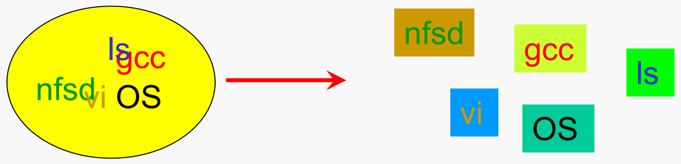
Processes for Simplicity

Lots of stuff's going on in the system…



Make it simple…

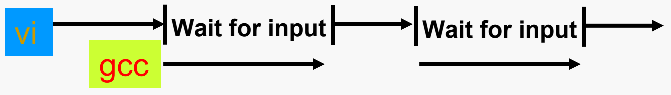
Separate each in isolated process.

OS deals with one thing at a time; they just deal with the OS.

Manage complexity by decomposition.

Processes for Speed

I/O parallelism:



Overlap execution, effectively turn 1 CPU into many.

Completion times are reduced.

Processes in the Real World

Processes and parallelism have been a fact of everyday life much longer than OSes have existed.

Industrial processes use multiple production lines operating simultaneously to increase throughput (# of items completed per unit time).

Can you always partition work to speed up the overall job?

Ideal speedup would be N-fold if N production lines are used.

Reality is that there are bottlenecks and overhead to coordinate lines.

Example: project groups in CS 3204

What is a thread?

In practice, what's needed to run code on a CPU.

"execution stream in an execution context"

execution stream: sequence of instructions

CPU execution context (1 thread)

state: stack, heap, registers

position: PC

```
add    $s0, $s1, $s3
sub    $s0, $s4, $s0
sw     $s0, 12($t7)
```

OS execution context (in threads)

identity + open file descriptors, VM page table, …

What is a process?

process: thread + address space

or, abstraction representing what you need to run a thread on an OS

address space: encapsulates protection, passive notion

Why separate notion of thread from notion of process?

In many situations, you want multiple threads sharing a common address space (servers, OS, parallel programs).

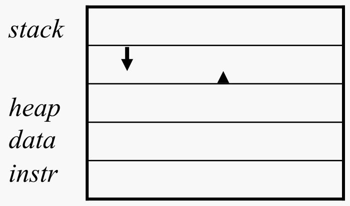
A process is not a program

program: code + data, passive

```
int foo;
int main() {
   printf("foo");
}
```

process: program in execution

may execute multiple instances of a single program at once



How to make a process?

Creation:

Load code and data into memory; create empty call stack.

Initialize state to same as after a process switch.

Add to OS's list of processes.

Clone:

Stop current process and save state.

Make copy of current instructions, data, stack and OS state.

Add new process to OS's list of processes.

Unix

Fork clones a process.

Exec overlays the current process.

No create! Fork, then exec.

```
if ( (pid = fork()) == 0 ) {
   // this is child process
   exec(…);    // exec does not return
}
else {
   // this is parent
   wait( pid );  // wait for child to finish
}
```

Pros: simple, clean

Cons: duplicate operations

Process environments

uniprogramming: one process at a time

- "cooperative timesharing"
- mostly PCs, vintage OSes
- easy for OS, usually hard for user
- violates isolation
- when should process yield?
- uniprogramming != uniprocessing

multiprogramming: > 1 process at a time

- time-sharing
- CTSS, Multics, Unix, VMS, NT, …
- multiprogramming != multiprocessing

The Grand Illusion

Each thread has the illusion of its own CPU, even on a uniprocessor machine.

- How does this work?

Two key pieces:

- thread control block (Pintos: thread class)
  - one per thread, holds execution state
- dispatching loop:

```
while ( 1 )
    interrupt running thread
    save state
    get next thread
    load state, jump to it
```

Multiprogramming problems

Tracking state…

- PCB (*process control block*)
- thread state + OS state

N processes… who to run?

- Need to schedule whenever we have one resource and many "things" that want to use it.

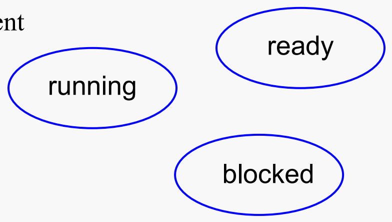
Protection…

- Prevent processes from getting at another's state
- Fairness: make sure each process gets to run
- No protection --> system crashes are about O(# processes)

Process states

Processes in three states:

- *running*: executing now
- *ready*: waiting for CPU
- *blocked*: waiting for another event



Which ready process do we pick?

- 0 ready processes: run idle loop
- 1 ready process: only one choice!
- > 1 ready process: what to do???

Picking a process to run

Scan process table for first runnable?

- expensive, weird priorities (small PIDs are better??)
- divide into runnable and blocked processes

FIFO?

- put threads on back of list, pull them off front
- Pintos does this (thread.c)
- what about priorities?

Priority?

- give some threads a better shot at the CPU
- problem?

How to get control?

traps: events generated by the current process

- system calls
- errors (illegal instructions)
- page faults

interrupts: events external to the process

- I/O interrupts
- timer interrupt (periodic, fairly frequent)

Process perspective

- explicit: process yields processor to another
- implicit: causes an expensive blocking event, gets switched

How to switch contexts?

Very machine dependent… must save

- general-purpose and FP registers, any co-processor state, …

Tricky

- OS code must save state without changing any state.
- How to run w/o touching any registers?
  - Some CISC machines have single instructions to save all registers on the stack
  - RISC machines reserve some registers for the kernel, or have a way to carefully save one and then continue

How expensive? direct cost of saving + indirect cost of flushing user caches

Fundamentals of process switching

Execution is THE vital issue.

- procedure calls, threads, processes are just variations

What's the minimum to execute code?

- position pointer to current instruction
- +
- state captures result of computation

What's the minimum to switch from one to another?

- save old instruction pointer and load new one

What about state?

- if state is per-thread, have to save and restore
- in practice, can save everything, nothing or something in between

Switching between procedures

Procedure call

- save active caller registers
- call `Foo`

```
saves callee registers
... do stuff ...
restores callee registers
jumps back to return address
```

- restore caller registers

How is state saved?

- saved proactively? saved lazily? not saved?

Aside: threads vs procedures

Threads may resume out of order

- cannot use LIFO stack to save state
- general solution: duplicate stack

Threads switch less often

- don't partition registers (why??)

Threads involuntarily interrupted

- *synchronous*: proc call can use compiler to save state
- *asynchronous*: thread switch code saves all registers

More than one thread can run

- scheduling: what to overlay on CPU next?
- proc call scheduling obvious: run called procedure

Synchronous thread switching

Switch threads: #(Arguments: CurrentThread, NextThread)

# Save caller's register state.

# Save current stack pointer to old thread's stack.

# Restore stack pointer for new thread's stack.

# Restore caller's register state.

# Return

Asynchronous thread switching

Switch threads: #(Arguments: CurrentThread, NextThread)

```
# Save current state:
  # triggered by interrupt
  # Save registers
  # Set up kernel environment
  # Call interrupt handler

# Restore new state:
  # Restore registers
```

Process vs threads

Different address space

switch page table, etc.

problems: how to share data? how to communicate?

Different processes have different privileges

switch OS's idea of who's running

Protection

have to save state in safe place

need support to forcibly revoke processor, prevent imposters

Different than procedures?

OS, not compiler manages state saving

Real OS permutations

One or many address spaces?

One or many threads per address space?

# address spaces: # threads per space	1	many
1	MS/DOS Macintosh	traditional Unix
many	embedded systems Pilot	VMS, Mach, OS2 NT, Solaris, OS X, Linux

Generic abstraction template

abstraction:	how OS abstracts underlying resource
virtualization:	how OS makes small number of resources seem like an infinite number
partitioning:	how OS divides resources
protection:	how OS prevents bad people from using pieces they shouldn't
sharing:	how different instances are shared
speed:	how OS reduces management overhead

How the CPU is abstracted

CPU state represented as a process

Virtualization:	processes interleaved transparently
Partitioning:	CPU shared across time
Protection:	pigs are forcibly interrupted, process's state saved in OS space, identities are protected
Sharing:	yield your CPU time slice to another process
Speed:	large scheduling quanta, minimize state needed to switch, share common state (code), duplicate state lazily