

# CE 440 Introduction to Operating System

Lecture 4: Thread  
Fall 2025

Prof. Yigong Hu

BOSTON  
UNIVERSITY

Slides courtesy of Manuel Egele, Ryan Huang and Baris Kasikci

# Recap: Process Creation

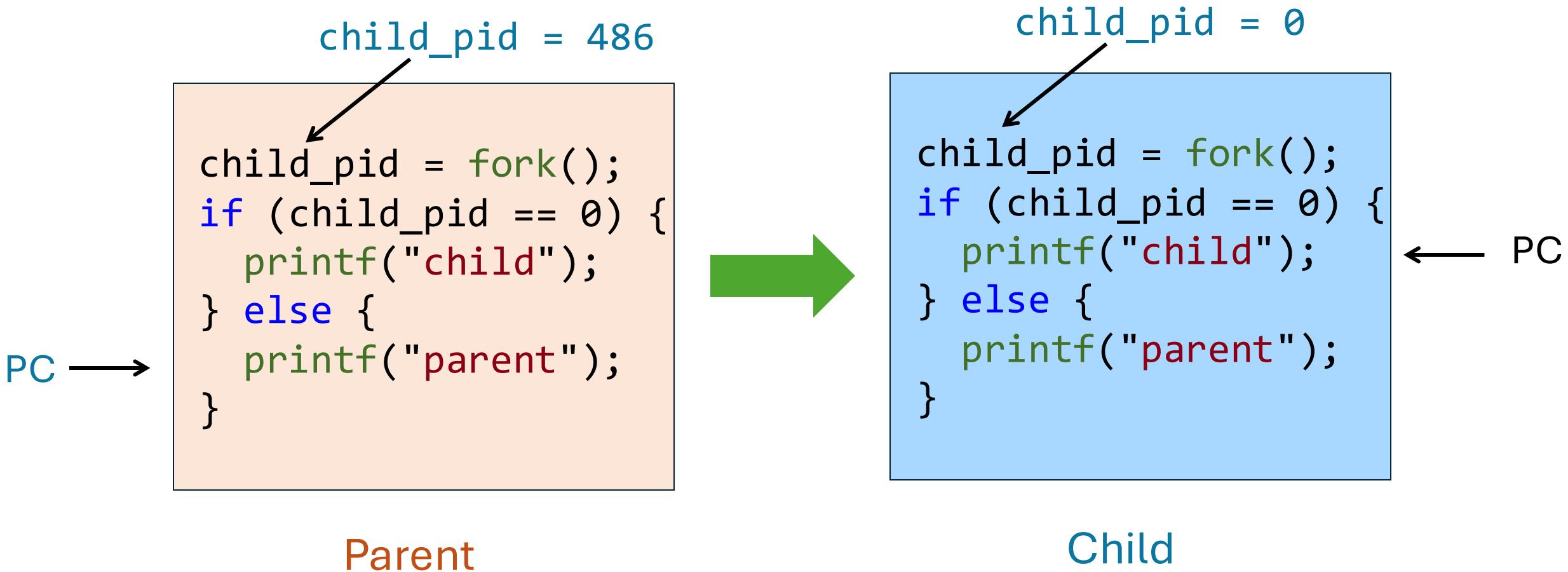
`int fork()`

1. Creates and initializes a new PCB
2. Creates a new address space
3. **Initializes the address space with a copy of the address space of the parent**
4. Initializes the kernel resources to point to the parent's resources (e.g., open files)
5. Places the PCB on the ready queue

**Fork returns twice**

- Huh?
- Returns the child's PID to the parent, "0" to the child

# Divergence



# Process Creation: Unix (2)

**Wait a second. How do we actually start a new program?**

```
int execv(char *prog, char *argv[])
int execve(const char *filename, char *const argv[], char *const envp[])
```

execv ()

1. Stops the current process
  2. Loads the program “prog” into the process’ address space
  3. Initializes hardware context and args for the new program
  4. Places the PCB onto the ready queue
- Note: It does not create a new process

**What does it mean for exec to return?**

**Warning: Pintos exec more like combined fork/exec**

# Why fork()?

**Most calls to fork followed by exec**

- could also combine into one `spawn` system call

**Very useful when the child...**

- Is cooperating with the parent
- Relies upon the parent's data to accomplish its task

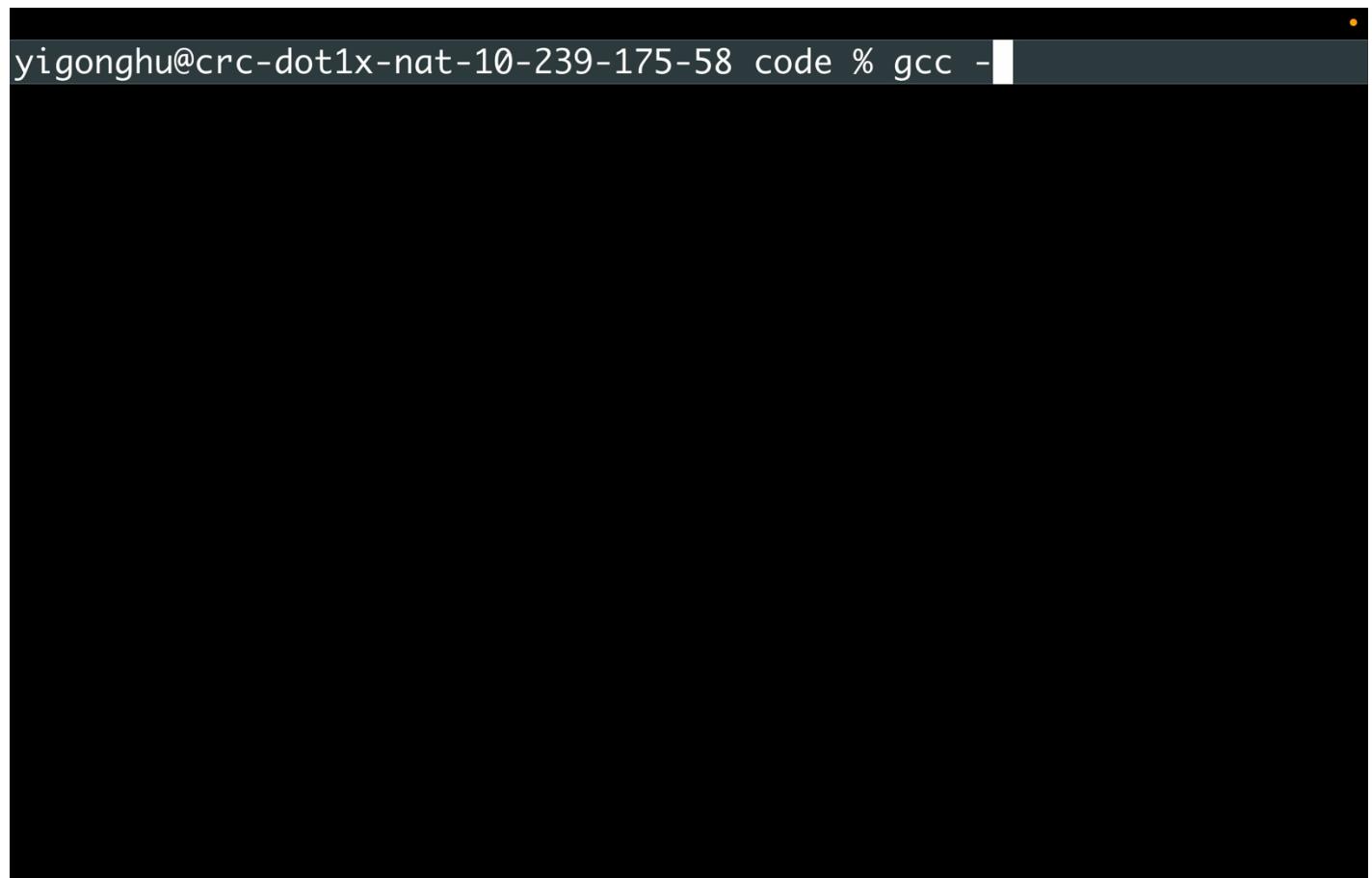
# Example: Web Server

```
while (1) {
    int sock = accept();
    if ((child_pid = fork()) == 0) {
        // Handle client request
    } else {
        // Close socket
    }
}
```

```
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,
        switch (pid = fork ()) {
            case -1:
                perror ("fork"); break;
            case 0:
                doexec ();
            default:
                waitpid (pid, NULL, 0);
                break;
        }
}
```

# Example: Shell

[https://yigonghu.github.io/\\_pages/ec440/fall25/code/minish.c](https://yigonghu.github.io/_pages/ec440/fall25/code/minish.c)



A terminal window showing the compilation of the C program minish.c using the gcc compiler. The command entered is "gcc -". The terminal prompt is "yigonghu@crc-dot1x-nat-10-239-175-58 code %". The window has a dark background and light-colored text.

# Why fork()?

**Most calls to fork followed by exec**

- could also combine into one **spawn** system call

**Very useful when the child...**

- Is cooperating with the parent
- Relies upon the parent's data to accomplish its task

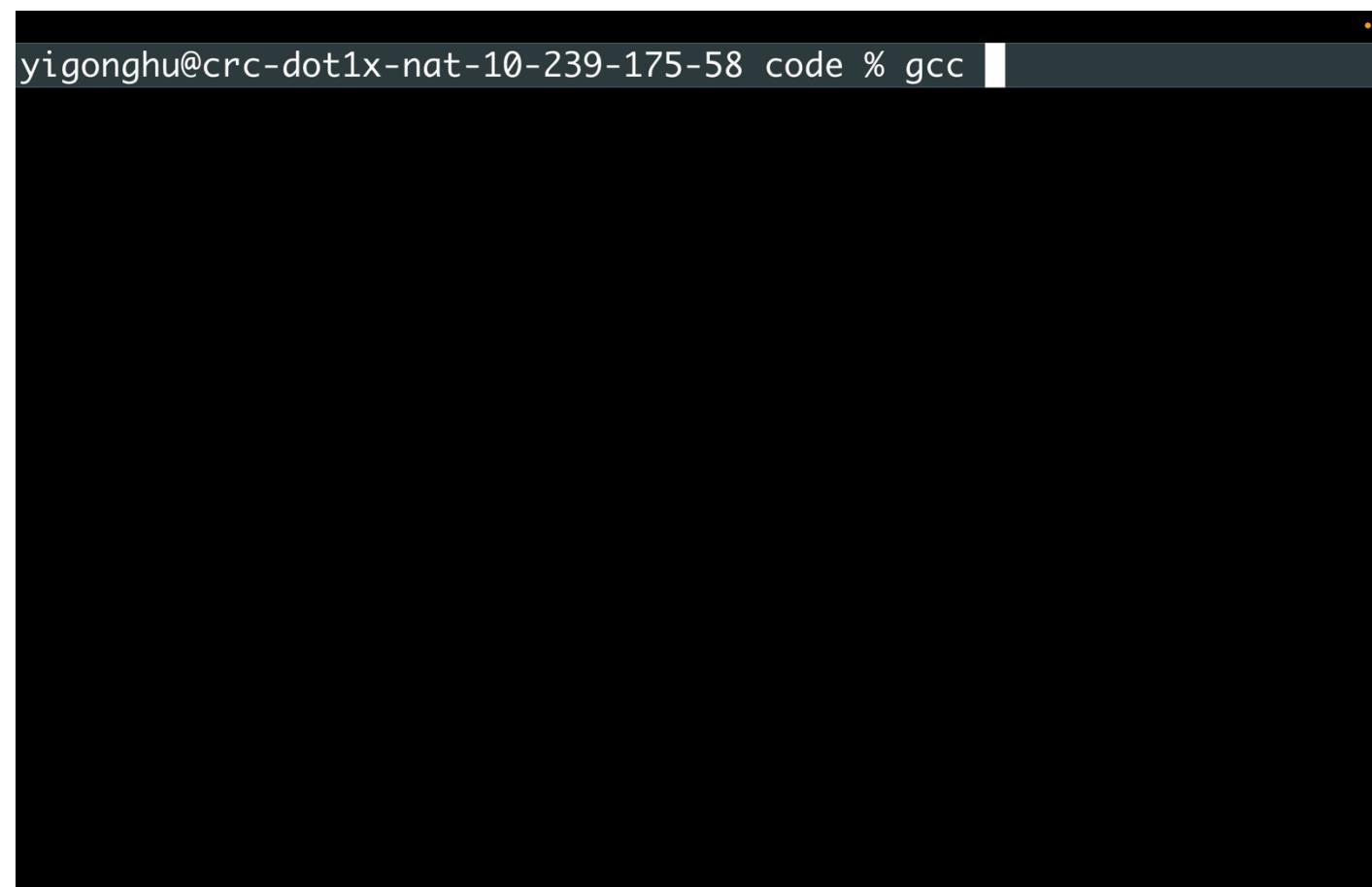
**Real win is simplicity of interface**

- 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

# Example: redirect

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

[https://yigonghu.github.io/\\_pages/ec440/fall25/code/redirsh.c](https://yigonghu.github.io/_pages/ec440/fall25/code/redirsh.c)



A terminal window showing the compilation of a C program named redirsh.c using the gcc compiler. The command entered is "gcc redirsh.c". The output shows the compilation process, including preprocessing, compilation, assembly generation, and linking. The assembly code is displayed in a monospaced font.

```
.  
yigonghu@crc-dot1x-nat-10-239-175-58 code % gcc redirsh.c  
yigonghu@crc-dot1x-nat-10-239-175-58 code % gcc -fno-pie -no-pie -o redirsh redirsh.c  
yigonghu@crc-dot1x-nat-10-239-175-58 code % ./redirsh  
yigonghu@crc-dot1x-nat-10-239-175-58 code %
```

# 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
);
```

# Questions

**Why Windows use CreateProcess while Unix uses fork/exec?**

- different OS design philosophy

**What happens if you run “exec csh” in your shell?**

**What happens if you run “exec ls” in your shell? Try it.**

**fork() can return an error. Why might this happen?**

# Process Termination

**All good processes must come to an end. But how?**

- Unix: `exit(int status)`, Windows: `ExitProcess(int status)`

**Essentially, free resources and terminate**

1. Terminate all threads (next lecture)
2. Close open files, network connections
3. Allocated memory (and VM pages out on disk)
4. Remove PCB from kernel data structures, delete

**Note that a process does *not* need to clean up itself**

- Why does the OS have to do it?

# **wait() a second...**

**Often it is convenient to pause until a child process has finished**

- Think of executing commands in a shell

**Unix `wait(int *wstatus)` (Windows: `WaitForSingleObject`)**

- Suspends the current process until *any* child process ends
- `waitpid()` suspends until the specified child process ends

**wait() has a return value...what is it?**

**Unix: Every process must be “reaped” by a parent**

- What happens if a parent process exits before a child?
- What do you think a “zombie” process is?

# **Problem with Process**

## **Creating a new process is costly**

- all of the data structures that must be allocated and initialized
- recall struct proc in Solaris

## **Communicating between processes is also costly**

- because most communication goes through the OS
- overhead of system calls and copying data

# Problem with fork()

**forks off copies of itself**

**To execute these programs we need to**

- Create several processes that execute in parallel
- Cause each to map to the same address space to share data
- They are all part of the same computation
- Have the OS schedule these processes in parallel (logically or physically)

**This situation is very inefficient**

- **Space:** PCB, page tables, etc.
- **Time:** create data structures, fork and copy addr space, etc.

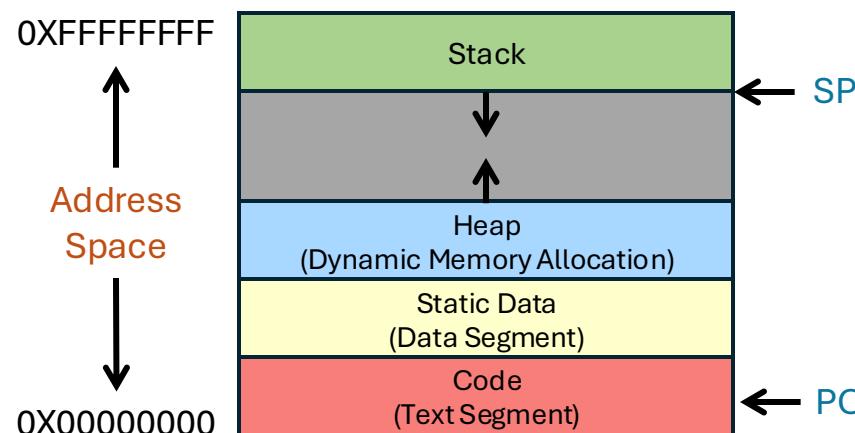
# Rethinking Process

**What is similar in these cooperating processes?**

- They all share the same code and data (address space)
- They all share the same privileges
- They all share the same resources (files, sockets, etc.)

**What don't they share?**

- Each has its own execution state: PC, SP, and registers



# Rethinking Process

**Idea: Why not separate the process concept from its execution state?**

- **Process:** address space, privileges, resources, etc.
- **Execution state:** PC, SP, registers

**Exec state also called **thread of control**, or **thread****

# Threads

**Modern OSes separate the concepts of processes and threads**

- The **thread** defines a sequential execution stream within a process (PC, SP, registers)
- The **process** defines the address space and general process attributes

**A thread is bound to a single process**

- Processes, however, can have multiple threads

**Threads become the unit of scheduling**

- Processes are now the **containers** in which threads execute
- Processes become static, threads are the dynamic entities

**Data structure: *Thread Control Block (TCB)***

# Small and Fast...

## Pintos thread class

```
struct thread {
    tid_t tid; /* Thread identifier. */
    enum thread_status status; /* Thread state. */
    char name[16]; /* Name (for debugging purposes). */
    uint8_t *stack; /* Saved stack pointer. */
    int priority; /* Priority. */
    struct list_elem allelem; /* List element for all
threads list. */
    struct list_elem elem; /* List element. */
    unsigned magic; /* Detects stack overflow. */
};
```

### Struct proc (Solaris)

```
/* One structure allocated per active process. It contains all
* data needed about the process while the process may be swapped
* out. Other per-process data (user.h) is also inside the proc structure.
* Lightweight-process data (lp.h) and the kernel stack may be swapped out.
*/
typedef struct proc {
    /* Fields requiring no explicit locking
     */
    struct vnode *p_vnode; /* pointer to v-node mode */
    struct vnode *p_vnlock; /* pointer to process specific address space pointer */
    struct plock *p_lock; /* ptr to process structure's mutex lock */
    kbaudx_t p_crlimit; /* lock for p_cred */
    struct cred *p_cred; /* process credentials */
    /* Fields protected by pidlock
     */
    int p_swapped; /* number of swapped out lugs */
    char p_stat; /* status of process */
    char p_crstate; /* current state */
    uchar_t p_pidle; /* flags protected only by pidlock */
    int p_wstatus; /* current wait return value */
    pid_t p_lwpid; /* forward link */
    struct proc *p_link; /* forward link to next process */
    struct proc *p_child; /* ptr to first child process */
    struct proc *p_sibling; /* ptr to sibling on same chain */
    struct proc *p_siblings; /* ptr to siblings with new state */
    struct proc *p_next; /* active child link next */
    struct proc *p_nextchild; /* active child link next */
    struct proc *p_nearestkin; /* gets accounting info at exit */
    struct proc *p_nearestch; /* */
    /* Fields protected by pidlock
     */
    /* Fields defined below */
    clock_t p_utime; /* user time, this process */
    clock_t p_stime; /* system time, this process */
    clock_t p_cutime; /* sun of children's user time */
    clock_t p_cstime; /* sun of children's system time */
    caddr_t p_rbase; /* base address of heap */
    size_t p_rbytesize; /* heap size in bytes */
    /* Per process signal stuff.
     */
    k_sigset_t p_sig; /* signals pending to this process */
    k_sigset_t p_iprice; /* ignore when generated */
    k_sigset_t p_iprime; /* get signal info with signal */
    struct sigqueue *p_ipq; /* queue of pending signals */
    struct sigqueue *p_ipqhigh; /* hrd to sigqueue structure pool */
    struct sigqueue *p_ipqlow; /* hrd to sigqueue structure pool */
    uchar_t p_stopsig; /* jedoncontrol stop cycle */
}
```

### Struct proc (Solaris) (2)

```
* Special per-process flag when set will fix misaligned memory
* references.
*/
char p_fixalignment;
/* Per process lwp and kernel thread stuff
*/
id_t p_lwpid; /* most recently allocated lwpid */
int p_lwpcnt; /* number of lugs in the process */
int p_lwpmax; /* max number of lugs */
int p_lwpwait; /* number of lugs in lwp.wait */
int p_zombcnt; /* number of zombie lugs */
int p_zomb_max; /* number of entries in p_zomb_tid */
id_t p_zomb_id; /* array of lwpids */
kthread_t *p_list; /* circular list of threads */
/* /proc (process filerelay) debugger interface stuff
*/
k_sigset_t p_ipqhigh; /* highest priority pending (ipq) */
k_sigset_t p_ipqlow; /* mask of traced faults (ipq) */
struct vnode *p_vroot; /* pointer to primary vnode node */
struct vnode *p_vroot; /* list of vnode nodes for process */
kthread_t *p_lwp; /* thread ptr for /proc agent lwp */
struct watched_area *p_bareas; /* watched areas */
along_t p_nareas; /* number of watched areas */
struct watched_page *p_npaged; /* remembered watched pages (vroot) */
int p_npaged; /* number of pages in watched areas */
int p_napages; /* number of active pr_npaged() */
struct proc *p_link; /* linked list for server */
kbaudx_t p_arwshas_cv; /* process stack size in bytes */
size_t p_arwshas; /* process stack size in bytes */

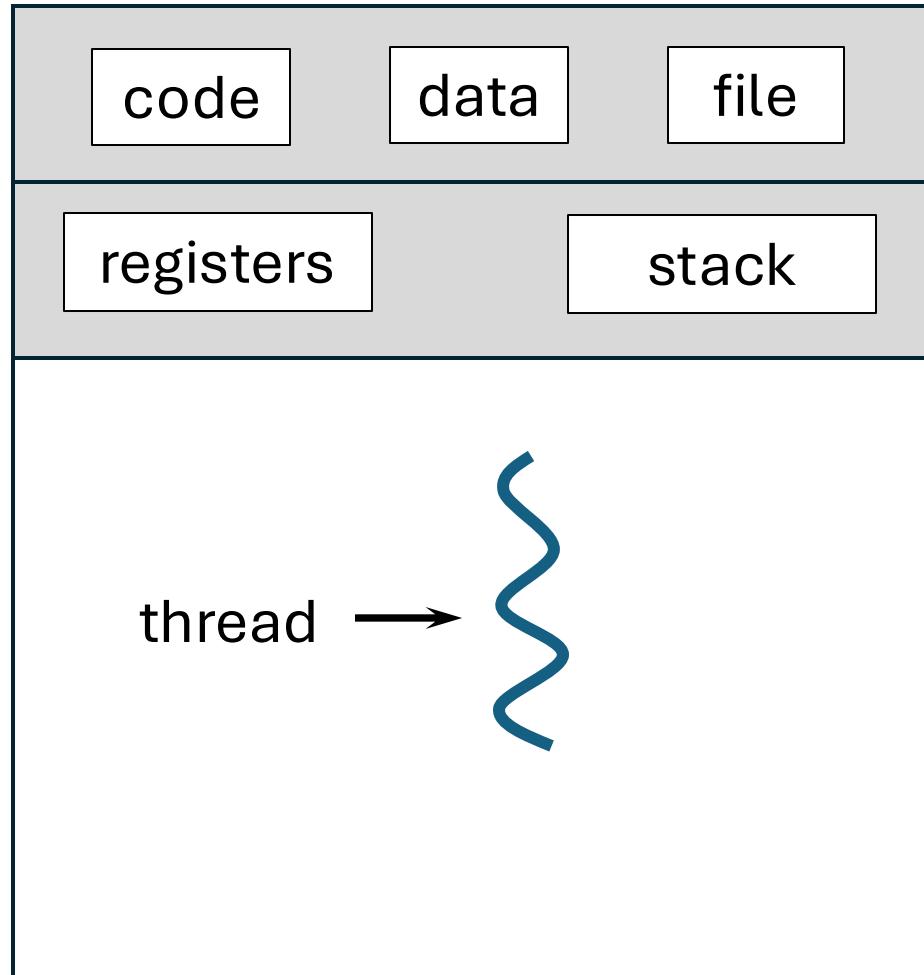
```

### Struct proc (Solaris) (3)

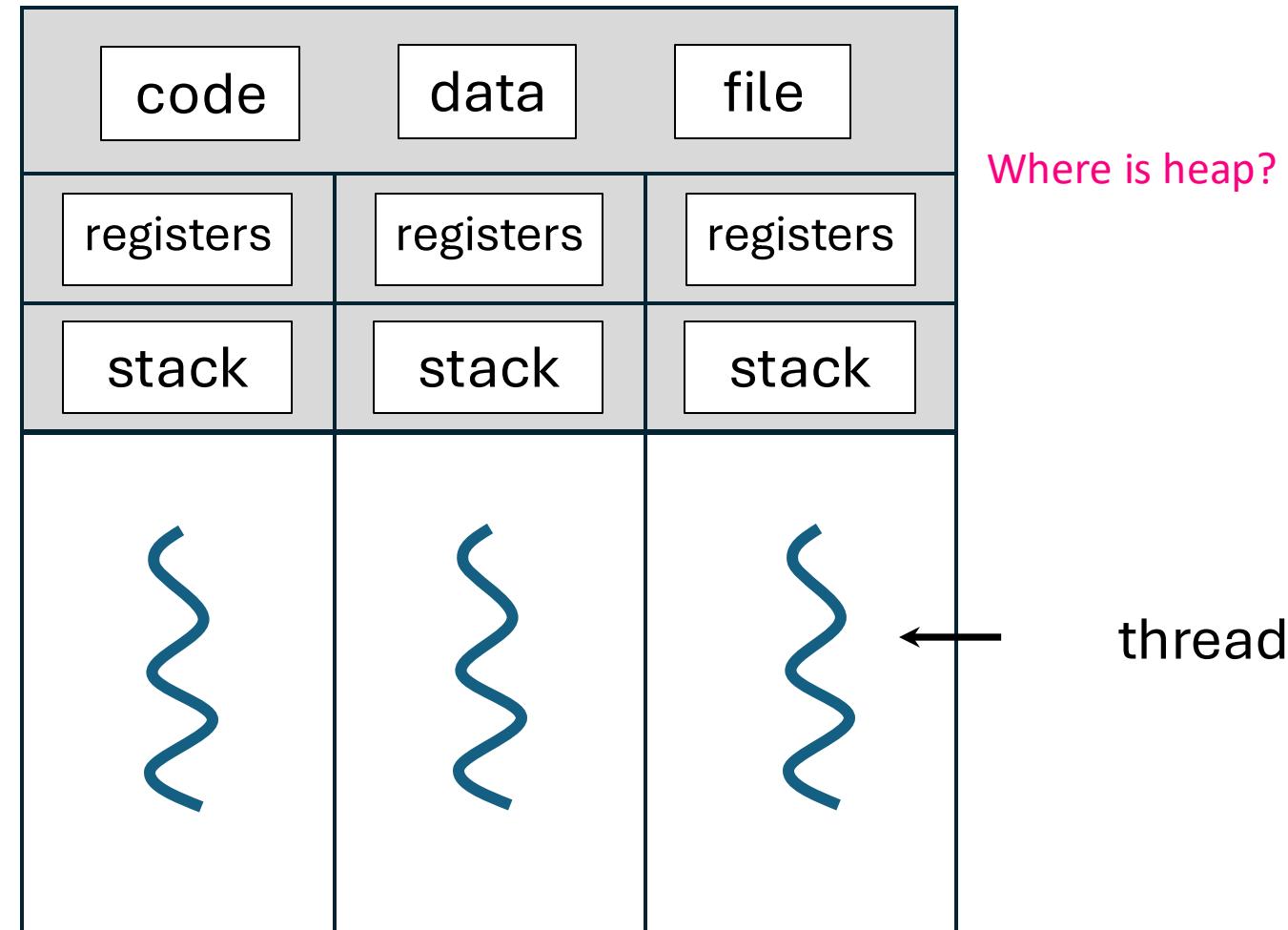
```
/*
 * Kernel probes
 */
uchar_t p_tif_flags;
/* C2 Security (C2_AUDIT)
 */
caddr_t p_audit_data; /* per process audit structure */
kthread_t *p_aslvpptp; /* thread ptr representing "aslvp" */
defined(_I386) || defined(_IA32) || defined(_IA64)
/* LWP support.
 */
maxx_t p_ldlock; /* protects the following fields */
struct seg_desc *p_ldt; /* Pointer to private LDT */
struct seg_desc p_ldt_desc; /* segment descriptor for private LDT */
int p_ldtindex; /* highest selector used */
f
size_t p_pvers; /* page version size before last swap */
struct alldesc *p_ldt; /* pointer to aligned LDT struct */
struct timer *p_gtimer; /* interval timer */
k_sigset_t p_notifications; /* signals in notification set */
kbaudx_t p_notifications; /* notify ev to synchronize with aslvp */
kbaudx_t p_notifications; /* */
uchar_t p_ldtindex; /* highest selector used */
wint_t p_sc_unblocked; /* number of unblocked threads */
struct vnode *p_sc_door; /* scheduler activations door */
caddr_t p_sc_door; /* */
wint_t p_sc_stropts; /* stack memory protection */
wint_t p_model; /* data model determined at exec time */
struct lwpchan_data *p_lwp; /* lwpchan cache */
/* */

/* protects unmapping and initialisation of robust locks.
 */
kbaudx_t p_top_munmaplock;
uchar_t p_top_munmaplock; /* pointer to user trap handlers */
uchar_t p_corefile; /* pattern for core file */
#endif defined(_IA64)
caddr_t p_upstack; /* base of the upward-growing stack */
size_t p_upstacksize; /* size of that stack, in bytes */
uchar_t p_las; /* which instruction set is utilized */
#endif
void *p_rce; /* resource control extension data */
struct task *p_task; /* our containing task */
struct task *p_taskp; /* tasks in task */
struct proc *p_tasknext; /* ptr to next process in task */
int p_lwpcount; /* number of TP_DNINH lugs */
int p_lwpexit; /* number of lugs in lwp.wait */
kthread_t *p_ac_data; /* ac (lwpid) lookup hash table */
p_ac_data; /* available esemantics structures */
)
```

# Threads in a Process

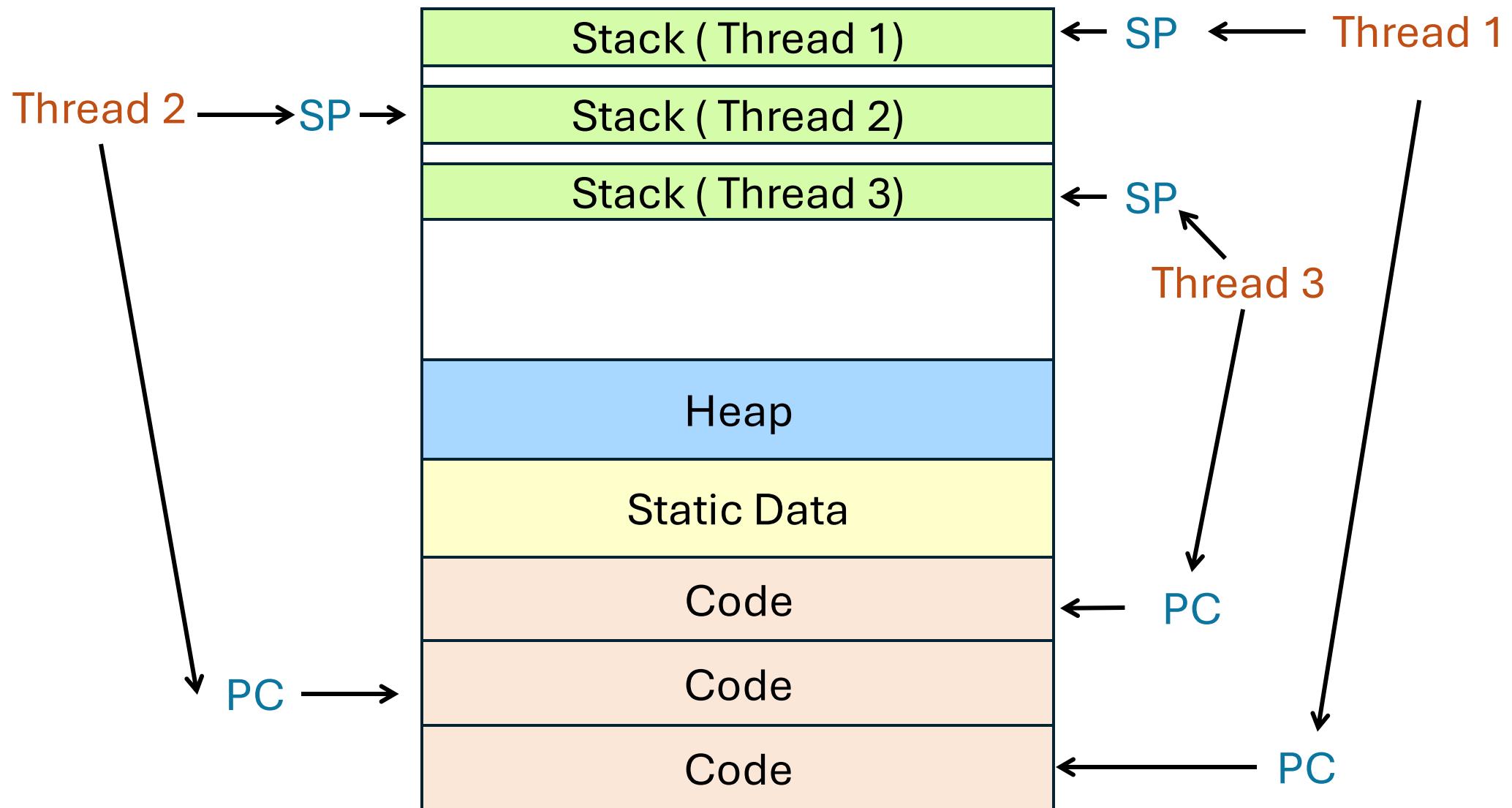


Single-threaded process



Multithreaded process

# Thread address space



# **Process/Thread Separation**

**Easier to support multithreaded applications**

- Concurrency does not require creating new processes

**Concurrency (multithreading) can be very useful**

- Improving program structure
- Allowing one process to use multiple CPUs/cores
- Handling concurrent events (e.g., Web requests)
- Allowing program to overlap I/O and computation

**So multithreading is even useful on a uniprocessor**

- Although today even cell phones are multicore

**But, brings a whole new meaning to Spaghetti Code**

- Forcing OS students to learn about synchronization...

# Recall fork

**fork() to create new processes to handle requests is overkill**

**Recall our forking Web server:**

```
while (1) {
    int sock = accept();
    if ((child_pid = fork()) == 0) {
        // Handle client request
        // Close socket and exit
    } else {
        // Close socket
    }
}
```

# Threads: Concurrent Servers

Instead, we can create a new thread for each request

```
web_server() {  
    while (1) {  
        int sock = accept();  
        thread_fork(handle_request, sock);  
    }  
}  
  
handle_request(int sock) {  
    Process request  
    close(sock);  
}
```

# Thread Primitives

```
tid thread_create (void (*fn) (void *), void *);
```

- Create a new thread, run fn with arg
- Allocate Thread Control Block (TCB)
- Allocate stack
- Build stack frame for base of stack
- Put func, args on stack
- Put thread on ready list

```
void thread_exit ();
```

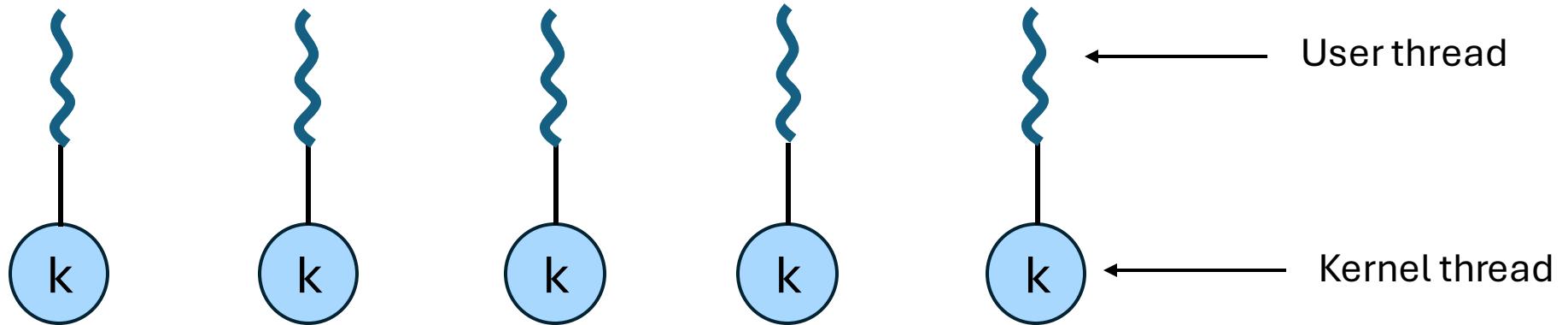
- Destroy current thread

```
void thread_join (tid thread);
```

- Wait for thread thread to exit

# Thread Implementation

Threads can be implemented in kernel



The OS schedules all the threads in the system

Also known as **lightweight processes**

- Windows: [threads](#)
- Solaris: [lightweight processes \(LWP\)](#)
- POSIX Threads (pthreads): [PTHREAD\\_SCOPE\\_SYSTEM](#)

# Limitations of Kernel Thread

## Every thread operation must go through kernel

- create, exit, join, synchronize, or switch for any reason
- On my laptop: syscall takes 100 cycles, function 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

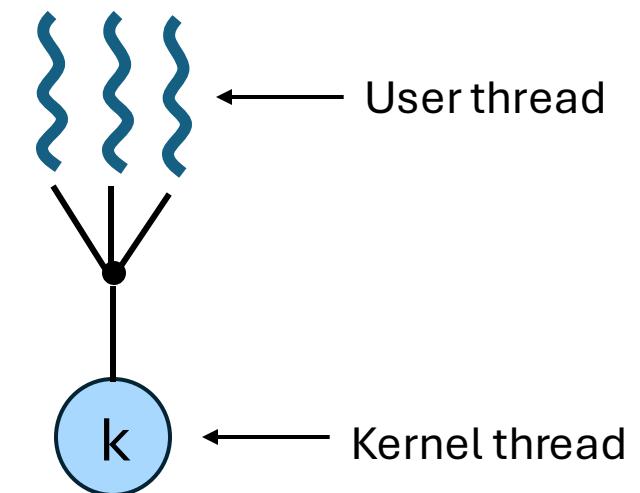
# Alternative: User-Level Threads

**Implement as user-level library (a.k.a. green threads)**

- One kernel thread per process
- `thread_create`, `thread_exit`, etc., just **library functions**
- library does thread context switch

**User-level threads are small and fast**

- pthreads: `PTHREAD_SCOPE_PROCESS`
- Java: `Thread`



# Limitation of User-level Threads

Can't take advantage of multiple CPUs or cores

User-level threads are **invisible** to the OS

- They are not well integrated with the OS

As a result, the OS can make poor decisions

- Scheduling a process with idle threads
- A blocking system call (e.g., disk read) blocks all threads
  - Even if the process has other threads that can execute
- Unscheduling a process with a thread holding a lock

How to solve this?

# Kernel and User Threads

## Use both kernel and user-level threads

- Can associate a user-level thread with a kernel-level thread
- Or, multiplex user-level threads on top of kernel-level threads

### Kernel-level threads

- Integrated with OS (informed scheduling)
- Slower to create, manipulate, synchronize

### User-level threads

- Faster to create, manipulate, synchronize
- Not integrated with OS (uninformed scheduling)

# Use Case: Java Virtual Machine

## Java Virtual Machine (JVM) (also C#, others)

- Java threads are user-level threads
- On older Unix, only one “kernel thread” per process
  - Multiplex all Java threads on this one kernel thread
- On modern OSes
  - Can multiplex Java threads on multiple kernel threads
  - Can have more Java threads than kernel threads
  - Why?

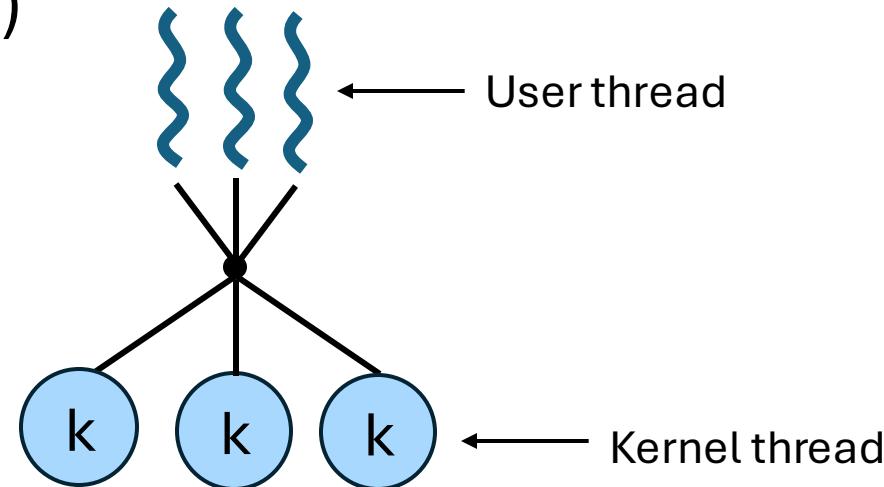
# User Threads on Kernel Threads

## User threads implemented on kernel threads

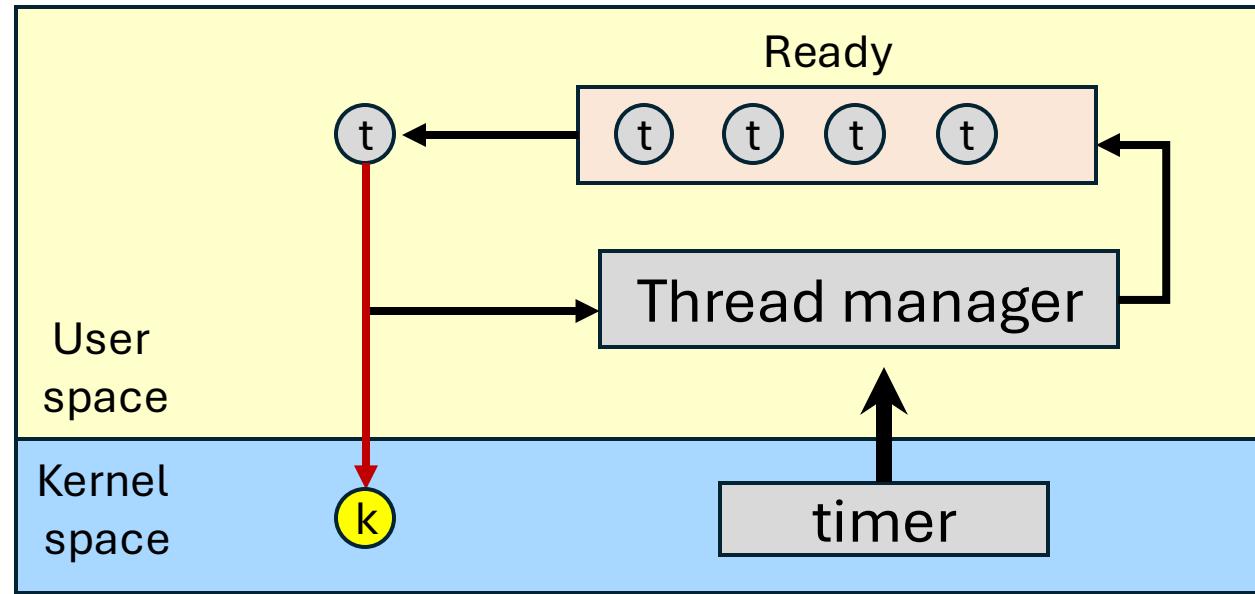
- Multiple kernel-level threads per process
- `thread_create`, `thread_exit` still library functions as before

## Sometimes called **n : m** threading

- Have n user threads per m kernel threads (Simple user-level threads are  $n : 1$ , kernel threads  $1 : 1$ )



# Implementing User-Level Threads



**Allocate a new stack for each thread\_create**

**Keep a queue of runnable threads**

**Schedule periodic timer signal (setitimer)**

- Switch to another thread on timer signals (preemption)

**Replace blocking system calls (read/write) to non-blocking calls**

- If operation would block, switch and run different thread

# Implementing User-Level Threads

The thread scheduler determines when a thread runs

It uses queues to keep track of what threads are doing

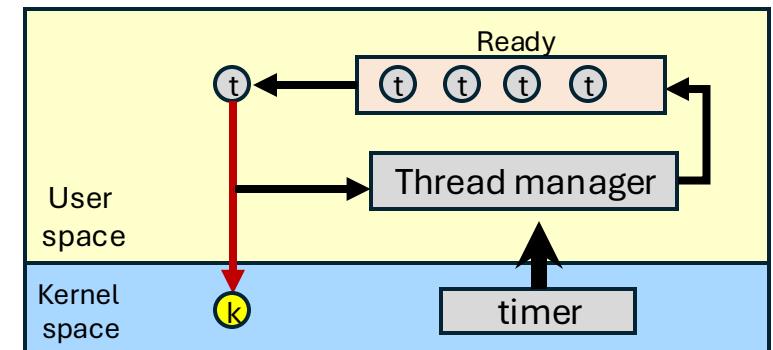
- Just like the OS and processes
- But it is implemented at user-level in a library

Run queue: Threads currently running (usually one)

Ready queue: Threads ready to run

Are there wait queues?

- How might you implement sleep(time)?



# Non-preemptive Thread Scheduling

Threads voluntarily give up the CPU with yield

Ping thread

```
While (1) {  
    printf("ping\n");  
    yield();  
}
```

Pong Thread

```
While (1) {  
    printf("pong\n");  
    yield();  
}
```

What is the output of running these two threads?

# `yield()`

**Wait a second. How does `yield()` work?**

**It gives up the CPU to another thread**

- In other words, it **context switches** to another thread

**So what does it mean for `yield` to return?**

- It means that *another thread* called `yield`!

**Execution trace of ping/pong**

- `printf("ping\n");`
- `yield();`
- `printf("pong\n");`
- `yield();`
- ...

# Preemptive Thread Scheduling

**Non-preemptive threads have to voluntarily give up CPU**

- A long-running thread will take over the machine
- Only voluntary calls to yield, sleep, or finish cause a context switch

**Preemptive scheduling causes an involuntary context switch**

- Need to regain control of processor asynchronously
- Use timer interrupt
- Timer interrupt handler forces current thread to “call” yield

# Thread Context Switch

**The context switch routine does all of the magic**

- Saves context of the currently running thread (old\_thread)
  - Push all machine state onto its stack
- Restores context of the next thread
  - Pop all machine state from the next thread's stack
- The next thread becomes the current thread
- Return to caller as new thread

**This is all done in assembly language**

- It works at the level of the procedure calling convention, so it cannot be implemented using procedure calls

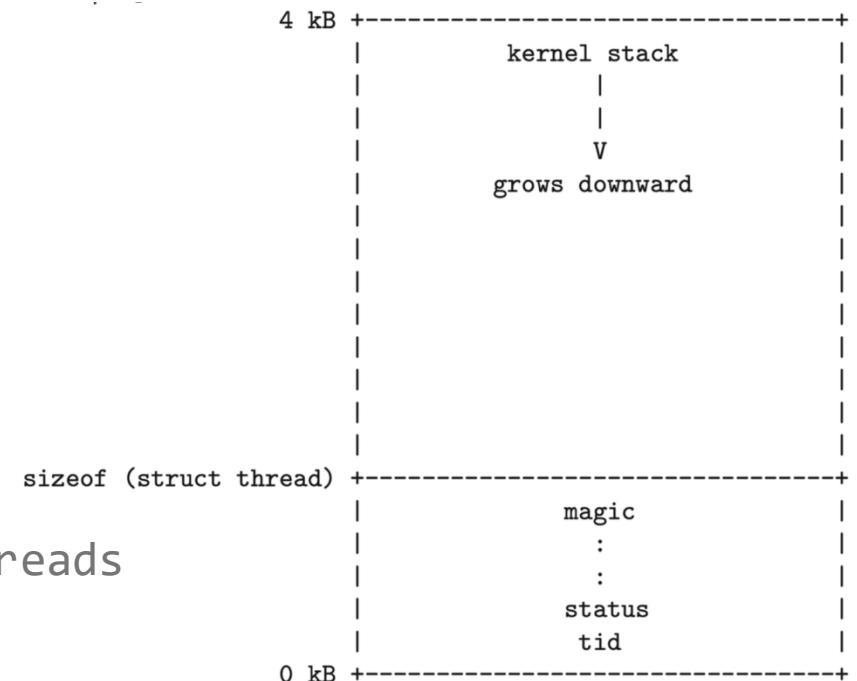
# Pintos switch\_threads

C declaration for thread-switch function:

- struct thread \*switch\_threads (struct thread \*cur,  
struct thread \*next);

Recall: Thread control block structure

```
struct thread {  
    tid_t tid; /* Thread identifier. */  
    enum thread_status status; /* Thread state. */  
    char name[16]; /* Name (for debugging purposes). */  
    uint8_t *stack; /* Saved stack pointer. */  
    int priority; /* Priority. */  
    struct list_elem allelem; /* List element for all threads  
list. */  
    struct list_elem elem; /* List element. */  
    unsigned magic; /* Detects stack overflow. */  
};  
uint32_t thread_stack_ofs = offsetof(struct thread, stack);
```



# Pintos switch\_threads implementation

Pintos's implements switch\_thread in i386 assembly

```
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
```

# Calling Conventions

## Calling Conventions is

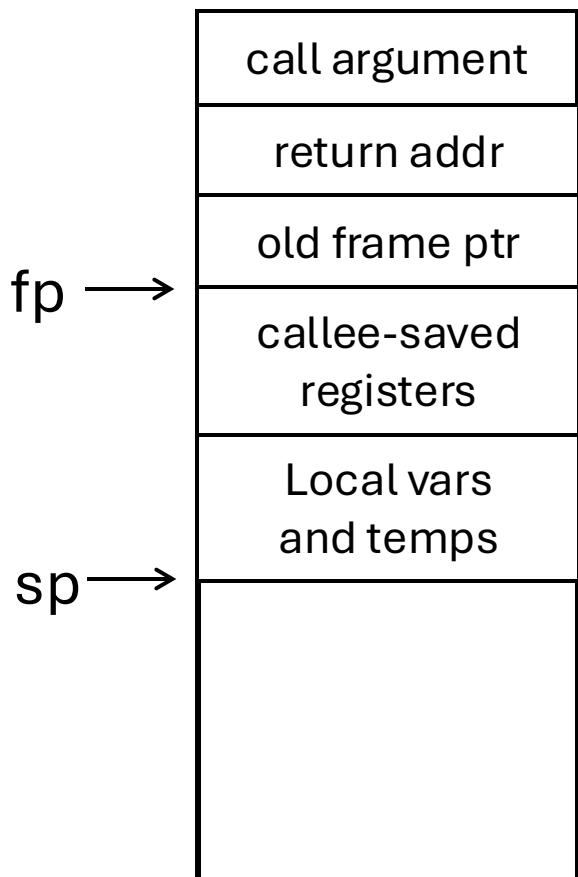
- a standard on how functions should be called by the *machine*
- how a function call in C/C++ gets converted into assembly language
- Compilers need to obey this standard in compiling code into assembly

## Use case

- A program calls functions across many object files and libraries
- For these codes to be interfaced together, we need a standardization for calls

# Calling Conventions

## x86 calling convention stack setup



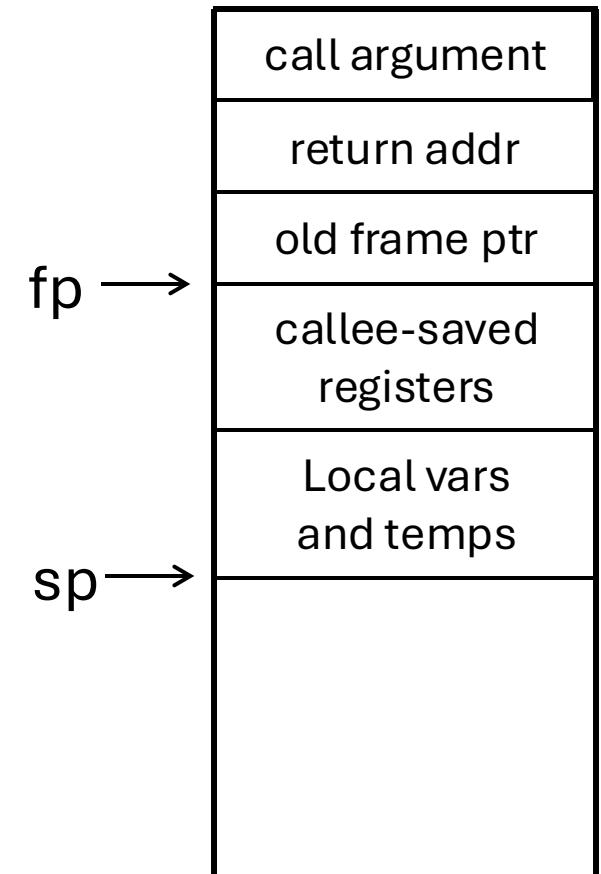
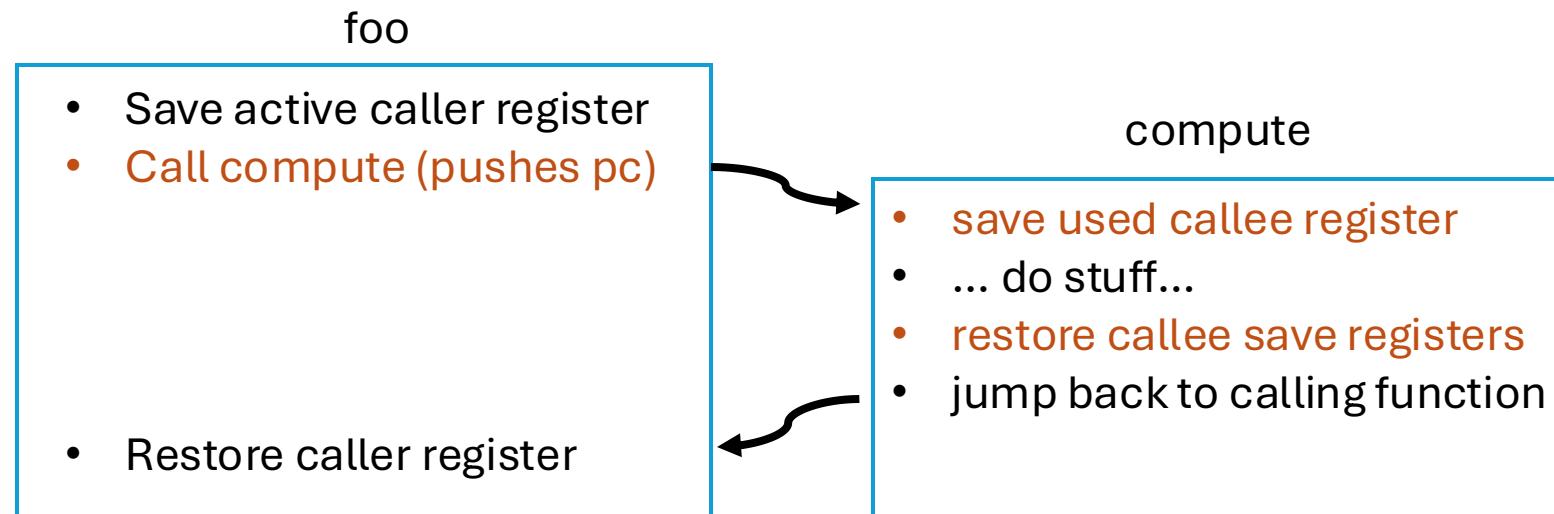
```
int compute(int a, int b)
{
    int i, result;
    result = 0;
    for (i = 0; i < a; i++)
        result = result + b - i;
    return result;
}

void foo()
{
    int x, y, z;
    x = 3;
    y = 5;
    z = compute(x, y);
    printf("compute(%d, %d)=%d\n", x, y, z);
}
```

# Calling Conventions

## Registers divided into 2 groups

- **caller-saved** regs: callee function free to modify
  - on x86, %eax [return val], %edx, & %ecx
- **callee-saved** regs: callee function must restore to original value upon return
  - on x86, %ebx, %esi, %edi, plus %ebp and %esp



# Pintos switch\_threads implementation

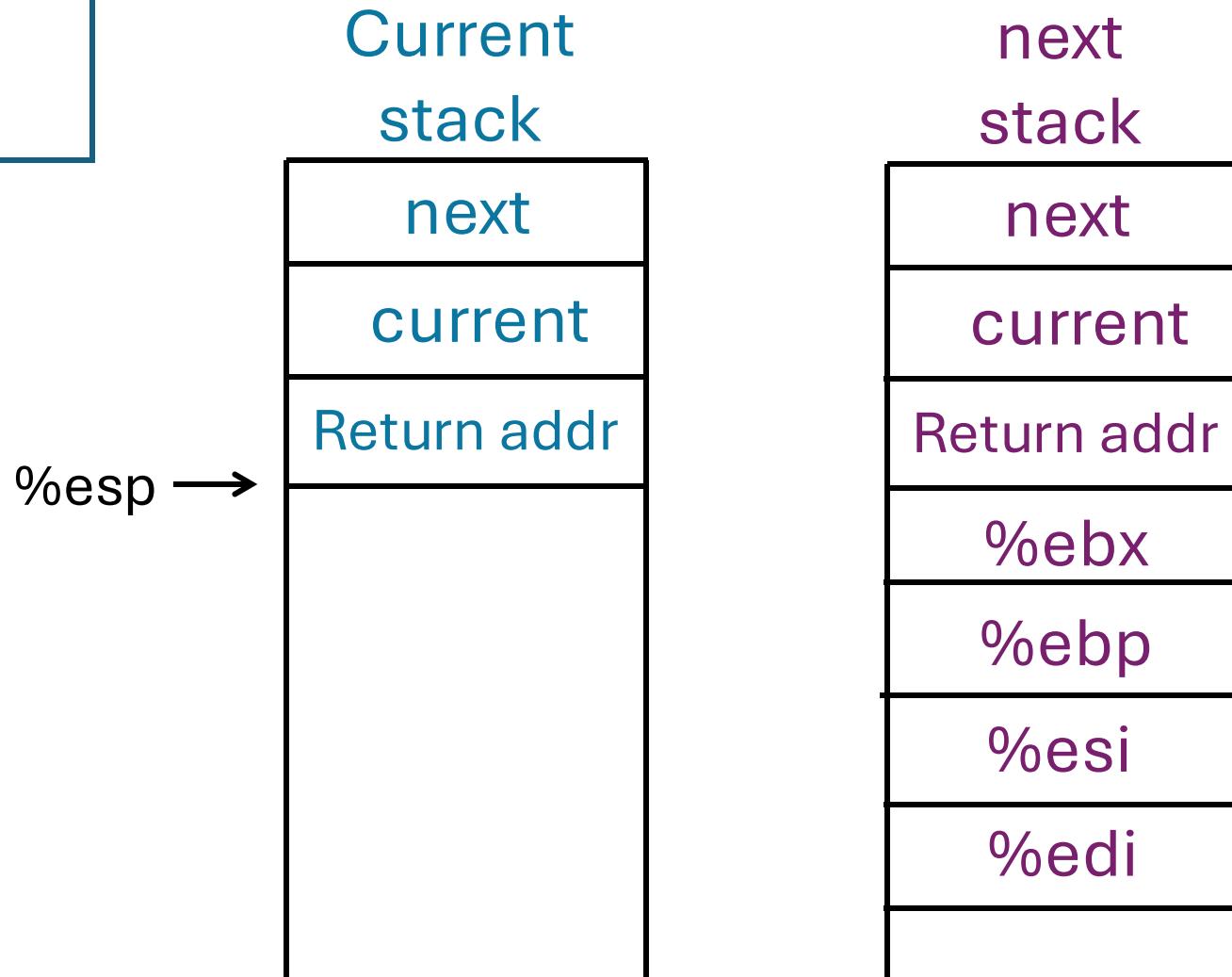
Pintos's implements switch\_thread in i386 assembly

```
pushl %ebx; pushl %ebp # Save caller-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 calle-saved regs
popl %ebp; popl %ebx
ret # Resume execution
```

# Pintos switch\_thread

```
struct thread *switch_threads (struct thread *cur, struct thread *next);
```

```
pushl %ebx; pushl %ebp  
pushl %esi; pushl %edi
```



# Pintos switch\_thread

```
struct thread *switch_threads (struct thread *cur, struct thread *next);
```

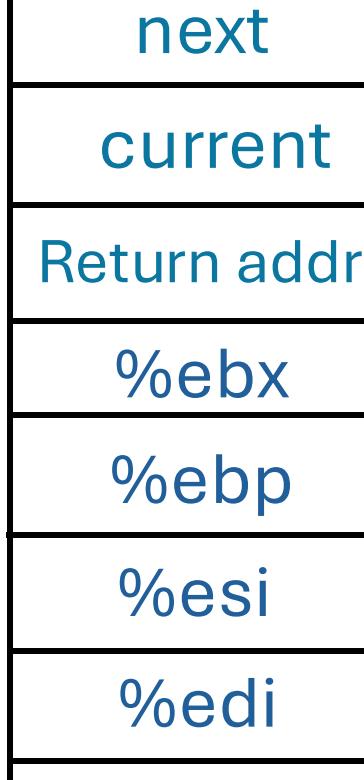
```
pushl %ebx; pushl %ebp  
pushl %esi; pushl %edi
```

```
mov thread_stack_ofs, %edx  
movl 20(%esp), %eax  
movl %esp, (%eax,%edx,1)  
movl 24(%esp), %ecx  
movl (%ecx,%edx,1), %esp
```

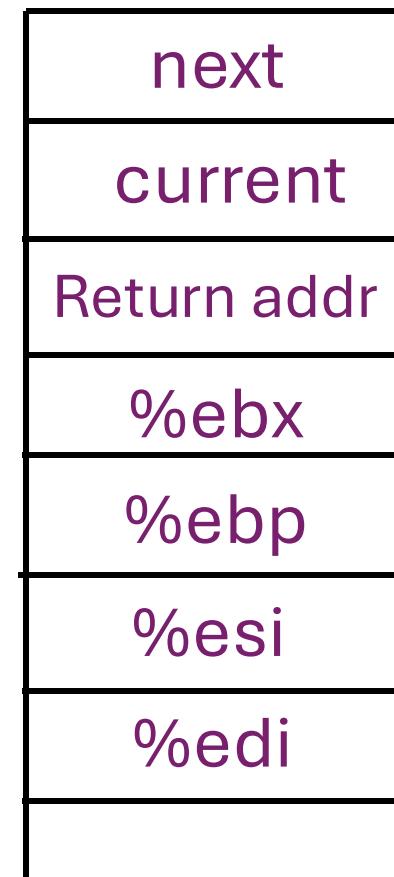
```
# cur->stack = %esp  
# %esp = next->stack
```

%esp →

Current  
stack



next  
stack



# Pintos switch\_thread

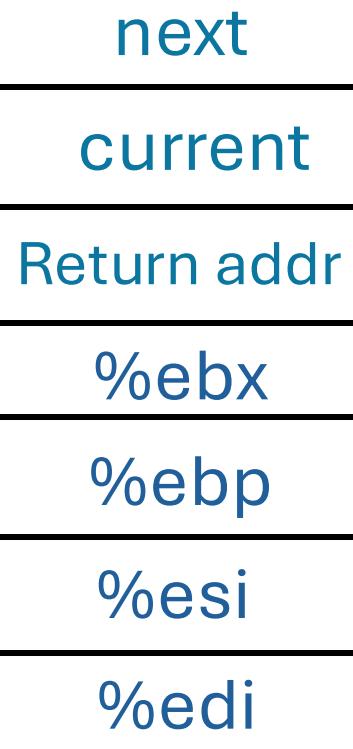
```
struct thread *switch_threads (struct thread *cur, struct thread *next);
```

```
pushl %ebx; pushl %ebp  
pushl %esi; pushl %edi
```

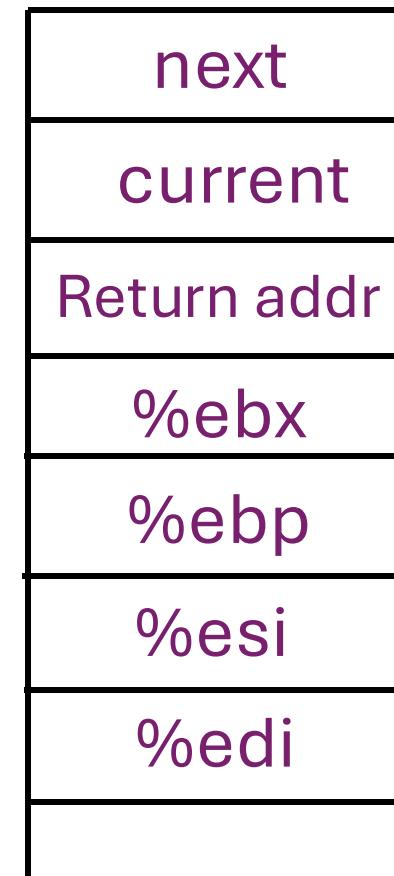
```
mov thread_stack_ofs, %edx  
movl 20(%esp), %eax  
movl %esp, (%eax,%edx,1)  
movl 24(%esp), %ecx  
movl (%ecx,%edx,1), %esp
```

```
popl %edi; popl %esi  
popl %ebp; popl %ebx
```

Current  
stack



next  
stack



# Pintos switch\_thread

```
struct thread *switch_threads (struct thread *cur, struct thread *next);
```

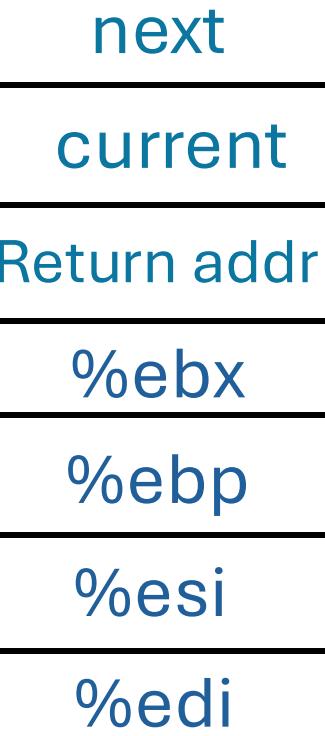
```
pushl %ebx; pushl %ebp  
pushl %esi; pushl %edi
```

```
mov thread_stack_ofs, %edx  
movl 20(%esp), %eax  
movl %esp, (%eax,%edx,1)  
movl 24(%esp), %ecx  
movl (%ecx,%edx,1), %esp
```

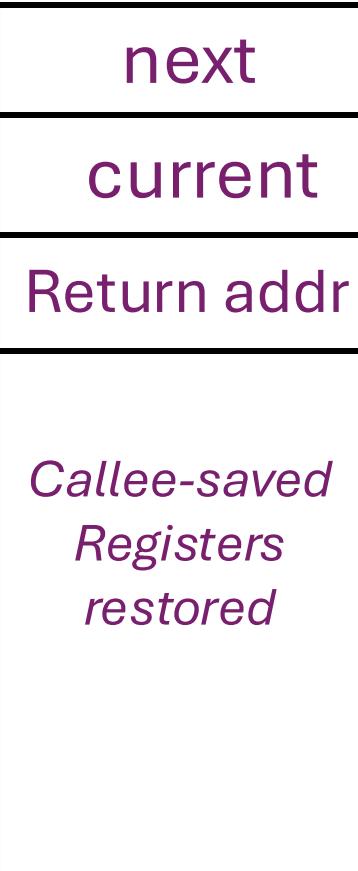
```
popl %edi; popl %esi  
popl %ebp; popl %ebx
```

```
ret
```

Current  
stack



next  
stack



# Thread Summary

**The operating system as a large multithreaded program**

- Each process executes as a thread within the OS

**Multithreading is also very useful for applications**

- Efficient multithreading requires fast primitives
- Processes are too heavyweight

**Solution is to separate threads from processes**

- Kernel-level threads much better, but still significant overhead
- User-level threads even better, but not well integrated with OS

**Now, how do we get our threads to correctly cooperate with each other?**

- Synchronization...

# **Next Time...**

**Read Chapters 28, 29**