

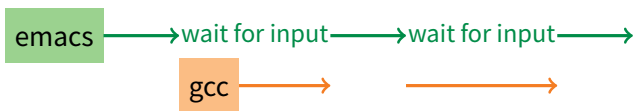
Processes

- **A *process* is an instance of a program running**
- **Modern OSes run multiple processes simultaneously**
- **Examples (can all run simultaneously):**
 - `gcc file_A.c` – compiler running on file A
 - `gcc file_B.c` – compiler running on file B
 - `emacs` – text editor
 - `firefox` – web browser
- **Non-examples (implemented as one process):**
 - Multiple firefox windows or emacs frames (still one process)
- **Why processes?**
 - Simplicity of programming
 - Speed: Higher throughput, lower latency

Speed

- **Multiple processes can increase CPU utilization**

- Overlap one process's computation with another's wait



- **Multiple processes can reduce latency**

- Running *A* then *B* requires 100 sec for *B* to complete



- Running *A* and *B* concurrently makes *B* finish faster



- *A* is slower than if it had whole machine to itself, but still < 100 sec unless both *A* and *B* completely CPU-bound

Processes in the real world

- **Processes and parallelism have been a fact of life much longer than OSeS have been around**
 - E.g., say takes 1 worker 10 months to make 1 widget
 - Company may hire 100 workers to make 100 widgets
 - Latency for first widget \gg 1/10 month
 - Throughput may be $<$ 10 widgets per month (if can't perfectly parallelize task)
 - And 100 workers making 10,000 widgets may achieve $>$ 10 widgets/month (e.g., if workers never idly wait for paint to dry)
- **You will see these effects in you Pintos project group**
 - May block waiting for partner to complete task
 - Takes time to coordinate/explain/understand one another's code
 - Labs won't take 1/3 time with three people
 - But you will graduate faster than if you took only 1 class at a time

A process's view of the world

- **Each process has own view of machine**

- Its own address space
- Its own open files
- Its own virtual CPU (through preemptive multitasking)

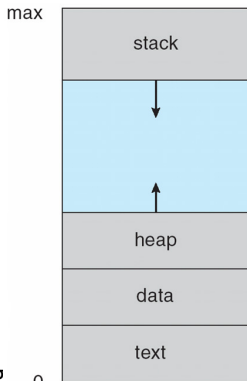
- `*(char *)0xc000` **different in P_1 & P_2**

- **Simplifies programming model**

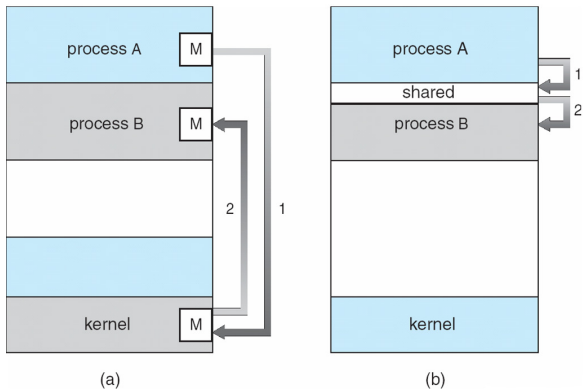
- `gcc` does not care that `firefox` is running 0

- **Sometimes want interaction between processes**

- Simplest is through files: `emacs` edits file, `gcc` compiles it
- More complicated: Shell/command, Window manager/app.



Inter-Process Communication



- **How can processes interact in real time?**
 - (a) By passing messages through the kernel
 - (b) By sharing a region of physical memory
 - (c) Through asynchronous signals or alerts

Outline

- 1 (UNIX-centric) User view of processes
- 2 Kernel view of processes
- 3 Threads
- 4 Thread implementation details

Creating processes

- **Original UNIX paper** is a great reference on core system calls
- `int fork (void);`
 - Create new process that is exact copy of current one
 - Returns *process ID* of new process in “parent”
 - Returns 0 in “child”
- `int waitpid (int pid, int *stat, int opt);`
 - `pid` – process to wait for, or -1 for any
 - `stat` – will contain exit value, or signal
 - `opt` – usually 0 or `WNOHANG`
 - Returns process ID or -1 on error

Deleting processes

- `void exit (int status);`
 - Current process ceases to exist
 - `status` shows up in `waitpid` (shifted)
 - By convention, `status` of 0 is success, non-zero error
- `int kill (int pid, int sig);`
 - Sends signal `sig` to process `pid`
 - `SIGTERM` most common value, kills process by default (but application can catch it for “cleanup”)
 - `SIGKILL` stronger, kills process always

Running programs

- `int execve (char *prog, char **argv, char **envp);`
 - `prog` – full pathname of program to run
 - `argv` – argument vector that gets passed to `main`
 - `envp` – environment variables, e.g., `PATH`, `HOME`
- **Generally called through a wrapper functions**
 - `int execvp (char *prog, char **argv);`
Search `PATH` for `prog`, use current environment
 - `int execlp (char *prog, char *arg, ...);`
List arguments one at a time, finish with `NULL`
- **Example:** `minish.c`
 - Loop that reads a command, then executes it
- **Warning:** Pintos `exec` more like combined `fork/exec`

```
pid_t pid; char **av;
void doexec () {
    execvp (av[0], av);
    perror (av[0]);
    exit (1);
}

/* ... main loop: */
for (;;) {
    parse_next_line_of_input (&av, stdin);
    switch (pid = fork ()) {
        case -1:
            perror ("fork"); break;
        case 0:
            doexec ();
        default:
            waitpid (pid, NULL, 0); break;
    }
}
```

Manipulating file descriptors

- `int dup2 (int oldfd, int newfd);`
 - Closes `newfd`, if it was a valid descriptor
 - Makes `newfd` an exact copy of `oldfd`
 - Two file descriptors will share same offset (`lseek` on one will affect both)
- `int fcntl (int fd, F_SETFD, int val)`
 - Sets *close on exec* flag if `val = 1`, clears if `val = 0`
 - Makes file descriptor non-inheritable by spawned programs
- **Example:** `redirsh.c`
 - Loop that reads a command and executes it
 - Recognizes `command < input > output 2> errlog`

```
void doexec (void) {
    int fd;
    if (infile) {      /* non-NULL for "command < infile" */
        if ((fd = open (infile, O_RDONLY)) < 0) {
            perror (infile);
            exit (1);
        }
        if (fd != 0) {
            dup2 (fd, 0);
            close (fd);
        }
    }

    /* ... do same for outfile→fd 1, errfile→fd 2 ... */

    execvp (av[0], av);
    perror (av[0]);
    exit (1);
}
```

Pipes

- `int pipe (int fds[2]);`
 - Returns two file descriptors in `fds[0]` and `fds[1]`
 - Data written to `fds[1]` will be returned by `read` on `fds[0]`
 - When last copy of `fds[1]` closed, `fds[0]` will return EOF
 - Returns 0 on success, -1 on error
- **Operations on pipes**
 - `read/write/close` – as with files
 - When `fds[1]` closed, `read(fds[0])` returns 0 bytes
 - When `fds[0]` closed, `write(fds[1])`:
 - ▷ Kills process with `SIGPIPE`
 - ▷ Or if signal ignored, fails with `EPIPE`
- **Example:** `pipesh.c`
 - Sets up pipeline `command1 | command2 | command3 ...`

pipesh.c (simplified)

```
void doexec (void) {
    while (outcmd) {
        int pipefds[2]; pipe (pipefds);
        switch (fork ()) {
            case -1:
                perror ("fork"); exit (1);
            case 0:
                dup2 (pipefds[1], 1);
                close (pipefds[0]); close (pipefds[1]);
                outcmd = NULL;
                break;
            default:
                dup2 (pipefds[0], 0);
                close (pipefds[0]); close (pipefds[1]);
                parse_command_line (&av, &outcmd, outcmd);
                break;
        }
    }
}
:
```

Why fork?

- **Most calls to `fork` followed by `execve`**
- **Could also combine into one *spawn* system call (like Pintos `exec`)**
- **Occasionally useful to fork one process**
 - Unix *dump* utility backs up file system to tape
 - If tape fills up, must restart at some logical point
 - Implemented by forking to revert to old state if tape ends
- **Real win is simplicity of interface**
 - Tons of things you might want to do to child: Manipulate file descriptors, set environment variables, reduce privileges, ...
 - Yet `fork` requires *no* arguments at all

Spawning a process without fork

- Without fork, needs tons of different options for new process
- Example: Windows `CreateProcess` system call
 - Also `CreateProcessAsUser`, `CreateProcessWithLogonW`, `CreateProcessWithTokenW`,...

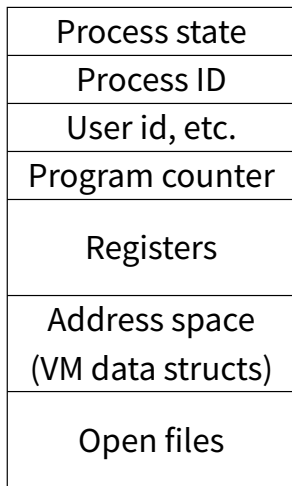
```
BOOL WINAPI CreateProcess(  
    _In_opt_      LPCTSTR lpApplicationName,  
    _Inout_opt_  LPTSTR lpCommandLine,  
    _In_opt_     LPSECURITY_ATTRIBUTES lpProcessAttributes,  
    _In_opt_     LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    _In_         BOOL bInheritHandles,  
    _In_         DWORD dwCreationFlags,  
    _In_opt_     LPVOID lpEnvironment,  
    _In_opt_     LPCTSTR lpCurrentDirectory,  
    _In_         LPSTARTUPINFO lpStartupInfo,  
    _Out_        LPPROCESS_INFORMATION lpProcessInformation  
);
```


Outline

- 1 (UNIX-centric) User view of processes
- 2 Kernel view of processes
- 3 Threads
- 4 Thread implementation details

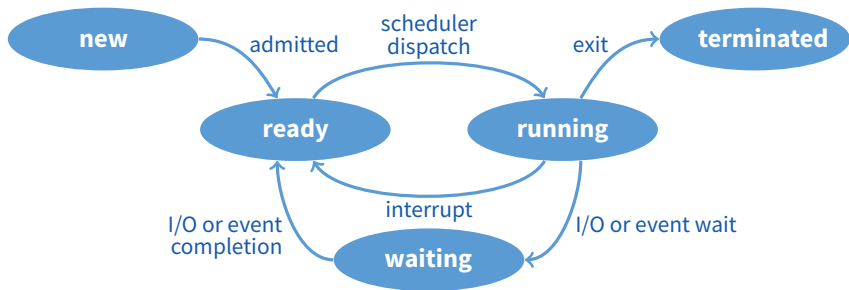
Implementing processes

- **Keep a data structure for each process**
 - Process Control Block (PCB)
 - Called `proc` in Unix, `task_struct` in Linux, and just `struct thread` in Pintos
- **Tracks *state* of the process**
 - Running, ready (runnable), waiting, etc.
- **Includes information necessary to run**
 - Registers, virtual memory mappings, etc.
 - Open files (including memory mapped files)
- **Various other data about the process**
 - Credentials (user/group ID), signal mask, controlling terminal, priority, accounting statistics, whether being debugged, which system call binary emulation in use, ...



PCB

Process states



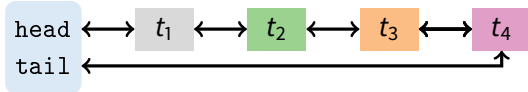
- **Process can be in one of several states**
 - *new* & *terminated* at beginning & end of life
 - *running* – currently executing (or will execute on kernel return)
 - *ready* – can run, but kernel has chosen different process to run
 - *waiting* – needs async event (e.g., disk operation) to proceed
- **Which process should kernel run?**
 - if 0 runnable, run idle loop (or halt CPU), 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 do better)
 - Divide into runnable and blocked processes

- **FIFO?**

- Put threads on back of list, pull them from front:



- Pintos does this—see `ready_list` in `thread.c`
- **Priority?**
 - Give some threads a better shot at the CPU

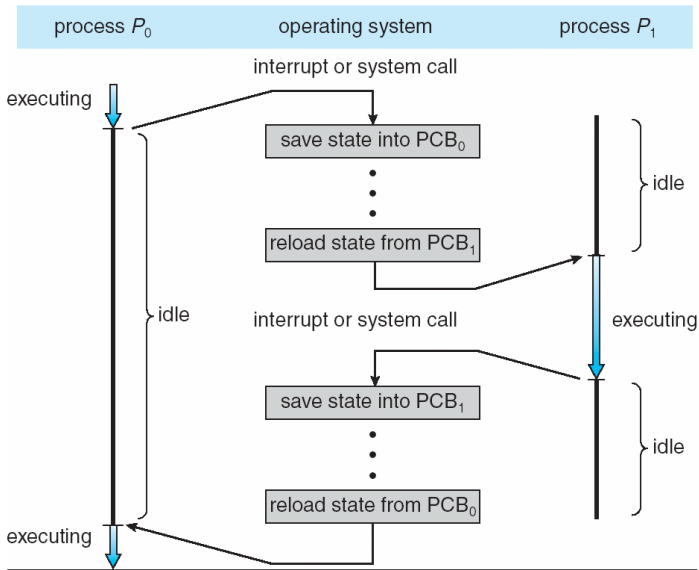
Scheduling policy

- **Want to balance multiple goals**
 - *Fairness* – don't starve processes
 - *Priority* – reflect relative importance of procs
 - *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 a whole lecture 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



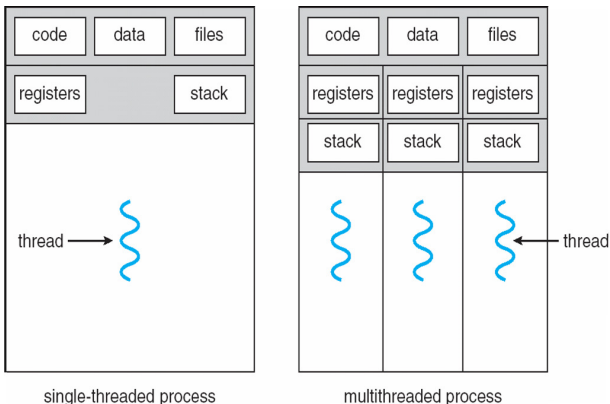
Context switch details

- **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)
 - ▶ HW Optimization 1: don't flush kernel's own data from TLB
 - ▶ HW Optimization 2: use tag to avoid flushing any data
 - Usually causes more cache misses (switch working sets)

Outline

- 1 (UNIX-centric) User view of processes
- 2 Kernel view of processes
- 3 **Threads**
- 4 Thread implementation details

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?

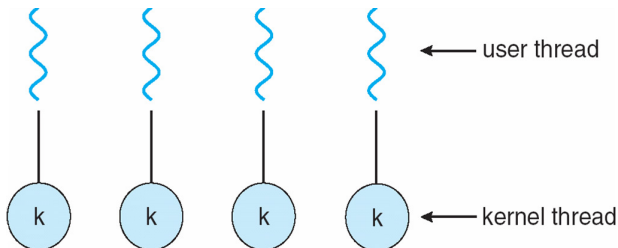
- **Most popular abstraction for concurrency**
 - Lighter-weight abstraction than processes
 - All threads in one process share memory, file descriptors, etc.
- **Allows one process to use multiple CPUs or cores**
- **Allows program to overlap I/O and computation**
 - Same benefit as OS running `emacs` & `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
 - Switch kernel threads when preempting 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 [in 3 weeks]**
- **See [Birell] for good introduction**
- **Can have preemptive or non-preemptive threads**
 - Preemptive causes more race conditions
 - Non-preemptive can't take advantage of multiple CPUs
 - Before prevalence of multicore, most kernels non-preemptive

Kernel threads

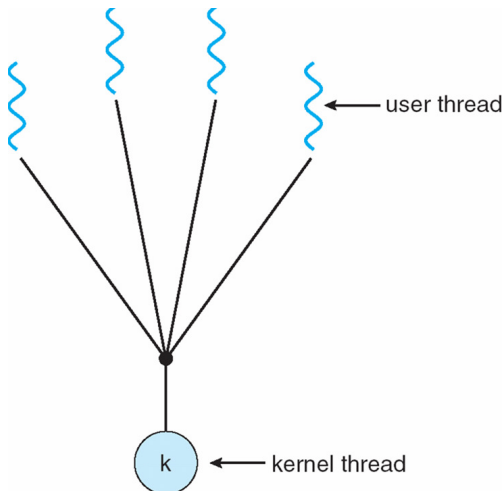


- **Can implement `thread_create` as a system call**
- **To add `thread_create` to an OS that doesn't have it:**
 - Start with process abstraction in kernel
 - `thread_create` like process creation with features stripped out
 - ▷ Keep same address space, file table, etc., in new process
 - ▷ `rfork/clone`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

Limitations of kernel-level threads

- **Every thread operation must go through kernel**
 - create, exit, join, synchronize, or switch for any reason
 - On my laptop: syscall takes 100 cycles, fn call 5 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

Alternative: User threads



- **Implement as user-level library (a.k.a. *green* threads)**
 - One kernel thread per process
 - `thread_create`, `thread_exit`, etc., just library functions

Implementing user-level threads

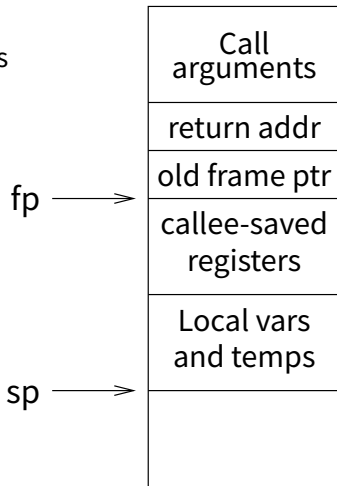
- **Allocate a new stack for each** `thread_create`
- **Keep a queue of runnable threads**
- **Replace networking system calls** (`read/write/etc.`)
 - If operation would block, switch and run different thread
- **Schedule periodic timer signal** (`setitimer`)
 - Switch to another thread on timer signals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

Outline

- 1 (UNIX-centric) User view of processes
- 2 Kernel view of processes
- 3 Threads
- 4 Thread implementation details

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 (on x86, %ebx, %esi, %edi, plus %ebp and %esp)
- **sp register always base of stack**
 - Frame pointer (*fp*) is old *sp*
- **Local variables stored in registers and on stack**
- **Function arguments go in caller-saved regs and on stack**
 - With 32-bit x86, all arguments on stack



Background: procedure calls

Procedure call

save active caller registers

call foo (pushes pc)

save needed callee registers

...do stuff...

restore callee saved registers

jump back to calling function

restore caller registers

- **Caller must save some state across function call**
 - Return address, caller-saved registers
- **Other state does not need to be saved**
 - Callee-saved regs, global variables, stack pointer

Pintos thread implementation

- **Pintos implements user processes on top of its own threads**
 - Same technique can be used to implement user-level threads, too
- **Per-thread state in thread control block structure**

```
struct thread {  
    ...  
    uint8_t *stack; /* Saved stack pointer. */  
    ...  
};  
uint32_t thread_stack_ofs = offsetof(struct thread, stack);
```

- **C declaration for asm thread-switch function:**
 - `struct thread *switch_threads (struct thread *cur, struct thread *next);`
- **Also thread initialization function to create new stack:**
 - `void thread_create (const char *name, thread_func *function, void *aux);`

i386 switch_threads

```
pushl %ebx; pushl %ebp           # Save callee-saved regs
pushl %esi; pushl %edi

mov thread_stack_ofs, %edx      # %edx = offset of stack field
                                #      in thread struct

movl 20(%esp), %eax             # %eax = cur
movl %esp, (%eax,%edx,1)        # cur->stack = %esp

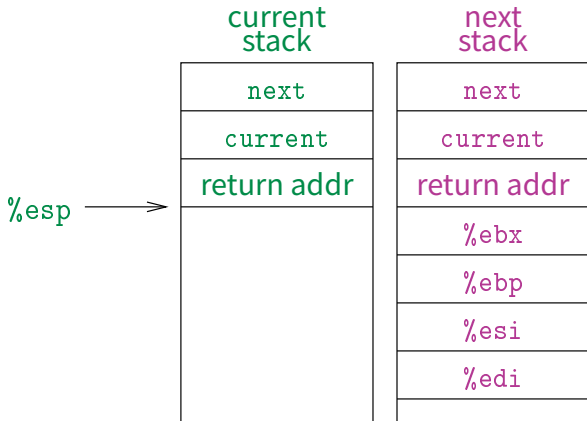
movl 24(%esp), %ecx            # %ecx = next
movl (%ecx,%edx,1), %esp        # %esp = next->stack

popl %edi; popl %esi           # Restore callee-saved regs
popl %ebp; popl %ebx

ret                              # Resume execution
```

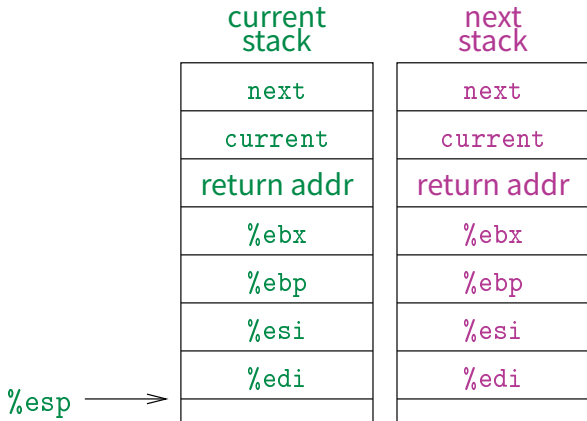
- **This is actual code from Pintos `switch.S` (slightly reformatted)**
 - See [Thread Switching](#) in documentation

i386 switch_threads



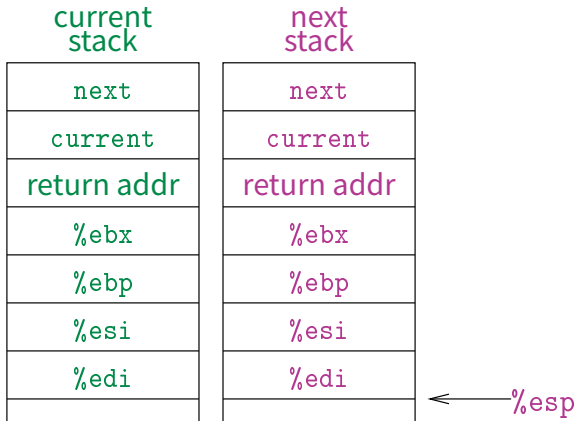
- This is actual code from Pintos `switch.S` (slightly reformatted)
 - See [Thread Switching](#) in documentation

i386 switch_threads



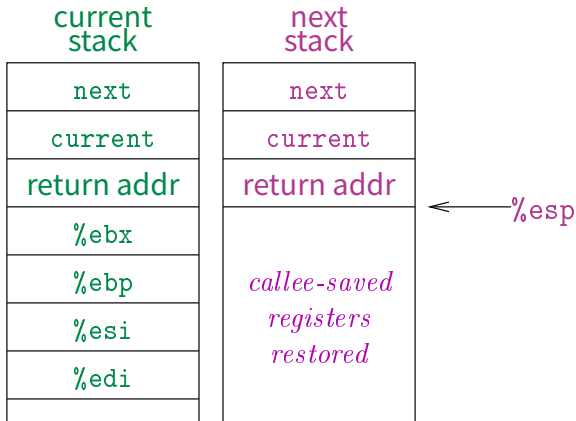
- This is actual code from Pintos `switch.S` (slightly reformatted)
 - See [Thread Switching](#) in documentation

i386 switch_threads



- This is actual code from Pintos `switch.S` (slightly reformatted)
 - See [Thread Switching](#) in documentation

i386 switch_threads

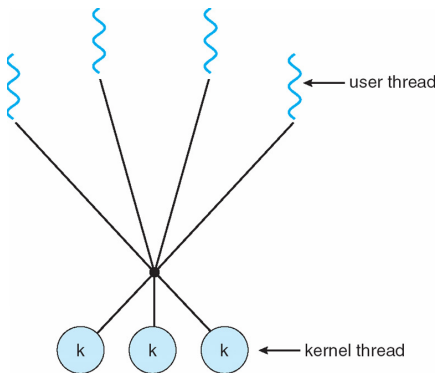


- This is actual code from Pintos `switch.S` (slightly reformatted)
 - See [Thread Switching](#) in documentation

Limitations of user-level threads

- **A user-level thread library can do the same thing as Pintos**
- **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 uncached 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 in future lectures.]

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 # kthreads 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)
- **Standard threads still 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
- **But concurrency greatly increases complexity**
 - More on that in concurrency, synchronization lectures...