CS 318 Principles of Operating Systems

Fall 2017

Lecture 5: Thread

Ryan Huang

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Administrivia

- **HW1 solution released on Piazza resources**
- **Lab 0 grading**
  - In progress
  - Cheating policy
- **Lab 1 review session by Guoye**
  - today 3pm in Malone 228
- **Group request**

Processes

- **Recall that a process includes many things**
  - An address space (defining all the code and data pages)
  - OS resources (e.g., open files) and accounting information
  - Execution state (PC, SP, regs, etc.)
- **Creating a new process is costly**
  - because of 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

Concurrent Programs

- **Recall our Web server example (or any parallel program)…**
  - forks off copies of itself to handle multiple simultaneous requests
- **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.

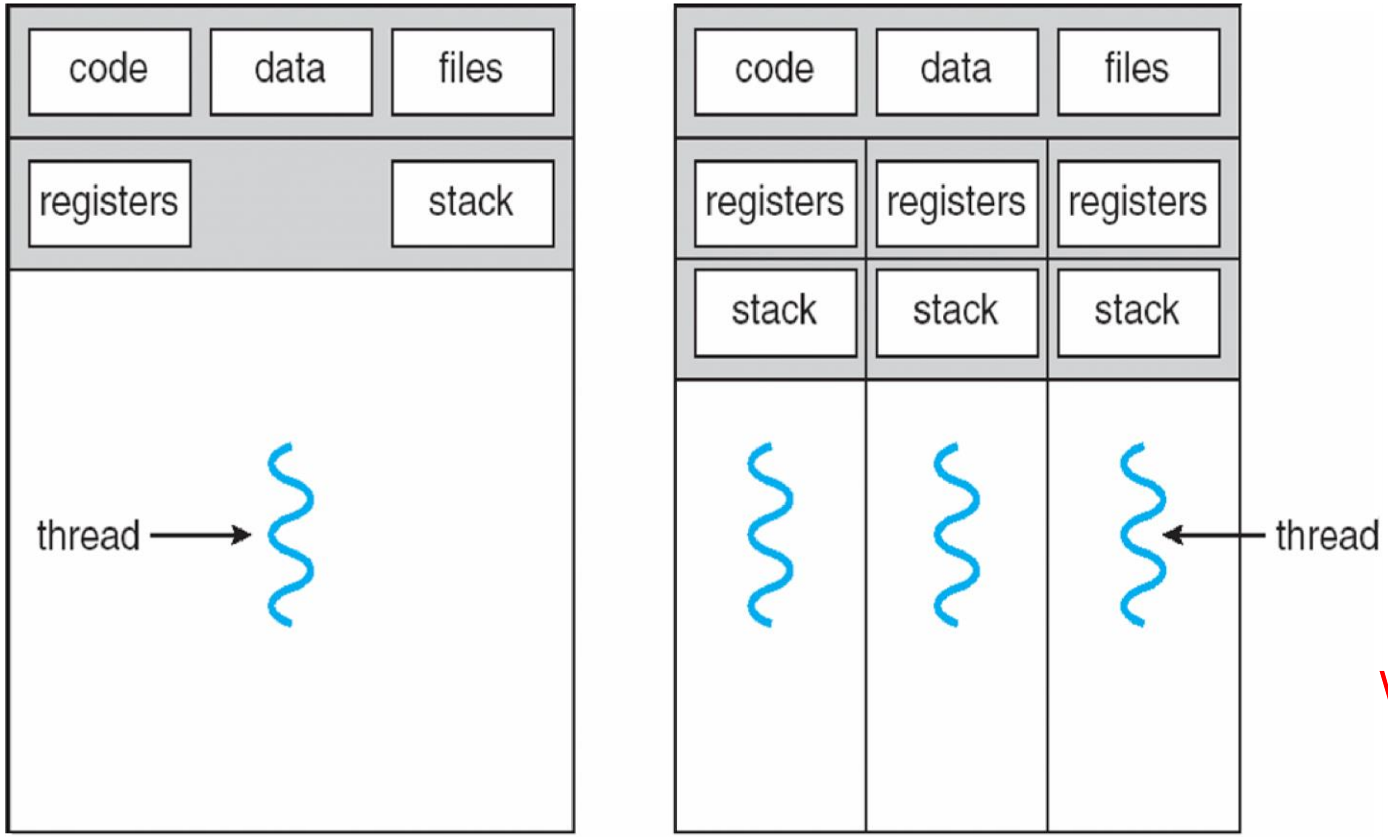
Rethinking Processes

- **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
- **Key idea: Why don't we separate the concept of a process 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 (everything but threads of execution)
- **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

Threads in a Process

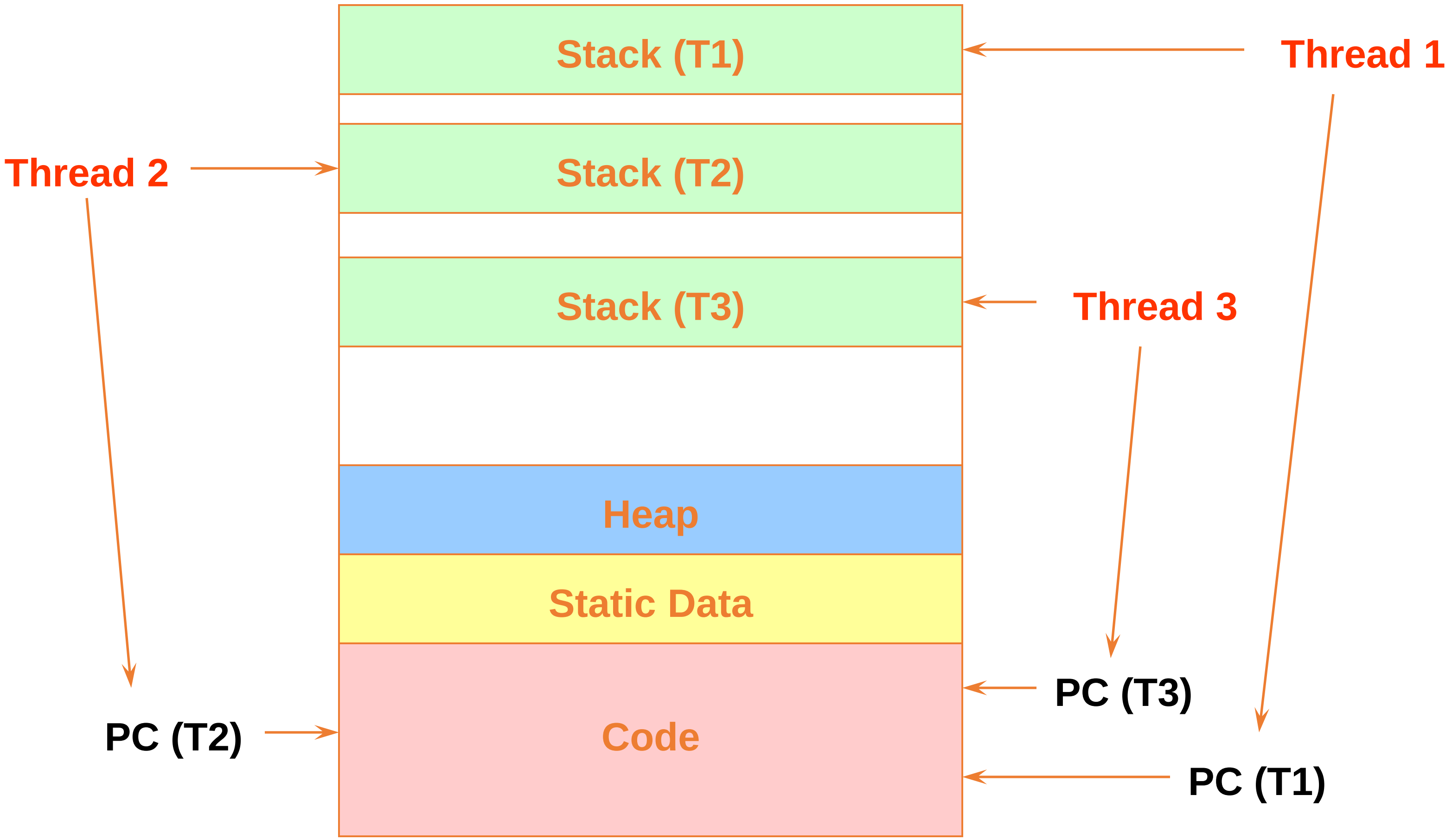


single-threaded process

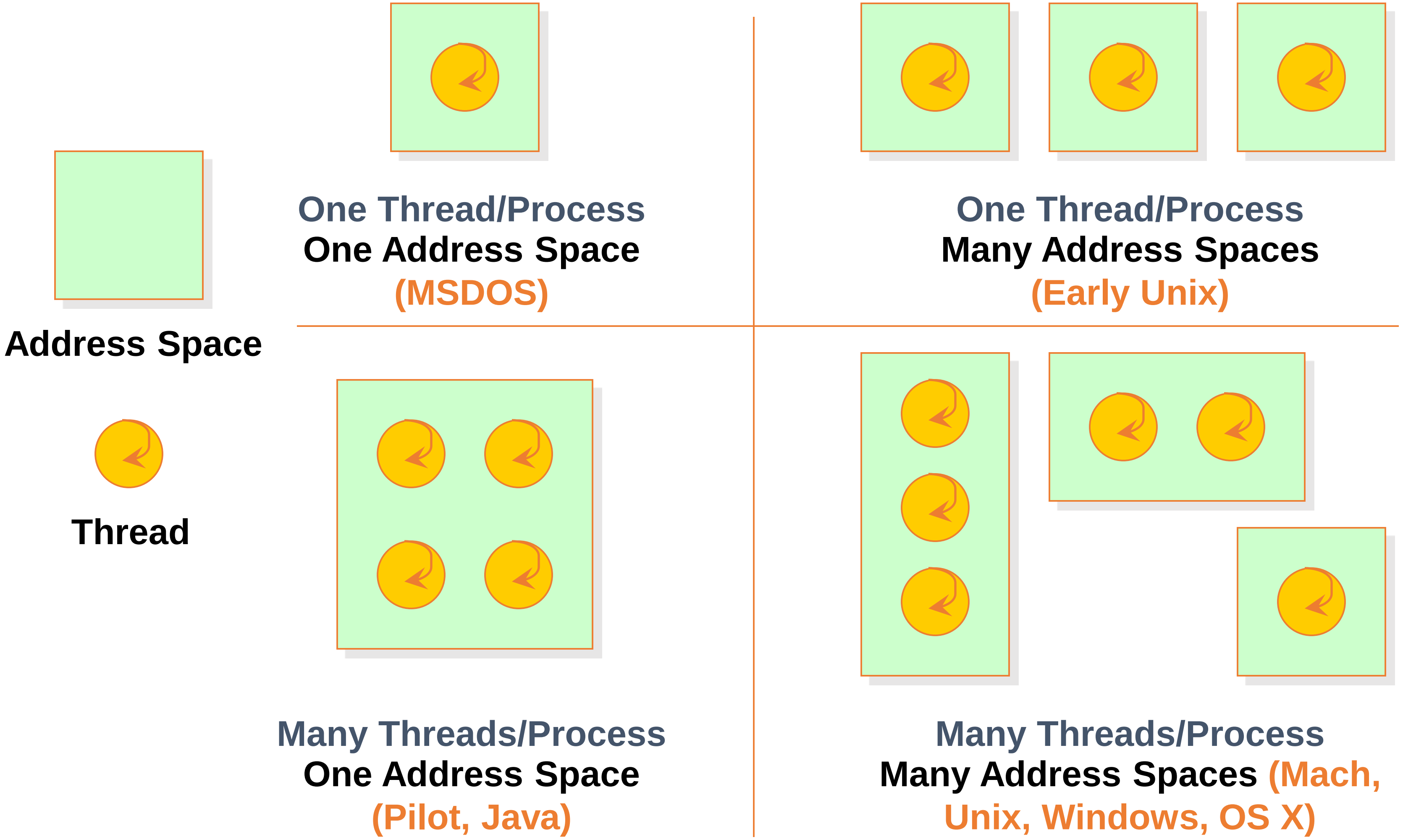
multithreaded process

What about heap?

Threads in a Process



Thread Design 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…

Threads: Concurrent Servers

- **Using `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**

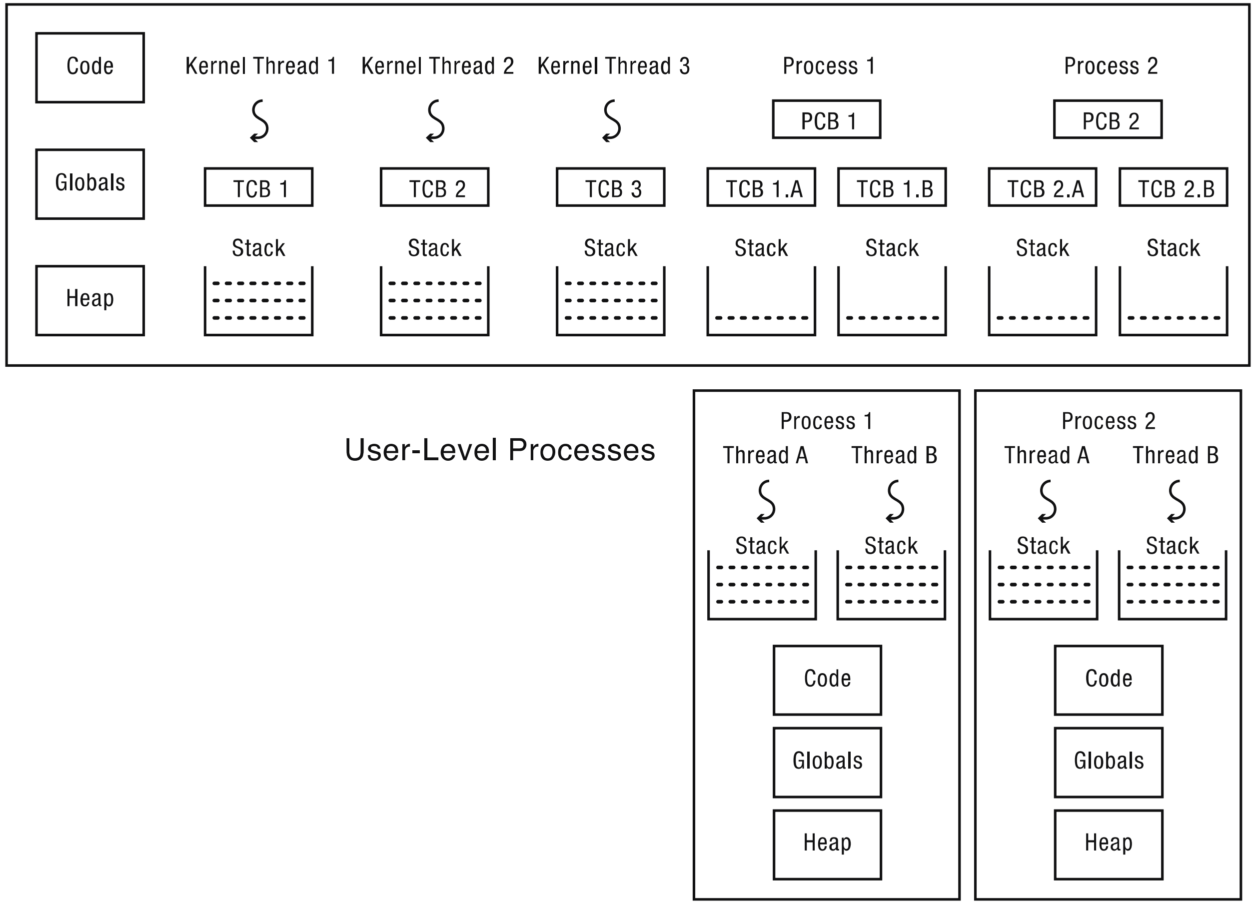
```
web_server() {
    while (1) {
        int sock = accept();
        thread_fork(handle_request, sock);
    }
}

handle_request(int sock) {
    Process request
    close(sock);
}
```

Thread Package API

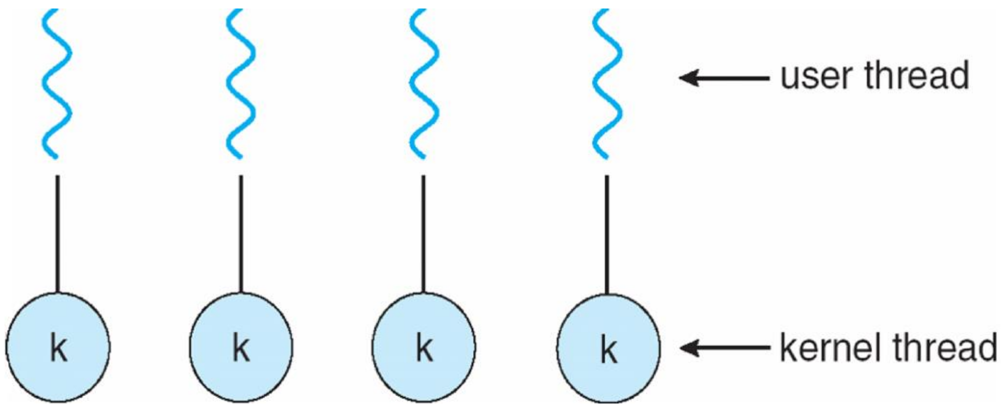
- `tid thread_create (void (*fn) (void *), void *);`
  - Create a new thread, run fn with arg
- `void thread_exit ();`
  - Destroy current thread
- `void thread_join (tid thread);`
  - Wait for thread thread to exit
- **See [Birrell] for good introduction**

Implementing Threads

- `thread_create(fun, args)`
  - Allocate thread control block
  - Allocate stack
  - Build stack frame for base of stack
  - Put func, args on stack
  - Put thread on ready list



Kernel-Level Threads



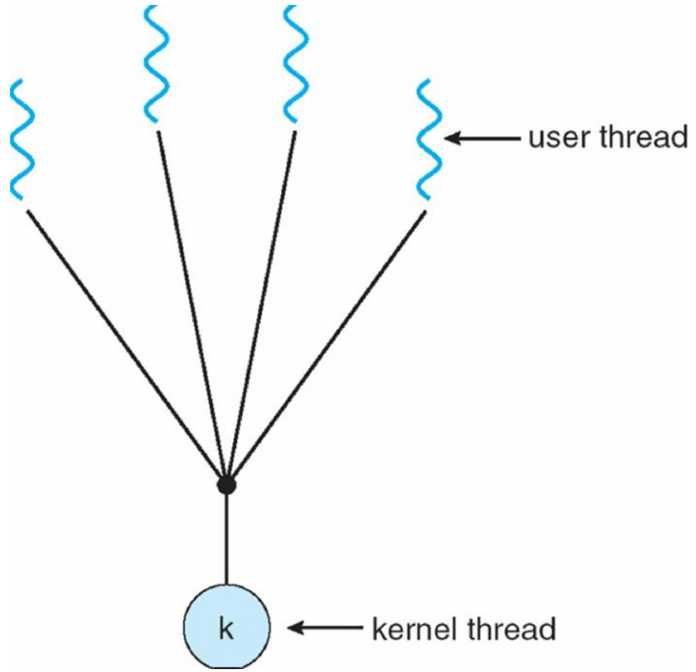
- **All thread operations are implemented in the kernel**
- **The OS schedules all of the threads in the system**
- **Also known as lightweight processes**
  - Windows: threads
  - Solaris: lightweight processes (LWP)
  - POSIX Threads (pthreads): PTHREAD_SCOPE_SYSTEM

Kernel Thread Limitations

- **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-Level Threads

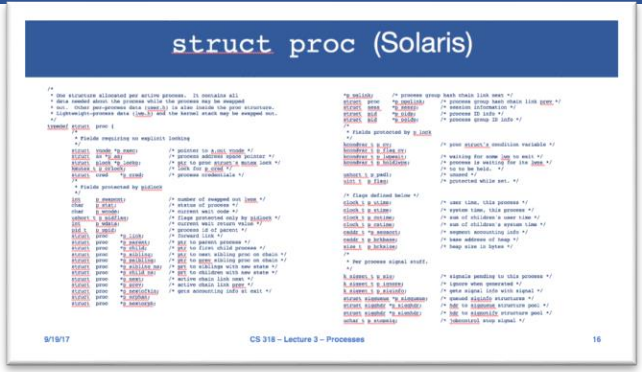
- **Implement as user-level library (a.k.a. green threads)**
  - One kernel thread per process
  - `thread_create`, `thread_exit`, etc., just library functions
  - library does thread context switch
- **User-level threads are small and fast**
  - pthreads: PTHREAD_SCOPE_PROCESS
  - Java: Thread

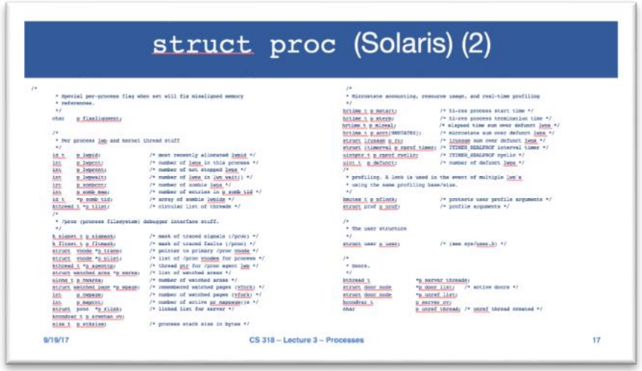


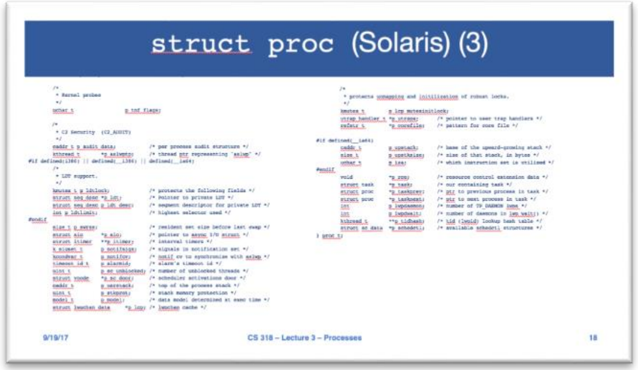
Small and Fast…

- **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. */
  };
```







U/L Thread Limitations

- **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 blocks all threads
    - Can replace read to handle network connections, but usually OSes don’t let you do this for disk
  - Unscheduling a process with a thread holding a lock
- **How to solve this?**
  - communication between the kernel and the user-level thread manager (Windows 8)
    - [Scheduler Activation]

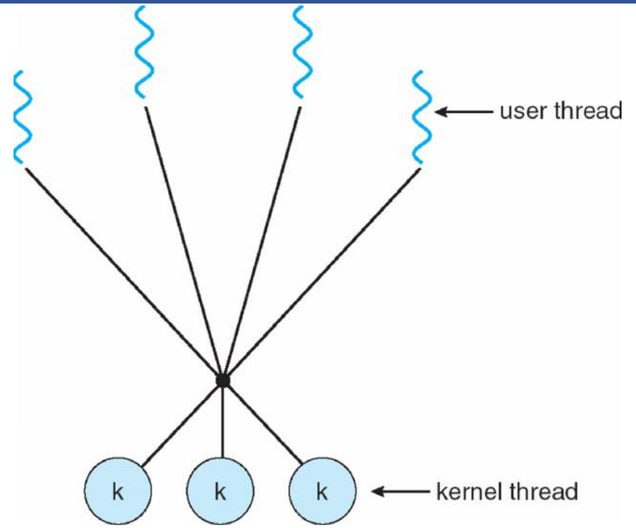
Kernel vs. User Threads

- **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)
- **Understanding their differences is important**
  - Correctness, performance

Kernel and User Threads

- **Or 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
- **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**
- **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

Thread Scheduling

- **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 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?**
- **The semantics of yield are that 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();`
  - …

Implementing `yield()`

```
yield() {
        thread_t old_thread = current_thread;
        current_thread = get_next_thread();
        append_to_queue(ready_queue, old_thread);
        context_switch(old_thread, current_thread);
        return;
}
```

As old thread

As new thread

- **The magic step is invoking context_switch()**
- **Why do we need to call append_to_queue()?**

Preemptive 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

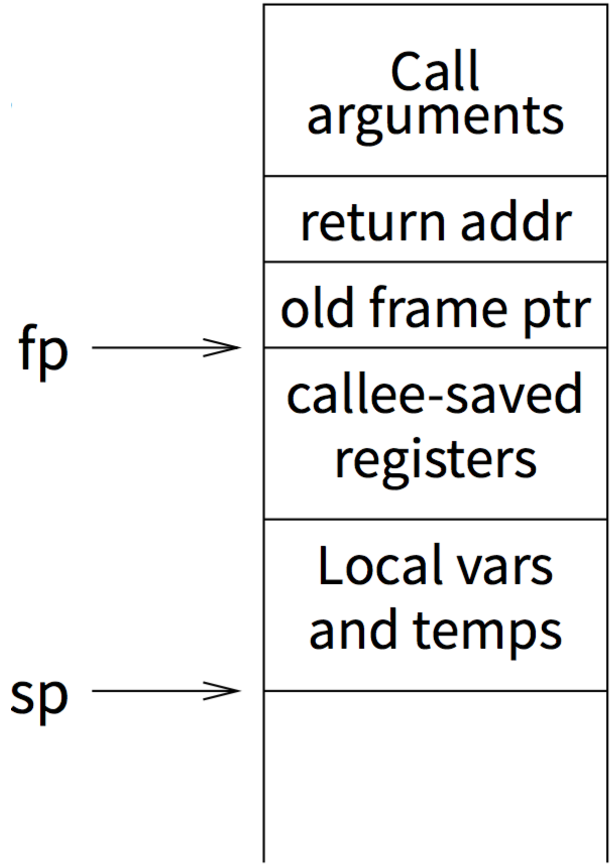
- **The context switch routine does all of the magic**
  - Saves context of the currently running thread (old_thread)
    - Push all machine state onto its stack
  - Restores context of the next thread
    - Pop all machine state from the next thread's stack
  - The next thread becomes the current thread
  - 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

Background: 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

- save active caller registers
- call *foo* (pushes pc)
- restore caller registers

- save used callee registers
- ...do stuff...
- restore callee saved registers
- jump back to calling function



Pintos Thread Implementation

- **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);`

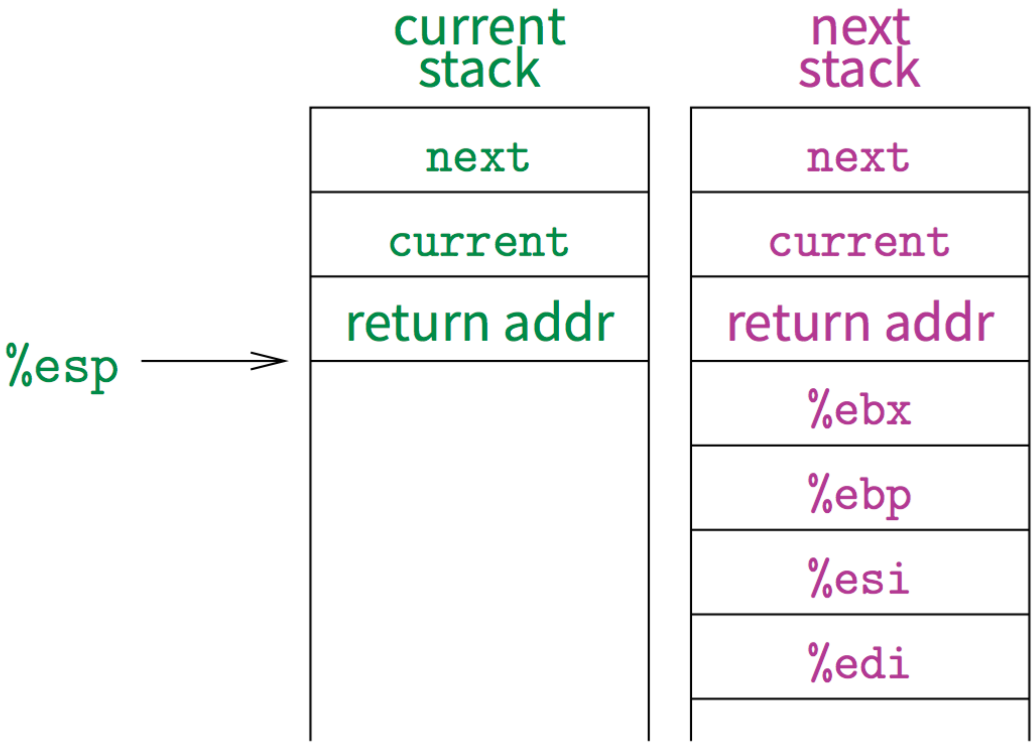
- **Also thread initialization function to create new stack:**
  - `void thread_create (const char *name, thread_func *function, void *aux);`

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

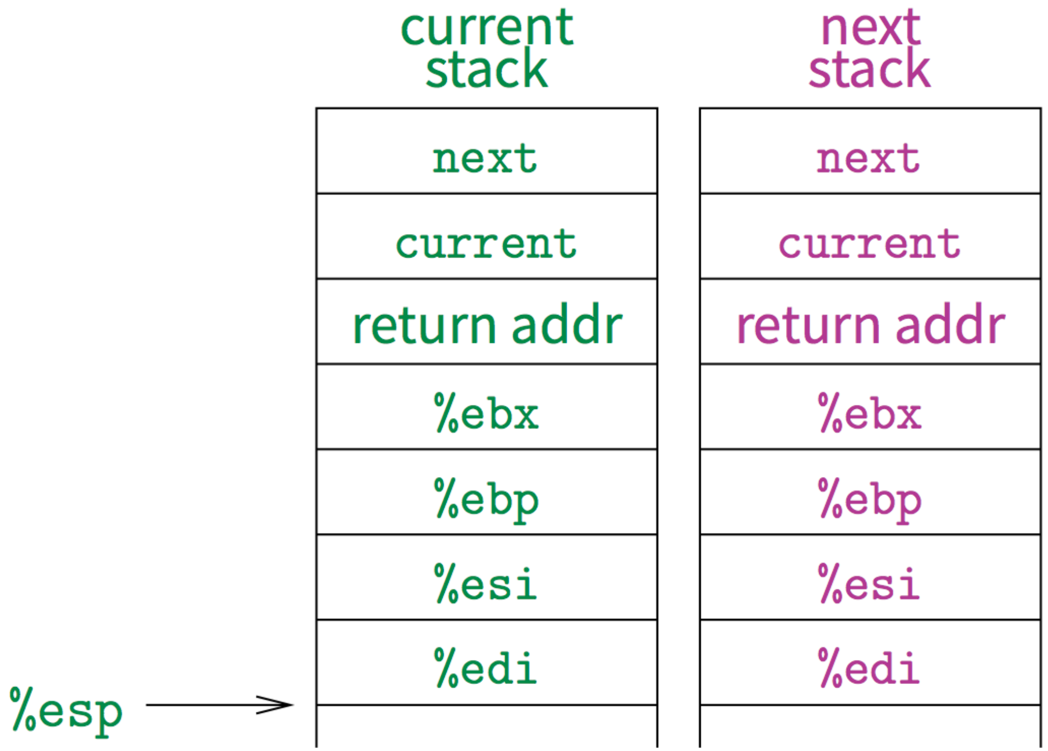
- **This is actual code from Pintos** `switch.S` **(slightly reformatted)**
  - See Thread Switching in documentation

i386 switch_threads



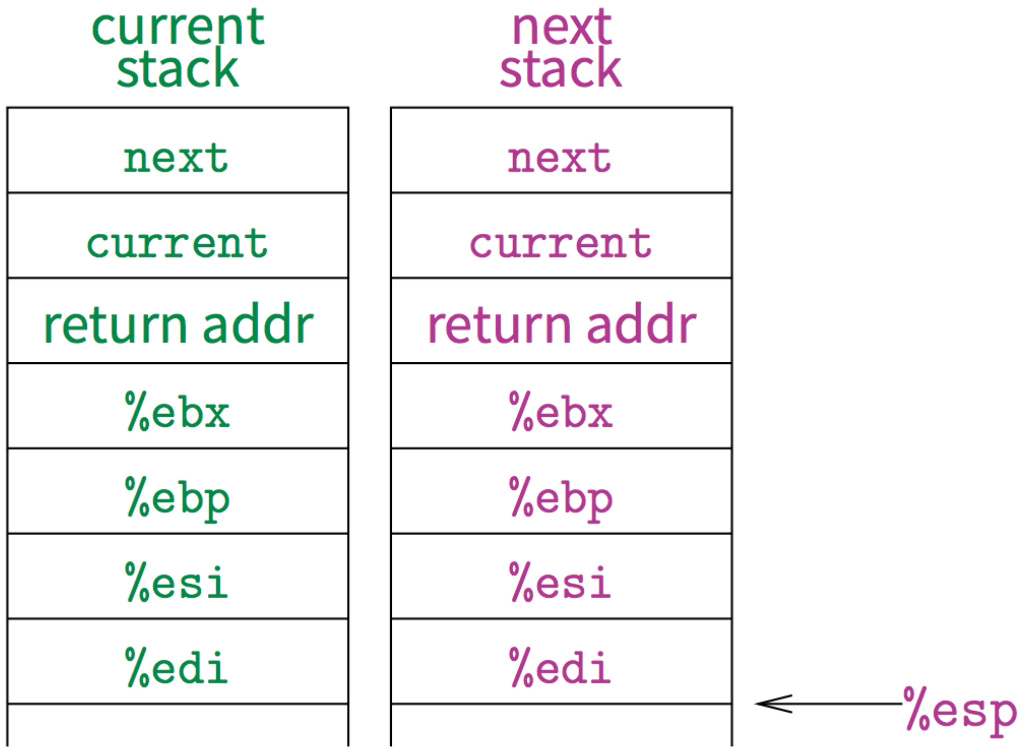
- **This is actual code from Pintos `switch.S` (slightly reformatted)**
  - See Thread Switching in documentation

i386 switch_threads



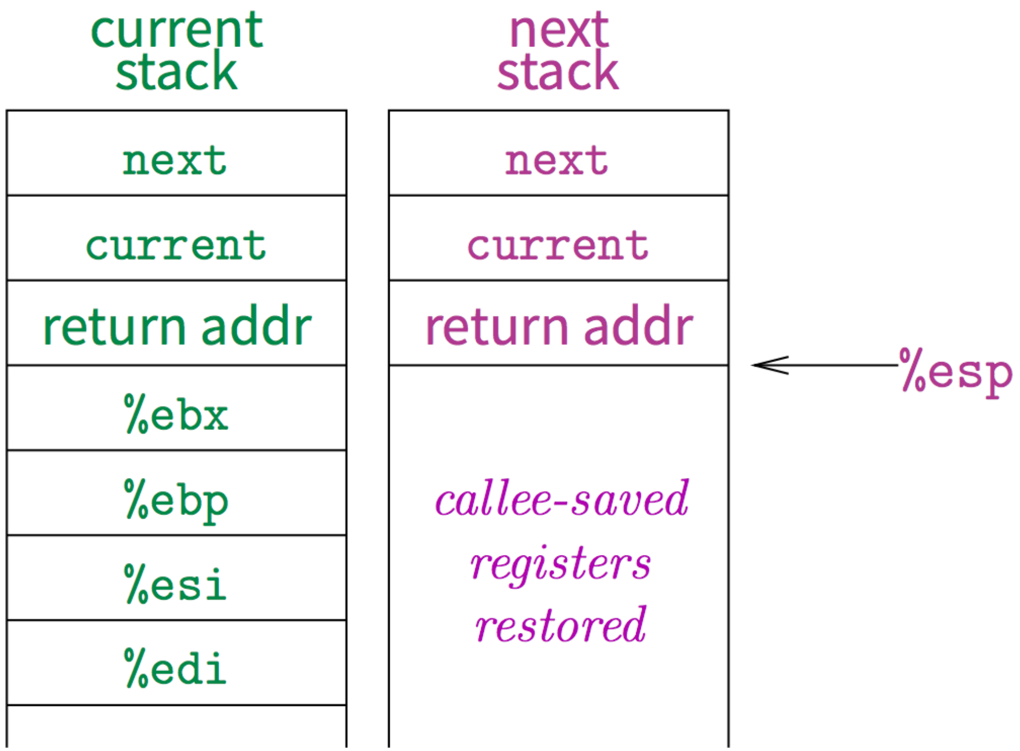
- **This is actual code from Pintos `switch.S` (slightly reformatted)**
  - See [Thread Switching] in documentation

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

Threads 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**