Lecture 4: Scheduling

Instructor: Mitch Neilsen

Office: N219D

Quote of the Day

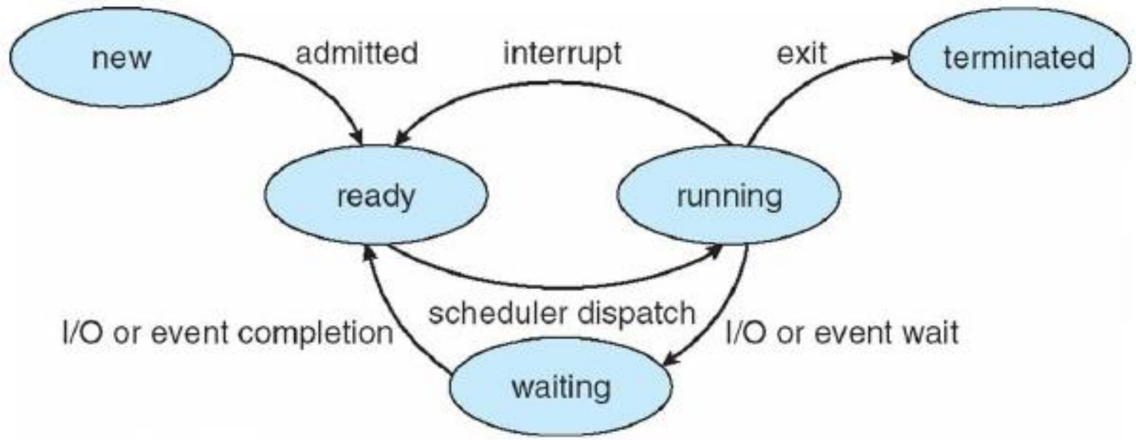
"There is nothing so annoying as to have two people talking when you're busy interrupting."

--Mark Twain

Outline

- To Do:
  - Read Ch. 1-5
  - Create Team of 2-3 for Project 1 – start this Friday
- Last time: Processes and Threads
  - Threads (Ch. 4)
- Today: Process Scheduling (Ch. 5)
  - Project 1 – Scheduling and Synchronization

Process states



- **Process can be in one of several states**
  - *new* & *terminated* at beginning & end of life
  - *running* – currently executing (or will execute on kernel return)
  - *ready* – can run, but kernel has chosen different process to run
  - *waiting* – needs async event (e.g., disk operation) to proceed
- **Which process should kernel run?**
  - if 0 runnable, run idle loop, if 1 runnable, run it
  - if >1 runnable, must make scheduling decision

Scheduling

- **How to pick which process to run**
- **Scan process table for first runnable?**
  - Expensive. Weird priorities (small pids 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

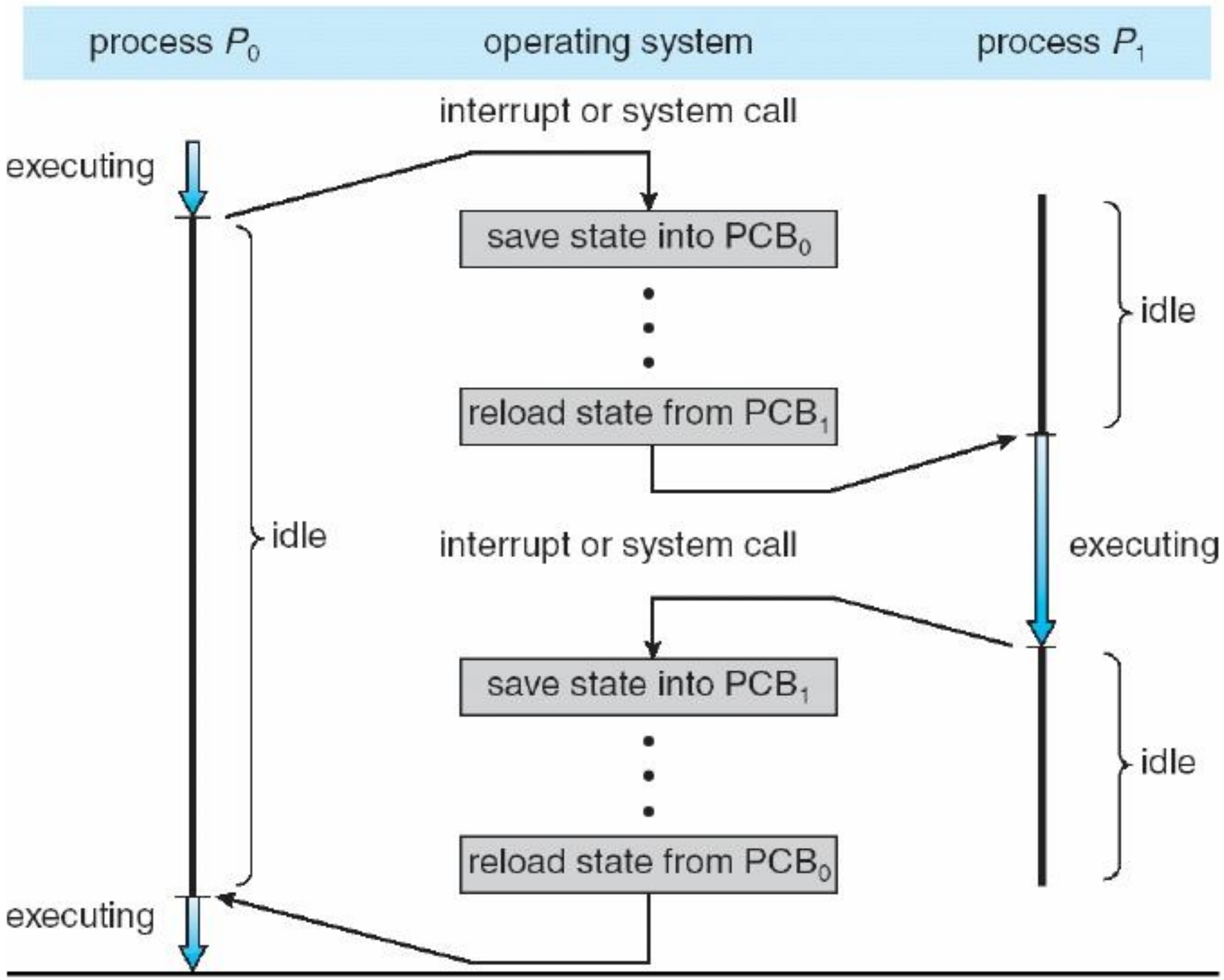
Scheduling policy

- **Want to balance multiple goals**
  - *Fairness* – don’t starve processes
  - *Priority* – reflect relative importance of processes
  - *Deadlines* – must do *x* (play audio) by certain time
  - *Throughput* – want good overall performance
  - *Efficiency* – minimize overhead of scheduler itself
- **No universal policy**
  - Many variables, can’t optimize for all
  - Conflicting goals (e.g., throughput or priority vs. fairness)
- **We will spend several lectures on this topic**

Preemption

- **Can preempt a process when kernel gets control**
- **Running process can vector control to kernel**
  - System call, page fault, illegal instruction, etc.
  - May put current process to sleep—e.g., read from disk
  - May make other process runnable—e.g., fork, write to pipe
- **Periodic timer interrupt**
  - If running process used up quantum, schedule another
- **Device interrupt**
  - Disk request completed, or packet arrived on network
  - Previously waiting process becomes runnable
  - Schedule if higher priority than current running proc.
- **Changing running process is called a *context switch***

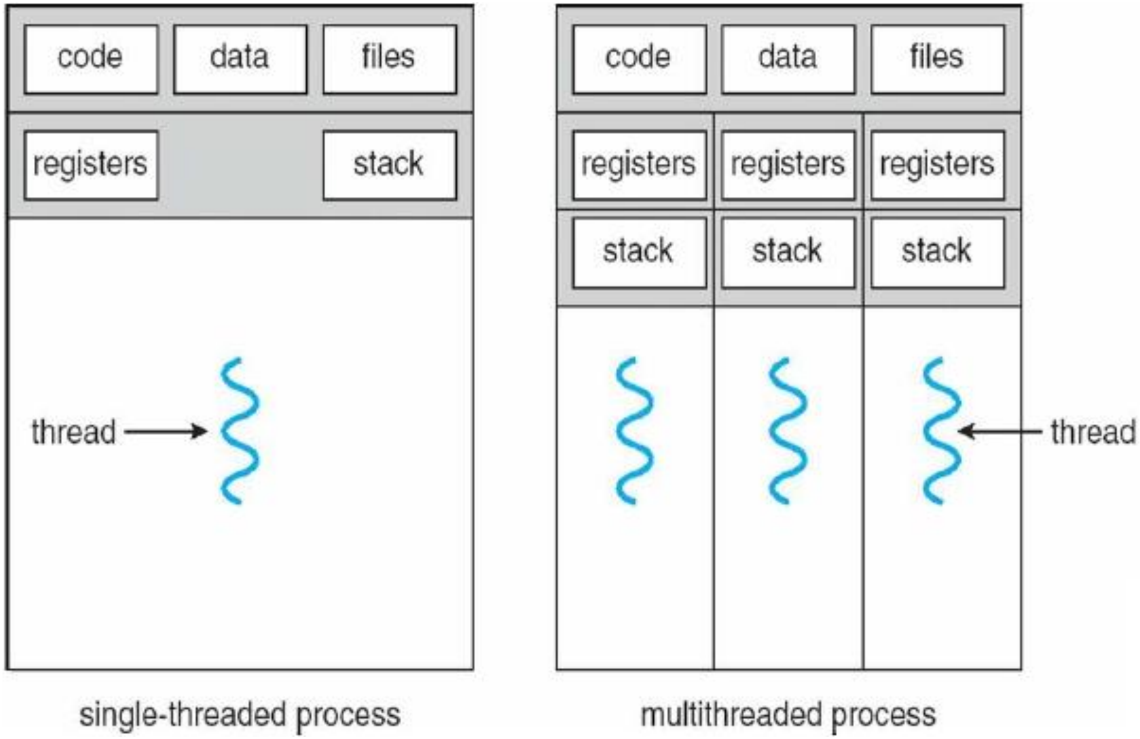
Context switch



Context switch details

- **Very machine dependent. Typical things include:**
  - Save program counter and integer registers (always)
  - Save floating point or other special registers
  - Save condition codes
  - Change virtual address translations
- **Non-negligible cost**
  - Save/restore floating point registers expensive
    - Optimization: only save if process used floating point
  - May require flushing TLB (memory translation hardware)
    - Optimization: don’t flush kernel’s own data from TLB
  - Usually causes more cache misses (switch working sets)

Threads



- **A thread is a schedulable execution context**
  - Program counter, stack, registers, . . .
- **Simple programs use one thread per process**
- **But can also have multi-threaded programs**
  - Multiple threads running in same process's address space

Why threads?

- **Most popular abstraction for concurrency**
  - Lighter-weight abstraction than processes
  - All threads in one process share memory, file descriptors, etc.
- **Allows one process to use multiple CPUs or cores**
- **Allows program to overlap I/O and computation**
  - Same benefit as OS running emacs & gcc simultaneously
  - E.g., threaded web server services clients simultaneously:

```
for (;;) {
    fd = accept_client ();
    thread_create (service_client, &fd);
}
```

- **Most kernels have threads, too**
  - Typically at least one kernel thread for every process

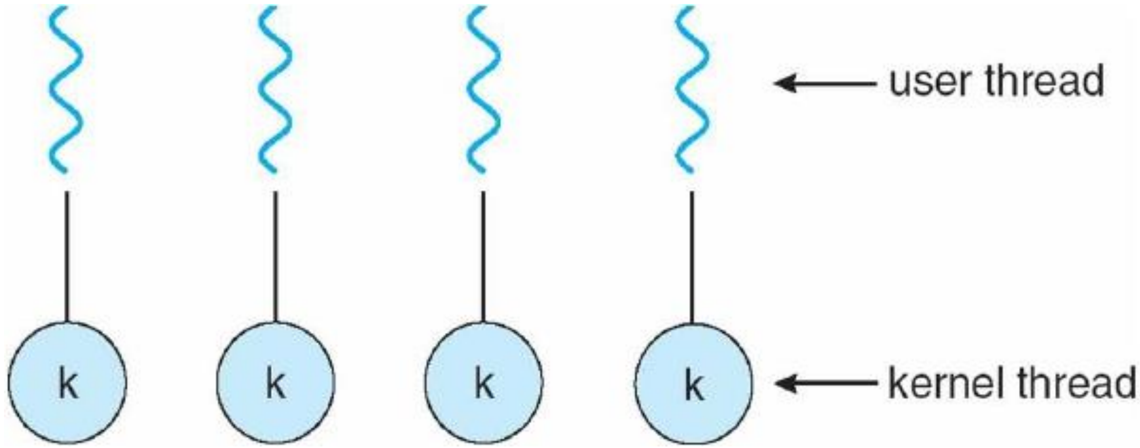
Thread package API

- tid thread_create (void (*fn) (void *), void *);
  - Create a new thread, run fn with arg
- void thread_exit ();
  - Destroy current thread
- void thread_join (tid thread);
  - Wait for thread thread to exit
- **Plus lots of support for synchronization [next week]**
- **Can have preemptive or non-preemptive threads**
  - Preemptive causes more race conditions
  - Non-preemptive can't take advantage of multiple CPUs
  - Before prevalent SMPs, most kernels non-preemptive

Limitations of kernel-level threads

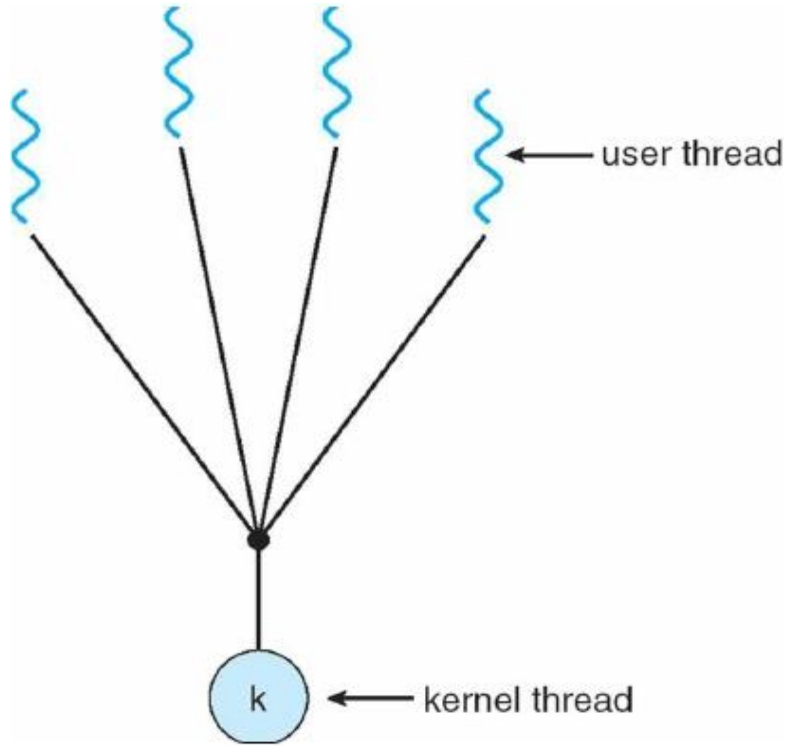
- **Every thread operation must go through kernel**
  - create, exit, join, synchronize, or switch for any reason
  - On Athlon 3400+: syscall takes 359 cycles, fn call 6 cycles
  - Result: threads 10x-30x slower when implemented in kernel
- **One-size fits all thread implementation**
  - Kernel threads must please all people
  - Maybe pay for fancy features (priority, etc.) you don't need
- **General heavy-weight memory requirements**
  - E.g., requires a fixed-size stack within kernel
  - Other data structures designed for heavier-weight processes

Kernel threads



- **Can implement** thread_create **as a system call**
- **To add** thread_create **to an OS that doesn't have it:**
  - Start with process abstraction in kernel
  - thread_create like process creation with features stripped out
    - Keep same address space, file table, etc., in new process
    - rfork/clone syscalls actually allow individual control
- **Faster than a process, but still very heavy weight**

User threads



- **An alternative: implement in user-level library**
  - One kernel thread per process
  - thread_create, thread_exit, etc., just library functions

Background: procedure calls

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

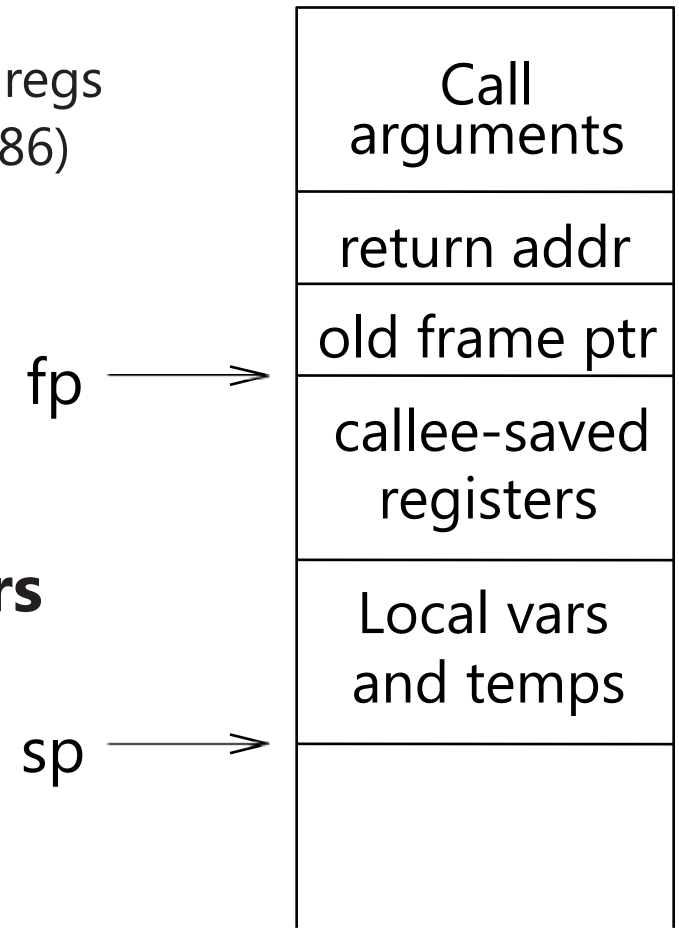
- **Some state saved on stack**
  - Return address, caller-saved registers
- **Some state not saved**
  - Callee-saved regs, global variables, stack pointer

Implementing user-level threads

- **Allocate a new stack for each** thread_create
- **Keep a queue of runnable threads**
- **Replace networking system calls (**read**/**write**/etc.)**
  - If operation would block, switch and run different thread
- **Schedule periodic timer signal (**setitimer**)**
  - Switch to another thread on timer signals (preemption)
- **Multi-threaded web server example**
  - Thread calls read to get data from remote web browser
  - “Fake” user-level read make read syscall in non-blocking mode
  - No data? schedule another thread
  - On timer or when idle check which connections have new data
- **How to switch threads?**

Background: calling conventions

- **Registers divided into 2 groups**
  - Functions free to clobber *caller-saved* regs (%eax [return val], %edx, & %ecx on x86)
  - But must restore *callee-saved* ones to original value upon return
- ***sp* register always base of stack**
  - Frame pointer (*fp*) is old *sp*
- **Local variables stored in registers and on stack**
- **Function arguments go in callee-saved regs and on stack**
  - With x86, all arguments on stack



Background: procedure calls

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

- **Some state saved on stack**
  - Return address, caller-saved registers
- **Some state not saved**
  - Callee-saved regs, global variables, stack pointer

Threads vs. procedures

- **Threads may resume out of order:**
  - Cannot use LIFO stack to save state
  - General solution: one stack per thread
- **Threads switch less often:**
  - Don’t partition registers (why?)
- **Threads can be involuntarily interrupted:**
  - Synchronous: procedure call can use compiler to save state
  - Asynchronous: thread switch code saves all registers
- **More than one than one thread can run at a time**
  - Thread scheduling: What to run next and on which CPU?
  - Procedure call scheduling obvious: Run called procedure

Example user threads implementation

- **Per-thread state in thread control block structure**

```
typedef struct tcb {
   unsigned long md_esp; /* Stack pointer of thread */
   char *t_stack;             /* Bottom of thread's stack */
   /* ... */
};
```

- **Machine-dependent thread-switch function:**
  - void thread_md_switch (tcb *current, tcb *next);
- **Machine-dependent thread initialization function:**
  - void thread_md_init (tcb *t,
    void (*fn) (void *), void *arg);

i386 thread_md_switch

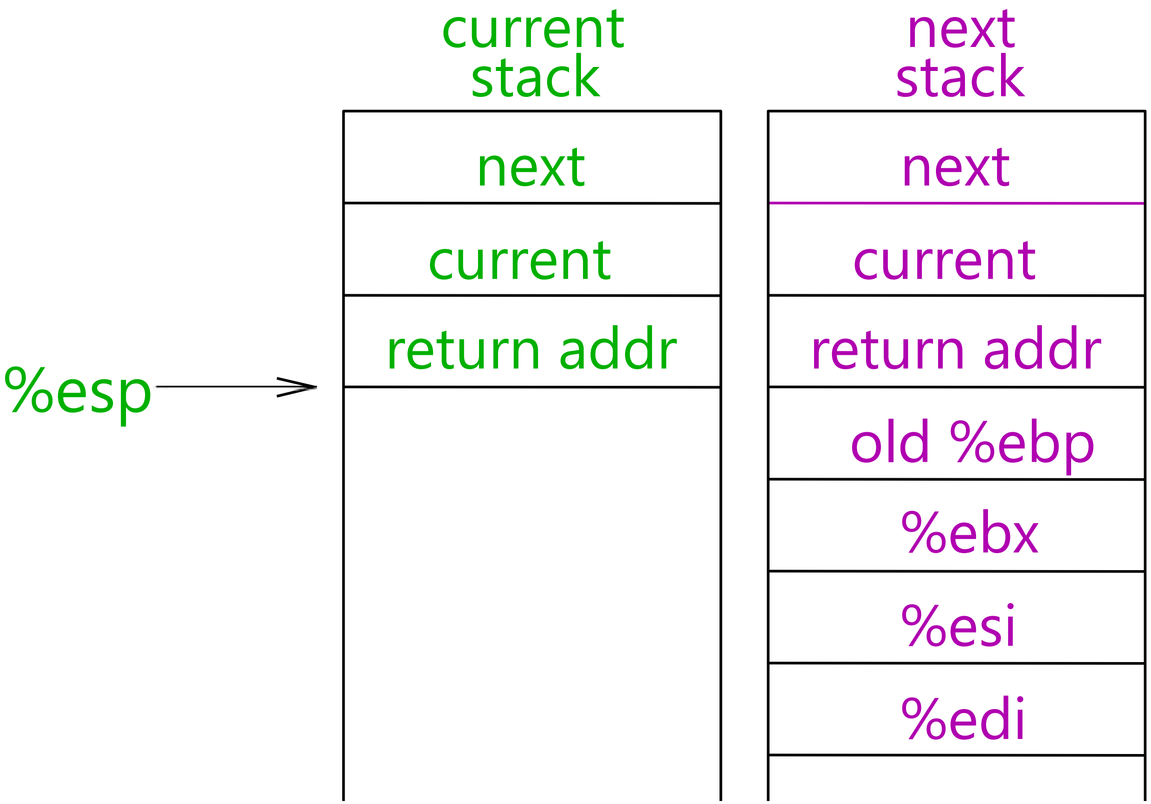
```
pushl %ebp; movl %esp,%ebp              # Save frame pointer
pushl %ebx; pushl %esi; pushl %edi      # Save callee-saved regs

movl 8(%ebp),%edx                       # %edx = thread_current
movl 12(%ebp),%eax                      # %eax = thread_next
movl %esp,(%edx)                        # %edx->md_esp = %esp
movl (%eax),%esp                        # %esp = %eax->md_esp

popl %edi; popl %esi; popl %ebx         # Restore callee saved regs
popl %ebp                               # Restore frame pointer
ret                                     # Resume execution
```

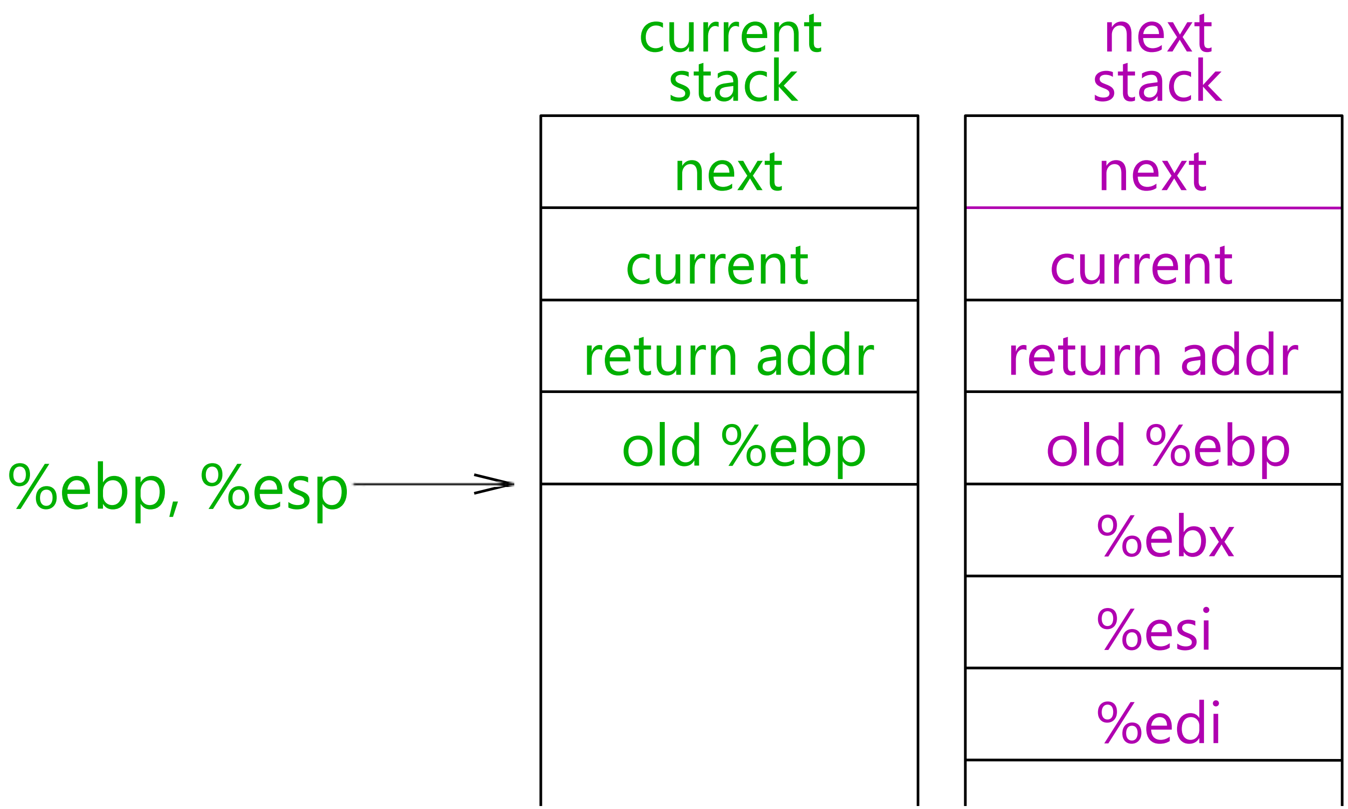
- **This is literally switch code from simple thread lib**
  - Nothing magic happens here
  - You will see very similar code in Pintos switch.S

i386 thread_md_switch



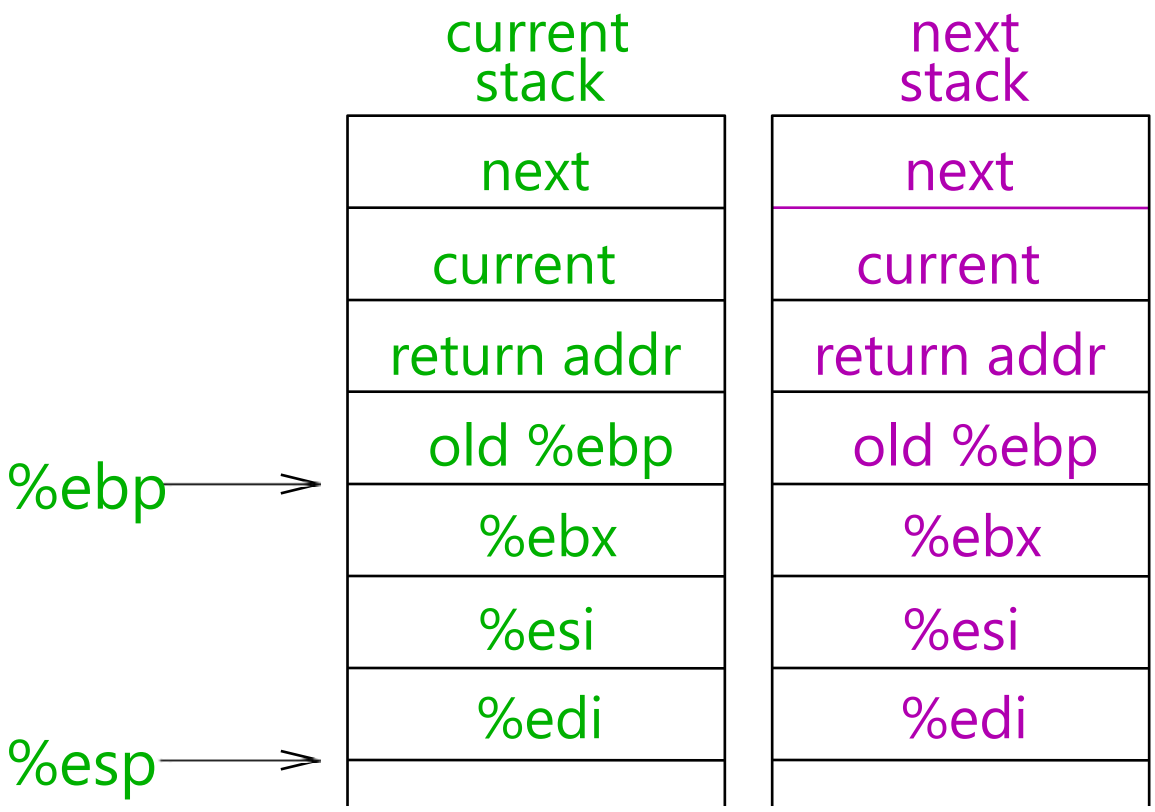
- **This is literally switch code from simple thread lib**
  - Nothing magic happens here
  - You will see very similar code in Pintos switch.S

i386 thread_md_switch



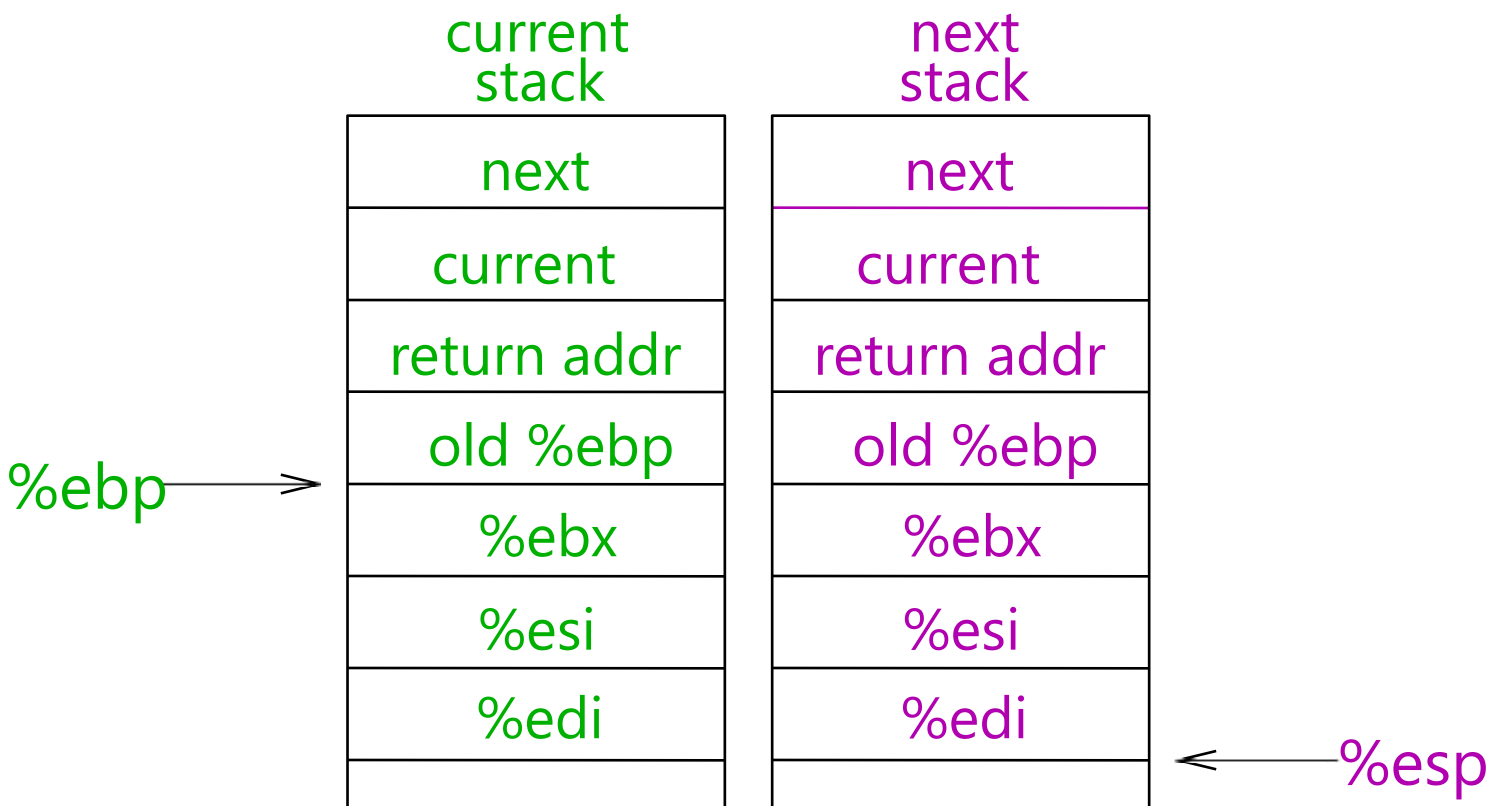
- **This is literally switch code from simple thread lib**
  - Nothing magic happens here
  - You will see very similar code in Pintos switch.S

i386 thread_md_switch



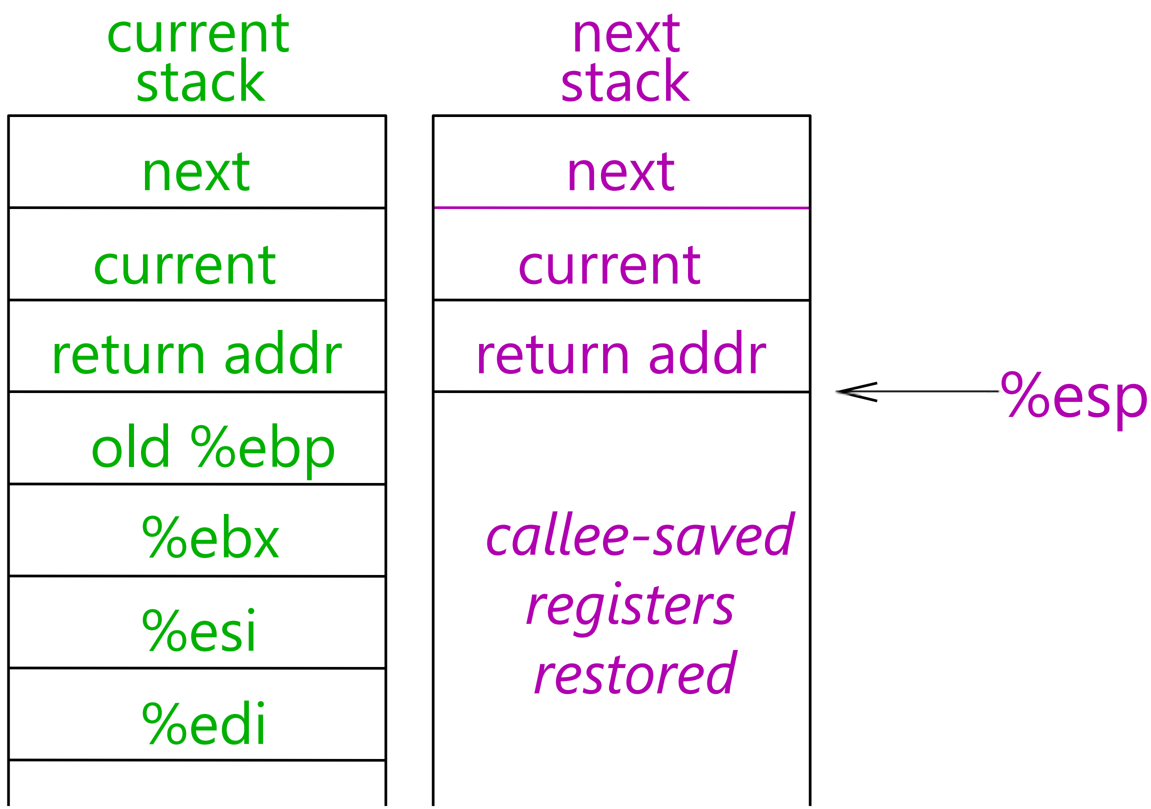
- **This is literally switch code from simple thread lib**
  - Nothing magic happens here
  - You will see very similar code in Pintos switch.S

i386 thread_md_switch



- **This is literally switch code from simple thread lib**
  - Nothing magic happens here
  - You will see very similar code in Pintos switch.S

i386 thread_md_switch

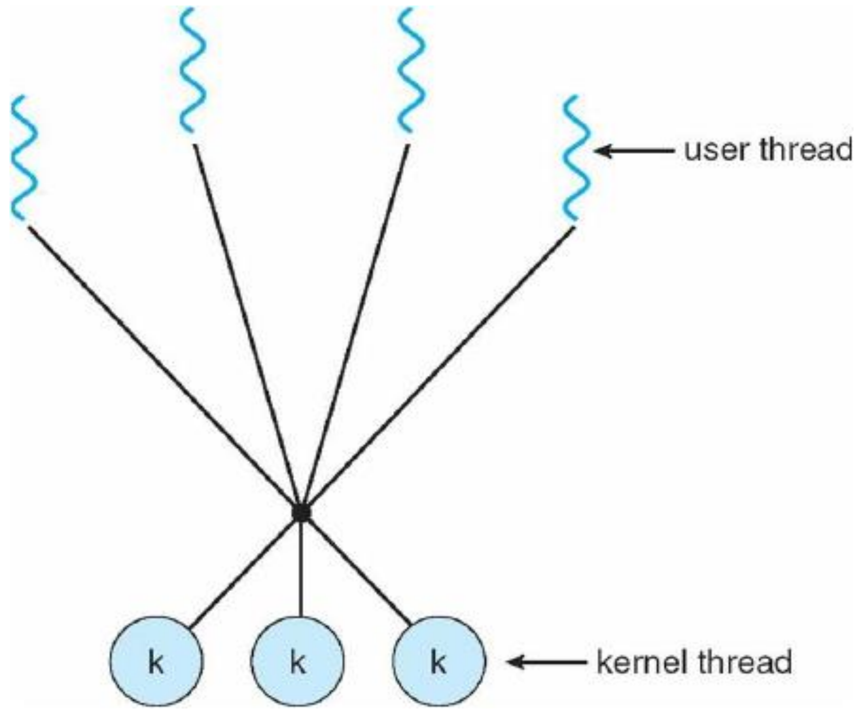


- **This is literally switch code from simple thread lib**
  - Nothing magic happens here
  - You will see very similar code in Pintos switch.S

Limitations of user-level threads

- **Can't take advantage of multiple CPUs or cores**
- **A blocking system call blocks all threads**
  - Can replace read to handle network connections
  - But usually OSes don't let you do this for disk
  - So one un-cached disk read blocks all threads
- **A page fault blocks all threads**
- **Possible deadlock if one thread blocks on another**
  - May block entire process and make no progress
  - [More on deadlock next week.]

User threads on kernel threads



- **User threads implemented on kernel threads**
  - Multiple kernel-level threads per process
  - thread_create, thread_exit still library functions as before
- **Sometimes called *n* : *m* threading**
  - Have *n* user threads per *m* kernel threads
    (Simple user-level threads are *n* : 1, kernel threads 1 : 1)

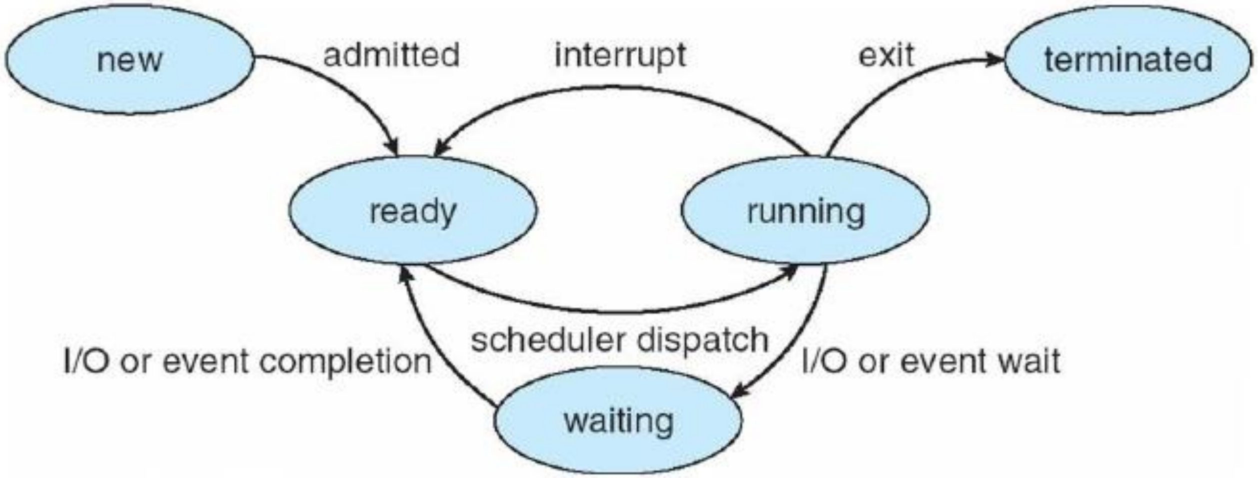
Limitations of $n : m$ threading

- **Many of same problems as $n : 1$ threads**
  - Blocked threads, deadlock, . . .
- **Hard to keep same # ktrheads as available CPUs**
  - Kernel knows how many CPUs available
  - Kernel knows which kernel-level threads are blocked
  - But tries to hide these things from applications for transparency
  - So user-level thread scheduler might think a thread is running while underlying kernel thread is blocked
- **Kernel doesn’t know relative importance of threads**
  - Might preempt kthread in which library holds important lock

Lessons

- **Threads best implemented as a library**
  - But kernel threads not best interface on which to do this
- **Better kernel interfaces have been suggested**
  - See Scheduler Activations [Anderson et al.]
  - Maybe too complex to implement on existing OSes (some have added then removed such features)
- **Today shouldn’t dissuade you from using threads**
  - Standard user or kernel threads are fine for most purposes
  - Use kernel threads if I/O concurrency main goal
  - Use *n* : *m* threads for highly concurrent (e.g,. scientific applications) with many thread switches
- **. . . though the next lecture may disuade you**
  - Concurrency greatly increases the complexity of a program!
  - Leads to all kinds of nasty race conditions

CPU Scheduling



- **Scheduling decisions may take place when a process:**
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Exits
- **Non-preemptive schedules use 1 & 4 only**
- **Preemptive schedulers run at all four points**

Scheduling criteria

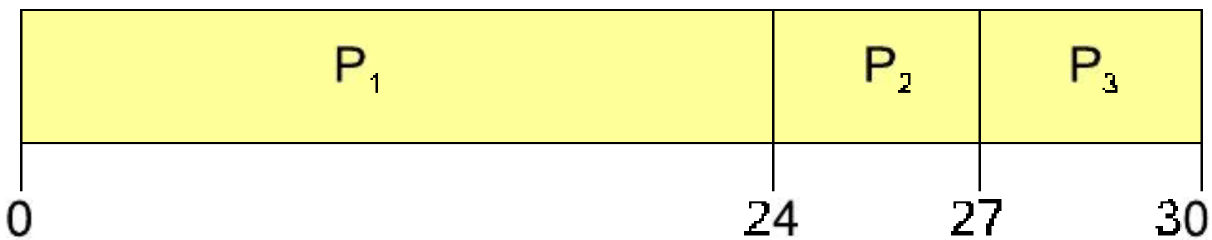
- **Why do we care?**
  - What goals should we have for a scheduling algorithm?

Scheduling criteria

- **Why do we care?**
  - What goals should we have for a scheduling algorithm?
- ***Throughput*** **– # of procs that complete per unit time**
  - Higher is better
- ***Turnaround time*** **– time for each proc to complete**
  - Lower is better
- ***Response time*** **– time from request to first response (e.g., key press to character echo, not launch to exit)**
  - Lower is better
- **Above criteria are affected by secondary criteria**
  - *CPU utilization* – keep the CPU as busy as possible
  - *Waiting time* – time each proc waits in ready queue

Example: FCFS Scheduling

- **Run jobs in order that they arrive**
  - Called "*First-come first-served*" (FCFS)
  - E.g.., Say P_1 needs 24 sec, while P_2 and P_3 need 3.
  - Say P_2 , P_3 arrived immediately after P_1 , get:



- **Dirt simple to implement—how good is it?**
- **Throughput: 3 jobs / 30 sec = 0.1 jobs/sec**
- **Turnaround Time:** P_1 : 24**,** P_2 : 27**,** P_3 : 30
  - Average TT: (24 + 27 + 30)/3 = 27
- **Can we do better?**

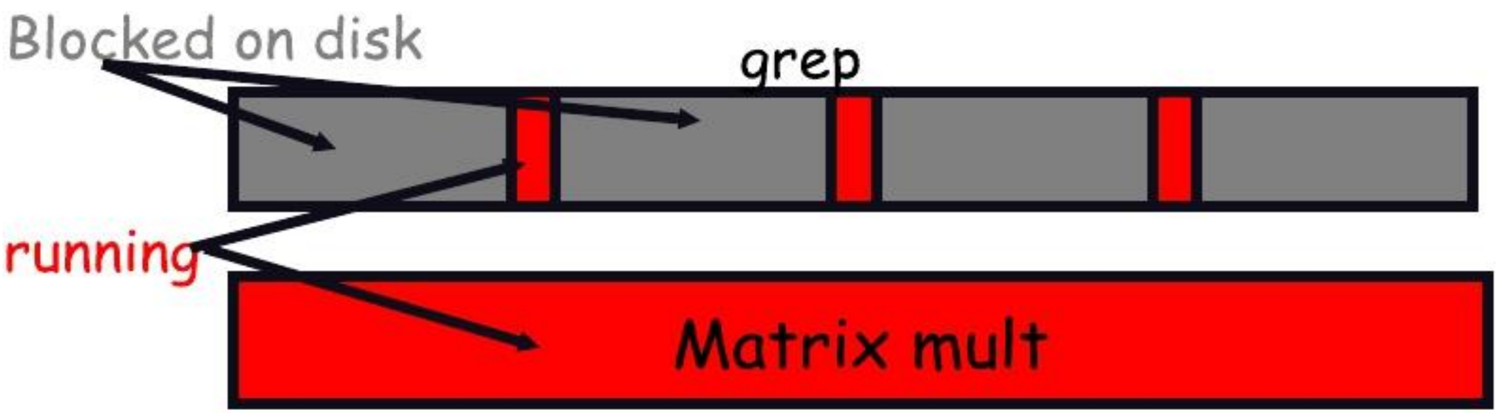
FCFS continued

- **Suppose we scheduled** P_2**,** P_3**, then** P_1
  - Would get:

- **Throughput: 3 jobs / 30 sec = 0.1 jobs/sec**
- **Turnaround time:** $P_1 : 30$**,** $P_2 : 3$**,** $P_3 : 6$
  - Average TT: (30 + 3 + 6)/3 = 13 – much less than 27
- **Lesson: scheduling algorithm can reduce TT**
  - Minimizing waiting time can improve RT and TT
- **What about throughput?**

View CPU and I/O devices the same

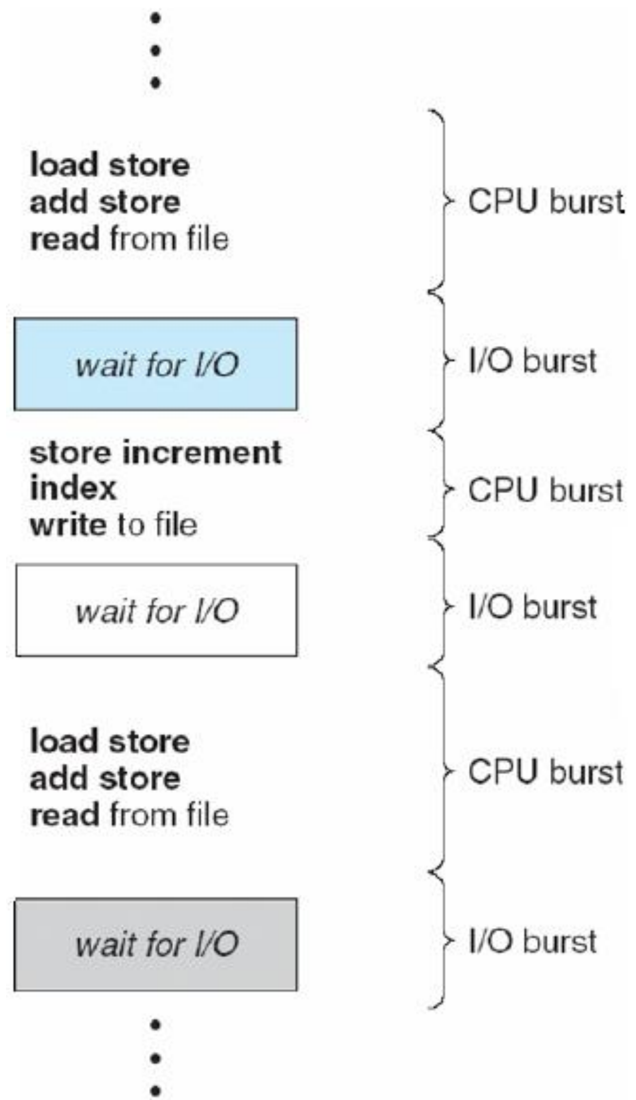
- **CPU is one of several devices needed by users' jobs**
  - CPU runs compute jobs, Disk drive runs disk jobs, etc.
  - With network, part of job may run on remote CPU
- **Scheduling 1-CPU system with *n* I/O devices like scheduling asymmetric *n* + 1-CPU multiprocessor**
  - Result: all I/O devices + CPU busy =⇒ n+1 fold speedup!



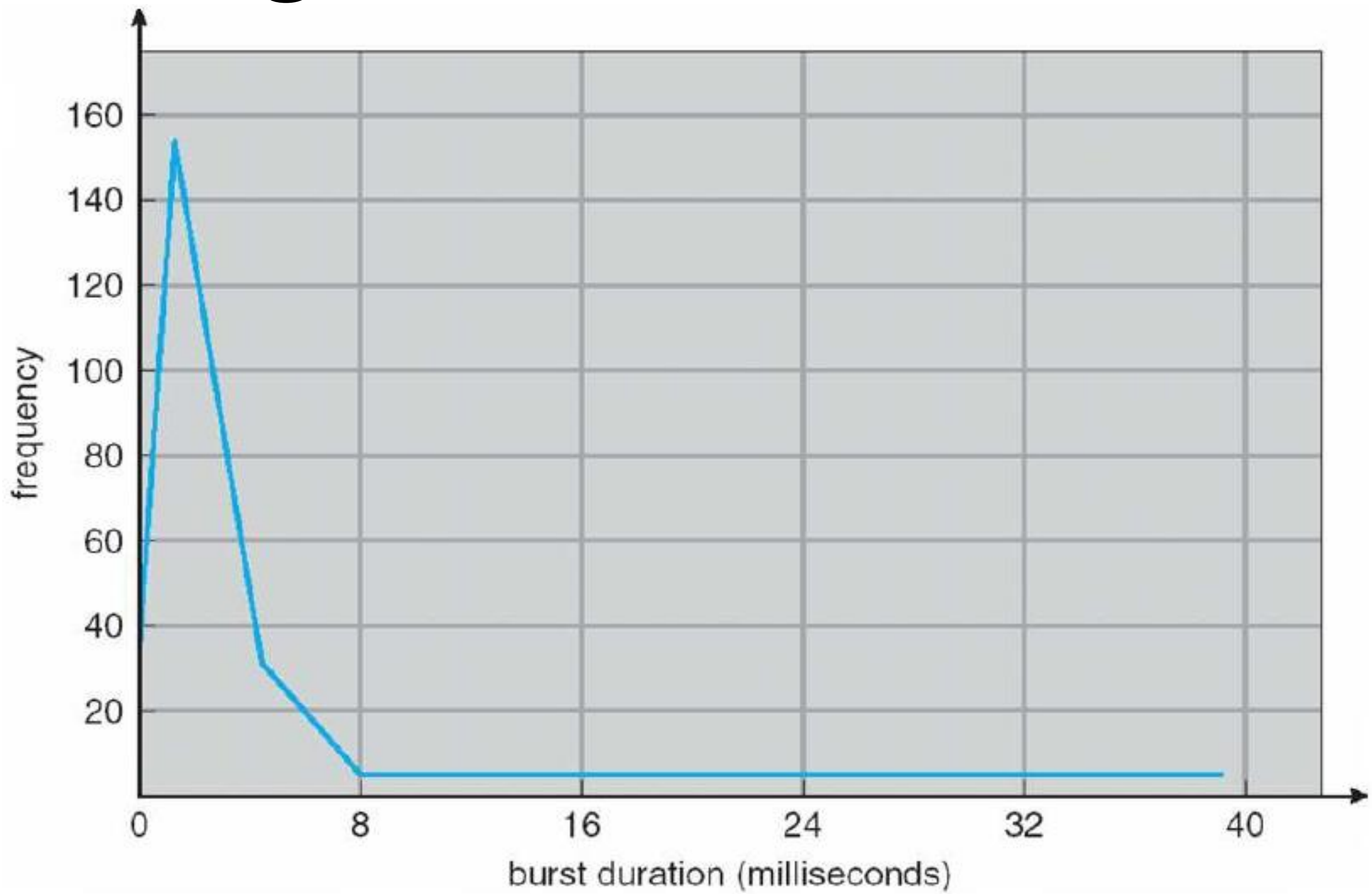
  - Overlap them just right? throughput will be almost doubled

Bursts of computation & I/O

- **Jobs contain I/O and computation**
  - Bursts of computation
  - Then must wait for I/O
- **To Maximize throughput**
  - Must maximize CPU utilization
  - Also maximize I/O device utilization
- **How to do?**
  - Overlap I/O & computation from multiple jobs
  - Means *response time* very important for I/O-intensive jobs: I/O device will be idle until job gets small amount of CPU to issue next I/O request



Histogram of CPU-burst times



- **What does this mean for FCFS?**

FCFS Convoy effect

- **CPU bound jobs will hold CPU until exit or I/O (but I/O rare for CPU-bound thread)**
  - long periods where no I/O requests issued, and CPU held
  - Result: poor I/O device utilization
- **Example: one CPU-bound job, many I/O bound**
  - CPU bound runs (I/O devices idle)
  - CPU bound blocks
  - I/O bound job(s) run, quickly block on I/O
  - CPU bound runs again
  - I/O completes
  - CPU bound job continues while I/O devices idle
- **Simple hack: run process whose I/O completed?**
  - What is a potential problem?

Adding a new list of threads in Pintos

- In src/lib/kernel/list.c:

```
/* Our doubly linked lists have two header elements: the "head"
   just before the first element and the "tail" just after the
   last element.  The `prev' link of the front header is null, as
   is the `next' link of the back header.  Their other two links
   point toward each other via the interior elements of the list.

   An empty list looks like this:

                      +------+     +------+
                  <---| head |<--->| tail |--->
                      +------+     +------+

   A list with two elements in it looks like this:

        +------+     +-------+     +-------+     +------+
    <---| head |<--->|   1   |<--->|   2   |<--->| tail |<--->
        +------+     +-------+     +-------+     +------+

   The symmetry of this arrangement eliminates lots of special
   cases in list processing.
```

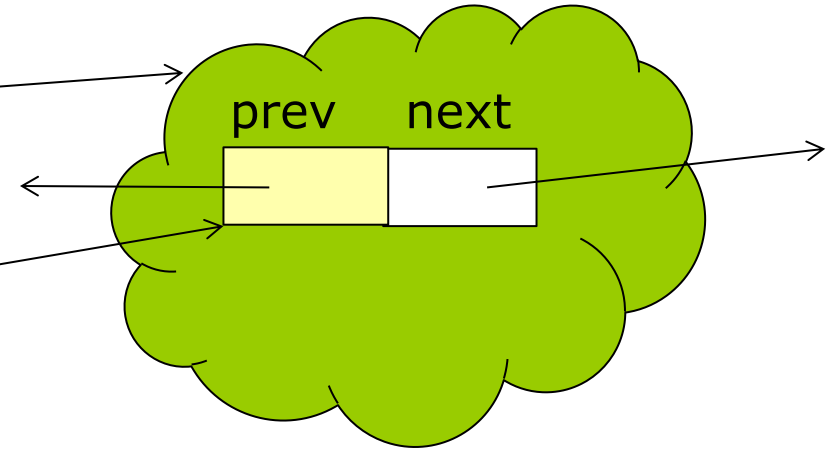
src/lib/kernel/list.h

```
/* Doubly linked list.

   This implementation of a doubly linked list does not require
   use of dynamically allocated memory.  Instead, each structure
   that is a potential list element must embed a struct list_elem
   member.  All of the list functions operate on these `struct
   list_elem's.  The list_entry macro allows conversion from a
   struct list_elem back to a structure object that contains it.

   For example, suppose there is a needed for a list of `struct
   foo'.  `struct foo' should contain a `struct list_elem'
   member, like so:

      struct foo
        {
          struct list_elem elem;
          int bar;
          ...other members...
        };
```



src/lib/kernel/list.h

```
Then a list of `struct foo' can be be declared and initialized
like so:
      struct list foo_list;
      list_init (&foo_list);

Iteration is a typical situation where it is necessary to
convert from a struct list_elem back to its enclosing
structure.  Here's an example using foo_list:

      struct list_elem *e;

      for (e = list_begin (&foo_list); e != list_end (&foo_list);
           e = list_next (e))
        {
          struct foo *f = list_entry (e, struct foo, elem);
          ...do something with f...
        }
```

- In **src/threads/thread.h**, add: `struct list_elem timer_list_elem;`
- In **src/devices/timer.c**, add: `static struct list wait_list;`
- In **src/devices/timer.c**, in timer_init(), add: `list_init(&wait_list);`

src/lib/kernel/list.h

- In src/threads/timer.c, add the current thread to the wait list in timer_sleep():

```
void timer_sleep (int64_t ticks)
{
  struct thread *t = thread_current ();

  /* Schedule our wake-up time. */
  t->wakeup_time = …

  /* Insert the current thread into the wait list. */
  intr_disable ();
  list_insert_ordered (&wait_list, &t->timer_list_elem,
                       compare_threads_by_wakeup_time, NULL);
  intr_enable ();

  /* Block this thread until timer expires. */
  …
}
```

Summary

- Read Ch. 1-5
- Watch GitHub Video
- Processes and Threads (Ch. 4)
- Process Scheduling (Ch. 5)
- Project 1 – Scheduling and Synchronization