

# CE 440 Introduction to Operating System

## Lecture 4: Thread Fall 2025

**Prof. Yigong Hu**



Slides courtesy of Manuel Egele, Ryan Huang and Baris Kasikci

# Recap: Process Creation

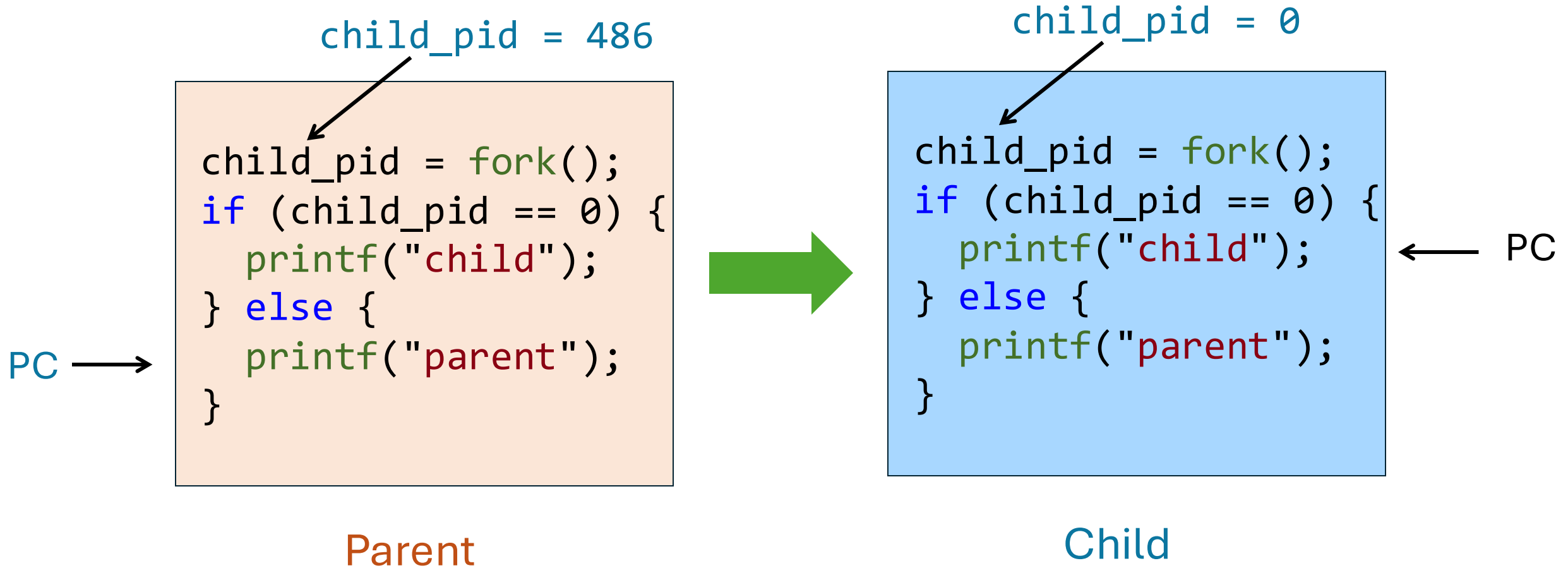
`int fork()`

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

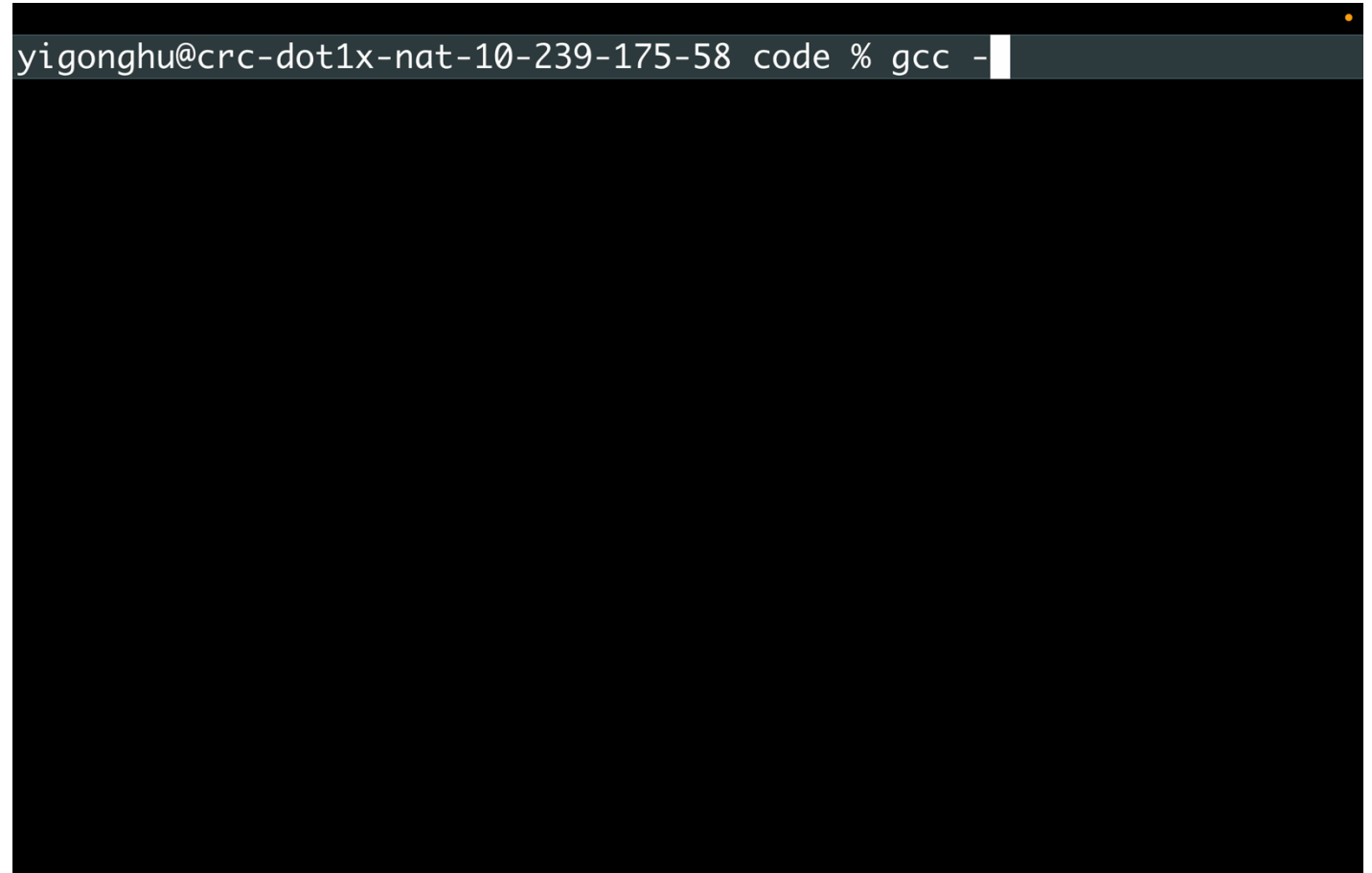
# Example: Web Server

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

# Example: Shell

```
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;
    }
}
```

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



# 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

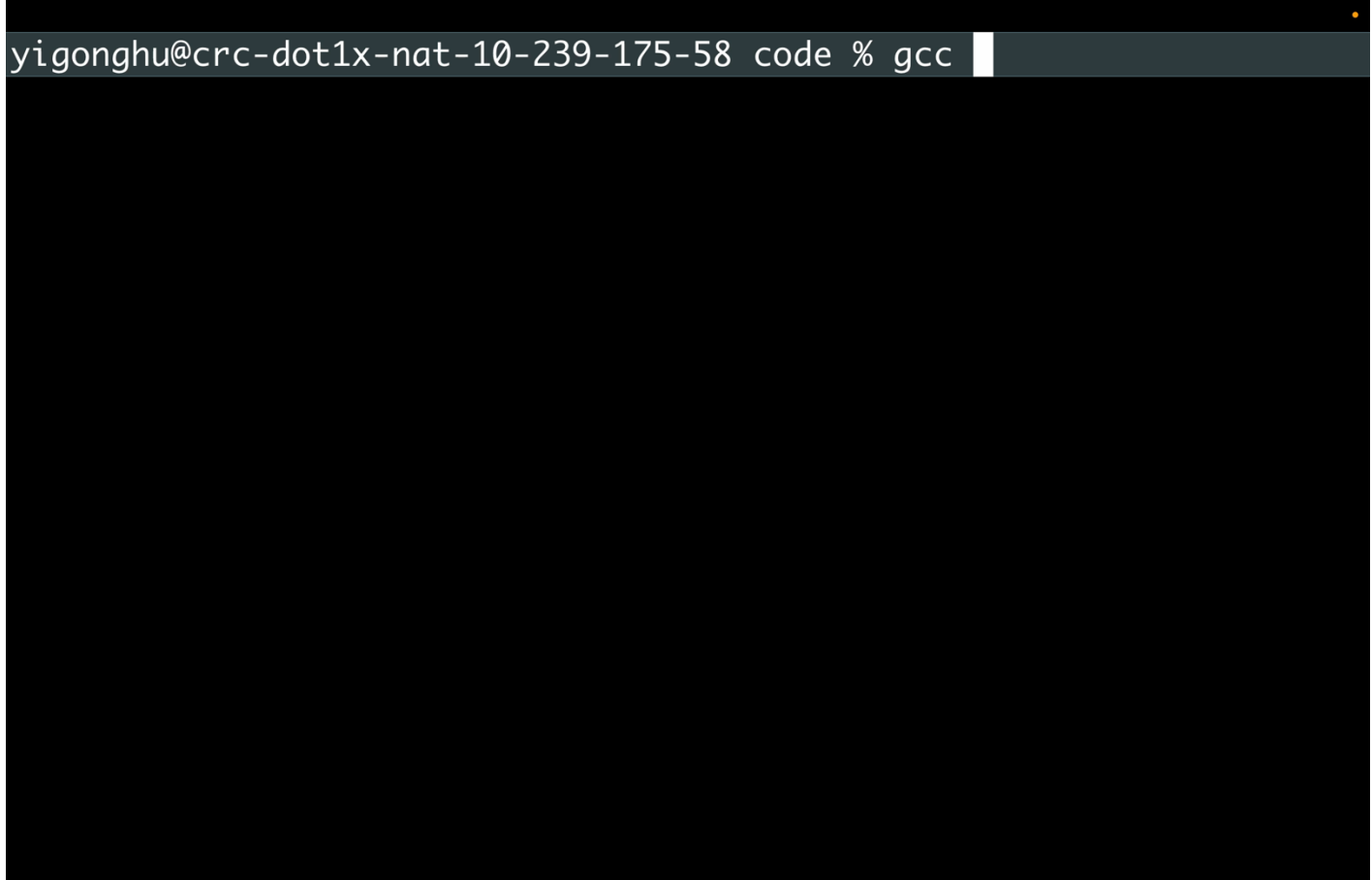
- Tons of things you might want to do to child:
  - **manipulate file descriptors, set environment variables, reduce privileges, ...**
- Yet fork requires no arguments at all



# Example: redirect

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

```
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);
}
```



yigonghu@crc-dot1x-nat-10-239-175-58 code % gcc

# 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()

**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.

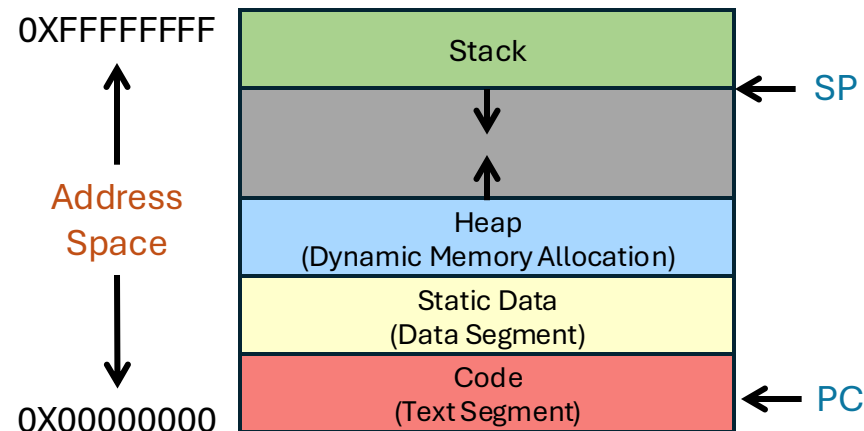
# Rethinking Process

## 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 (such as) is in the kernel structure.
 * Lightweight-process data (lwp_h) and the kernel stack may be swapped out.
 */
typedef struct proc {
    /*
     * Fields requiring no explicit locking
     */
    struct proc *_p_next; /* pointer to next node */
    struct proc *_p_prev; /* process address above pointer */
    struct proc *_p_lock; /* ptr to process's mutex lock */
    klist_t p_child; /* lock for p_cred */
    struct cred *_p_cred; /* process credentials */

    /*
     * Fields protected by p_lock
     */
    kmemod_t p_flag; /* proc struct's condition variable */
    kmemod_t p_priority; /* waiting for some lock to wait */
    kmemod_t p_nblock; /* process is waiting for its lwa */
    kmemod_t p_nblock2; /* used to be held */
    u_int u_arg; /* user's arg set */
    /*
     * Fields defined below */
    clock_t p_startime; /* user time, this process */
    clock_t p_rtime; /* system time, this process */
    clock_t p_uptime; /* sum of children's uptime */
    clock_t p_ttime; /* sum of children's system time */
    caddr_t p_account; /* segment accounting info */
    caddr_t p_kbytes; /* heap size in kbytes */
    size_t p_heap; /* heap size in bytes */

    /*
     * Per process signal stuff.
     */
    k_sigset_t p_sigmask; /* signals pending to this process */
    k_sigset_t p_ignore; /* ignore when generated */
    struct k_sigset p_siginfo; /* get signal info with signal */
    struct k_sigset p_sigqueue; /* queue signal info with signal */
    struct k_sigset p_sigpend; /* list to allocate structure pool */
    struct k_sigset p_sigpending; /* allocate structure pool */
    u_int p_nsigpend; /* jobcontrol stop signal */

```

## Struct proc (Solaris) (2)

```

// Special per-process flag when set will xla illegal memory
// references.
char p_flagillegal;

//
// Per process log and kernel thread stuff
//
int s_logid; // most recently allocated logid +1
int p_logcnt; // number of logs in this process +1
int p_logsize; // number of not stopped logs
int p_logwait; // number of logs in log_wait() +1
int p_logsumbit; // number of sombit logs +1
int p_logsum; // number of entries in p_logsum_id +1
int s_logsum_id; // array of sombit logids +1
int s_logsum_id_list; // circular list of threads +1

//
// /proc (process filesystem) debugger interface stuff.
//
k_logset_t p_logmask; // mask of traced signals (/proc) +1
k_logset_t p_logmask; // mask of traced files (/proc) +1
k_logset_t p_logmask; // pointer to primary (/proc) +1
struct vmde_t p_logvms; // list of /proc vmde's for processes +1
kthread_t *p_logentry; // thread ptr for /proc agent log +1
struct watched_area *p_wareas; // list of watched areas +1
ulong t_n_wareas; // number of watched areas +1
struct watched_page *p_wpages; // unwatched watched page (vfork) +1
int p_wpages; // number of watched pages (vfork) +1
p_mcontext_t p_mcontext; // number of active pc_mcontext +1
struct proc *p_loglink; // linked list for server +1
knodehdr_t p_arwhash; // //
size_t p_exhibit; // process stack size in bytes +1

//
// Microstate accounting, resource usage, and real-time profiling
//
khrtime_t p_startut; // h/proc process start time +1
khrtime_t p_endutime; // h/proc process termination time +1
khrtime_t p_nhrtime; // elapsed time sum over default logs +1
khrtime_t p_defutime; // microstate sum over default logs +1
struct lrusage_t p_ru; // lrusage sum over default logs +1
struct timeval_t p_ru_tval; // ITIMER_REAL/PROF interval timer +1
uint32_t p_defvcpu; // ITIMER_REAL/PROF vcpu +1
uint32_t p_deflog; // number of default logs +1
//
// profiling. A lock is used in the event of multiple logs +1
// using the same profiling base/size.
kmutex_t p_gproflock; //
struct prof_t p_prof; // protects user profile arguments +1
// profile arguments +1

//
// The user structure
//
struct user_t p_user; // (see user/user.h) +1

//
// Doors.
//
kthread_t p_door; //
struct door_node_t p_door_list; // active doors +1
struct door_node_t p_door_list; // active doors +1
struct door_node_t p_door_list; // active doors +1
kthread_t p_door; //
char p_unref_thread; // unref thread created +1

```

### Struct proc (Solaris) (3)

```

/*
 * Kernel probes
 */
uwhar_t p_tf_flags;

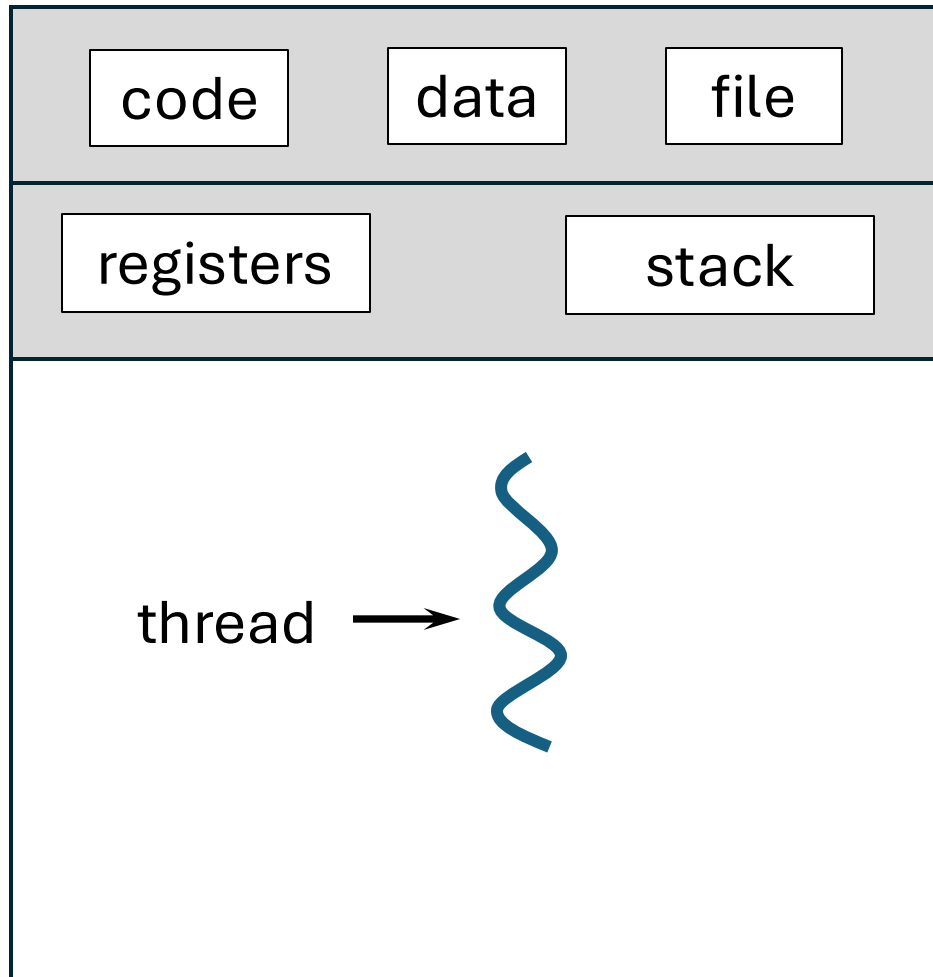
/*
 * C2 Security (C2_AUDIT)
 */
caddr_t p_audit_data; /* per process audit structure */
kthread_t p_asleep; /* thread ptr representing 'asleep' */
before(198) || defined(_LSE) || defined(_L464)
uwhar_t p;

/*
 * L2P support.
 */
kmutex_t p_lock; /* protects the following fields */
struct sleep_desc p_id; /* pointer to private L2P */
struct sleep_desc p_id_desc; /* segment descriptor for private L2P */
int p_id_limit; /* highest selector used */

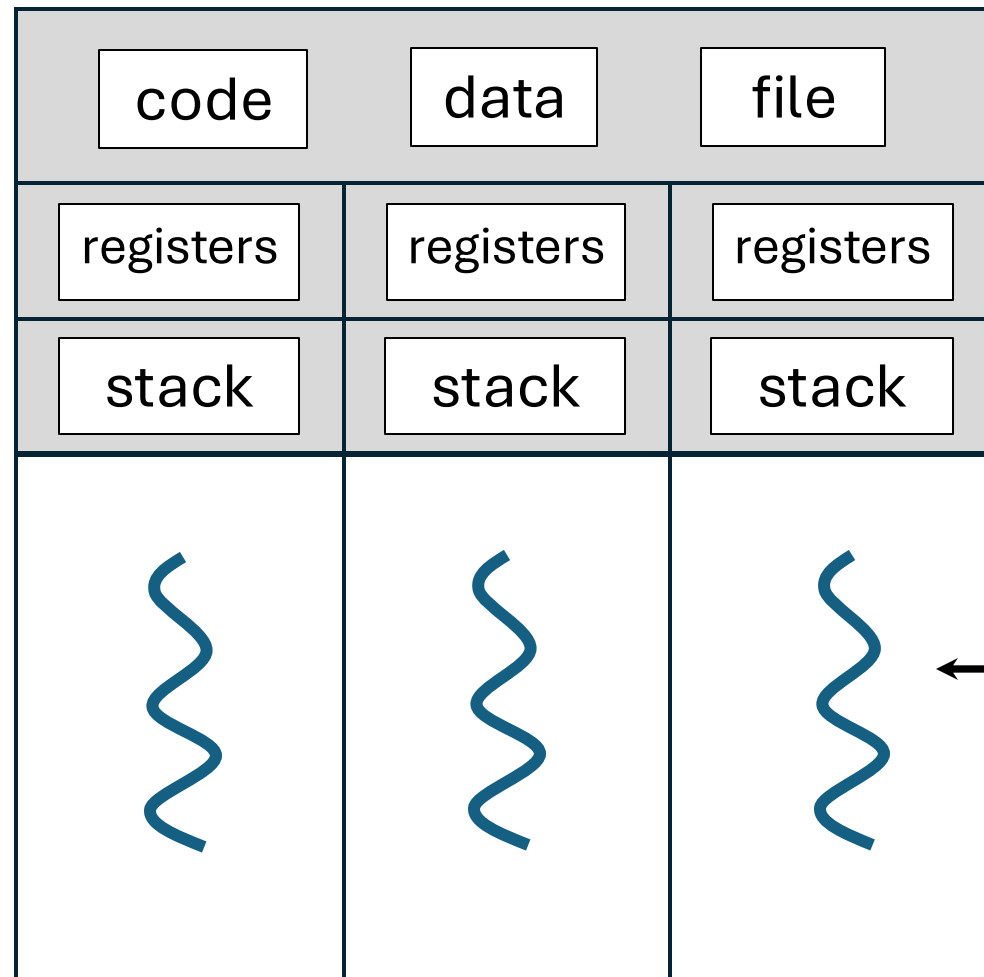
/*
 * L2P support.
 */
p_resident set size before last sweep */
struct aio p_aio; /* pointer to aio */
struct limer p_limer; /* interval timer */
k_mutex_t p_outflags; /* signals in notification set */
k_mutex_t p_outflag; /* notify ev to synchronize with asleep */
kmutex_t p_limer; /* state's limer */
uwhar_t p_as_unlocked; /* number of unlocked threads */
struct vnode_desc p_as_desc; /* scheduler activations desc */
p_lstart; /* top of the process stack */
uwhar_t p_tgroot; /* stack memory protection */
model_t p_model; /* data model determined at exec time */
struct bop_data p_bop; /* bop data cache */

```

# Threads in a Process



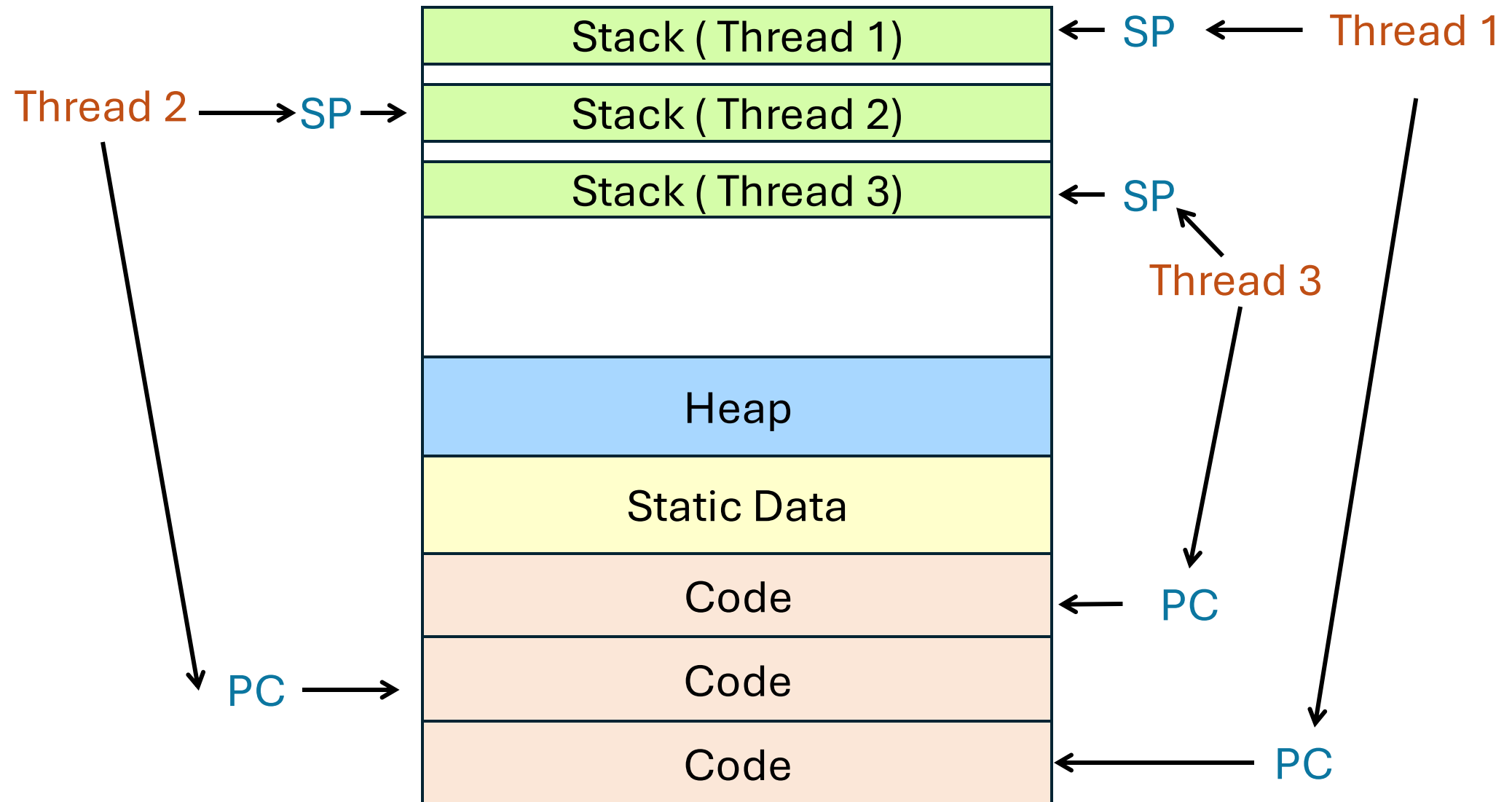
Single-threaded process



Where is heap?

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 ();`

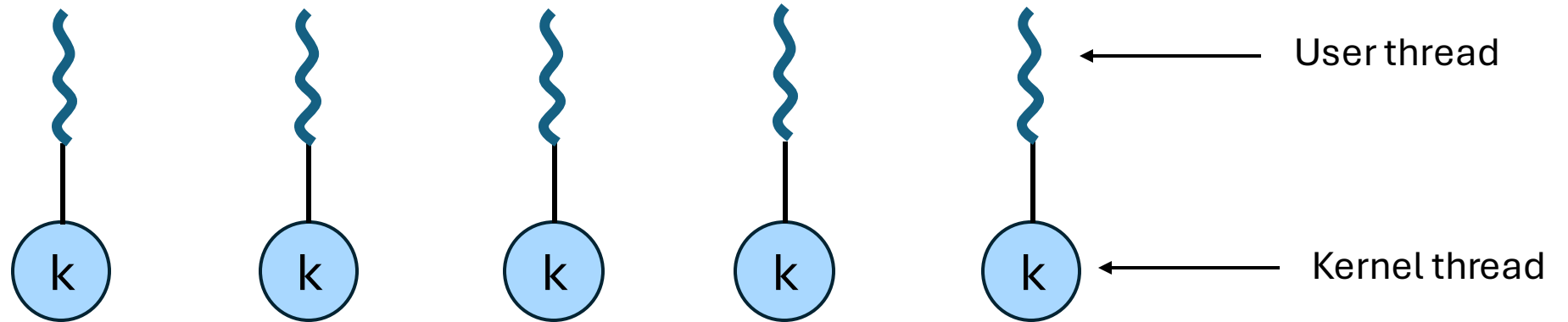
- Destroy current thread

`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

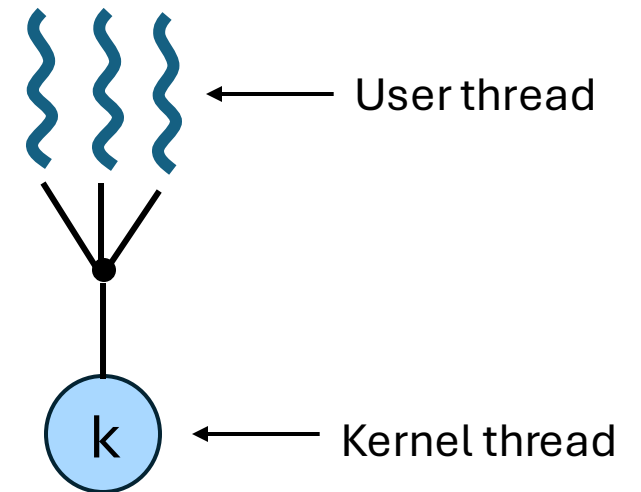
# Alternative: User-Level Threads

## 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)

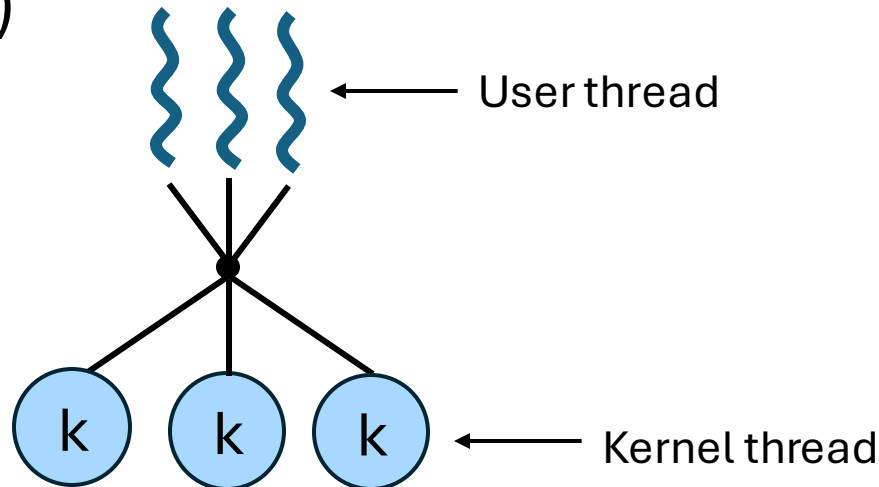
# User Threads on Kernel Threads

## 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)



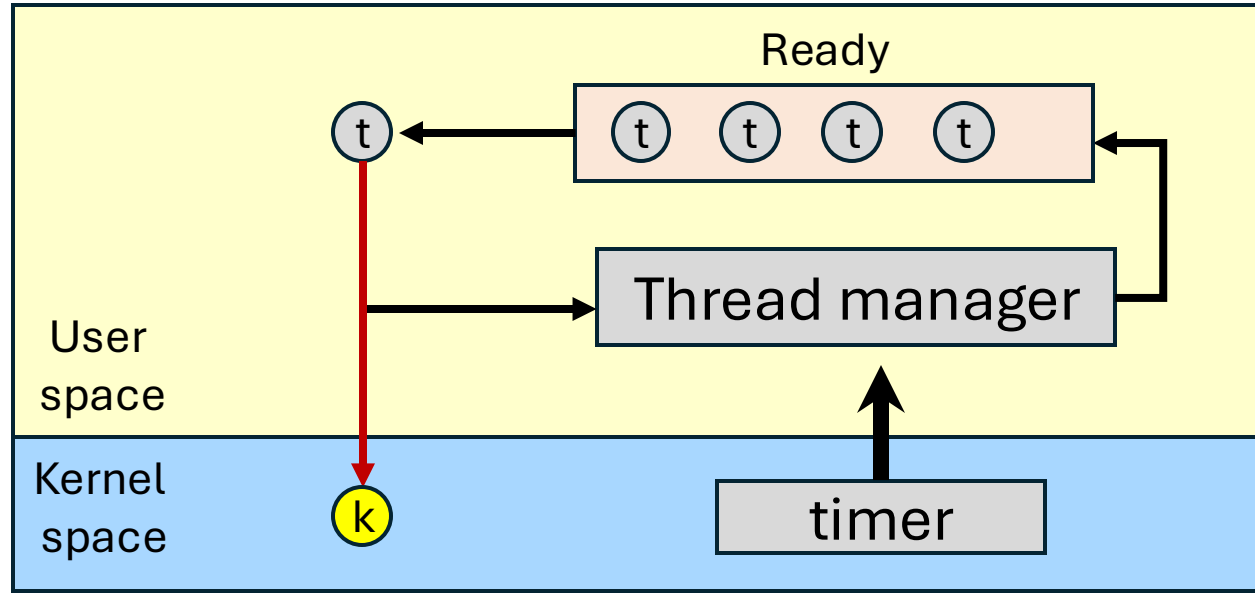
# Use Case: Java Virtual Machine

## 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?



# 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

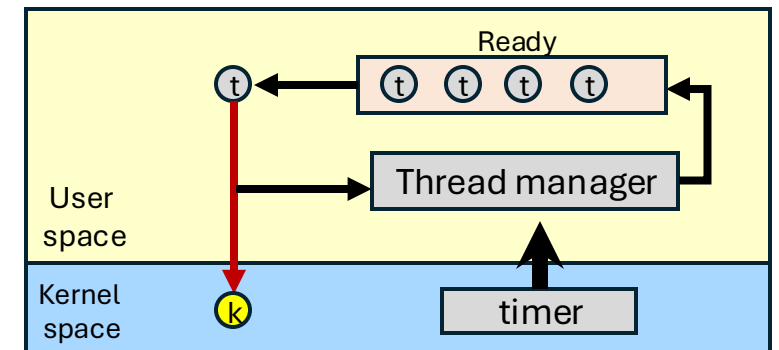
- Just like the OS and processes
- But it is implemented at user-level in a library

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

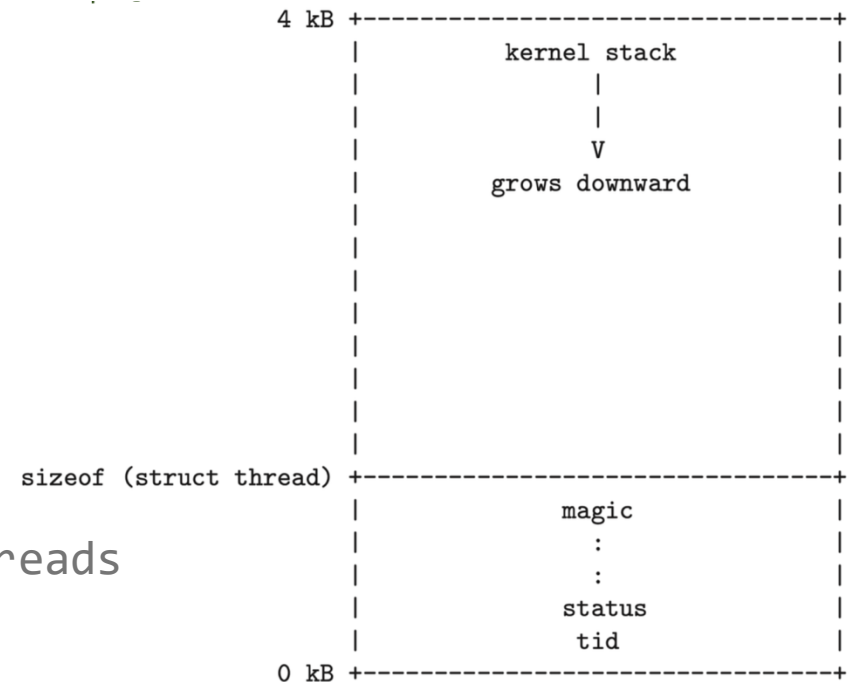
# Pintos switch\_threads

## 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

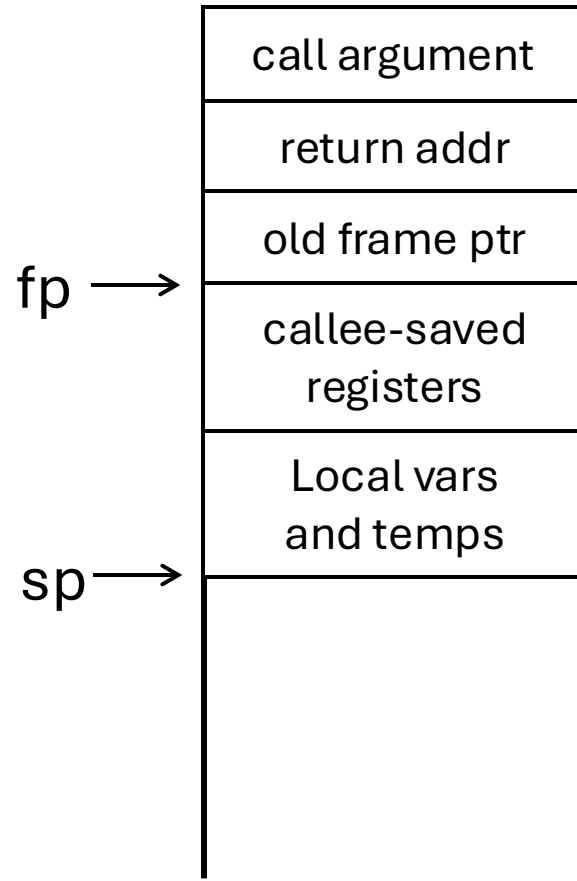
- a standard on how functions should be called by the *machine*
- how a function call in C/C++ gets converted into assembly language
- Compilers need to obey this standard in compiling code into assembly

## 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



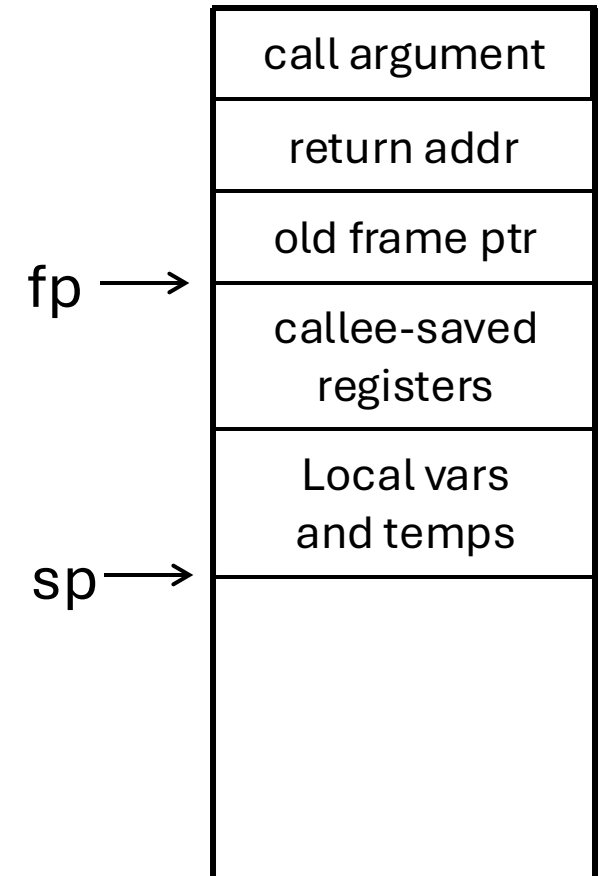
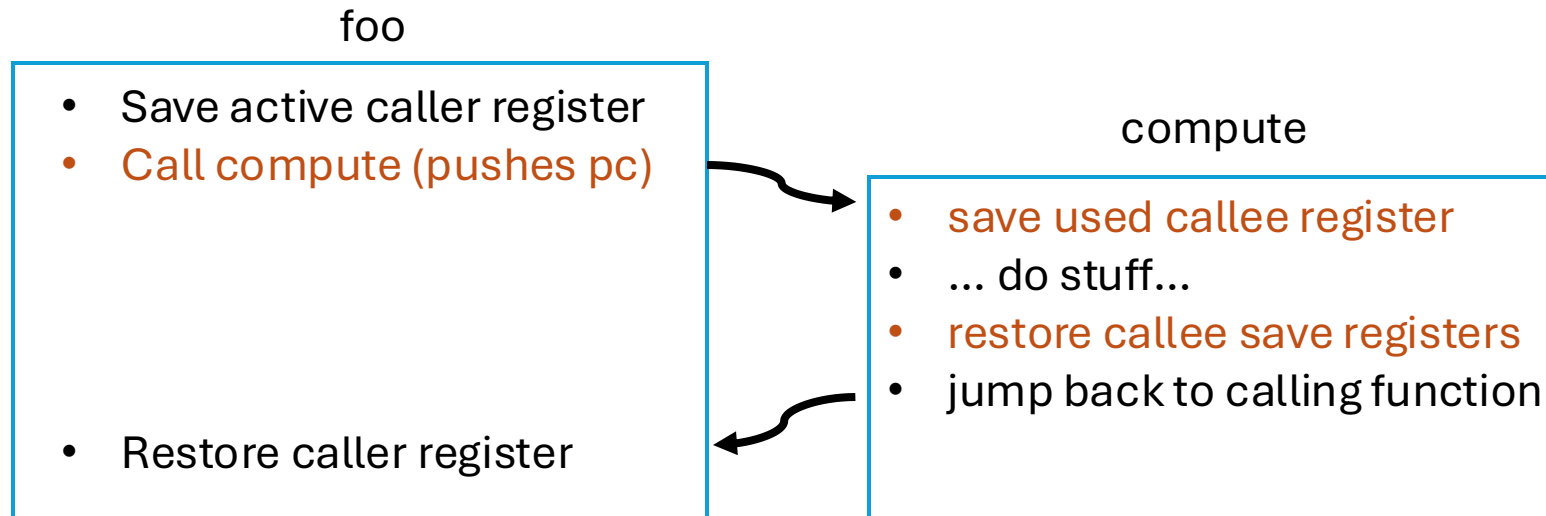
```
int compute(int a, int b)
{
    int i, result;
    result = 0;
    for (i = 0; i < a; i++)
        result = result + b - i;
    return result;
}

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

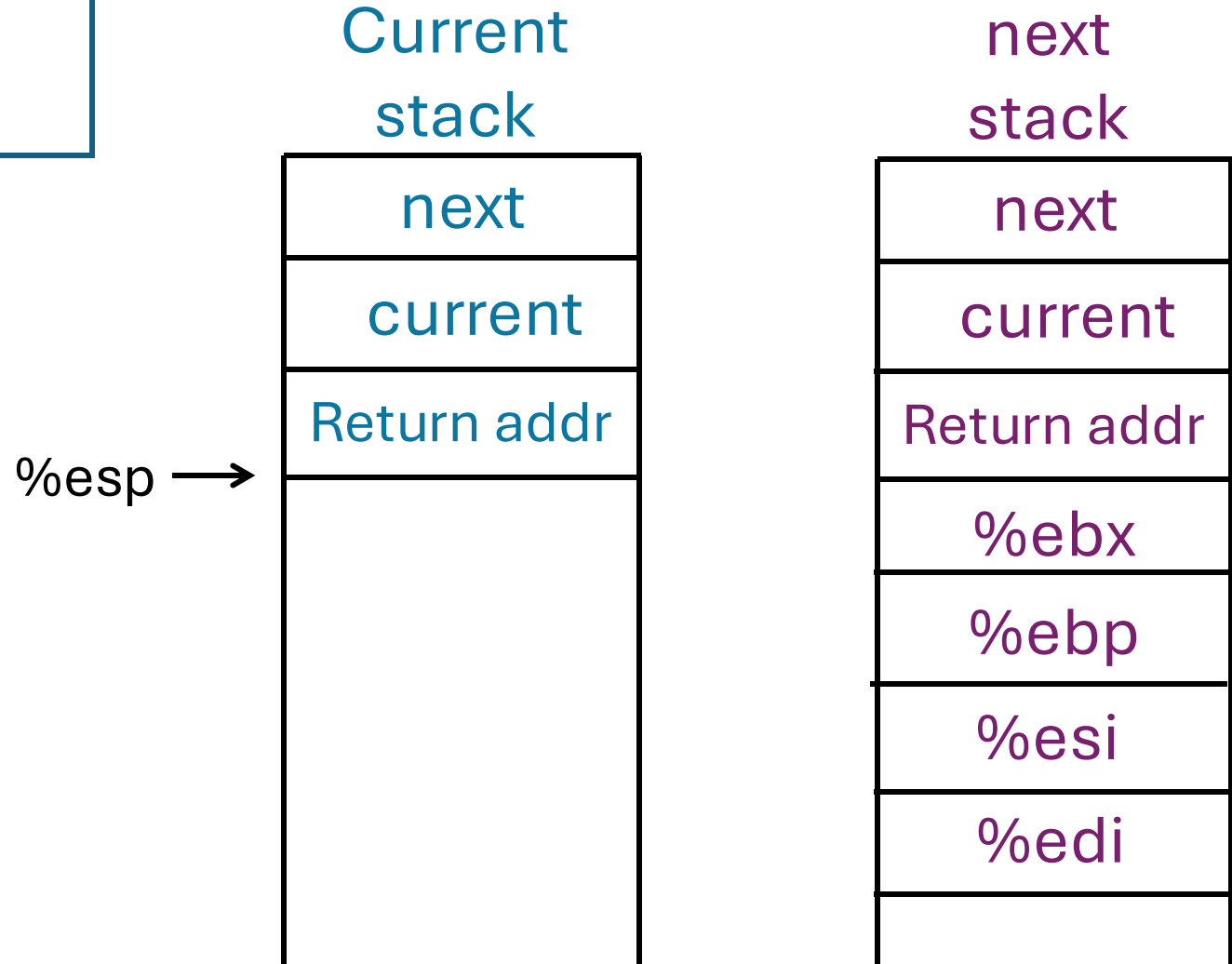
```
pushl %ebx; pushl %ebp # Save caller-saved regs
pushl %esi; pushl %edi
mov thread_stack_ofs, %edx # %edx = offset of stack field
                           # in thread struct

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 callee-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

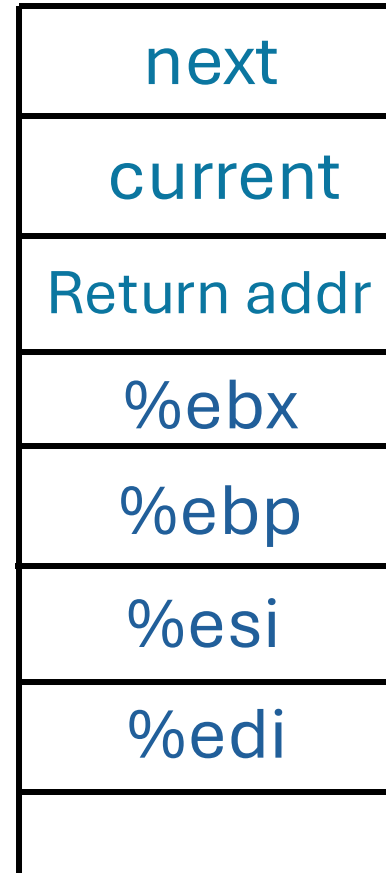
```
struct thread *switch_threads (struct thread *cur, struct thread *next);
```

```
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
```

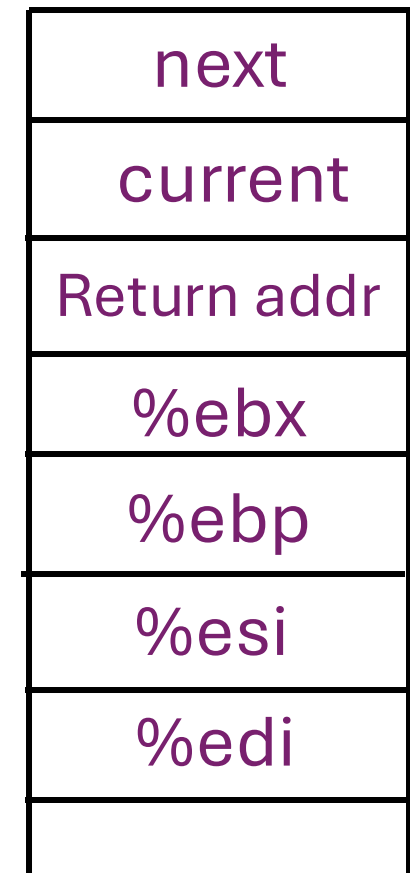
```
# cur->stack = %esp  
# %esp = next->stack
```

Current  
stack



%esp →

next  
stack



# Pintos switch\_thread

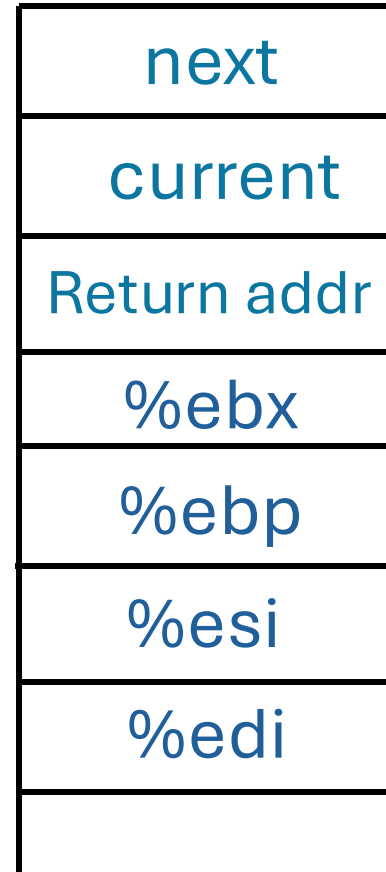
```
struct thread *switch_threads (struct thread *cur, struct thread *next);
```

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

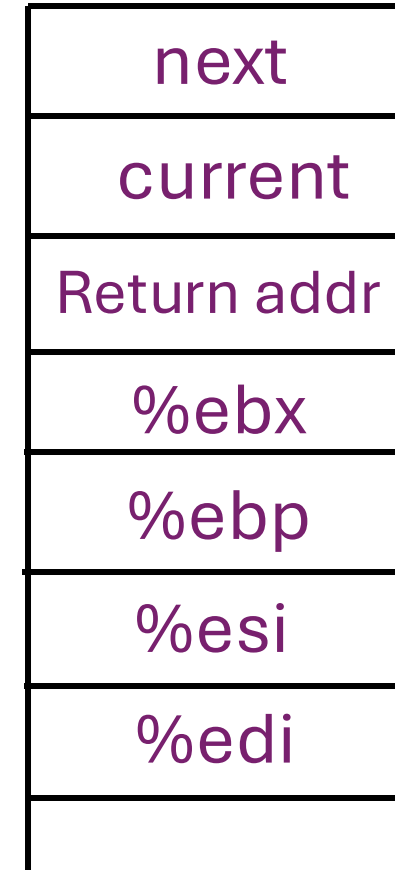
```
mov thread_stack_ofs, %edx  
movl 20(%esp), %eax  
movl %esp, (%eax,%edx,1)  
movl 24(%esp), %ecx  
movl (%ecx,%edx,1), %esp
```

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

Current  
stack



next  
stack



← %esp

# Pintos switch\_thread

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

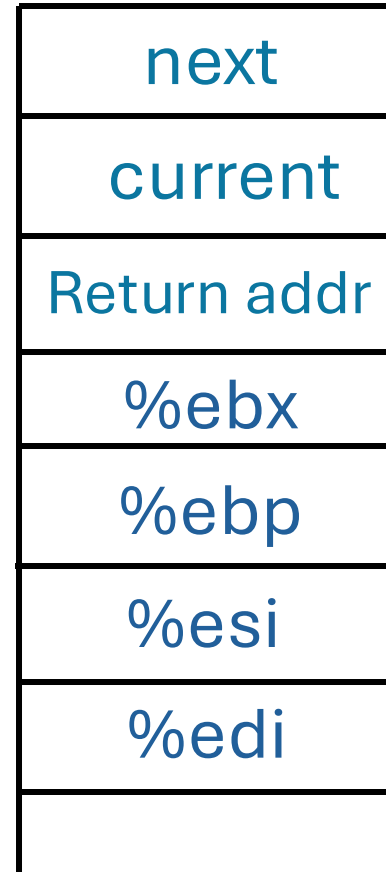
```
pushl %ebx; pushl %ebp  
pushl %esi; pushl %edi
```

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

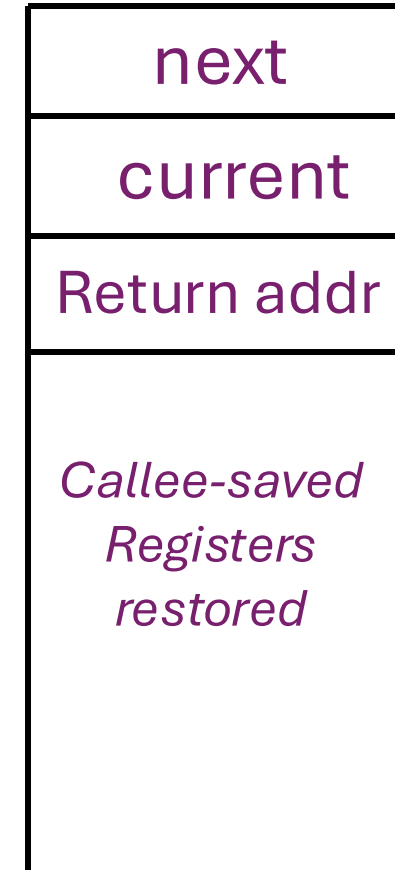
```
popl %edi; popl %esi  
popl %ebp; popl %ebx
```

```
ret
```

Current  
stack



next  
stack



← %esp



# 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 7, 8**