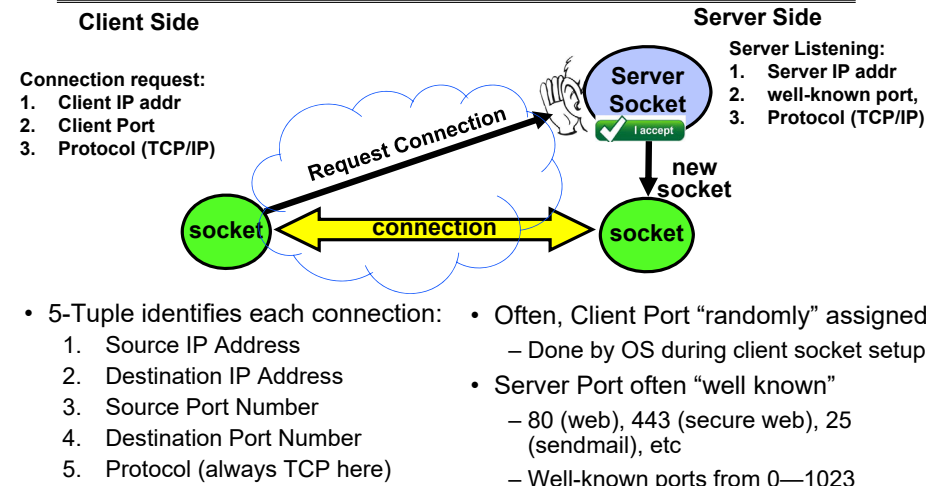


CS162  
 Operating Systems and  
 Systems Programming  
 Lecture 6

Synchronization 1: Concurrency

February 2<sup>nd</sup>, 2023  
 Prof. John Kubiawicz  
<http://cs162.eecs.Berkeley.edu>

Recall: Connection Setup over TCP/IP

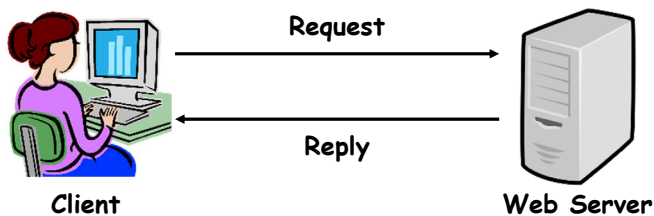


2/2/2023

Kubiawicz CS162 © UCB Spring 2023

Lec 6.2

Recall: Web Server

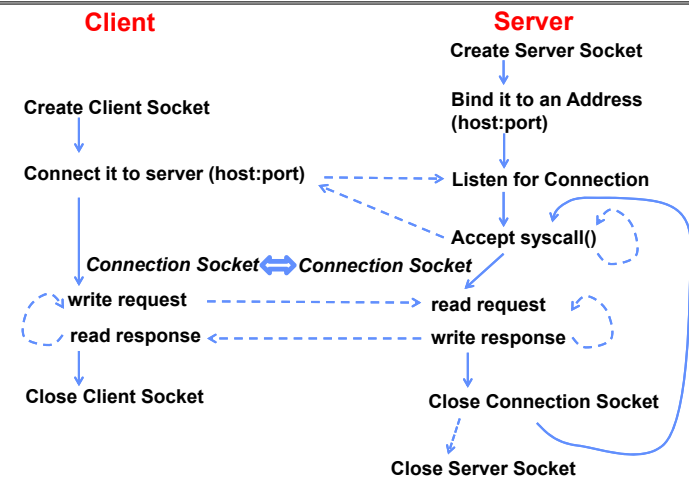


2/2/2023

Kubiawicz CS162 © UCB Spring 2023

Lec 6.3

Recall: Sockets in concept



2/2/2023

Kubiawicz CS162 © UCB Spring 2023

Lec 6.4

## Recall: Client Protocol

```
char *host_name, *port_name;

// Create a socket
struct addrinfo *server = lookup_host(host_name, port_name);
int sock_fd = socket(server->ai_family, server->ai_socktype,
                    server->ai_protocol);

// Connect to specified host and port
connect(sock_fd, server->ai_addr, server->ai_addrlen);

// Carry out Client-Server protocol
run_client(sock_fd);

/* Clean up on termination */
close(sock_fd);
```

2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.5

## Recall Client-Side: Getting the Server Address

```
struct addrinfo *lookup_host(char *host_name, char *port) {
    struct addrinfo *server;
    struct addrinfo hints;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;      /* Includes AF_INET and AF_INET6 */
    hints.ai_socktype = SOCK_STREAM; /* Essentially TCP/IP */

    int rv = getaddrinfo(host_name, port_name,
                        &hints, &server);

    if (rv != 0) {
        printf("getaddrinfo failed: %s\n", gai_strerror(rv));
        return NULL;
    }
    return server;
}
```

2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.6

## Recall: Server Protocol (v1)

```
// Create socket to listen for client connections
char *port_name;
struct addrinfo *server = setup_address(port_name);
int server_socket = socket(server->ai_family,
                          server->ai_socktype, server->ai_protocol);

// Bind socket to specific port
bind(server_socket, server->ai_addr, server->ai_addrlen);
// Start listening for new client connections
listen(server_socket, MAX_QUEUE);

while (1) {
    // Accept a new client connection, obtaining a new socket
    int conn_socket = accept(server_socket, NULL, NULL);
    serve_client(conn_socket);
    close(conn_socket);
}
close(server_socket);
```

2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.7

## Recall: Server Address: Itself (wildcard IP), Passive

```
struct addrinfo *setup_address(char *port) {
    struct addrinfo *server;
    struct addrinfo hints;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;      /* Includes AF_INET and AF_INET6 */
    hints.ai_socktype = SOCK_STREAM; /* Essentially TCP/IP */
    hints.ai_flags = AI_PASSIVE;     /* Set up for server socket */

    int rv = getaddrinfo(NULL, port, &hints, &server); /* No address! (any local IP) */
    if (rv != 0) {
        printf("getaddrinfo failed: %s\n", gai_strerror(rv));
        return NULL;
    }
    return server;
}
```

- Accepts any connections on the specified port

2/2/2023

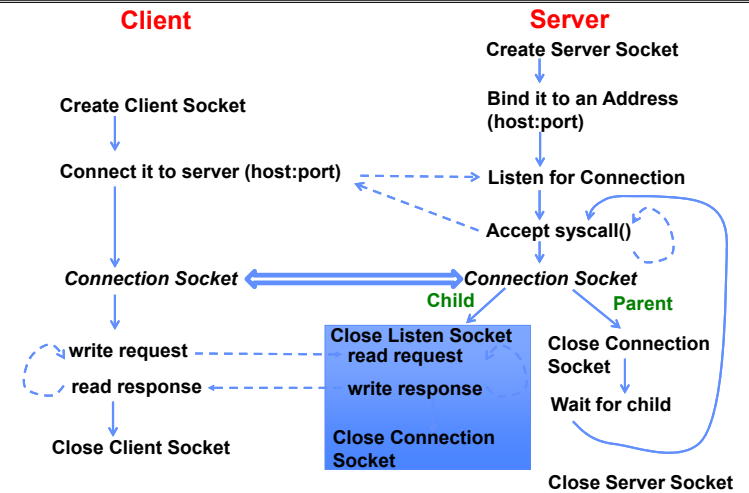
Kubiatowicz CS162 © UCB Spring 2023

Lec 6.8

## How Could the Server Protect Itself?

- Handle each connection in a separate process
  - This will mean that the logic serving each request will be “sandboxed” away from the main server process
- In the following code, keep in mind:
  - `fork()` will duplicate *all* of the parent’s file descriptors (i.e. pointers to sockets!)
  - We keep control over accepting new connections in the parent
  - New child connection for each remote client

## Sockets With Protection (each connection has own process)



## Server Protocol (v2)

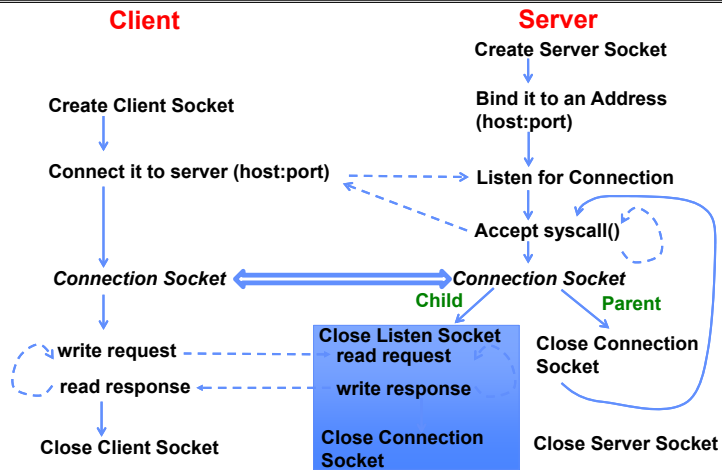
```

// Socket setup code elided...
listen(server_socket, MAX_QUEUE);
while (1) {
    // Accept a new client connection, obtaining a new socket
    int conn_socket = accept(server_socket, NULL, NULL);
    pid_t pid = fork();
    if (pid == 0) {
        close(server_socket);
        serve_client(conn_socket);
        close(conn_socket);
        exit(0);
    } else {
        close(conn_socket);
        wait(NULL);
    }
}
close(server_socket);
    
```

## How to make a Concurrent Server

- So far, in the server:
  - Listen will queue requests
  - Buffering present elsewhere
  - But server *waits* for each connection to terminate before servicing the next
    - » This is the standard shell pattern
- A concurrent server can handle and service a new connection before the previous client disconnects
  - Simple – just don’t wait in parent!
  - Perhaps not so simple – multiple child processes better not have data races with one another through file system/etc!

## Sockets With Protection and Concurrency



2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.13

## Server Protocol (v3)

```
// Socket setup code elided...
listen(server_socket, MAX_QUEUE);
while (1) {
    // Accept a new client connection, obtaining a new socket
    int conn_socket = accept(server_socket, NULL, NULL);
    pid_t pid = fork();
    if (pid == 0) {
        close(server_socket);
        serve_client(conn_socket);
        close(conn_socket);
        exit(0);
    } else {
        close(conn_socket);
        //wait(NULL);
    }
}
close(server_socket);
```

2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.14

## Faster Concurrent Server (without Protection)

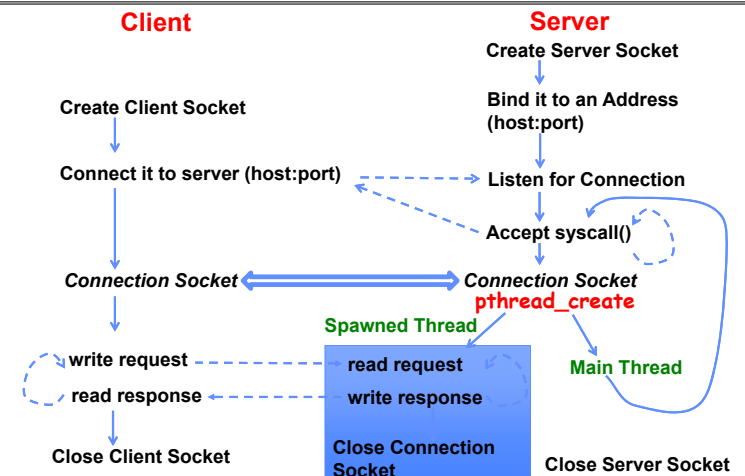
- Spawn a new *thread* to handle each connection
  - Lower overhead spawning process (less to do)
- Main *thread* initiates new client connections without waiting for previously spawned threads
- Why give up the protection of separate processes?
  - More efficient to create new threads
  - More efficient to switch between threads
- Even more potential for data races (need synchronization?)
  - Through shared memory structures
  - Through file system

2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.15

## Sockets with Concurrency, without Protection



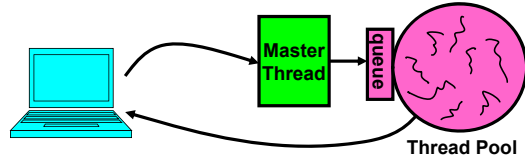
2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.16

## Thread Pools: More Later!

- Problem with previous version: Unbounded Threads
  - When web-site becomes too popular – throughput sinks
- Instead, allocate a bounded “pool” of worker threads, representing the maximum level of multiprogramming



```

master() {
    allocThreads(worker, queue);
    while(TRUE) {
        con=AcceptCon();
        Enqueue(queue, con);
        wakeUp(queue);
    }
}

worker(queue) {
    while(TRUE) {
        con=Dequeue(queue);
        if (con==null)
            sleepOn(queue);
        else
            ServiceWebPage(con);
    }
}
    
```

2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.17

## Administrivia

- Project 1 in full swing! Released Yesterday!
  - We expect that your design document will give intuitions behind your designs, not just a dump of pseudo-code
  - Think of this you are in a company and your TA is your manager
- Paradox: need code for design document?
  - Not full code, just enough prove you have thought through complexities of design
- Should be attending your permanent discussion section!
  - Discussion section attendance is mandatory, but don't come if sick!!
    - » We have given a mechanism to make up for missed sections—see EdStem
- Midterm 1: February 16<sup>th</sup>, 7-9PM (Two weeks from today!)
  - Fill out conflict request by tomorrow!

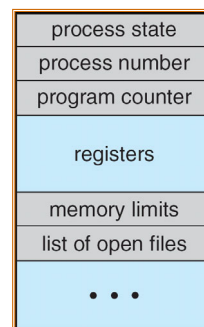
2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.18

## Recall: The Process Control Block

- Kernel represents each process as a process control block (PCB)
  - Status (running, ready, blocked, ...)
  - Register state (when not ready)
  - Process ID (PID), User, Executable, Priority, ...
  - Execution time, ...
  - Memory space, translation, ...
- Kernel *Scheduler* maintains a data structure containing the PCBs
  - Give out CPU to different processes
  - This is a Policy Decision
- Give out non-CPU resources
  - Memory/I/O
  - Another policy decision



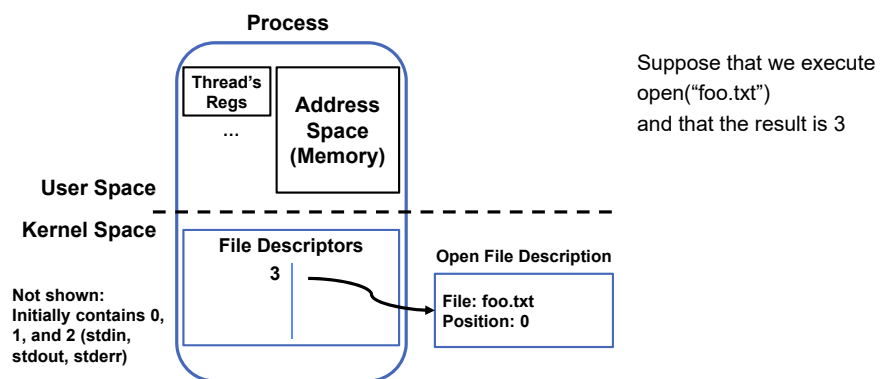
Process Control Block

2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.19

## Process-Specific File Descriptor Table inside Kernel

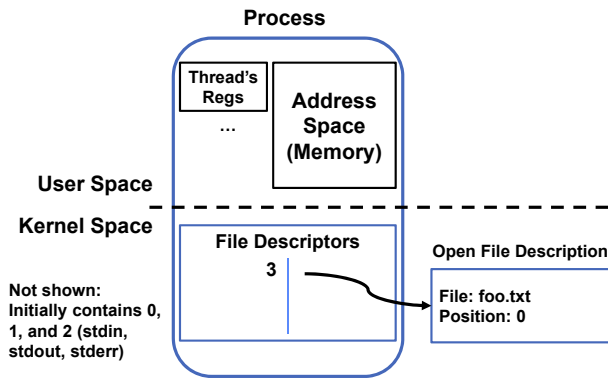


2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.20

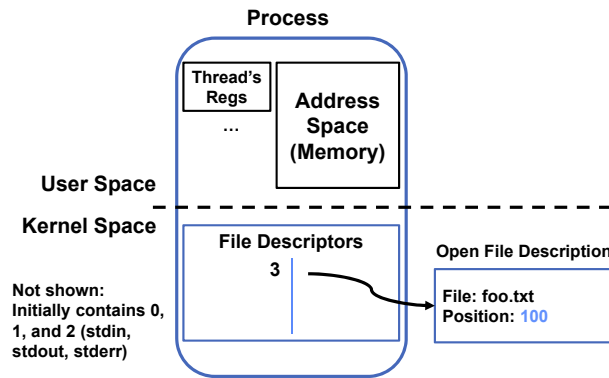
## Process-Specific File Descriptor Table inside Kernel



Suppose that we execute `open("foo.txt")` and that the result is 3

Next, suppose that we execute `read(3, buf, 100)` and that the result is 100

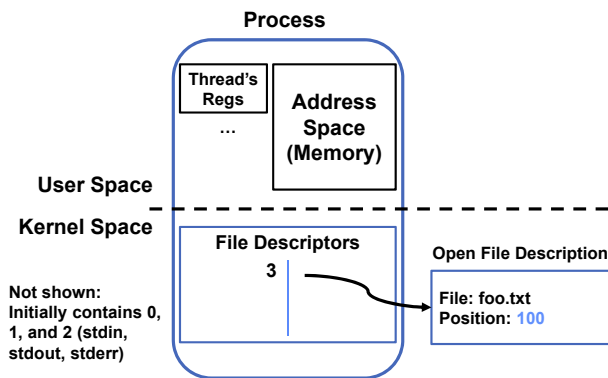
## Process-Specific File Descriptor Table inside Kernel



Suppose that we execute `open("foo.txt")` and that the result is 3

Next, suppose that we execute `read(3, buf, 100)` and that the result is 100

## Process-Specific File Descriptor Table inside Kernel

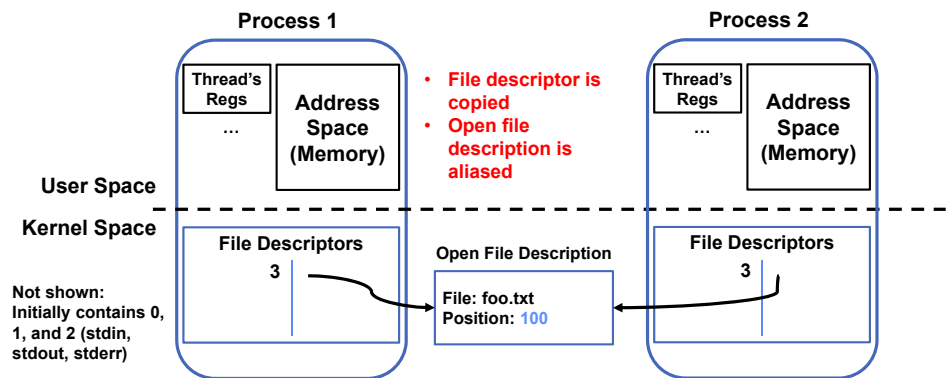


Suppose that we execute `open("foo.txt")` and that the result is 3

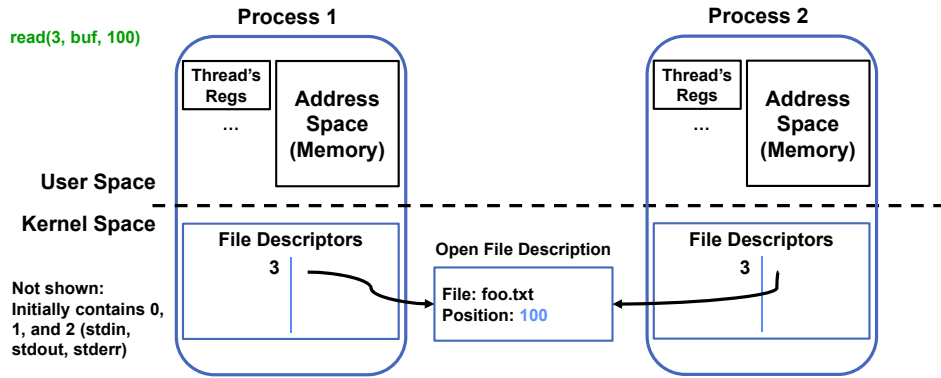
Next, suppose that we execute `read(3, buf, 100)` and that the result is 100

Finally, suppose that we execute `close(3)`

## Instead of Closing, let's fork(!)



## Open File Description is *Aliased*

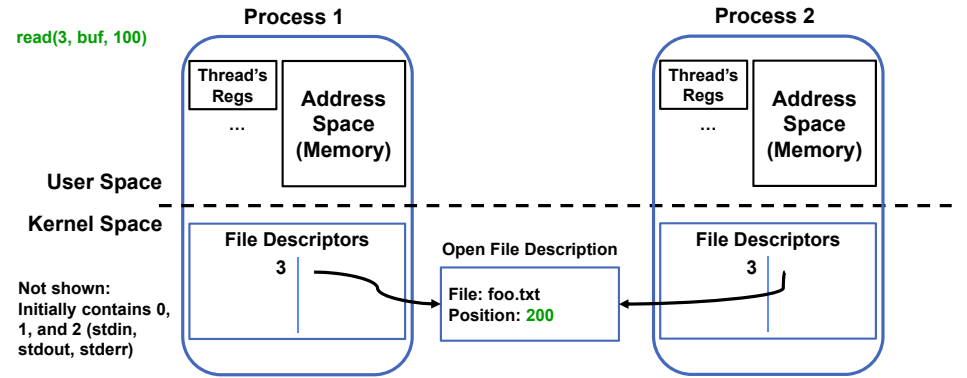


2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.25

## Open File Description is *Aliased*

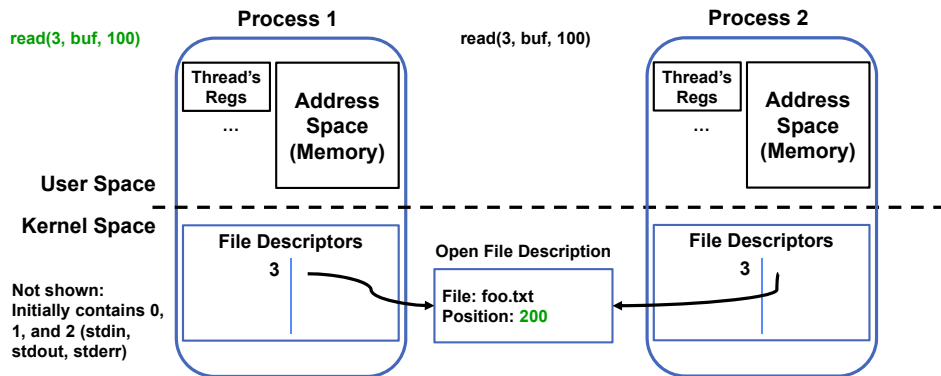


2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.26

## Open File Description is *Aliased*

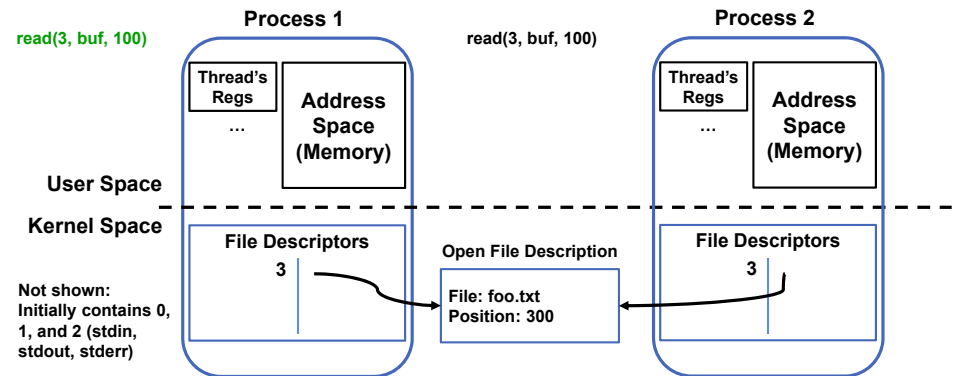


2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.27

## Open File Description is *Aliased*

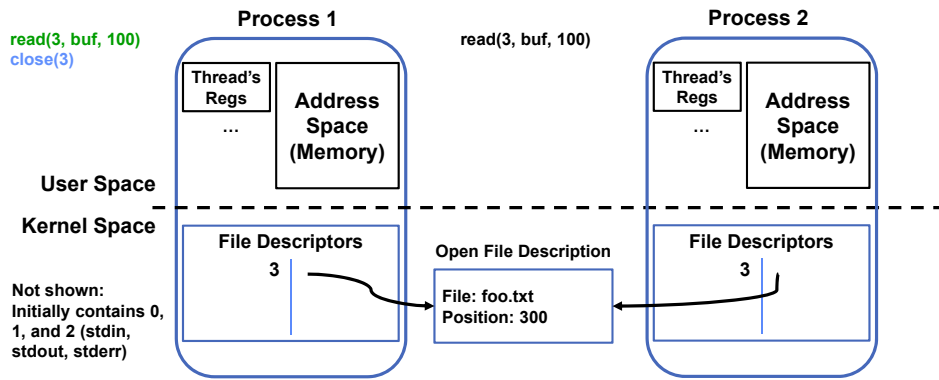


2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.28

## File Descriptor is Copied

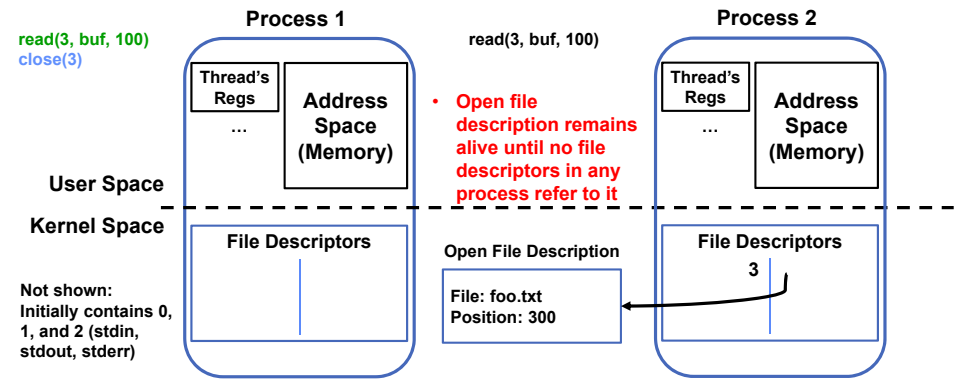


2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.29

## File Descriptor is Copied



2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.30

## Why is Aliasing the Open File Description a Good Idea?

- It allows for *shared resources* between processes

2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.31

## Example: Shared Terminal Emulator

- When you `fork()` a process, the parent's and child's `printf` outputs go to the same terminal

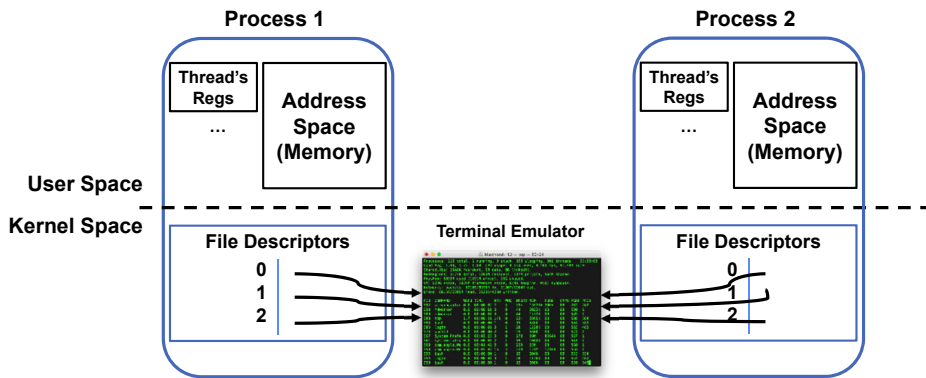
2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.32



## Example: Shared Terminal Emulator

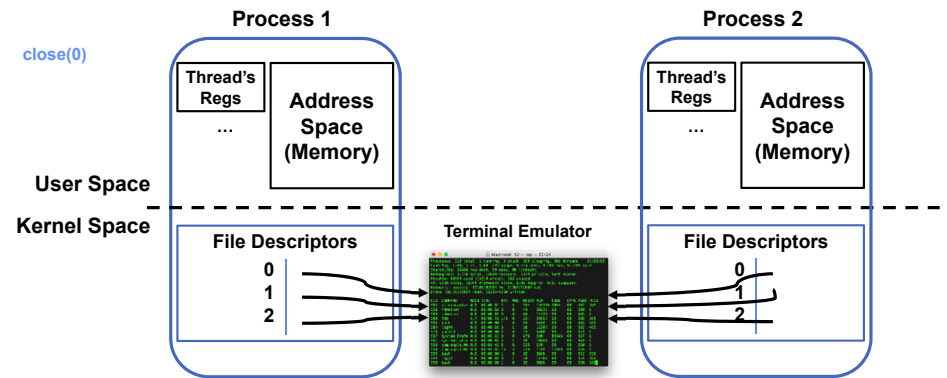


2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.33

## Example: Shared Terminal Emulator

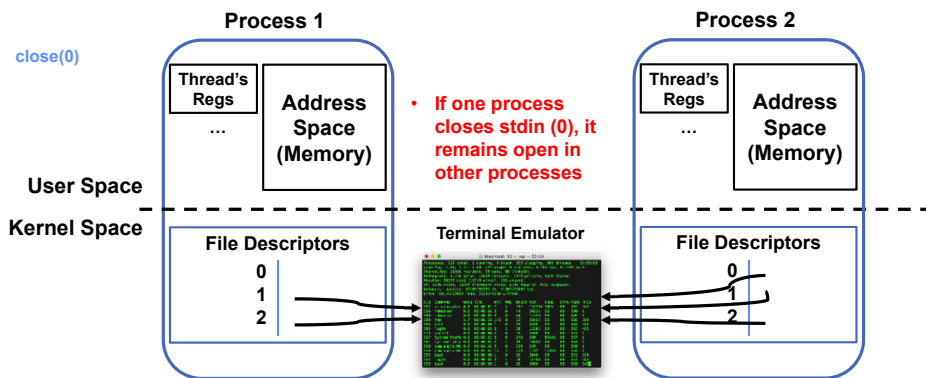


2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.34

## Example: Shared Terminal Emulator



2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.35

## Other Examples

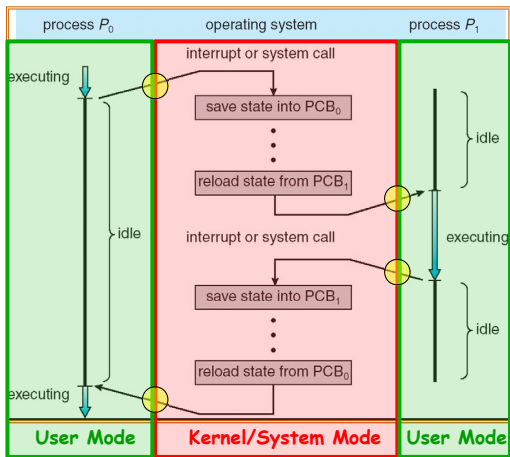
- Shared network connections after `fork()`
  - Allows handling each connection in a separate process
  - We saw this in our Webserver examples
- Pipes – channel for communication
  - `int pipe(int pipefd[2]); /* Create array of two file descriptors */`
  - Writes to `pipefd[1]` can be read from `pipefd[0]`
  - Useful for interprocess communication:
    - » after `fork()`, both parent and child can communicate (one can read what other one writes)
  - And in writing a shell (Homework 2)

2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.36

## Recall: CPU Switch From Process A to Process B

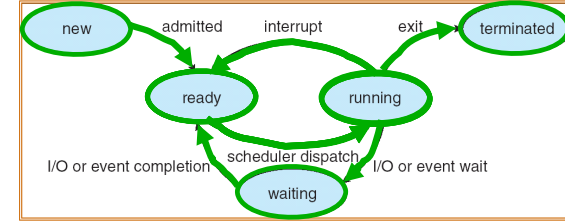


2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.37

## Lifecycle of a Process



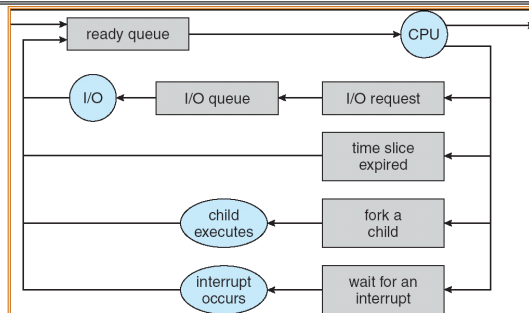
- As a process executes, it changes state:
  - **new**: The process is being created
  - **ready**: The process is waiting to run
  - **running**: Instructions are being executed
  - **waiting**: Process waiting for some event to occur
  - **terminated**: The process has finished execution

2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.38

## Process Scheduling



- PCBs move from queue to queue as they change state
  - Decisions about which order to remove from queues are **Scheduling** decisions
  - Many algorithms possible (few weeks from now)

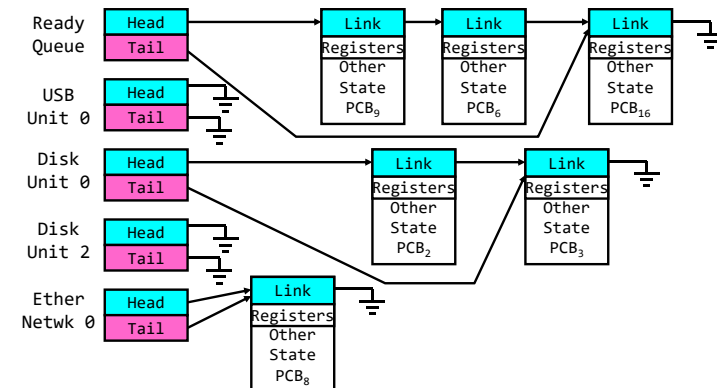
2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.39

## Ready Queue And Various I/O Device Queues

- Process not running  $\Rightarrow$  PCB is in some scheduler queue
  - Separate queue for each device/signal/condition
  - Each queue can have a different scheduler policy



2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.40

## Modern Process with Threads

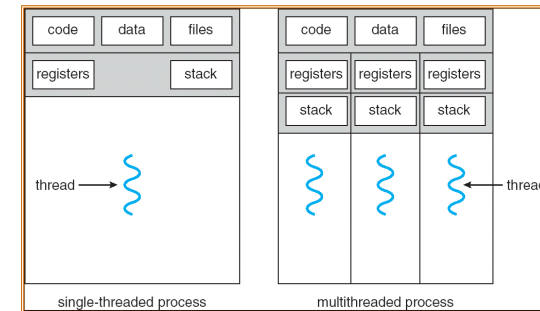
- Thread: *a sequential execution stream within process* (Sometimes called a “**Lightweight process**”)
  - Process still contains a single Address Space
  - No protection between threads
- Multithreading: *a single program made up of a number of different concurrent activities*
  - Sometimes called multitasking, as in Ada ...
- Why separate the concept of a thread from that of a process?
  - Discuss the “thread” part of a process (concurrency)
  - Separate from the “address space” (protection)
  - Heavyweight Process  $\equiv$  Process with one thread

2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.41

## Single and Multithreaded Processes



- Threads encapsulate concurrency: “Active” component
- Address spaces encapsulate protection: “Passive” part
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?

2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.42

## Thread State

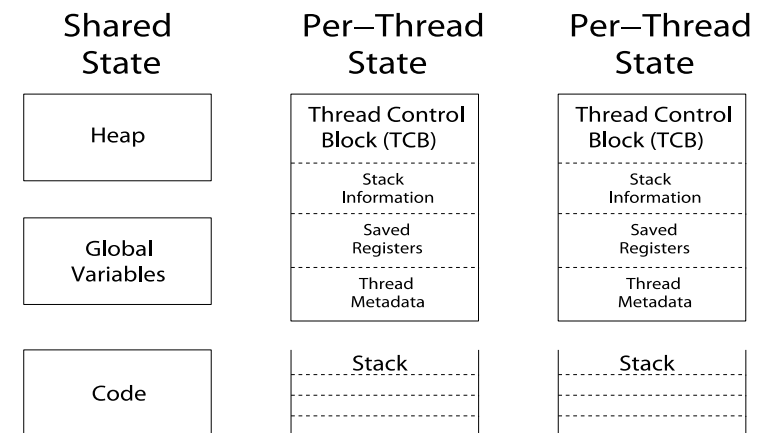
- State shared by all threads in process/address space
  - Content of memory (global variables, heap)
  - I/O state (file descriptors, network connections, etc)
- State “private” to each thread
  - Kept in **TCB  $\equiv$  Thread Control Block**
  - CPU registers (including, program counter)
  - Execution stack – what is this?
- Execution Stack
  - Parameters, temporary variables
  - Return PCs are kept while called procedures are executing

2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.43

## Shared vs. Per-Thread State



2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.44

## Example: Execution Stack Example

```
A(int tmp) {
A:   if (tmp<2)
A+1:   B();
A+2:   printf(tmp);
      }
      B() {
B:     C();
B+1:  }
      C() {
C:     A(2);
C+1:  }
      A(1);
exit: }
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

## Execution Stack Example

```
A(int tmp) {
A:   if (tmp<2)
A+1:   B();
A+2:   printf(tmp);
      }
      B() {
B:     C();
B+1:  }
      C() {
C:     A(2);
C+1:  }
      A(1);
exit: }
```

Stack  
Pointer →

```
A: tmp=1
   ret=exit
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

## Execution Stack Example

```
A(int tmp) {
A:   if (tmp<2)
A+1:   B();
A+2:   printf(tmp);
      }
      B() {
B:     C();
B+1:  }
      C() {
C:     A(2);
C+1:  }
      A(1);
exit: }
```

Stack  
Pointer →

```
A: tmp=1
   ret=exit
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

## Execution Stack Example

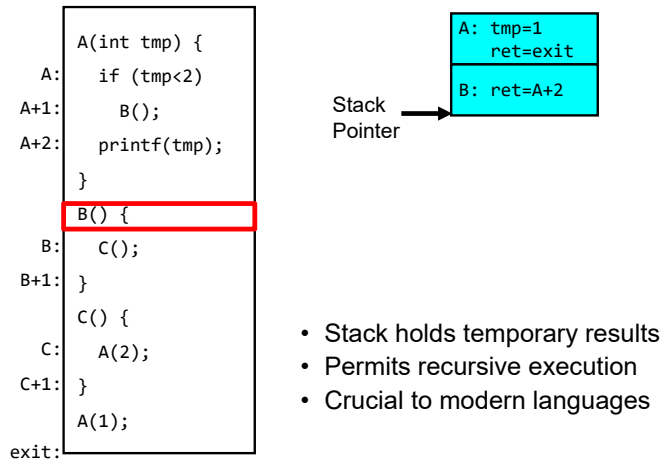
```
A(int tmp) {
A:   if (tmp<2)
A+1:   B();
A+2:   printf(tmp);
      }
      B() {
B:     C();
B+1:  }
      C() {
C:     A(2);
C+1:  }
      A(1);
exit: }
```

Stack  
Pointer →

```
A: tmp=1
   ret=exit
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

## Execution Stack Example

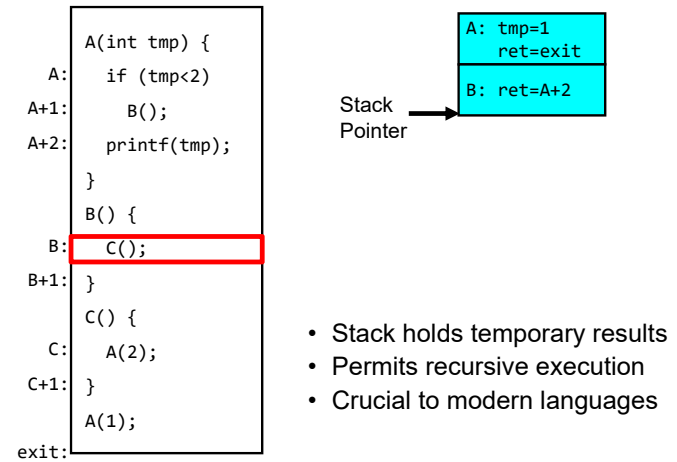


2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.49

## Execution Stack Example

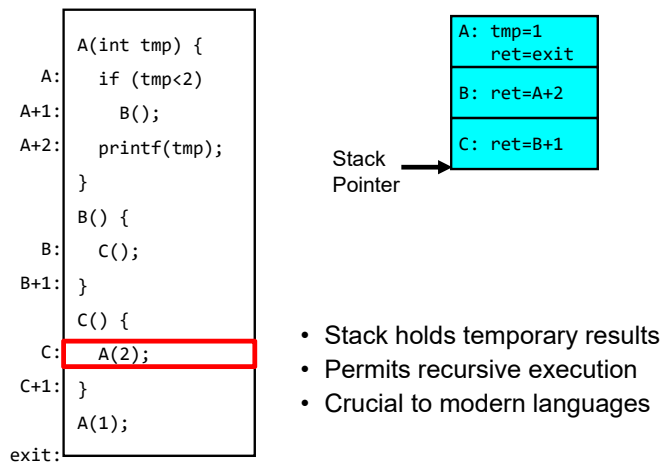


2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.50

## Execution Stack Example

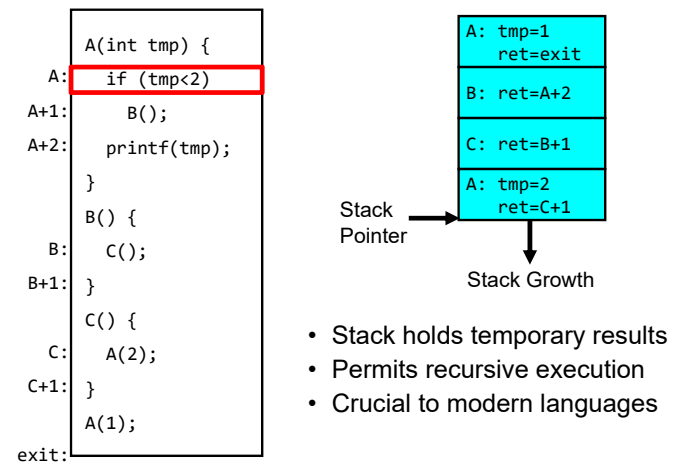


2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.51

## Execution Stack Example

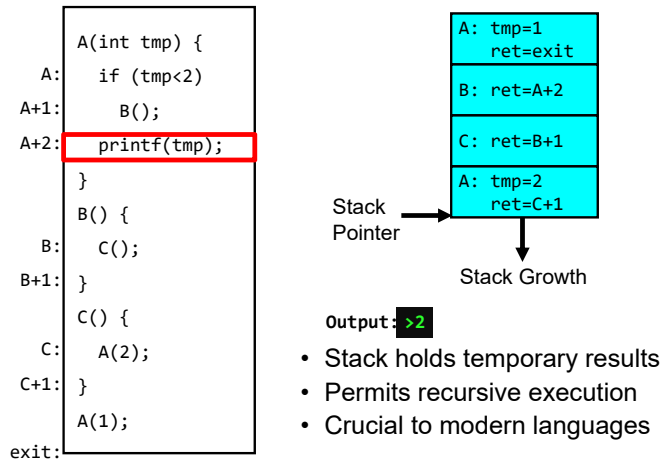


2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.52

## Execution Stack Example

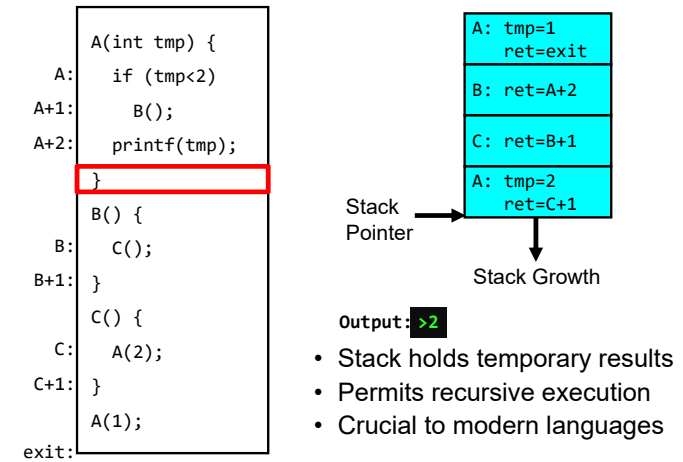


2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.53

## Execution Stack Example

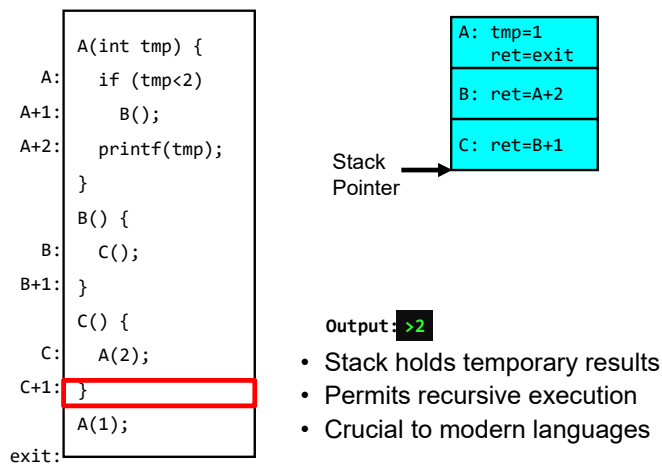


2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.54

## Execution Stack Example

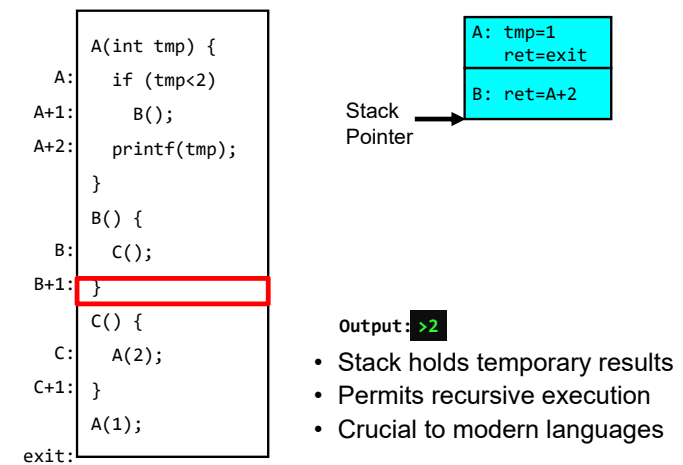


2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.55

## Execution Stack Example

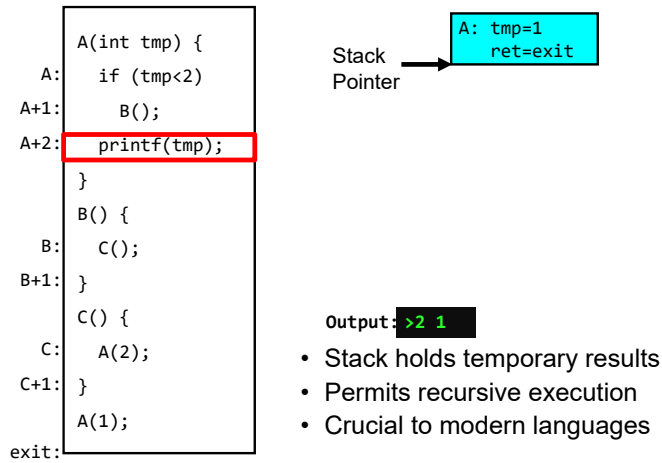


2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.56

## Execution Stack Example

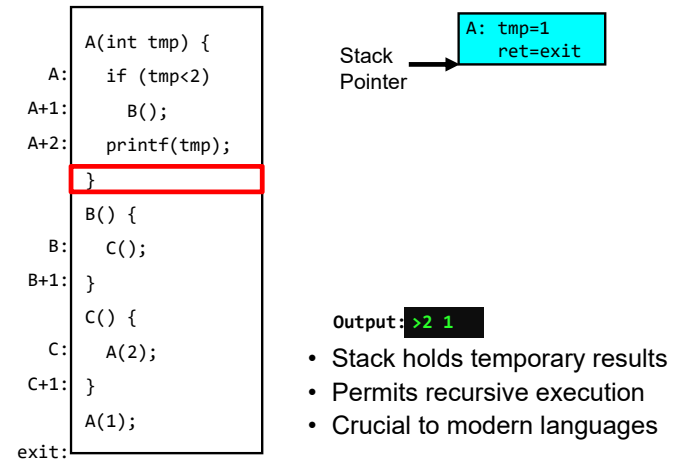


2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.57

## Execution Stack Example

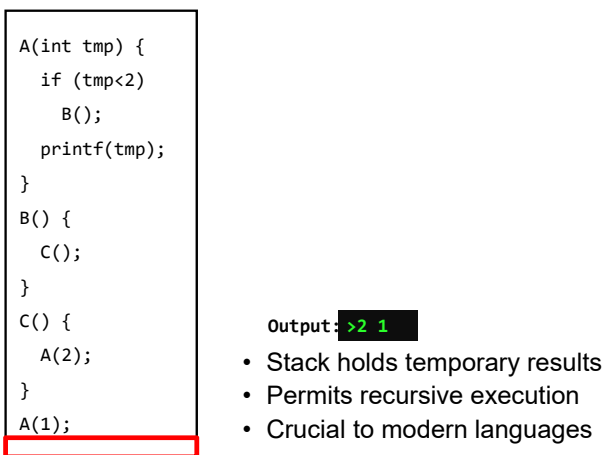


2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.58

## Execution Stack Example

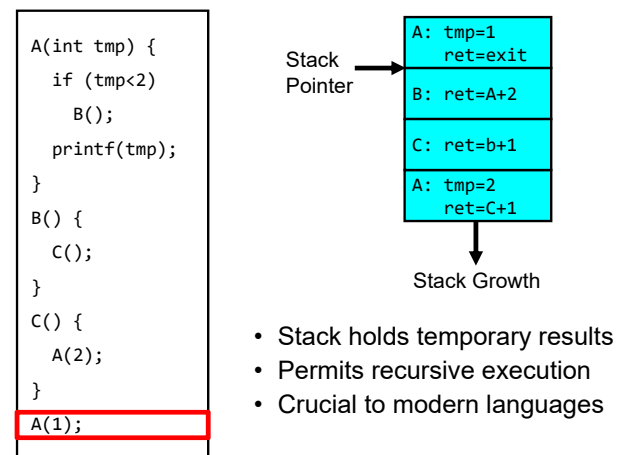


2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.59

## Execution Stack Example



2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.60

## Motivational Example for Threads

- Imagine the following C program:

```
main() {
    ComputePI("pi.txt");
    PrintClassList("classlist.txt");
}
```

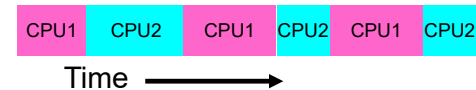
- What is the behavior here?
  - Program would never print out class list
  - Why? ComputePI would never finish

## Use of Threads

- Version of program with Threads (loose syntax):

```
main() {
    ThreadFork(ComputePI, "pi.txt");
    ThreadFork(PrintClassList, "classlist.txt");
}
```

- What does ThreadFork() do?
  - Start independent thread running given procedure
- What is the behavior here?
  - Now, you would actually see the class list
  - This *should* behave as if there are two separate CPUs



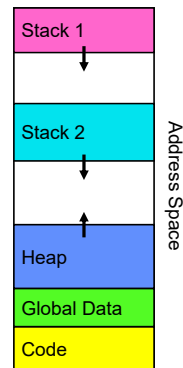
## Memory Footprint: Two-Threads

- If we stopped this program and examined it with a debugger, we would see

- Two sets of CPU registers
- Two sets of Stacks

- Questions:

- How do we position stacks relative to each other?
- What maximum size should we choose for the stacks?
- What happens if threads violate this?
- How might you catch violations?



## OS Library API for Threads: *pthread*s

**pThreads: POSIX standard for thread programming**  
**[POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)]**

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                  void *(*start_routine)(void*), void *arg);
```

- thread is created executing *start\_routine* with *arg* as its sole argument.
- return is implicit call to *pthread\_exit*

```
void pthread_exit(void *value_ptr);
```

- terminates the thread and makes *value\_ptr* available to any successful join

```
int pthread_yield();
```

- causes the calling thread to yield the CPU to other threads

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- suspends execution of the calling thread until the target *thread* terminates.
- On return with a non-NULL *value\_ptr* the value passed to *pthread\_exit()* by the terminating thread is made available in the location referenced by *value\_ptr*.

**prompt% man pthread**

**<https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html>**



## The Core of Concurrency: the Dispatch Loop

- Conceptually, the scheduling loop of the operating system looks as follows:

```
Loop {
  RunThread();
  ChooseNextThread();
  SaveStateOfCPU(curTCB);
  LoadStateOfCPU(newTCB);
}
```

- This is an *infinite* loop
  - One could argue that this is all that the OS does
- Should we ever exit this loop???
  - When would that be?

2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.65

## Running a thread

Consider first portion: RunThread()

- How do I run a thread?
  - Load its state (registers, PC, stack pointer) into CPU
  - Load environment (virtual memory space, etc)
  - Jump to the PC
- How does the dispatcher get control back?
  - Internal events: thread returns control voluntarily
  - External events: thread gets *preempted*

2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.66

## Internal Events

- Blocking on I/O
  - The act of requesting I/O implicitly yields the CPU
- Waiting on a “signal” from other thread
  - Thread asks to wait and thus yields the CPU
- Thread executes a `yield()`
  - Thread volunteers to give up CPU

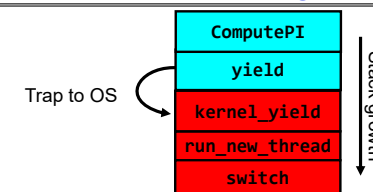
```
computePI() {
  while(TRUE) {
    ComputeNextDigit();
    yield();
  }
}
```

2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.67

## Stack for Yielding Thread



- How do we run a new thread?

```
run_new_thread() {
  newThread = PickNewThread();
  switch(curThread, newThread);
  ThreadHouseKeeping(); /* Do any cleanup */
}
```

- How does dispatcher switch to a new thread?
  - Save anything next thread may trash: PC, regs, stack pointer
  - Maintain isolation for each thread

2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.68

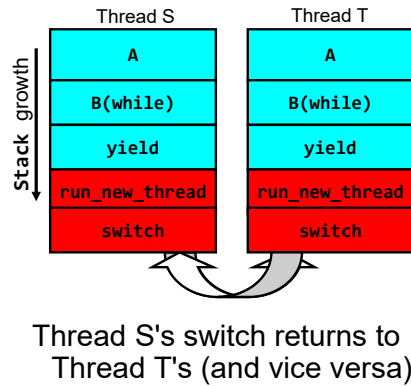
## What Do the Stacks Look Like?

- Consider the following code blocks:

```

proc A() {
    B();
}
proc B() {
    while(TRUE) {
        yield();
    }
}
    
```

- Suppose we have 2 threads:
  - Threads S and T



## Saving/Restoring state (often called "Context Switch")

```

Switch(tCur, tNew) {
    /* Unload old thread */
    TCB[tCur].regs.r7 = CPU.r7;
    ...
    TCB[tCur].regs.r0 = CPU.r0;
    TCB[tCur].regs.sp = CPU.sp;
    TCB[tCur].regs.retpc = CPU.retpc; /*return addr*/

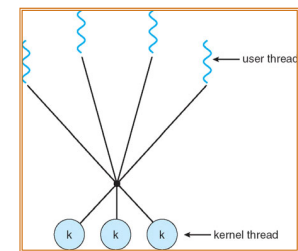
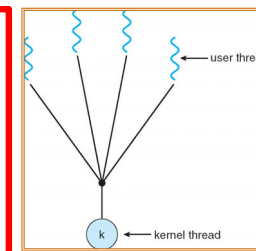
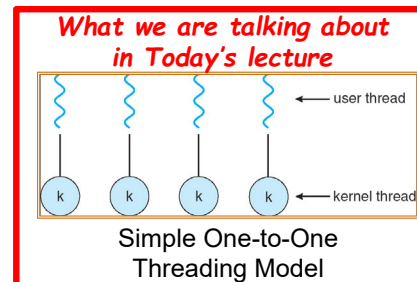
    /* Load and execute new thread */
    CPU.r7 = TCB[tNew].regs.r7;
    ...
    CPU.r0 = TCB[tNew].regs.r0;
    CPU.sp = TCB[tNew].regs.sp;
    CPU.retpc = TCB[tNew].regs.retpc;
    return; /* Return to CPU.retpc */
}
    
```

## Switch Details (continued)

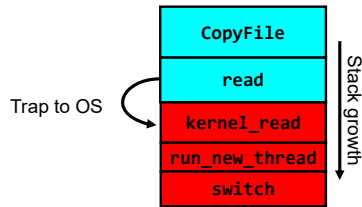
- What if you make a mistake in implementing switch?
  - Suppose you forget to save/restore register 32
  - Get intermittent failures depending on when context switch occurred and whether new thread uses register 32
  - System will give wrong result without warning
- Can you devise an exhaustive test to test switch code?
  - No! Too many combinations and inter-leavings
- Cautionary tale:
  - For speed, Topaz kernel saved one instruction in switch()
  - Carefully documented! Only works as long as kernel size < 1MB
  - What happened?
    - » Time passed, People forgot
    - » Later, they added features to kernel (no one removes features!)
    - » Very weird behavior started happening
  - Moral of story: Design for simplicity

## How expensive is context switching?

- Switching between threads in same process similar to switching between threads in different processes, but *much cheaper*:
  - No need to change address space
- Some numbers from Linux:
  - Frequency of context switch: 10-100ms
  - Switching between processes: 3-4 μsec.
  - Switching between threads: 100 ns
- Even cheaper: switch threads (using "yield") in user-space!



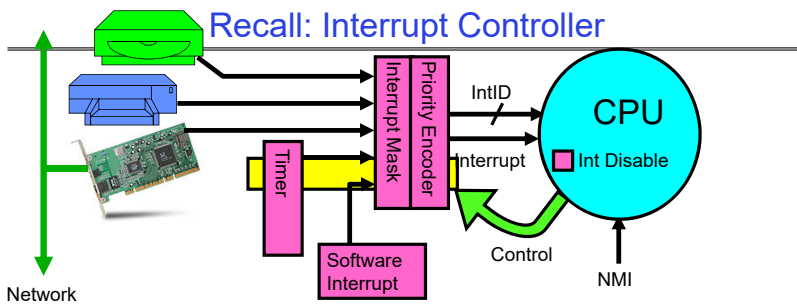
## What happens when thread blocks on I/O?



- What happens when a thread requests a block of data from the file system?
  - User code invokes a system call
  - Read operation is initiated
  - Run new thread/switch
- Thread communication similar
  - Wait for Signal/Join
  - Networking

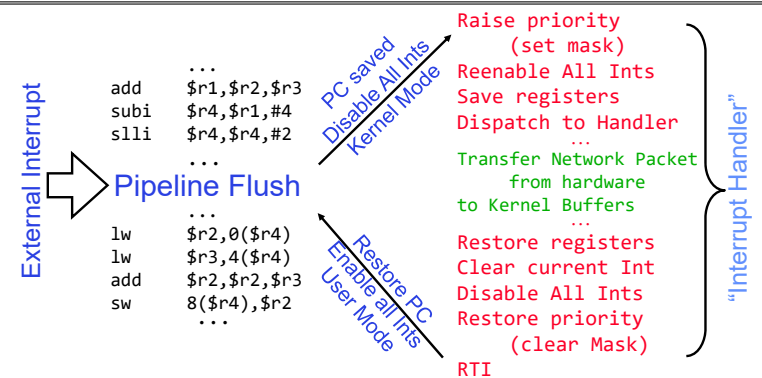
## External Events

- What happens if thread never does any I/O, never waits, and never yields control?
  - Could the ComputePI program grab all resources and never release the processor?
    - » What if it didn't print to console?
  - Must find way that dispatcher can regain control!
- Answer: utilize external events
  - Interrupts: signals from hardware or software that stop the running code and jump to kernel
  - Timer: like an alarm clock that goes off every some milliseconds
- If we make sure that external events occur frequently enough, can ensure dispatcher runs



- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Interrupt identity specified with ID line
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software
- CPU can disable all interrupts with internal flag
- Non-Maskable Interrupt line (NMI) can't be disabled

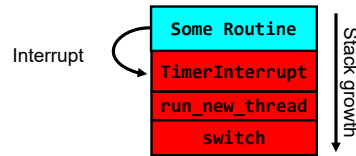
## Example: Network Interrupt



- An interrupt is a hardware-invoked context switch
  - No separate step to choose what to run next
  - Always run the interrupt handler immediately

## Use of Timer Interrupt to Return Control

- Solution to our dispatcher problem
  - Use the timer interrupt to force scheduling decisions



- Timer Interrupt routine:

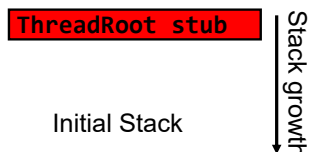
```
TimerInterrupt() {
    DoPeriodicHouseKeeping();
    run_new_thread();
}
```

## ThreadFork(): Create a New Thread

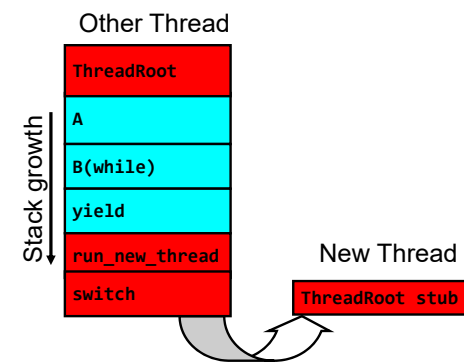
- ThreadFork() is a user-level procedure that creates a new thread and places it on ready queue
- Arguments to ThreadFork()
  - Pointer to application routine (fcnPtr)
  - Pointer to array of arguments (fcnArgPtr)
  - Size of stack to allocate
- Implementation
  - Sanity check arguments
  - Enter Kernel-mode and Sanity Check arguments again
  - Allocate new Stack and TCB
  - Initialize TCB and place on ready list (Runnable)

## How do we initialize TCB and Stack?

- Initialize Register fields of TCB
  - Stack pointer made to point at stack
  - PC return address  $\Rightarrow$  OS (asm) routine ThreadRoot()
  - Two arg registers (a0 and a1) initialized to fcnPtr and fcnArgPtr, respectively
- Initialize stack data?
  - Minimal initialization  $\Rightarrow$  setup return to go to beginning of ThreadRoot()
    - » Important part of stack frame is in registers for RISC-V (ra)
    - » X86: need to push a return address on stack
  - Think of stack frame as just before body of ThreadRoot() really gets started

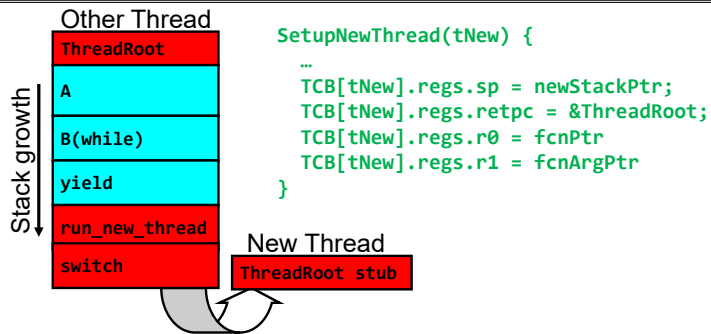


## How does Thread get started?



- Eventually, run\_new\_thread() will select this TCB and return into beginning of ThreadRoot()
  - This really starts the new thread

## How does a thread get started?



- How do we make a **new** thread?
  - Setup TCB/kernel thread to point at new user stack and ThreadRoot code
  - Put pointers to start function and args in registers or top of stack
    - » This depends heavily on the calling convention (i.e. RISC-V vs x86)
- Eventually, run\_new\_thread() will select this TCB and return into beginning of ThreadRoot()
  - This really starts the new thread

2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

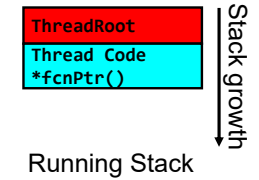
Lec 6.81

## What does ThreadRoot() look like?

- ThreadRoot() is the root for the thread routine:
 

```

ThreadRoot(fcnPTR, fcnArgPtr) {
    DoStartupHousekeeping();
    UserModeSwitch(); /* enter user mode */
    Call fcnPtr(fcnArgPtr);
    ThreadFinish();
}
            
```
- Startup Housekeeping
  - Includes things like recording start time of thread
  - Other statistics
- Stack will grow and shrink with execution of thread
- Final return from thread returns into ThreadRoot() which calls ThreadFinish()
  - ThreadFinish() wake up sleeping threads

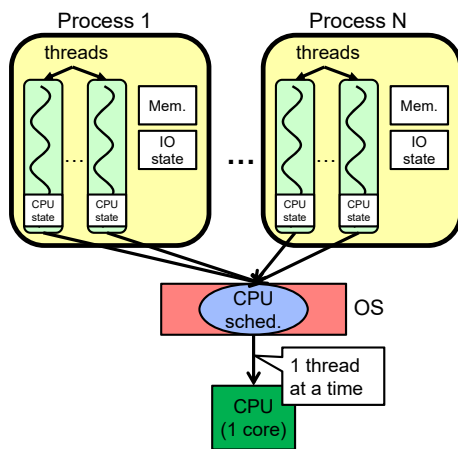


2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.82

## Processes vs. Threads: One Core



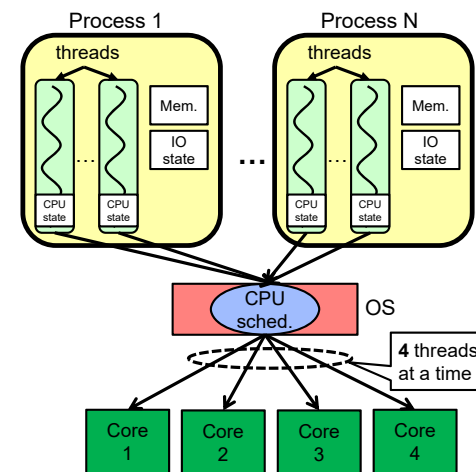
- Switch overhead:
  - Same process: **low**
  - Different proc.: **high**
- Protection
  - Same proc: **low**
  - Different proc: **high**
- Sharing overhead
  - Same proc: **low**
  - Different proc: **high**
- Parallelism: **no**

2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.83

## Processes vs. Threads: MultiCore



- Switch overhead:
  - Same process: **low**
  - Different proc.: **high**
- Protection
  - Same proc: **low**
  - Different proc: **high**
- Sharing overhead
  - Same proc: **low**
  - Different proc, simultaneous core: **medium**
  - Different proc, offloaded core: **high**
- Parallelism: **yes**

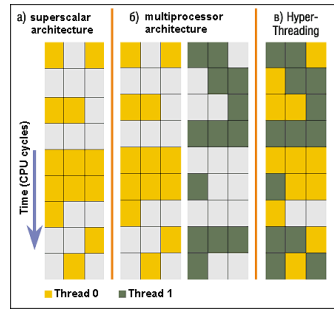
2/2/2023

Kubiatowicz CS162 © UCB Spring 2023

Lec 6.84

## Recall: Simultaneous MultiThreading/Hyperthreading

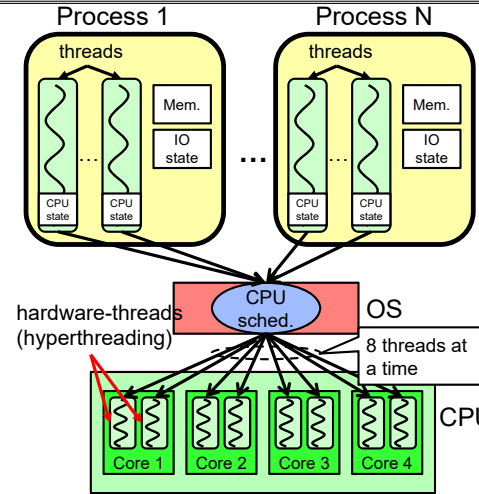
- Hardware scheduling technique
  - Superscalar processors can execute multiple instructions that are independent.
  - Hyperthreading duplicates register state to make a second “thread,” allowing more instructions to run.
- Can schedule each thread as if were separate CPU
  - But, sub-linear speedup!



Colored blocks show instructions executed

- Original technique called “Simultaneous Multithreading”
  - <http://www.cs.washington.edu/research/smt/index.html>
  - SPARC, Pentium 4/Xeon (“Hyperthreading”), Power 5

## Processes vs. Threads: Hyper-Threading



- Switch overhead between hardware-threads: *very-low* (done in hardware)
- Contention for ALUs/FPUs may hurt performance

## Threads vs Address Spaces: Options

| # threads<br>Per AS: | # of addr spaces: | One  | Many  |
|----------------------|-------------------|--|---|
| One                  |                   | MS/DOS, early Macintosh  | Traditional UNIX  |
| Many                 |                   | Embedded systems (Geoworks, VxWorks, JavaOS, etc)<br>JavaOS, Pilot(PC) | Mach, OS/2, Linux<br>Windows 10<br>Win NT to XP, Solaris, HP-UX, OS X |

- Most operating systems have either
  - One or many address spaces
  - One or many threads per address space

## Conclusion

- Recall: `open()`, `read()`, `write()`, and `close()` used for wide variety of I/O:
  - Files on disk
  - Devices (terminals, printers, etc.)
  - Regular files on disk
  - Networking (sockets)
  - Local interprocess communication (pipes, sockets)
- Processes have two parts
  - Threads (Concurrency)
  - Address Spaces (Protection)
- Stack is essential part of computation
  - Every thread has two stacks: user-level (in address space) and kernel
  - The kernel stack + support often called the “kernel thread”
- Various textbooks talk about *processes*
  - When this concerns concurrency, really talking about thread portion of a process
  - When this concerns protection, talking about address space portion of a process