, char * argv) {
    f(x);
}

void
f (int x) {
    ...
}
```

```
PC→ PC1   load
    PC2   jmp   10
    PC10  load
    PC11  load
    PC12  add
    PC13  return
    PC3   —
```

Process

dynamic flow of instructions
"prog in execution"

+ "process state"

- heap
- stack
  - main
  - foo
- registers
- PC

Dynamic

Thus,
- can have many processes of same program
- can have a program that invokes many processes

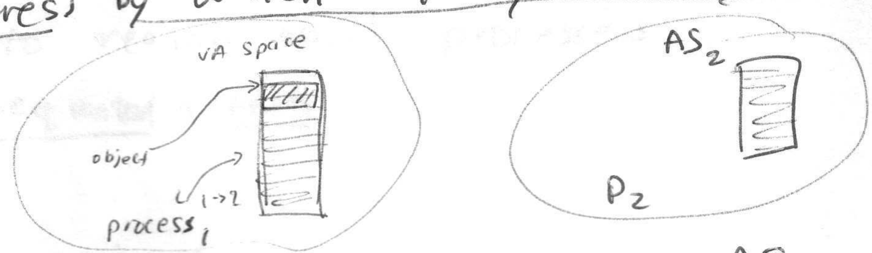
No 1→1 mapping

~~Threads~~

Address Space:

- will discuss in detail in mem mgmt
- "virtual" address space
  - => really large memory to use
- linear array of bytes : [0 ... N-1] $N \sim 2^{32}$ 2^{64}

def: all logical entities used by a process + address by which they are referenced



one-to-one mapping : Process <=> AS

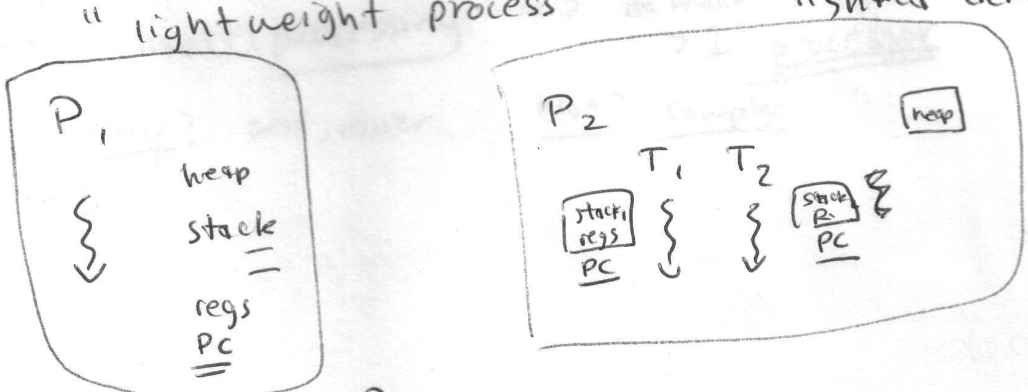
=> protection boundary

Threads

Process != Thread

Threads: Many execution streams w/in one process!

"lightweight process"   "shared address space"



> why do this?
- efficient communication if cooperating to do something
- express concurrency + ↑

{ Process₁ write item to file
  P₂ read from file }

{ T₁ write to mem
  T₂ read from mem }

Why use Processes?

1) Divide and Conquer
   Take large problem => many small ones
   (smaller => easier)

2) ~~Simple way to express~~ concurrency
   lots of users, devices, etc.

3) Easy to reason about processes
   sequential activity

System Classifications

Uniprogramming: One process @ a time
- early PCs
- Why? simple
- Not? inconvenient, poor perf.

Multiprogramming: many @ time
- All modern OSs
- ! multiprocessing => machine w/ >1 processor
- Why? perf, easier
- Not? complex

Multiprogramming

OS requirements

- **Policy**: when to schedule A, B, ...
- **Mechanism**: How to switch between processes
  - How to protect them from one another

Separation of P / M

- Policy: depends on (1) workload, other assumptions about env.
  - (2) metrics
- Mechanism: basic functionality for doing stuff

w/ processes

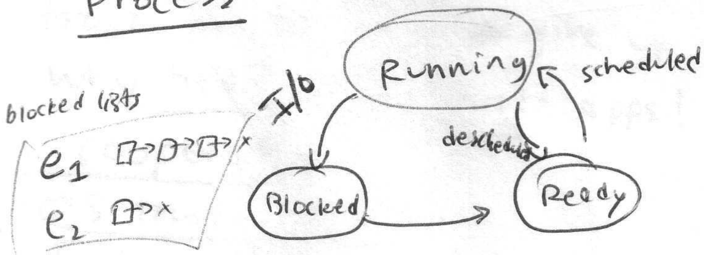
- P = scheduling (later)
- M = dispatching (today)

Java/C Project #1 out, due on Tuesday

Dispatch Mechanism

Data Structure: list of processes (per-process state)

Process: 3 modes



OS: "ready list"

Dispatcher loop:

How to desched A, run B

"gain control", save state of A, load state of B, run B (context switch)

How? what state?

How to gain control?

general

User Processes — "user mode" limited access to h/w

"OS" (dispatcher) — (system) "kernel mode" can access anything

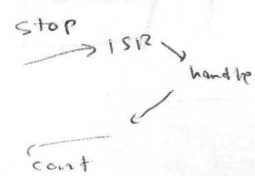
how: h/w support

H/W S/W interface

Problem:
only one CPU
CPU is running user program
dispatcher: how to run??

Two ways:

- **Traps**: internal event in user process
  e.g. sys call, error (illegal inst, I/O), fault (page fault)
- **Interrupt**: external event
  character typed, disk I/O done



Disp: gaining control

Cooperative approach
trust process to give up CPU (yield)
why bad? OS trusts apps! [Alto ⇒ Mac]

Non-cooperative
OS trusts no one
Timer interrupt ⇒ give OS control (dispatcher)
(every 10 ms)
Can count "ticks", decide to switch to diff job (policy)

(H/W support)

Key: user can't turn off interrupts
(user/sys mode)

⇒ Building OS is about being paranoid!

What state must be saved?

DBp: must track state of Procs when not running

on trap/interrupt,
save state to PCB (Process Control Block)

Info in PCB

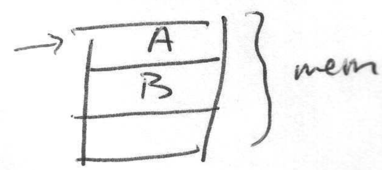
execution state : registers, status word, PC, stack ptr, etc)
i/o state : open files
sched info : state, priority
accounting : owner, pid

examine [memory?]

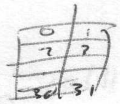
What needs to be saved? (on switch)

no protection
- 0a) Trust: early PCs, macs
- 1a) All of memory [Alto]
  Slow: 1 GB of mem => disk?
- 1b) Protect memory [later]

] trusting
] non-trusting



Context-switch: Implementation



Machine-dependent (assembly)
save/restore registers

why hard?

Code needs registers to run
=> H/W support
CISC: instr. to save regs => stack
RISC: convention: don't use R_1, R_2

H/W
S/W
interface

How expensive?

many loads/stores

Process Creation

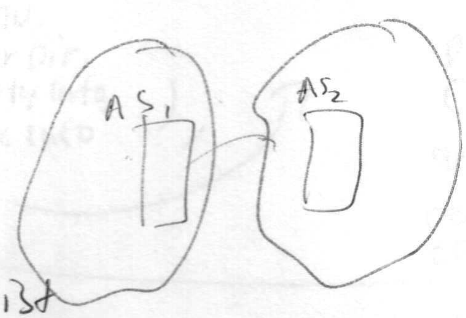
2 ways:
- from scratch
- by imitation (cloning)

Scratch:

Load code/data
Create stack, PCB
Process ⇒ ready list

Clone

Stop current, save state
Copy state
Add process ⇒ ready list



Unix: "clone approach"

fork() clones process
But: just copies same process

exec() overlays new image on calling process

shell/
shell.c

⋮

```
cmd = get_command()
int rc = fork();
if (rc<0) {
    //error
} else if (rc==0) {
    //child
    execv(cmd, args);
    printf("failed");
} else {
    //parent
    int pid = rc;
    wait(pid);
}
```

1) not an exact copy: why?

2) problem: copy can be expensive (esp. if exec() is next)

Skip

Example : Process Creation in Windows NT

Create Process (AppName, CmdLine, Process Attributes,
Thread Attributes, InheritHandles,
Creation Flags, Environment,
Current Directory, Startup Info,
Process Info);

History
VAX/VMS Mach ⟹ NT

As David Korn says

"There is a single primitive, named CreateProcess(), that takes ten arguments, yet still cannot perform the simple operation of overlaying the current process with a new program as execve() requires"