Announcements

- Project Session #1
  - video on-line, slides will be posted
- Assignment #1 out today
  - due Tue., July 8th, 10 p.m.
  - start now (tricky sync. debugging)
  - Andrew Birrell Paper (cs140/info)
- C/NC? allowed, but not advisable
  - you'll do the same amount of work
  - tell your teammates if C/NC!

More OS motivation

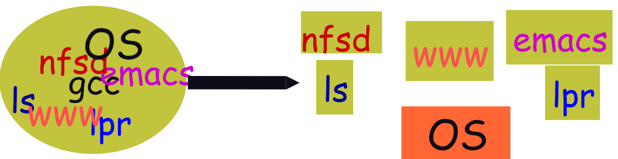
- ISCA keynote, Justin Rattner (intel)
  - processors at (3GHz) wall
  - emphasis on "meaningful" performance
  - processor architecture -> **system** architecture
  - BL: OS affects **responsiveness**
  - **when** you do things matters: scheduling
- Chrysler: 30 GB hard drive, WiFi hot spot
  - cars contain many embedded computers
  - reboot? slowdown? crash?
  - different time/reliability requirements vs. PC
  - protection, scheduling

Today's adventure

- What are threads, processes?
- What are they for?
- How do they work?
- Threads vs. processes?
- Concurrency (will continue Monday)
- Readings:
  7th ed: Silberschatz/Galvin: Ch. 3 (skip 3.{4-6}), 4, 6
  6th ed: Silberschatz/Galvin: Ch. 4 (skip 4.{5,6}), 5, 7

Why processes? Simplicity

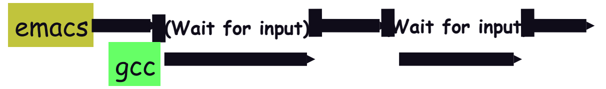
- Hundreds of things going on in system



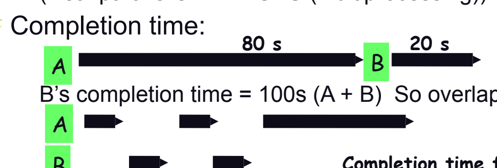
- How to make simple?
  - Separate each in isolated process. OS deals with one thing at a time, they just deal with OS
  - *THE* universal trick for managing complexity: decomposition ("reductionism")

Why processes? Speed

- I/O parallelism:



  Overlap execution: make 1 CPU into many
  (Real parallelism: > 1 CPU (multiprocessing))
- Completion time:



Processes in the real world

- Processes, parallelism fact of life much longer than OSes have been around
  - Companies use parallelism for more throughput: 1 worker = 100 widgets? hire 100 to make 10,000.
- Can you always partition work to speed up job?
  - Ideal: N-fold speedup
  - Reality: bottlenecks + coordination overhead
  - Example: CS140 group project
  - (More abstractly: easy to increase throughput, reducing latency more difficult)

What is a thread?

- In theory: Turing machine
  tape (state), tape head (position)
- In practice: What's needed to run code on CPU
  "execution stream in an execution context"
  Execution stream: sequential seq. of instructions
- CPU execution context (1 thread)
  State: stack, heap, registers
  Position: program counter register
- OS execution context (n threads):
  identity + open file descriptors, page table, …

```
add r1, r2, r3
sub r2, r3, r10
st r2, 0(r1)
…
```

What is a process?

- Process: thread + address space
  or, abstraction representing what you need to run thread on OS (open files, etc)
- Address space: encapsulates protection
  address state passive, threads active
- Why separate thread, process?
  Many situations where you want multiple threads per address space (servers, OS, parallel app, user app?)

Process != Program

- Program: code + data
  passive

```
int c;
int main() {
   printf("hello");
}
```

- Process: running program
  state: registers, stack, heap…
  position: Program counter
- We both run Firefox:
  same program, different process

stack	
heap	
data	int a;
code	main()

How to make one?

- Creation:
  Load code and data into memory; create empty call stack.
  Initialize state to same as after a process switch.
  Put on OS's list of processes.
- Clone:
  Stop current process and save state.
  Make copy of current code, data, stack and OS state.
  Add new process to OS's list of processes

gcc → gcc, gcc

Example: Unix

- How to make processes:
  Fork clones a process.
  Exec overlays the current process.
  No create! Fork then exec.

```
if((pid = fork()) == 0) {
        /* child process */
       exec("foo");  /* exec does not return */
else
       /* parent */
       wait(pid); /* wait for child to finish */
```

- Pros: Simple, clean. Con: duplicate operations

Examples: Windows

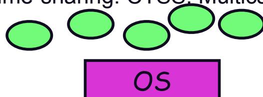
BOOL CreateProcess(
LPCTSTR *lpApplicationName*, // pointer to name of executable module
LPTSTR *lpCommandLine*, // pointer to command line string
LPSECURITY_ATTRIBUTES *lpProcessAttributes*, // process security attr.
LPSECURITY_ATTRIBUTES *lpThreadAttributes*, // thread security attr.
BOOL *bInheritHandles*, // handle inheritance flag
DWORD *dwCreationFlags*, // creation flags
LPVOID *lpEnvironment*, // pointer to new environment block
LPCTSTR *lpCurrentDirectory*, // pointer to current directory name
LPSTARTUPINFO *lpStartupInfo*, // pointer to STARTUPINFO
LPPROCESS_INFORMATION *lpProcessInformation* // pointer to PROCESS_INFORMATION **);**

Process environments

- Uniprogramming: 1 process at a time
  - "Cooperative timesharing": mostly PCs, vintage OSes
  - Easy for OS, hard for user (generally).
  - Violates isolation: Infinite loops?
  - When should process yield?

OS

- Multiprogramming: > 1 process at time
  - Time-sharing: CTSS, Multics, Unix, VMS, NT, ...



Multiprogramming != multiprocessing

The multithreading illusion

- Each thread has its illusion of own CPU
  - yet on a uni-processor all threads share the same physical CPU!
  - How does this work?

CPU

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

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

The multiprogramming problems

- Track state? PCB (process control block)
  - Thread state, plus OS state: identify, accounting, …

pcb

Priority	registers	open file descriptors, …

- N processes? Whom to run? ("Scheduling")
  - Need to schedule whenever 1 resource & many "things" (disk, net, CPU, classroom, …)
- Protection? Need two things:
  - Prevent process from getting at another's state
  - Fairness: make sure each process gets to run
  - (No protection? system crashes ~ O(# of processes)

Process states

- Processes in three states:

blocked

  - Running: executing now
  - Ready: waiting for CPU
  - Blocked: waiting for another event (I/O, lock)
- Which ready process to pick?
  - 0 ready processes: run idle loop
  - 1 ready process: easy!
  - > 1: what to do?

Picking a process to run

- Scan process table for first runnable?
  - Expensive. Weird priorities (small pid's better)
  - Divide into runnable and blocked processes
- FIFO?
  - Put threads on back of list, pull them off from front
  - (pintos does this: thread.c)
- Priority?
  - Give some threads a better shot at the CPU
  - problem?

Scheduling policies

- Scheduling issues
  - Fairness: don't starve process
  - Prioritize: more important first
  - Deadlines: must do by time 'x' (car brakes)
  - Optimization: some schedules >> faster than others
- No universal policy:
  - Many variables, can't maximize them all
  - Conflicting goals
    - **More important jobs vs. starving others**
    - **I want my job to run first, you want yours.**
- Given some policy, how to get control? Switch?

How to get control?

- Traps: events generated by current process
  - System calls
  - Errors (illegal instructions)
  - Page faults
- Interrupts: events external to the process
  - I/O interrupt
  - Timer interrupt (every 1 to 10 milliseconds or so)
- Process perspective:
  - Explicit: process yields processor to another
  - Implicit: causes an expensive blocking event, gets switched

How to "context switch"?

- Very machine dependent. Must save:
  - General-purpose & floating point registers, any co-processor state, shadow registers (Alpha, sparc)
- Tricky:
  - OS code must save state without changing any state
  - How to run without touching any registers??
    - **Some CISC machines have single instruction to save all registers on stack (e.g. x86's *pusha* instruction)**
    - **RISC: reserve registers for kernel (MIPS – k0,k1) or have way to carefully save one and then continue**
- How expensive?
  - Direct cost of saving; opportunity cost of flushing useful caches (cache, TLB, etc.)

Fundamentals of process switching

- "execution" *THE* Grand Theme of CS:
  - Procedure calls, threads, processes just variations
- What's the minimum to execute code?
  - Position (pointer to current instruction)
  - State (captures result of computation)
- Minimum to switch from one to another?
  - Save old instruction pointer and load new one
- What about state?
  - If per-thread state have to save and restore
  - In practice can save everything, nothing or combination.

Switching between procedures

- Procedure call:

save active caller registers
call foo → saves used callee registers
...do stuff...
restores callee registers
jumps back to pc → restore caller regs

- 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
  - alternate solution: activation record list(s)
- 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 than one thread can run
  - Scheduling: what to overlay on CPU next?

Synchronous thread switching

```
Switch_threads:  # (Arguments: CurrentThread; NextThread)

# Save caller's register state. ABI allows us to destroy eax,ecx,edx
        push ebx; push ebp; push esi; push edi
# Save current stack pointer to old thread's stack,
        mov eax, SWITCH_CUR[esp]
        mov [eax], esp
# Restore stack pointer from new thread's stack.
        mov ecx, SWITCH_NEXT[esp]
        mov esp, [ecx]
# Restore caller's register state.
        pop edi; pop esi; pop ebp; pop ebx
        ret
```

Asynchronous thread switching

Assume: x86 – kernel doesn't use FP, etc.

Save current state:	**Restore new state:**
# triggered by interrupt	
intr_entry:	**intr_exit**:
# Save registers. Trap already	# Restore registers.
# saved: cs, eip, eflags,ss,esp	popa # pop all gp regs
push ds; push es; etc.	pop es
pusha # Push all gp regs	pop ds;
# Set up kernel environment.	etc.
cld; mov eax, SEL_KDSEG	# iret restores cs,eip,eflags
mov ds, eax; mov es, eax	iret
# Call interrupt handler.	
push esp	
call intr_handler	

Process vs. threads

- Different address space:
  - switch page table, etc.
  - Problems: How to share data? How to communicate?
- Different process have different privileges:
  - switch OS's idea of who's running
- Protection:
  - have to save state in safe place (OS)
  - 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*

# of address spaces / # of threads/space	1	many
1	**MS/DOS** **Mac OS 1-6**	**Traditional UNIX**
many	**Embedded, Mac 7-9, Palm OS**	**VMS, Mach, OS/2, Win/NT** **Solaris, OS X, Linux**

Generic resource management template for OS (or anything)

- Abstraction: how OS abstracts underlying resource
- Virtualization: how OS makes small number of resources seem like an "infinite" number
- Partitioning: how OS divides resource
- Protection: how OS prevents bad people from using pieces they shouldn't
- Sharing: how different instances shared
- Speed: how OS reduces management overhead

How CPU managed

- Abstraction: CPU state represented as process
- Virtualization: processes interleaved transparently (run ~1/n slower than real CPU)
- Partitioning: CPU shared across time
- Protection: (1) pigs: forcibly interrupted; (2) corruption: process' state saved in OS; (3) imposter: cannot assume another's identity
- Sharing: yield your CPU time slice to another process
- Speed: (1) large scheduling quanta; (2) minimize state needed to switch; (3) share common state (code); (4) duplicate state lazily

Summary

- Thread = pointer to instruction + state
- Process = thread + address space
- Key aspects:
  - per-thread state
  - picking a thread to run
  - switching between threads
- The Future:
  - How to share state among threads?
  - How to pick the right thread to run?
  - next: sharing state (concurrency, synchronization)