Processes

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

"Virtual CPU"
"Activity of some kind"

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

(memory)

what goes here?

what goes here?

(what "hw" does prog. affect while running?)
Program vs Process

Different!

Program: static code + static data

Program:

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

Process:

dynamic flow of instructions
"prog in execution" + "process state"

heap
stack
main too
registers
Pc

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

Thus,

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

No 1→1 mapping
Address Space:
- will discuss in detail in mem mgmt
- "virtual" address space
  = large memory to use
  = linear array: \([0 \ldots N-1]\) \(N \approx 2^{32} \approx 2^{64}\)
- def: all logical entities used by a process + address by which they are referenced

one-to-one mapping: \(\text{All Process} \rightarrow \text{AS}\)

\(\Rightarrow\) protection boundary

Threads
- Process \(!=\) Thread
  - Threads: Many execution streams \(\cap\) in one process!
  - "lightweight process"
  - "shared address space"

\(\Rightarrow \text{why do this?}\)
- efficient communication if cooperating to do something
- express concurrency +
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
   sequenced activity

System Classifications

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

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


Multiprogramming

OS requirements

\[
\begin{align*}
\text{Policy} & : \text{when to schedule A, B, ...} \\
\text{Mechanism} & : \text{how to switch between processes} \\
& \quad \text{how to protect them from one another}
\end{align*}
\]

Separation of P/M

\[
\begin{align*}
\text{Policy} & : \text{depends on workload, other assumptions about env.} \\
\text{Mechanism} & : \text{basic functionality for doing stuff}
\end{align*}
\]

w/ processes

\[
P = \text{scheduling (later)} \\
M = \text{dispatching (today)}
\]

Dispatch Mechanism

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

Process: 3 modes

\[
\begin{align*}
\text{blocked} & \\
\text{running} & \\
\text{scheduled} & \\
\end{align*}
\]

Dispatcher

How to schedule A, run B

\[
\text{save state of A, load state of B, run B}
\]

How?

context switch

\[
\text{what state?}
\]
How to gain control?

User Processes

"user mode"
limited access to h/w
(system)
"kernel mode"
can access anything

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

Two ways:

Traps: internal event in user process
(legitimate, IO) (debug, fault)
  e.g., sys call, error, 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)

Can count "ticks", decide to switch to
diff job (policy)

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

Building OS is about being paranoid!
What state must be saved?

Disp: must track state of Pecs 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

What needs to be saved? (on switch) [Memory]

- [0a] Trace: empty Pcs, none
- [1a] All of memory [A to] (later)
- Show: 1 GB or mem \( \rightarrow \) disk?
- [1b] Protect memory

Context-switch Implementation

Machine-dependent (assembly)
- save/restore registers

Why hard?
- codes needs registers to run
  - H/W support
  - CISC: instr. to save regs \( \rightarrow \) stack
  - RISC: convention: don't use R1, R2

How expensive?
- many loads/stores
Process Creation

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

Scratch:
  - Load code/data
  - Create stack, PCB
  - Process \(\Rightarrow\) ready list

Clone:
  - Stop current, save state
  - Copy state
  - Add process \(\Rightarrow\) ready list

Unix: "clone approach"
  - `fork()` creates process
  - But: just copies same process
  - `exec()` overlays new image on calling process

```c
int rc = fork();
if (rc < 0) {
  // error
} else if (rc == 0) {
  // child
  execv(cmd, args);
} else {
  // parent
  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, CommandLine, Process Attributes,
Thread Attributes, InheritHandles,
Creation Flags, Environment,
CurrentDirectory, Startup Info,
Process Info);

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 execute() requires."