Scheduling

Today

- Introduction to scheduling
- Classical algorithms

Next Time

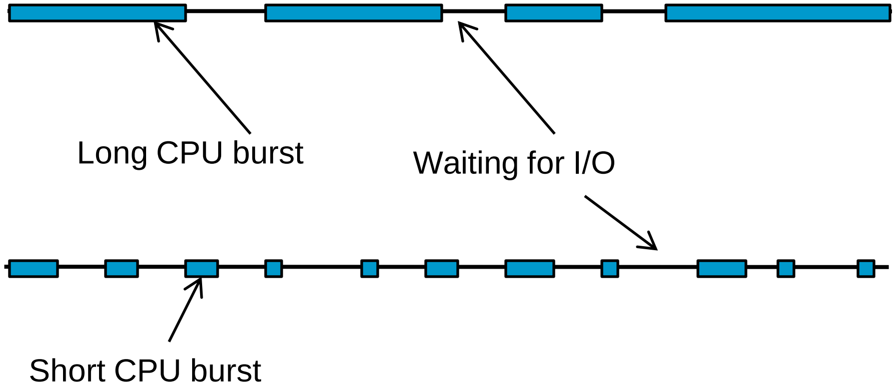
- Process interaction & communication

Scheduling

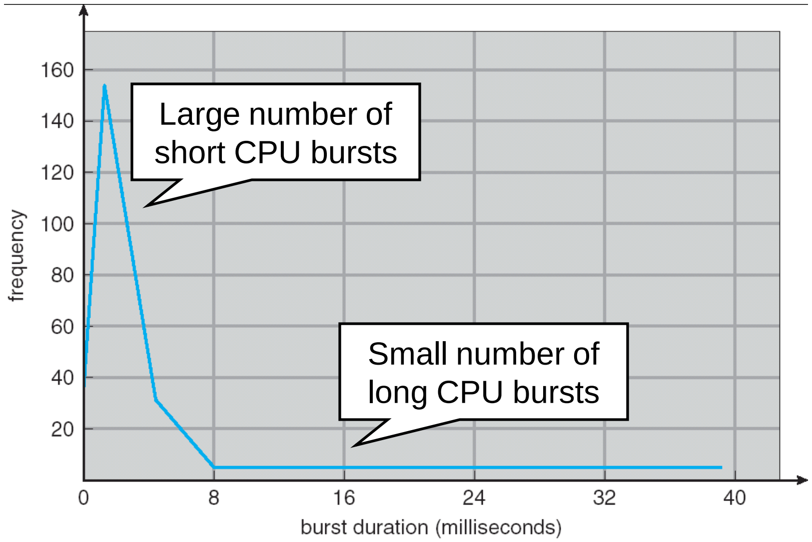
- Problem
  - Several ready processes & much fewer CPUs
- A choice has to be made
  - By the *scheduler,* using a *scheduling algorithm*
- Scheduling through time
  - Early batch systems – Just run the next job in the tape
  - Early timesharing systems – Scarce CPU time so scheduling is critical
  - PCs – Commonly one active process so scheduling is easy; with fast & per-user CPU scheduling is not critical
  - Networked workstations & servers – All back again, multiple ready processes & expensive CS, scheduling is critical

Process behavior

- Bursts of CPU usage alternate with periods of I/O wait
  - A property key to scheduling
  - CPU-bound & I/O bound process
- As CPU gets faster – more I/O bound processes



Histogram of CPU-burst times



Environments and goals

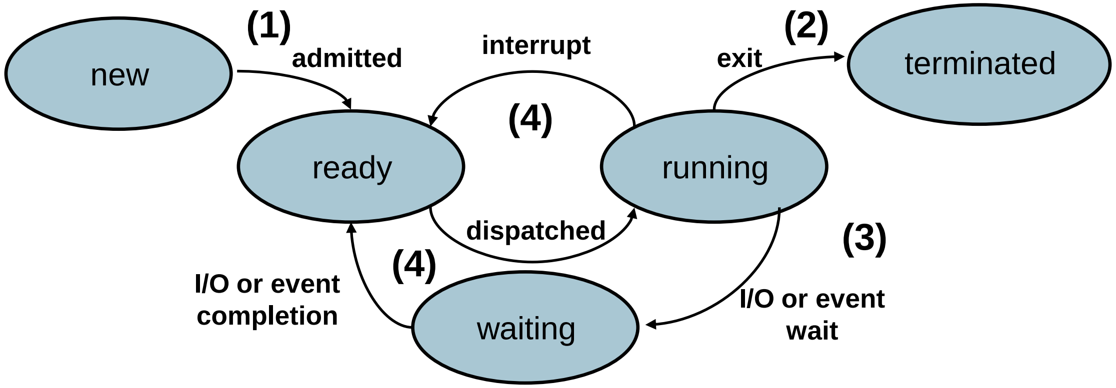
- Different scheduling algorithms for different application areas
- Worth distinguishing
  - Batch
  - Interactive
  - Real-time
- All systems
  - Fairness – comparable processes getting comparable service
  - Policy enforcement – seeing that stated policy is carried out
  - Balance – keeping all parts of the system busy (mix pool of processes)

Environments and goals

- Batch systems
  - Throughput – max. jobs per hour
  - Turnaround time – min. time bet/ submission & termination
    - Waiting time – sum of periods spent waiting in ready queue
  - CPU utilization – keep CPU busy all time *(anything wrong?)*
- Interactive systems
  - Response time – respond to requests quickly (time to start responding)
  - Proportionality – meet users' expectations
- Real-time system
  - Meeting deadlines – avoid losing data
  - Predictability – avoid quality degradation in multimedia systems
- Average, maximum, minimum or *variance*?

When to schedule?

- When to make scheduling decisions?
  1. At process creation
  2. When a process exits
  3. When a process blocks on I/O, a semaphore, etc
  4. When an I/O interrupts occurs
  5. A fix periods of time – Need a HW clock interrupting



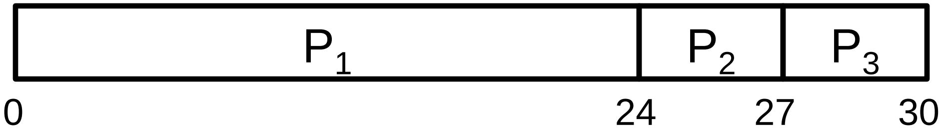
When to schedule?

- A fixed periods of times … preemptive and non-preemptive
  - No-preemptive
    - Once a process gets the CPU, it doesn't release it until the process terminates or switches to waiting
  - Preemptive
    - Using a timer, the OS can preempt the CPU even it the thread doesn't relinquish it voluntarily
    - Of course, re-assignment involves overhead

First-Come First-Served scheduling

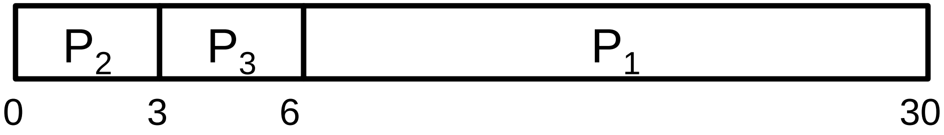
- First-Come First-Served (FCFS)
  - Simplest, easy to implement, non-preemptive

Process	Burst Time
P1	24
P2	3
P3	3



Average waiting time:
(0 + 24 + 27)/3 = 17

Change order of arrival ….



Average waiting time = 3

FCFS issues

- Potentially bad average response time
  - 1 CPU-bound process (burst of 1 sec.)
  - Many I/O-bound ones (needing to read 1000 records)
  - Each I/O-bound process reads one block per sec!
- May lead to poor utilization of resources
  - Poor overlap of CPU and I/O

Shortest Job/Remaining Time First sched.

- Shortest-Job First
  - Assumption – total time needed (or length of next CPU burst) is known
  - Provably optimal

First job finishes at time a

Second job at time a + b

…

Mean turnaround time

(4a + 3 b + 2c + d)/4

↑

Biggest
contributor

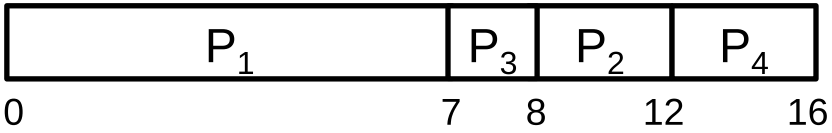
Job #	Finish time
1	a
2	b
3	c
4	d

Preempetive or not?

- A preemptive variation – Shortest Remaining Time (or SRPT)

SJF and SRT

- Shortest Job First – Non-preemptive

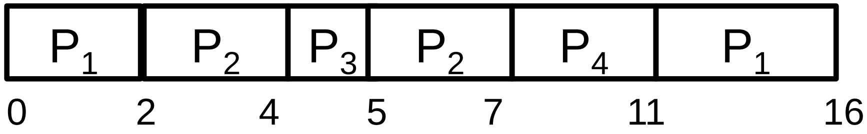

P_1 P_3 P_2 P_4

0 7 8 12 16

avg. waiting time = (0 + 6 + 3 + 7)/4 = 4

Process	Arrival	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

- Shortest Remaining Time First – Preemptive


P_1 P_2 P_3 P_2 P_4 P_1

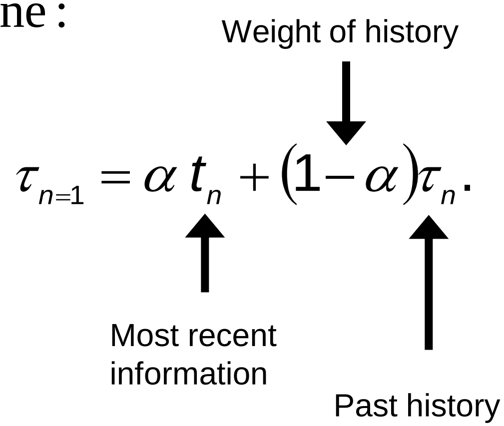
0 2 4 5 7 11 16

avg. waiting time = (9 + 1 + 0 +2)/4 = 3

Determining length of next CPU burst

- Can only *estimate* length
- Can be done using length of previous CPU bursts and exponential averaging

- $t_n =$ actual lenght of n^{th} CPU burst
- $\tau_{n+1} =$ predicted value for the next CPU burst
- $\alpha, 0 \le \alpha \le 1$
- Define :

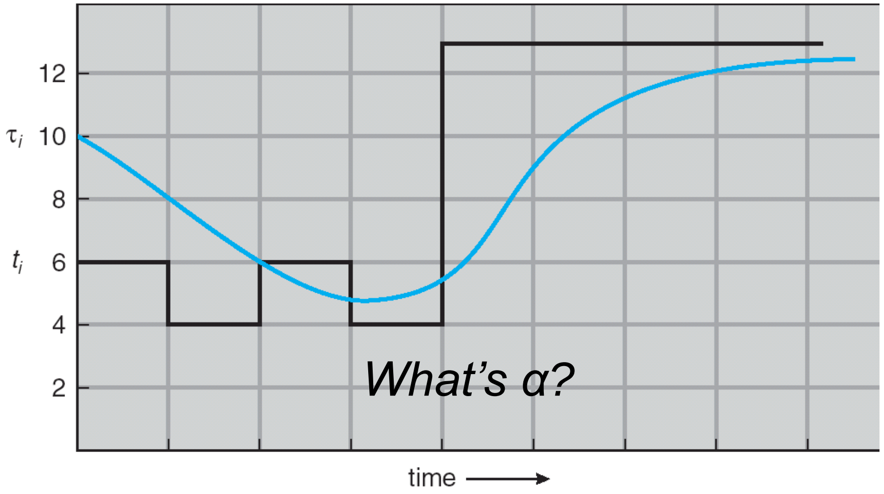


Weight of history

$$\tau_{n=1} = \alpha\, t_n + (1-\alpha)\tau_n .$$

Most recent information

Past history



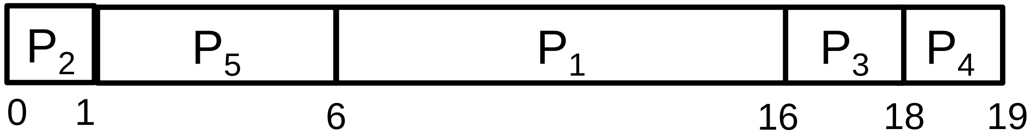
What's α?

CPU burst (t_i)	6	4	6	4	13	13	13	…
"guess" (τ_i) 10	8	6	6	5	9	11	12	…

Priority scheduling

- SJF is a special case of priority-based scheduling
  - Priority = reverse of predicted next CPU burst
- Pick process with highest priority (lowest number)

Process	Burst time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2



avg. waiting time = (6 + 0 + 16 +18 + 1)/5 = 8.2

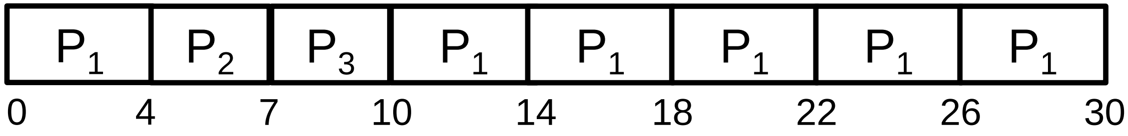
Priority scheduling issues

- And how do you assign priorities?
- Starvation
  - With an endless supply of high priority jobs, low priority processes may never execute
- Solution
  - Increases priority with age, i.e. accumulated waiting time
  - Decrease priority as a function of accumulated processing time
  - Assigned maximum quantum

Round-robin scheduling

- Simple, fair, easy to implement, & widely-used
- Each process gets a fix *quantum* or *time slice*
- When quantum expires, if running preempt CPU
- With *n* processes & quantum *q*, each one gets 1/*n* of the CPU time, no-one waits more than (*n*-1) *q*

q = 4



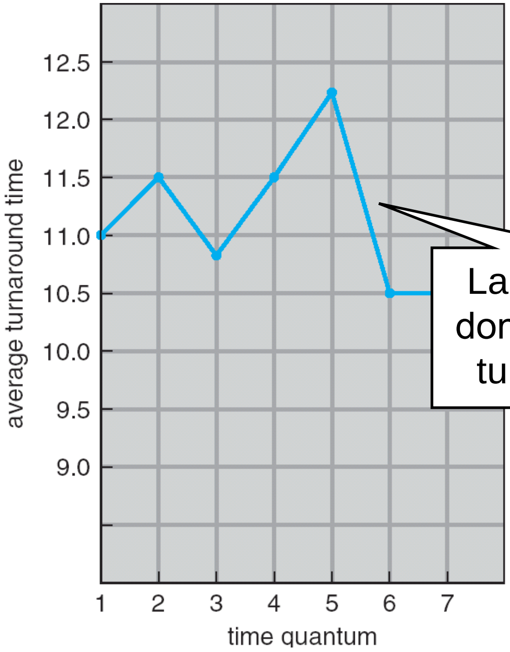
Process	Burst Time
P1	24
P2	3
P3	3

avg. waiting time = (6 + 4 +7)/3 = 5.66

Preempetive or not?

Quantum & Turnaround time

- Length of quantum
  - Too short – low CPU efficiency (*why?)*
  - Too long – low response time
    (*really long, what do you get?)*
  - Commonly ~ 50-100 msec.

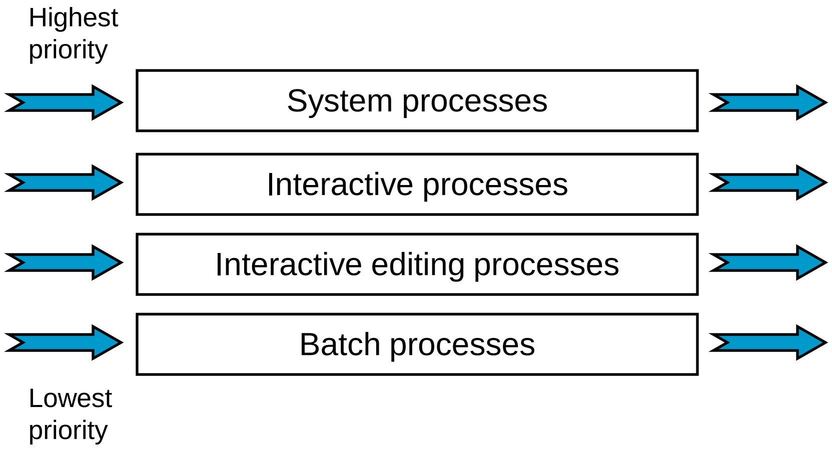


process	time
P_1	6
P_2	3
P_3	1
P_4	7

Largest quantums don’t imply shortest turnaround times

Combining algorithms

- In practice, any real system uses some hybrid approach, with elements of each algorithm
- Multilevel queue
  - Ready queue partitioned into separate queues
  - Each queue has its own scheduling algorithm
  - Scheduling must be done between the queues
    - Fixed priority scheduling; (i.e., foreground first); starvation?
    - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes



Multiple (feedback) queues

- Multiple queues, allow processes to move bet/ queues
- Example CTSS – Idea: separate processes based on CPU bursts
  - IBM 7094 had space for 1 process in memory (switch = swap)
  - Goals: low context switching cost & good response time
  - Priority classes: class *i* gets 2^i quantas
  - Scheduler executes first all processes in queue 0; if empty, all in queue 1, …
  - If process uses all its quanta → move to next lower queue (leave I/O-bound & interact. processes in high-priority queue)
  - What about process with long start but interactive after that?

Carriage-return hit → promote process to top class

Multiple-processor scheduling

- Scheduling more complex w/ multiple CPUs
- Asymmetric/symmetric (SMP) multiprocessing
  - Supported by most OSs (common or independent ready queues)
- Processor affinity – benefits of past history in a processor
- Load balancing – keep workload evenly distributed
  - Push migration – specific task pushes processes for balance
  - Pull migration – idle processor asks for/pulls work
- Symmetric multithreading (hyperthreading or SMT)
  - Multiple logical processors on a physical one
  - Each w/ own architecture state, supported by hardware
  - Shouldn’t require OS to know about it (but could benefit from)

Thread scheduling

- Now add threads – user or kernel level?
- User-level (process-contention scope)
  - Context switch is cheaper
  - You can have an application-specific scheduler at user level
  - Kernel doesn’t know of your threads
- Kernel-level (system-contention scope)
  - Any scheduling of threads is possible (since the kernel knows of all)
  - Switching threads inside same process is cheaper than switching processes

Real-time scheduling

- Different categories
  - *Hard RT* – not on time ~ not at all
  - *Soft RT* – important to meet guarantees but not critical
- Scheduling can be static or dynamic
- Schedulable real-time system
  - The events that a RT system may have to respond could be periodic or aperiodic
  - Given a set of *m* periodic events, can it handle it? schedulable
    - event *i* occurs within period P_i and requires C_i seconds

Then the load can only be handled if

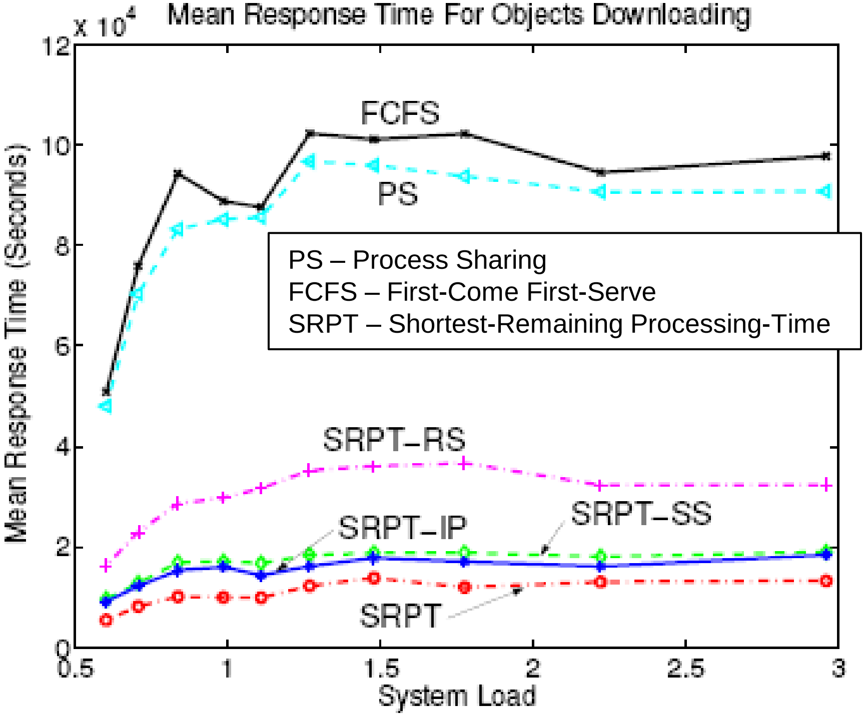
$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

P1: C = 50 msec, P = 100msec (.5)
P2: C = 30 msec, P = 200msec (.15)
P3: C = 100 msec, P = 500msec (.2)
P4: C = 200 msec, P= 1000msec (.2)

Scheduling the server-side of P2P systems

- P2P users' response is dominated by download
  - >80% download requests in Kazaa are rejected due to capacity saturation at server peers
  - >50% of all requests for large objects (>100MB) take more than one day & ~20% take over one week to complete
- Most implementations use FCFS or PS
- *Apply SRPT!* Work from Nortwestern

Mean response time of object download as a function of system load.



PS – Process Sharing
FCFS – First-Come First-Serve
SRPT – Shortest-Remaining Processing-Time

Some other algorithms

- Guaranteed sched. - e.g. proportional to # processes
  - Priority = amount used / amount promised
  - Lower ratio → higher priority
- Lottery scheduling – simple & predictable
  - Each process gets lottery tickets for resources (CPU time)
  - Scheduling – lottery, i.e. randomly pick a ticket
  - Priority – more tickets means higher chance
  - Processes may exchange tickets
- Fair-Share scheduling
  - Schedule aware of ownership
  - Owners get a % of CPU, processes are picked to enforce it

Policy vs. mechanism

- Separate what is done from how it is done
  - Think of parent process with multiple children
  - Parent process may knows relative importance of children (if, for example, each one has a different task)
- None of the algorithms presented takes the parent process input for scheduling
- Scheduling algorithm parameterized
  - Mechanism in the kernel
- Parameters filled in by user processes
  - Policy set by user process
  - Parent controls scheduling w/o doing it

Scheduling in xv6

```
void
scheduler(void)
{
  struct proc *p;

  for(;;){
    // Enable interrupts on this processor.
    sti();
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      // Switch to chosen process. It is the process's job to release
      // ptable.lock and then reacquire it before jumping back to us.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
     }
  release(&ptable.lock);
}
```

Enable interrupts to handle whatever is there before continuing

Switches hw page table and TSS registers to point to process

Switches hw page table register to the kernel-only page table, for when no process is running

Scheduling in xv6

```
void
sched(void)
{
    int intena;

    if(!holding(&ptable.lock))
       panic("sched ptable.lock");
    if(cpu->ncli != 1)
       panic("sched locks");
    if(proc->state == RUNNING)
       panic("sched running");
    if(readeflags()&FL_IF)
       panic("sched interruptible");
    intena = cpu->intena;
    swtch(&proc->context, cpu->scheduler);
    cpu->intena = intena;
 }
```

Convention in xv6: a process that wants to give up the CPU must acquire the process table loc, release any other lock it is holding, update its own state and call `sched`.

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
    acquire(&ptable.lock);
    proc->state = RUNNABLE;
    sched();
    release(&ptable.lock);
}
```

Scheduling in xv6

```
# Context switch
# void swtch(struct context **old, struct context *new);
# Save current register context in old
# and then load register context from new.

.globl swtch
swtch:
   movl 4(%esp), %eax
   movl 8(%esp), %edx

   # Save old callee-save registers
   pushl %ebp
   pushl %ebx
   pushl %esi
   pushl %edi

   # Switch stacks
   movl %esp, (%eax)
   movl %edx, %esp

   # Load new callee-save registers
   popl %edi
   popl %esi
   popl %ebx
   popl %ebp
   ret
```

Loads arguments off the stack into %eax and %edx before changing stack pointer

Pushes register state creating a context structure on the current stack;
%esp is save implicitly to *old;
%eip was saved by `call` instruction that invoked swtch and is above %ebp

Switch stacks

New stack has same format, so just undo;
`ret` has the %eip at the top

Next time

- Process synchronization
  - Race condition & critical regions
  - Software and hardware solutions
  - Review of classical synchronization problems
  - …
- *What really happened on Mars?*
  http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html