# CS162
# Operating Systems and Systems Programming
# Lecture 5

## Sockets and IPC (Finished)
## Concurrency: Processes and Threads

February 1st, 2022

Prof. Anthony Joseph and John Kubiatowicz

http://cs162.eecs.Berkeley.edu

---

## Recall: Key Unix I/O Design Concepts

- Uniformity – Everything Is a File!
  – file operations, device I/O, and interprocess communication through open, read/write, close
  – Allows simple composition of programs
    » find | grep | wc …
- Open before use
  – Provides opportunity for access control and arbitration
  – Sets up the underlying machinery, i.e., data structures
- Byte-oriented
  – Even if blocks are transferred, addressing is in bytes
- Kernel buffered reads
  – Streaming and block devices looks the same, read blocks yielding processor to other task
- Kernel buffered writes
  – Completion of out-going transfer decoupled from the application, allowing it to continue
- Explicit close



---

## Recall: Low-Level vs High-Level file API

- Low-level direct use of syscall interface: `open(), read(), write(), close()`
- Opening of file returns file descriptor:
  `int myfile = open(…);`
- File descriptor only meaningful to kernel
  – Index into process (PDB) which holds pointers to kernel-level structure ("file description") describing file.
- Every `read()` or `write()` causes syscall no matter how small (could read a single byte)
- Consider loop to get 4 bytes at a time using `read()`:
  – Each iteration enters kernel for 4 bytes.

- High-level buffered access: `fopen(), fread(), fwrite(), fclose()`
- Opening of file returns ptr to FILE:
  `FILE *myfile = fopen(…);`
- FILE structure is user space contains:
  – a chunk of memory for a buffer
  – the file descriptor for the file (`fopen()` will call `open()` automatically)
- Every `fread()` or `fwrite()` filters through buffer and may not call `read()` or `write()` on every call.
- Consider loop to get 4 bytes at a time using `fread()`:
  – First call to `fread()` calls `read()` for block of bytes (say 1024). Puts in buffer and returns first 4 to user.
  – Subsequent `fread()` grab bytes from buffer



---

## Recall: Low-Level vs. High-Level File API

**Low-Level Operation:**

```
ssize_t read(…) {

    asm code … syscall # into %eax
    put args into registers %ebx, …
    special trap instruction
        Kernel:
          get args from regs
          dispatch to system func
          Do the work to read from the file
          Store return value in %eax
    get return values from regs

  Return data to caller
};
```

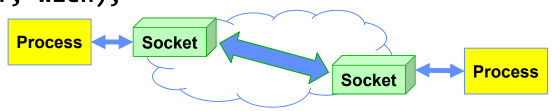
**High-Level Operation:**

```
ssize_t fread(…) {
  Check buffer for contents
  Return data to caller if available
    asm code … syscall # into %eax
    put args into registers %ebx, …
    special trap instruction
        Kernel:
          get args from regs
          dispatch to system func
          Do the work to read from the file
          Store return value in %eax
    get return values from regs
  Update buffer with excess data
  Return data to caller
};
```

## Recall: Sockets: An Endpoint for Communication

- **Key Idea:** Communication across the world looks like File I/O

```
write(wfd, wbuf, wlen);
```

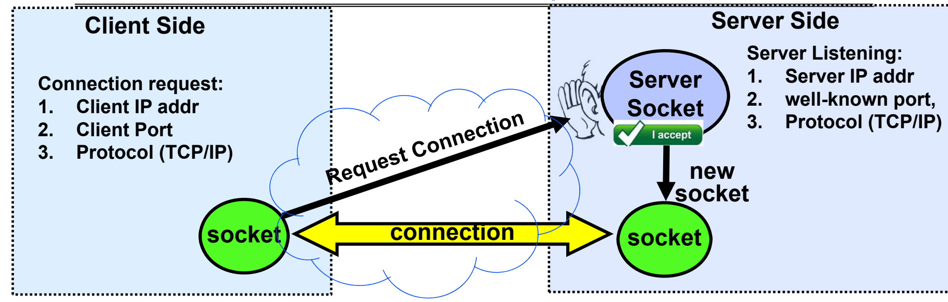



```
n = read(rfd, rbuf, rmax);
```

- Sockets: Endpoint for Communication
  – Queues to temporarily hold results
- Connection: Two Sockets Connected Over the network ⇒ IPC over network!
  – How to **open()**?
  – What is the namespace?
  – How are they connected in time?



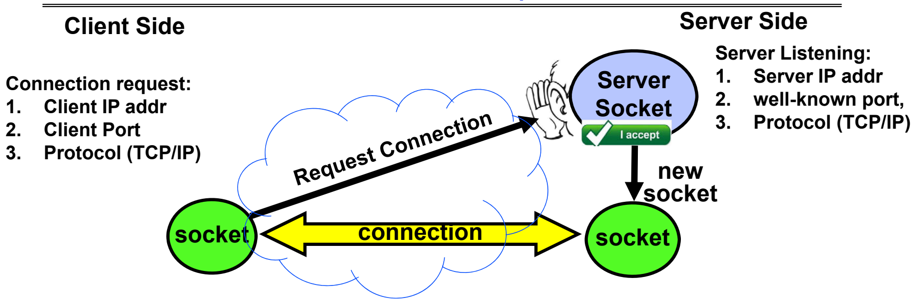
## Recall: Connection Setup over TCP/IP




- Special kind of socket: **server socket**
  – Has file descriptor
  – Can't read or write
- Two operations:
  1. **listen()**: Start allowing clients to connect
  2. **accept()**: Create a *new socket* for a *particular* client



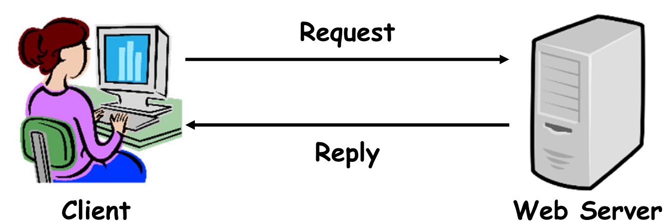
## Recall: Connection Setup over TCP/IP



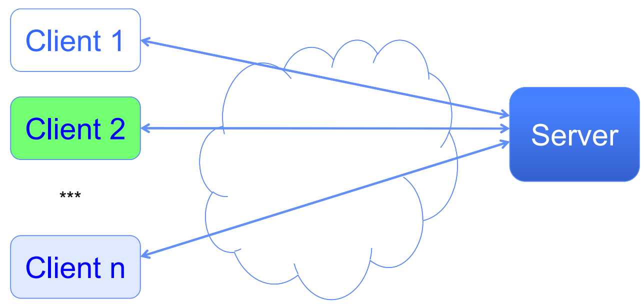

- 5-Tuple identifies each connection:
  1. Source IP Address
  2. Destination IP Address
  3. Source Port Number
  4. Destination Port Number
  5. Protocol (always TCP here)
- Often, Client Port "randomly" assigned
  – Done by OS during client socket setup
- Server Port often "well known"
  – 80 (web), 443 (secure web), 25 (sendmail), etc
  – Well-known ports from 0—1023



## Web Server



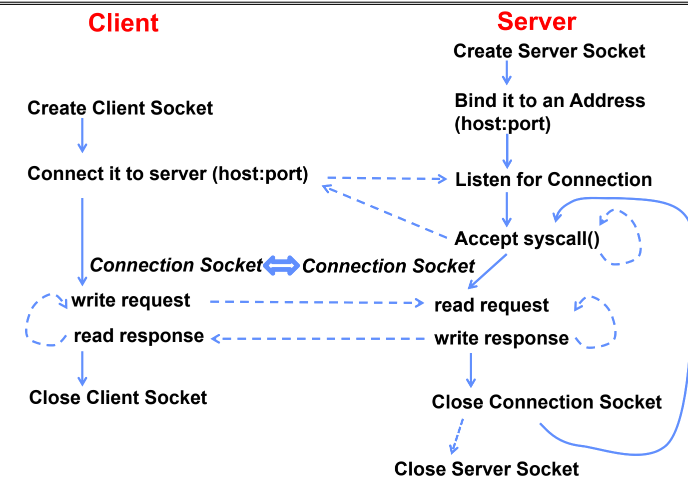
## Client-Server Models




- File servers, web, FTP, Databases, …
- Many clients accessing a common server



## Sockets in concept






## 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);
```



## Server Protocol (v1)

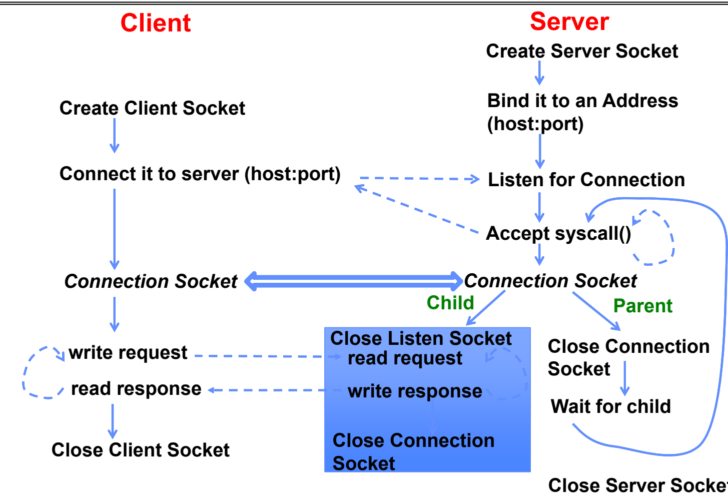
```
// Create socket to listen for client connections
char *port_name;
struct addrinfo *server = setup_address(port_name);
int server_socket = socket(server->ai_family,
                           server->ai_socktype, server->ai_protocol);
// Bind socket to specific port
bind(server_socket, server->ai_addr,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

## 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



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






## Server Protocol (v2)

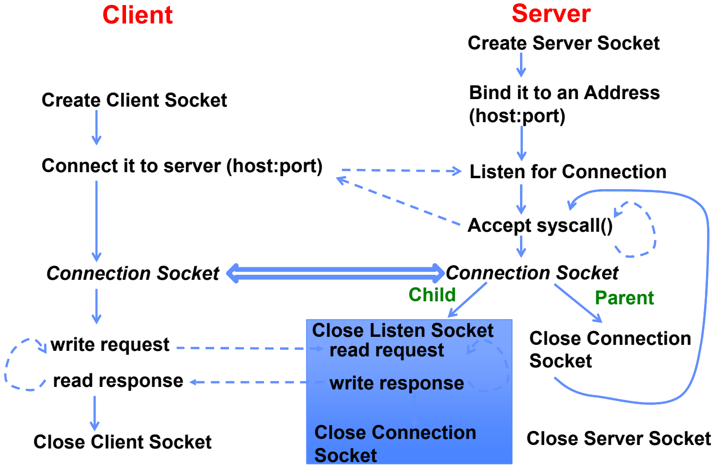
```
// Socket setup code elided…
listen(server_socket, MAX_QUEUE);
while (1) {
  // Accept a new client connection, obtaining a new socket
  int conn_socket = accept(server_socket, NULL, NULL);
  pid_t pid = fork();
  if (pid == 0) {
    close(server_socket);
    serve_client(conn_socket);
    close(conn_socket);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



## 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
- A concurrent server can handle and service a new connection before the previous client disconnects

## Sockets With Protection and Concurrency






## 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);
```



## Server Address: Itself

```
struct addrinfo *setup_address(char *port) {
  struct addrinfo *server;
  struct addrinfo hints;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_PASSIVE;
  getaddrinfo(NULL, port, &hints, &server);
  return server;
}
```

- Accepts any connections on the specified port



## Client: Getting the Server Address

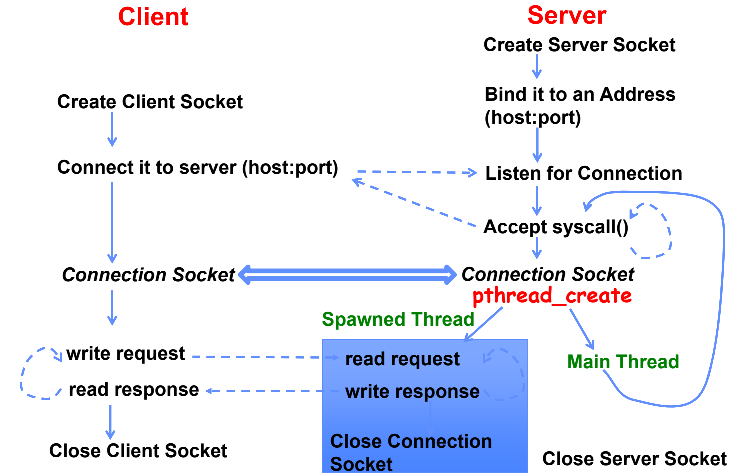
```
struct addrinfo *lookup_host(char *host_name, char *port) {
  struct addrinfo *server;
  struct addrinfo hints;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;

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

# Concurrent Server without Protection

- Spawn a new thread to handle each connection
- 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



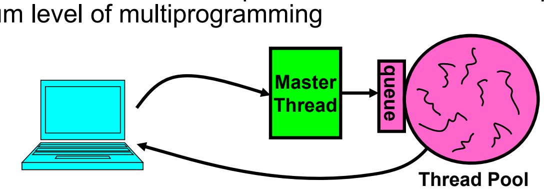
# Sockets with Concurrency, without Protection






# 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);
   }
}
```



# Administrivia

- Kubiatowicz Office Hours:
  – 1-2pm, Tuesday & Wednesday
- Friday was drop deadline. If you forgot to drop, we can't help you!
  – You need to speak with advisor services in your department about how to drop
- Recommendation: Read assigned readings *before* lecture
- Group sign up should have happened already
  – If you don't have 4 members in your group, we will try to find you other partners
  – Want everyone in your group to have the same TA
  – Go to your assigned section on Friday, starting this week!
- Midterm 1 conflicts
  – We will handle these conflicts next week

## Administrivia (Con't)

- Back in person this week!
  - Please be up-to-date with vaccinations and wear masks!
    - » You should have a green pass if you come to class
  - I will be in VLSB 2050 on Tuesday/Thursday 3:30-5:00
    - » Will be trying to get synchronous zoom working. May take a couple of tries to get right
    - » Screen Cast for sure. If I can project it, it will be recorded…
  - We will be trying to make virtual options available for people who are sick
- Start Planning on how your group will collaborate on projects!
  - Meet regularly, in person as regularly as possible
    - » We will have more suggestions on collaborating as term goes on
  - Virtual Interactions: Plan ways of *also* collaborating remotely
    - » Virtual Coffee Hours with your group (with camera)
    - » Regular Brainstorming meetings?
  - Try to meet multiple times a week




## Computers (Cars/other things) in the news

- Y2K22? January 2022 saw a whole new class of bugs:
  - Well, welcome to Y2K22 bugs. If you write a date/time in YYMMDDHHMM format (which is year, month, day, hour, and minute), it now exceeds 31 bits!
  - Meaning – if they use unsigned instead of signed 32-bit numbers it breaks!
  - So, a bunch of systems are now broken:


**Exchange (Email)**

SONICWALL

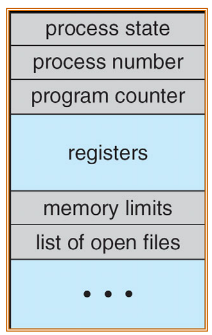
**Email Security/Firewall**


**Honda Car Clocks/Navigation Systems**



## 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/IO
  - Another policy decision



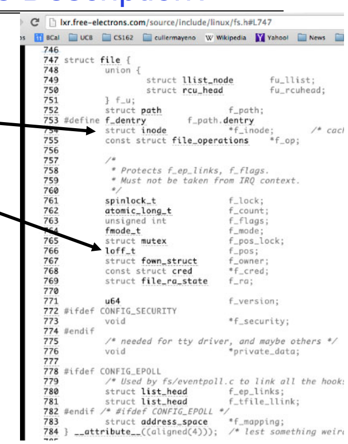

Process Control Block



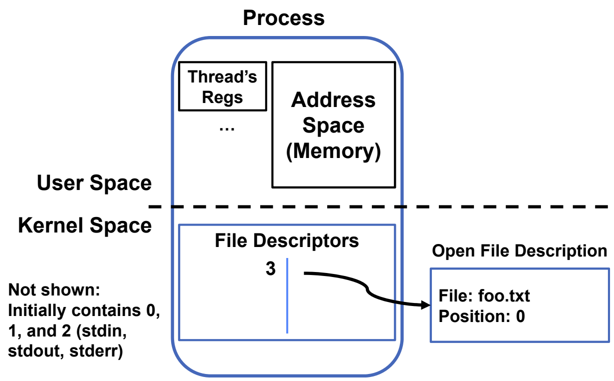
## Recall: What's in an Open File Description?

For our purposes, the two most important things are:

- Where to find the file data on disk
- The current position within the file



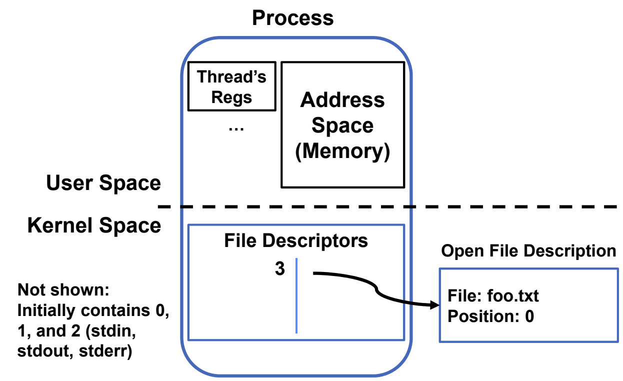
## Abstract Representation of a Process




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



## Abstract Representation of a Process

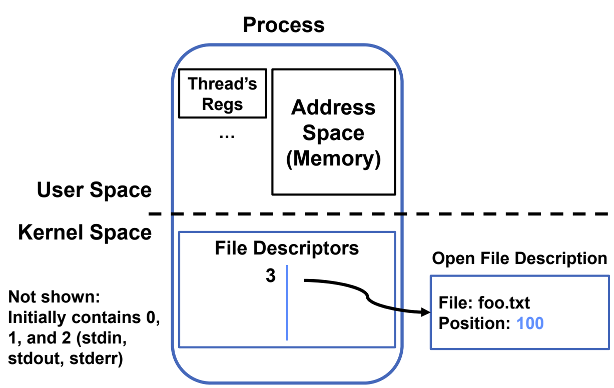



Suppose that we execute
open("foo.txt")
and that the result is 3

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



## Abstract Representation of a Process

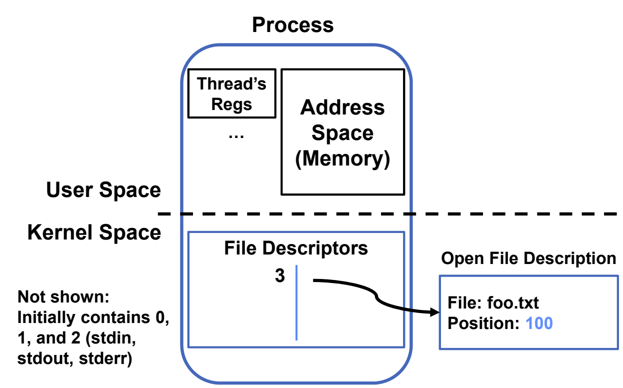



Suppose that we execute
open("foo.txt")
and that the result is 3

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



## Abstract Representation of a Process

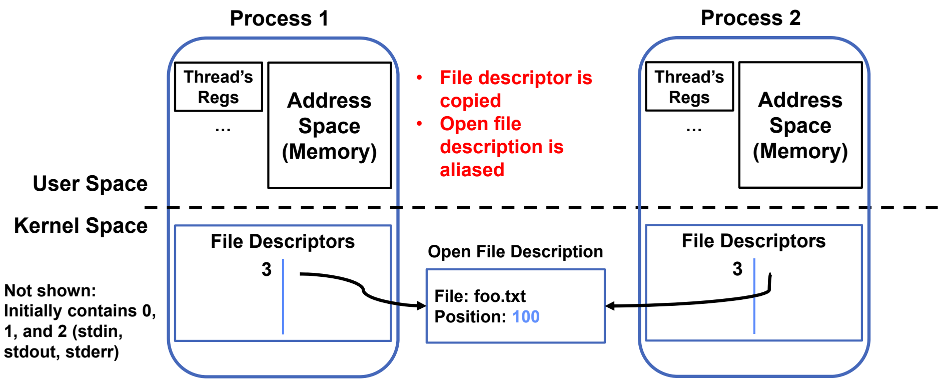



Suppose that we execute
open("foo.txt")
and that the result is 3

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

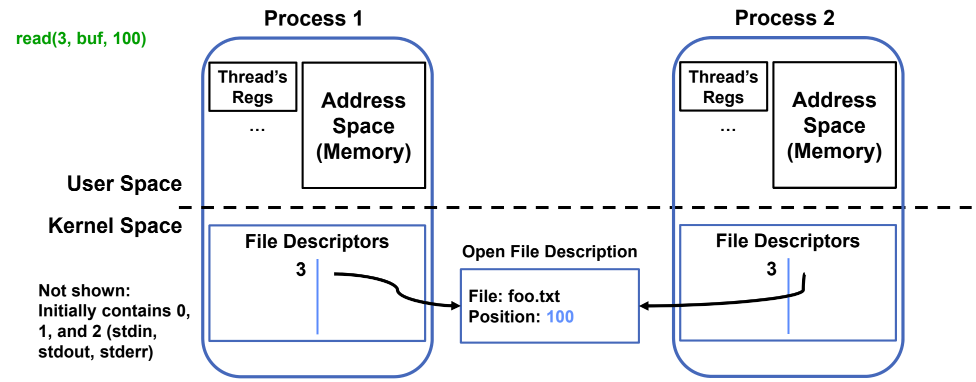
Finally, suppose that we execute
close(3)

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



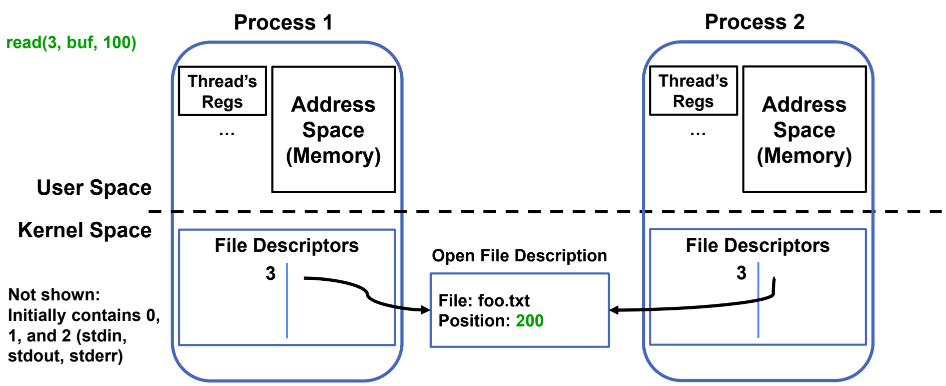



## Open File Description is *Aliased*



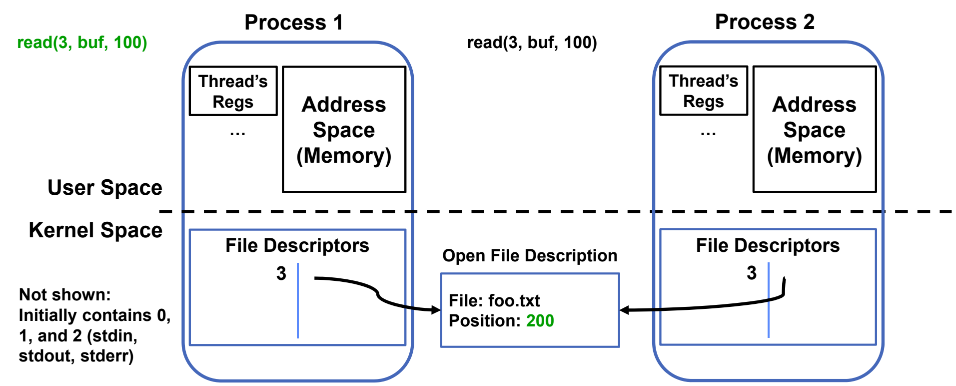



## Open File Description is *Aliased*

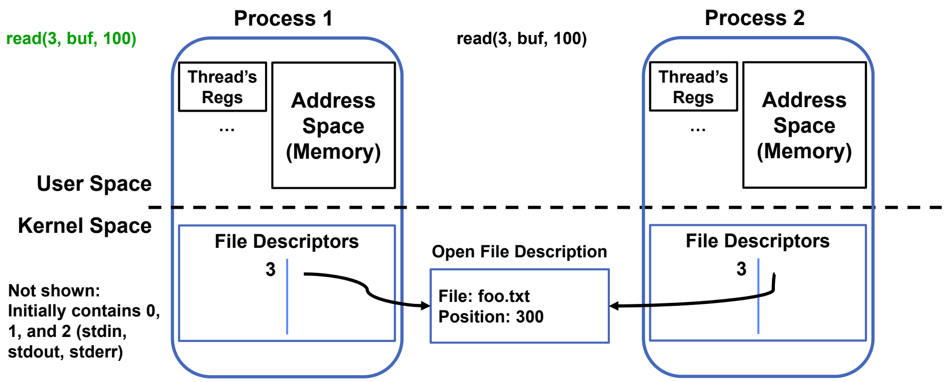





## Open File Description is *Aliased*

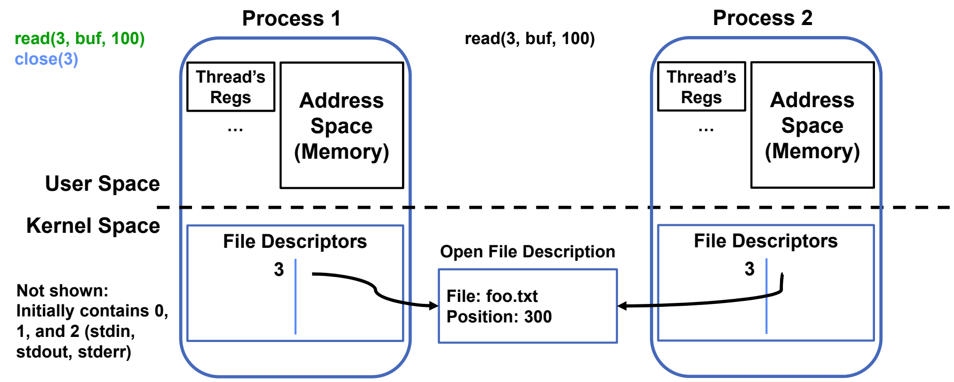


## Open File Description is *Aliased*



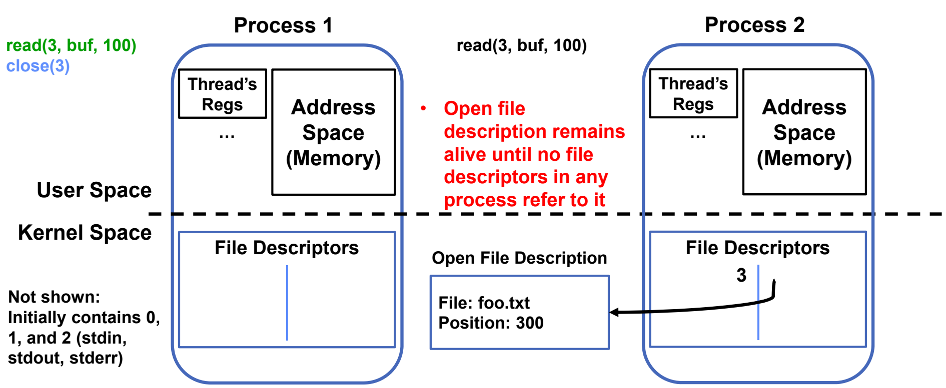



## File Descriptor is *Copied*






## File Descriptor is *Copied*






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

- It allows for *shared resources* between processes

## Recall: In POSIX, Everything is a "File"

- Identical interface for:
  - Files on disk
  - Devices (terminals, printers, etc.)
  - Regular files on disk
  - Networking (sockets)
  - Local interprocess communication (pipes, sockets)
- Based on the system calls **open()**, **read()**, **write()**, and **close()**

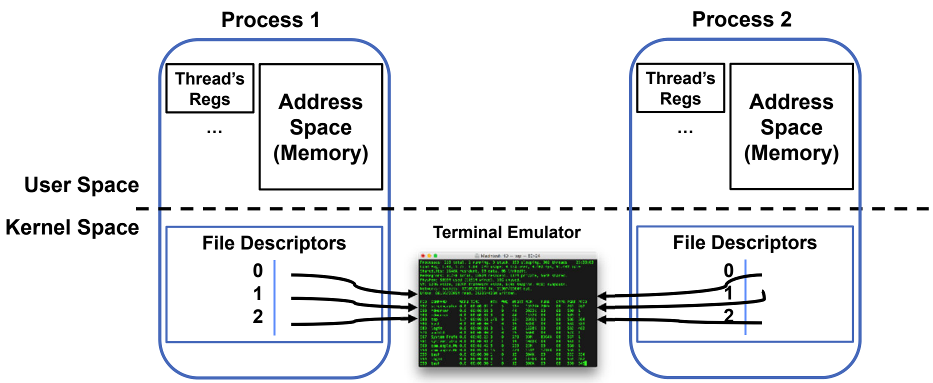


## Example: Shared Terminal Emulator

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

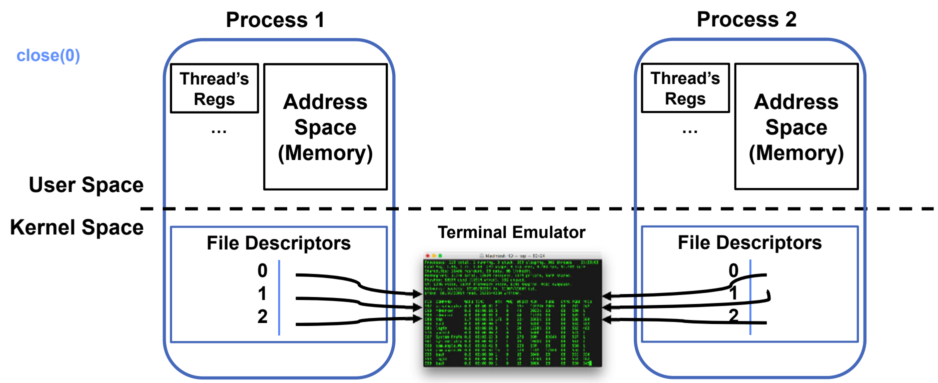


## Example: Shared Terminal Emulator

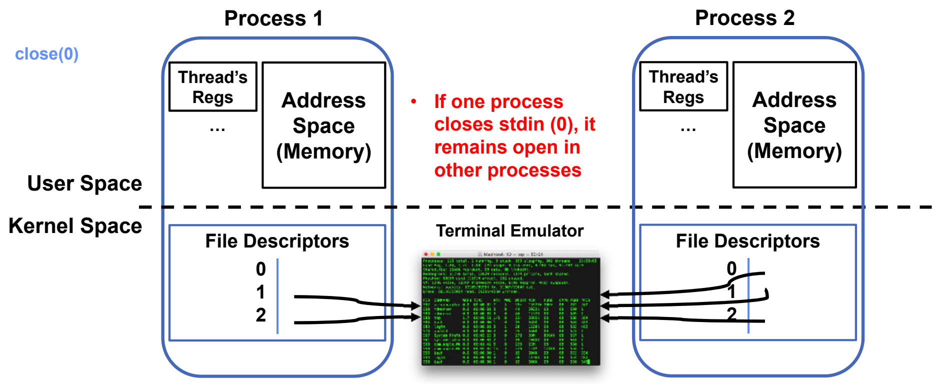





## Example: Shared Terminal Emulator



## Example: Shared Terminal Emulator





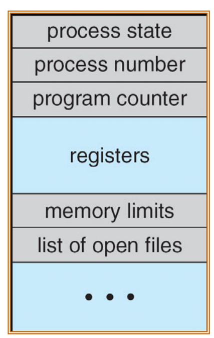

## Other Examples

- Shared network connections after `fork()`
  – Allows handling each connection in a separate process
  – We'll explore this next time
- Shared access to pipes
  – Useful for interprocess communication
  – And in writing a shell (Homework 2)



## Recall: How do we Multiplex Processes?

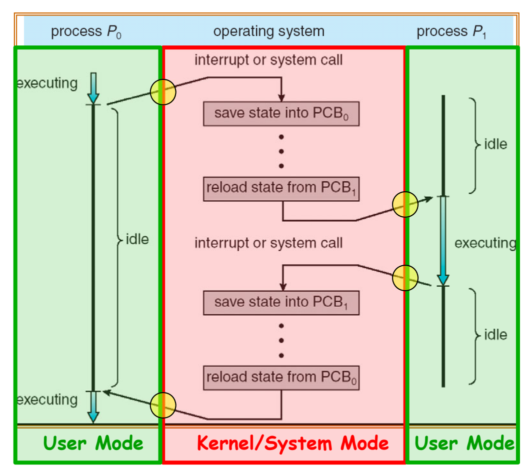
- The current state of process held in a process control block (PCB):
  – This is a "snapshot" of the execution and protection environment
  – Only one PCB active at a time
- Give out CPU time to different processes (Scheduling):
  – Only one process "running" at a time
  – Give more time to important processes
- Give pieces of resources to different processes (Protection):
  – Controlled access to non-CPU resources
  – Example mechanisms:
    » Memory Trnslation: Give each process their own address space
    » Kernel/User duality: Arbitrary multiplexing of I/O through system calls



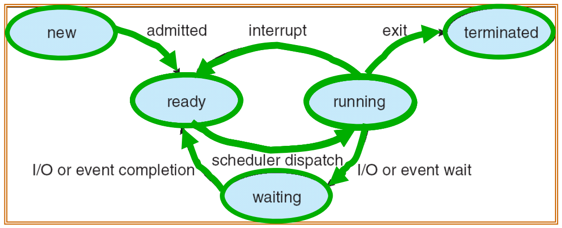

Process Control Block



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



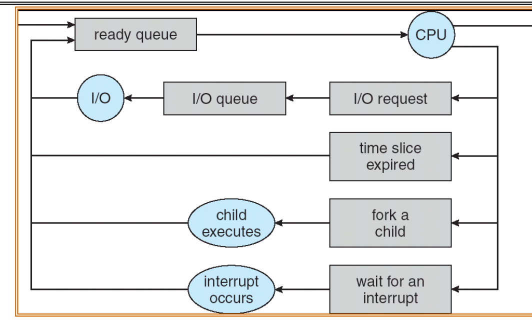
# Lifecycle of a Process




- 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



# Process Scheduling

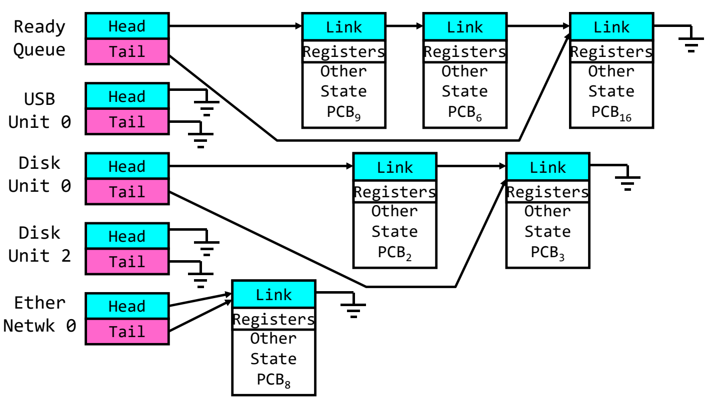



- PCBs move from queue to queue as they change state
  - Decisions about which order to remove from queues are Scheduling decisions
  - Many algorithms possible (few weeks from now)



# Ready Queue And Various I/O Device Queues

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

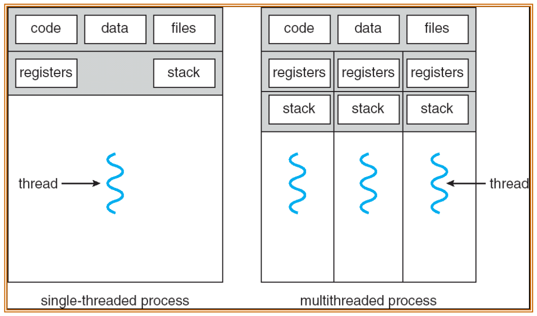





# Modern Process with Threads

- 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 ≡ Process with one thread

## 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?

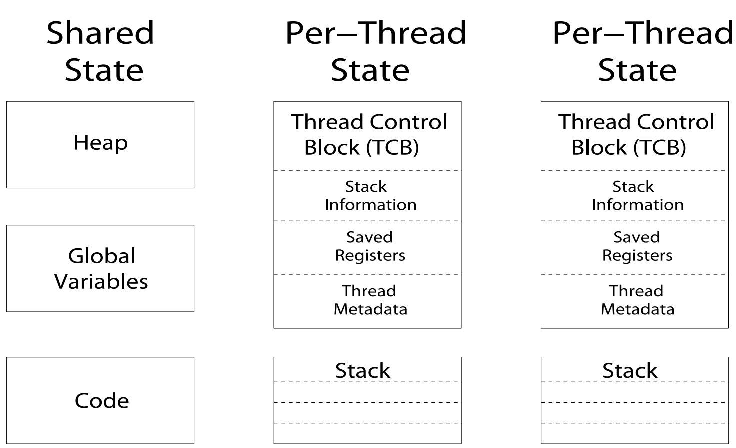


## Thread State

- 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 ≡ 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



## Shared vs. Per-Thread State






## 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

```
       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:
```

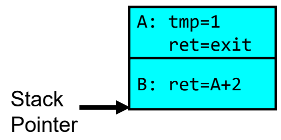
A: tmp=1 ret=exit

Stack Pointer → B: ret=A+2

- 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:
```

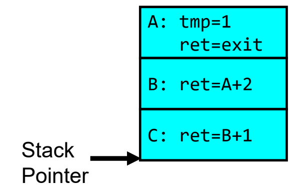



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages



# 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:
```

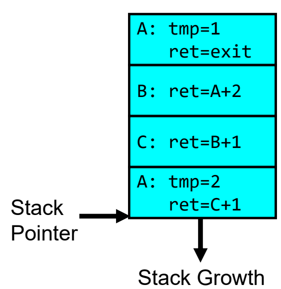



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages



# 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:
```

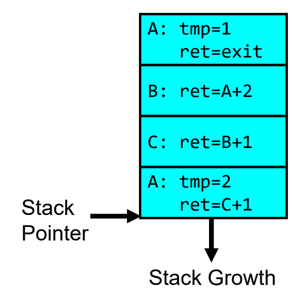



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages



# 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:
```

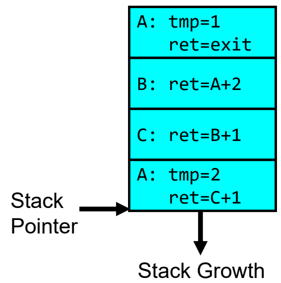



**Output:** >2

- 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:
```



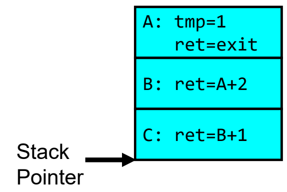

**Output:** >2

- 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:
```



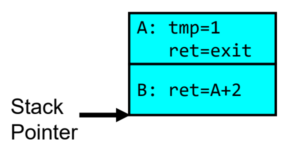

**Output:** >2

- 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:
```



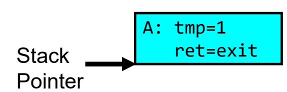

**Output:** >2

- 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:
```




**Output:** >2 1

- 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 |
|---|

**Output:** >2 1

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



## Execution Stack Example

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

**Output:** >2 1

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



## Execution Stack Example

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

Stack Pointer →

| A: tmp=1 ret=exit |
|---|
| B: ret=A+2 |
| C: ret=b+1 |
| A: tmp=2 ret=C+1 |

↓ Stack Growth

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



## Motivational Example for Threads

- Imagine the following C program:

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

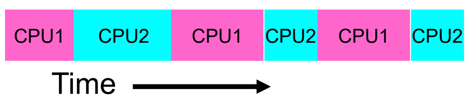
- What is the behavior here?
  - Program would never print out class list
  - Why? `ComputePI` would never finish

# Use of Threads

- Version of program with Threads (loose syntax):

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

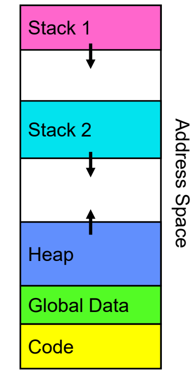
- What does `ThreadFork()` do?
  – Start independent thread running given procedure
- What is the behavior here?
  – Now, you would actually see the class list
  – This *should* behave as if there are two separate CPUs






# 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: *pthreads*

**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**



# Dispatch Loop

- Conceptually, the dispatching 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?

## 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*



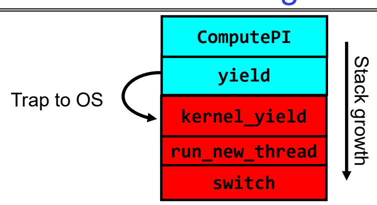
## Internal Events

- 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();
   }
}
```



## Stack for Yielding Thread




- How do we run a new thread?

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

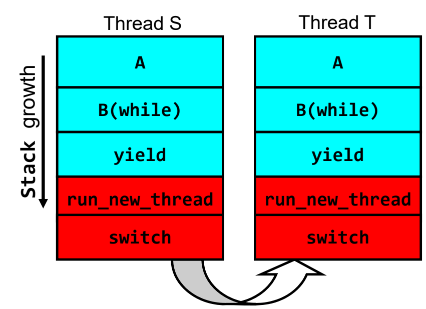
- How does dispatcher switch to a new thread?
  – Save anything next thread may trash: PC, regs, stack pointer
  – Maintain isolation for each thread



## 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



## Conclusion

- Socket: an abstraction of a network I/O queue (IPC mechanism)
- Processes have two parts
  - One or more Threads (Concurrency)
  - Address Spaces (Protection)
- Concurrency accomplished by multiplexing CPU Time:
  - Unloading current thread (PC, registers)
  - Loading new thread (PC, registers)
  - Such context switching may be voluntary (`yield()`, I/O operations) or involuntary (timer, other interrupts)