CSC 112: Computer Operating Systems
Lecture 2

Processes and Threads

Department of Computer Science,

Hofstra University

Acknowledgement: Lecture slides based on UC Berkeley CS 162: Operating Systems and System Programming

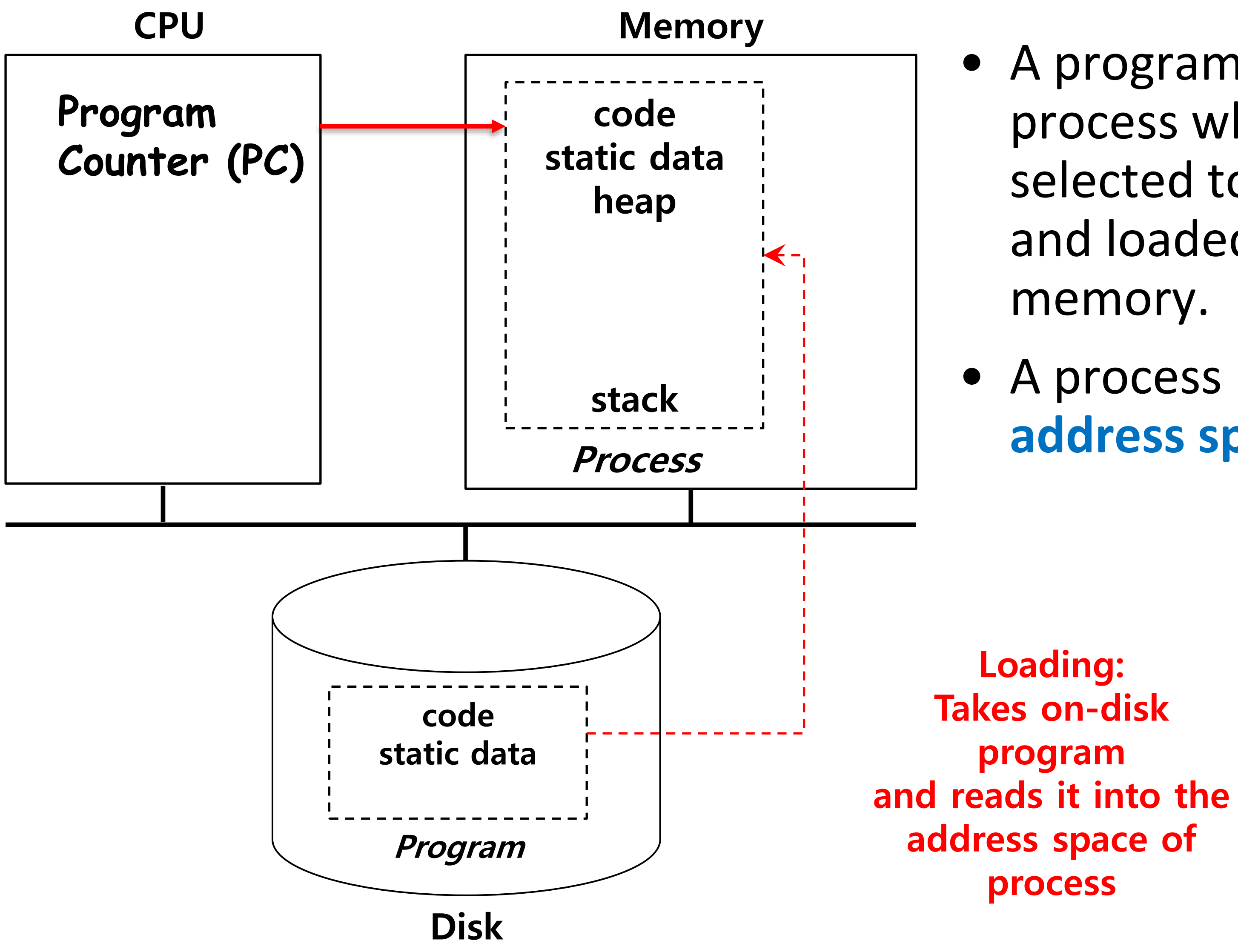
Overview

- Process concept
- Process state
- Process API (creation, wait)
- Process tree

Process

- Program is a ***static*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file loaded into memory
  - Process is an abstraction of CPU
- Execution of program started via Graphic User Interface (GUI) mouse clicks, command line entry of its name, etc
- A physical CPU is shared by many processes
  - Time sharing: run one process for a little while, then run another one, and so forth.
  - Processes believe they are using CPU alone

Process

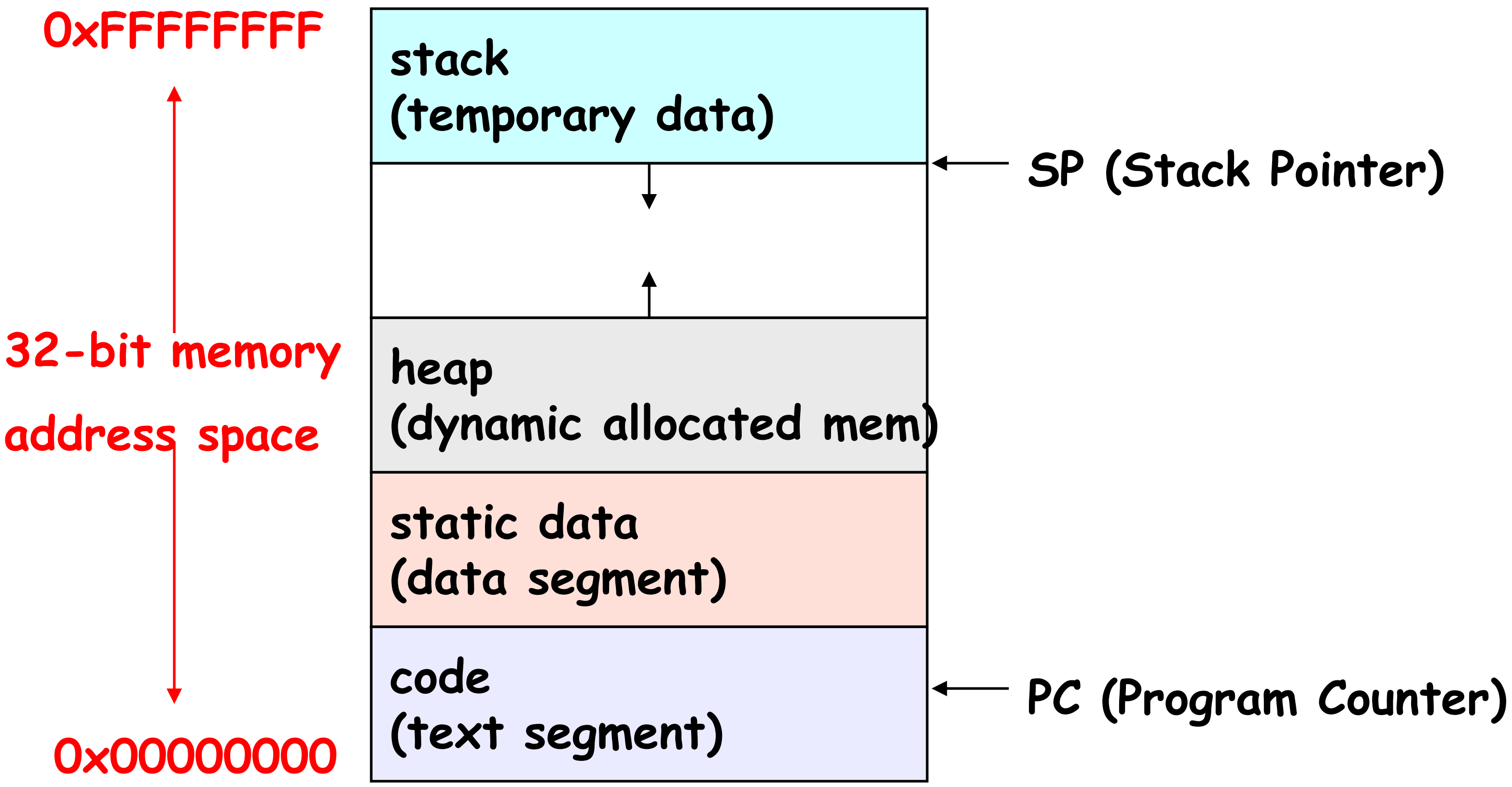


- A program becomes a process when it is selected to execute and loaded into memory.
- A process has an **address space**

Process

Process: a running program

- **Consists of:**
  - Stack: Temporary data, e.g., function parameters, return addresses, local variables
  - Heap: Dynamically allocated memory
  - Static data: Global variables
  - Code: Instructions
  - Registers: SP (Stack Pointer), PC (Program counter)



Process

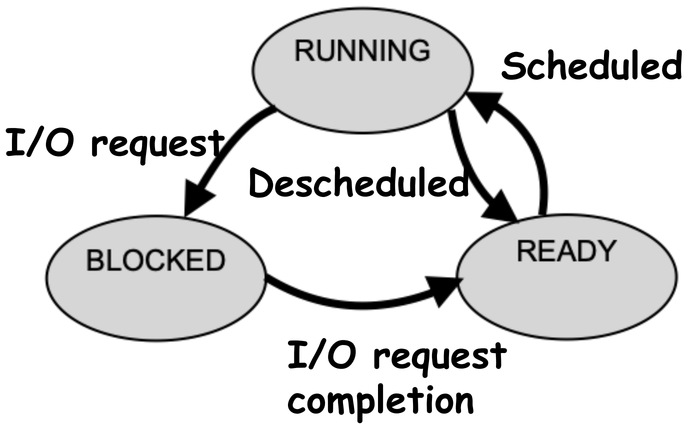
```c
struct proc {
        struct spinlock lock; // p->lock must be held when using these:
        enum procstate state; // Process state
        void *chan; // If non-zero, sleeping on chan
        int killed; // If non-zero, have been killed
        int xstate; // Exit status to be returned to parent's wait
        int pid; // Process ID
        // wait_lock must be held when using this:
        struct proc *parent; // Parent process
        // these are private to the process, so p->lock need not be
held.
        uint64 kstack; // Virtual address of kernel stack
        uint64 sz; // Size of process memory (bytes)
        pagetable_t pagetable; // User page table
        struct trapframe *trapframe; // data page for trampoline.S
        struct context context; // swtch() here to run process
        struct file *ofile[NOFILE]; // Open files
        struct inode *cwd; // Current directory
        char name[16]; // Process name (debugging)
};
```

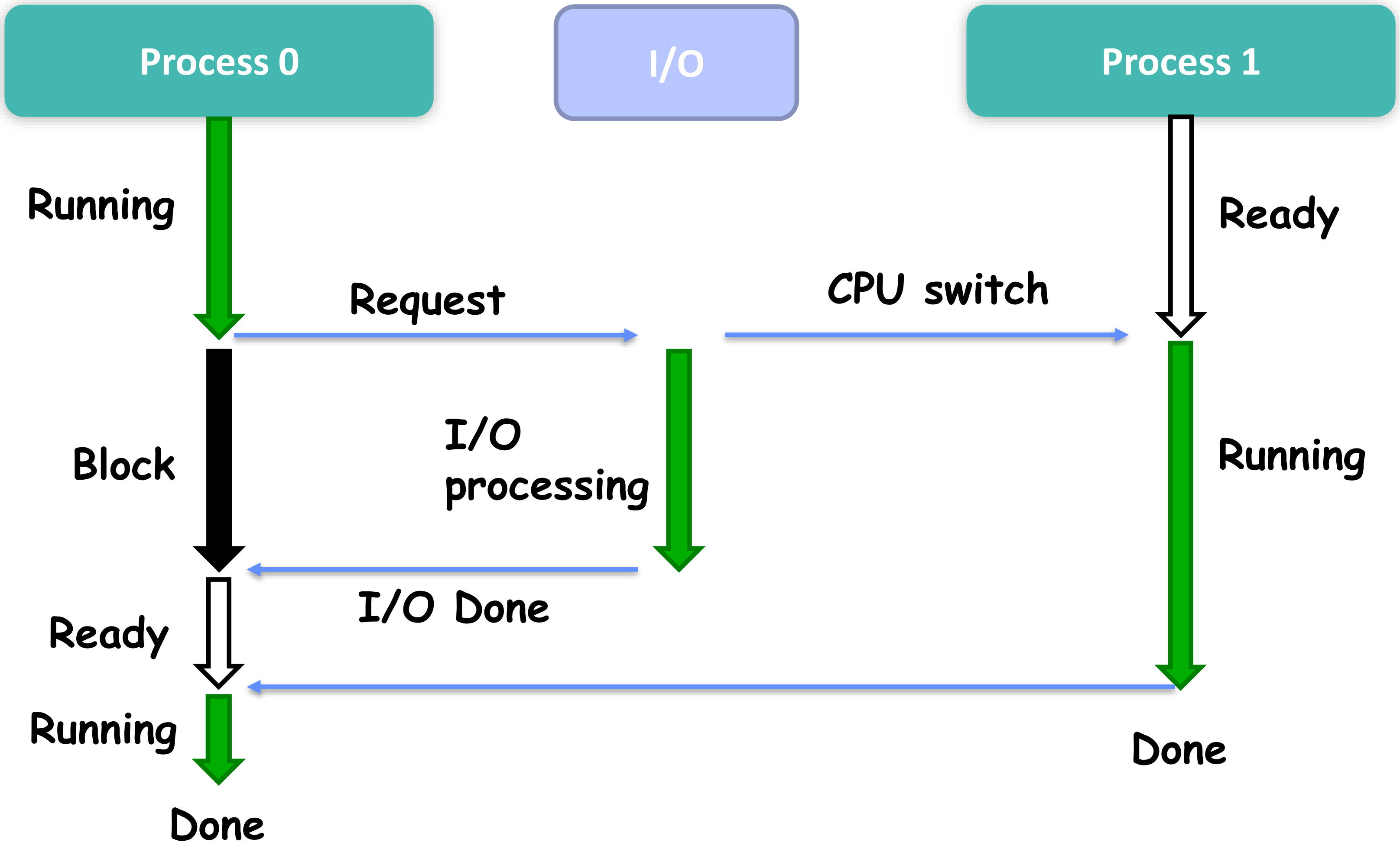
XV6 (proc.h)

- A process is represented by a **process control block (PCB)**
  - Process ID (PID, unique)
  - State
  - Parent process pointer
  - Opened files
  - Many other fields
  - PCB in XV6 does not include pointers to child processes for simplicity, but PCB in Linux include them for convenient references to its child processes

Process States

- Process has different states
  - READY
    - » Ready to run and pending for running
  - RUNNING
    - » Being executed by OS
  - BLOCKED
    - » Suspended due to some other events, e.g., I/O requests





What is a Process in an Operating System?
https://www.youtube.com/watch?v=vLwMl9qK4T8

Process API

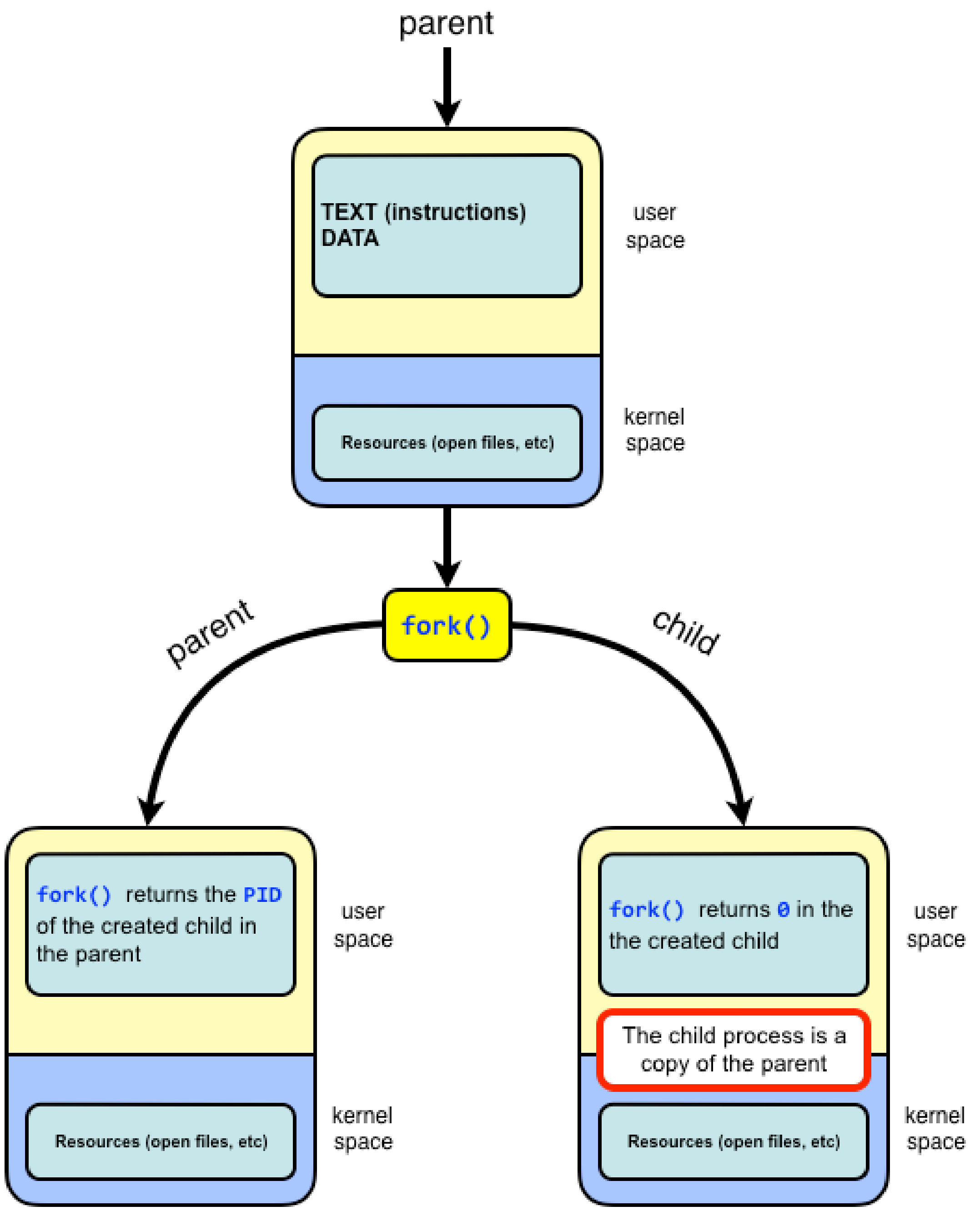
- Process API to manipulate processes
  - **CREATE**
    - » Create a new process, e.g., double click, a command in terminal
  - **WAIT**
    - » Wait for a process to stop
    - » Like I/O request
  - **DESTROY**
    - » Kill the processes
  - **STATUS**
    - » Obtain the information of a process
  - **OTHERS**
    - » Suspend or resume a process

Process Creation

- A process is created by another process, **parent process** or **calling process**
- Process creation relies on two system calls
  - **fork()**
    - » Create a new process and clone its parent process
  - **exec()**
    - » Overwrite the created process with a new program

fork()

- A function without any arguments
  - **ret = fork()**
- Both **parent process** and **child process** continue to execute **the instruction following the fork()**
- The return value indicates which process it is (**parent** or **child**)
  - **ret > 0** (pid of child process): code running in the parent process,
  - **ret == 0**: code running in the newly-created child process
  - **ret == -1**: an error or failure occurred when creating new process
- Fun analogy: imaging you are a process after fork, but you don't know if you are the child or parent process, as if you are running inside of a Matrix. But you can identify which process you are running, by looking up to the sky and see the ret value from fork()
- Child process is a duplicate of its parent process and has same
  - **instructions, data, stack**
- Child and parents have different
  - **PIDs, memory spaces**



fork()

```
int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int ret = fork();
    if (ret < 0) {
            // fork failed; exit
            fprintf(stderr, "fork failed\n"); exit(1);
    } else if (ret == 0) {
            // child (new process)
            printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {
            // parent goes down this path (original process)
            printf("hello, I am parent of %d (pid:%d)\n", ret, (int) getpid());
    }
    return 0;
}
```

Child Process

Parent Process

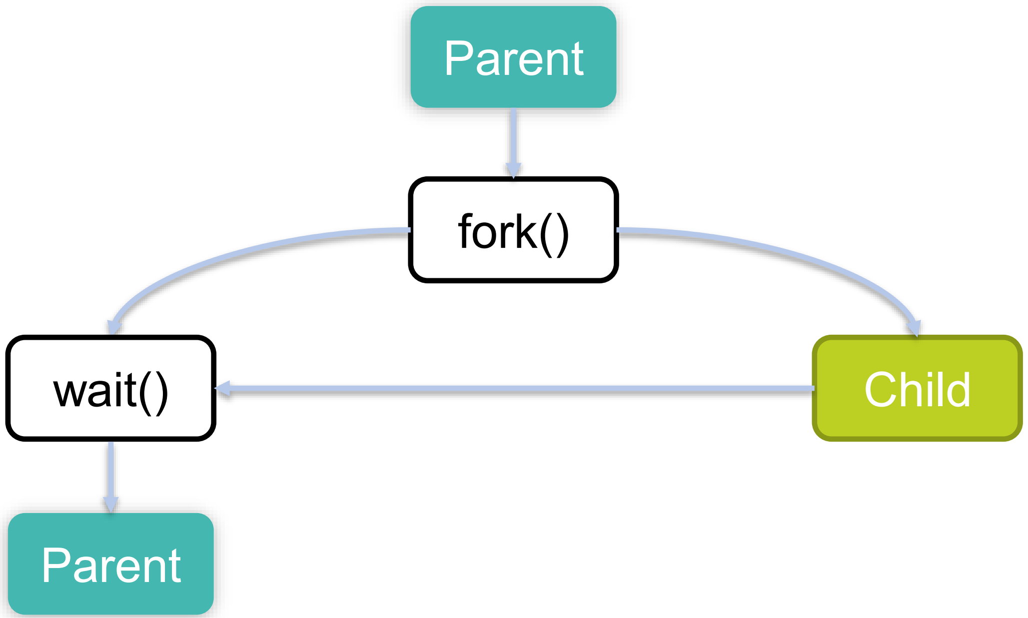
Output

```
hello world (pid:96744)
hello, I am parent of 96745 (pid:96744)
hello, I am child (pid:96745)
```

- Assuming no error (ret >= 0):
- Code inside if(ret == 0){} block is executed by Child process
- Code inside if(ret > 0){} block is executed by Parent process
- Code outside of if-then-else blocks is executed by both Child and Parent

wait()

- Let the parent process wait for the completion of the child process
  - **pid = wait()**
- **wait()** suspends the execution of the calling process until one of its child processes terminates.
  - When a child process terminates, **wait()** retrieves its termination status and allows the system to clean up the resources associated with that child. If the parent does not call **wait()** to collect the child's exit status, the child becomes a zombie process, which means its PCB persists in the process table, even though it is no longer running.
    - » While zombie processes do not consume processor or memory resources, they occupy entries in the process table. The process table is of finite size, and if too many zombie processes accumulate, it can prevent new processes from being created.
  - If there are multiple child processes, **wait()** does not allow the parent to specify which child process to wait for. **waitpid(pid)** is an advanced version of wait. It allows the parent process to specify which child process (or group of processes) it wants to wait for.



wait()

```
int main(int argc, char *argv[])
{
        printf("hello world (ret:%d)\n", (int) getpid());
        int pid = fork();
        if (pid < 0) {
                // fork failed; exit
                fprintf(stderr, "fork failed\n");
                exit(1);
        } else if (pid == 0) {
                // child (new process)
                printf("hello, I am child (pid:%d)\n", (int) getpid());
                sleep(1);
        } else {
                // parent goes down this path (original process). wc (wait child) stores pid of the child process waited by parent
                int ret = wait(NULL); //wc contains pid of the child process being waited for by parent process
                printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", pid, ret, (int) getpid());
        }
        return 0;
}
```

Child process sleeps for 1 second
Parent process waits for the child process to finish sleeping

Child Process

Parent Process

wait()

- **Without wait():** it is nondeterministic which process (parent or child) runs first

```
hello world (pid:96744)
hello, I am parent of 96745 (pid:96744)
hello, I am child (pid:96745)
```

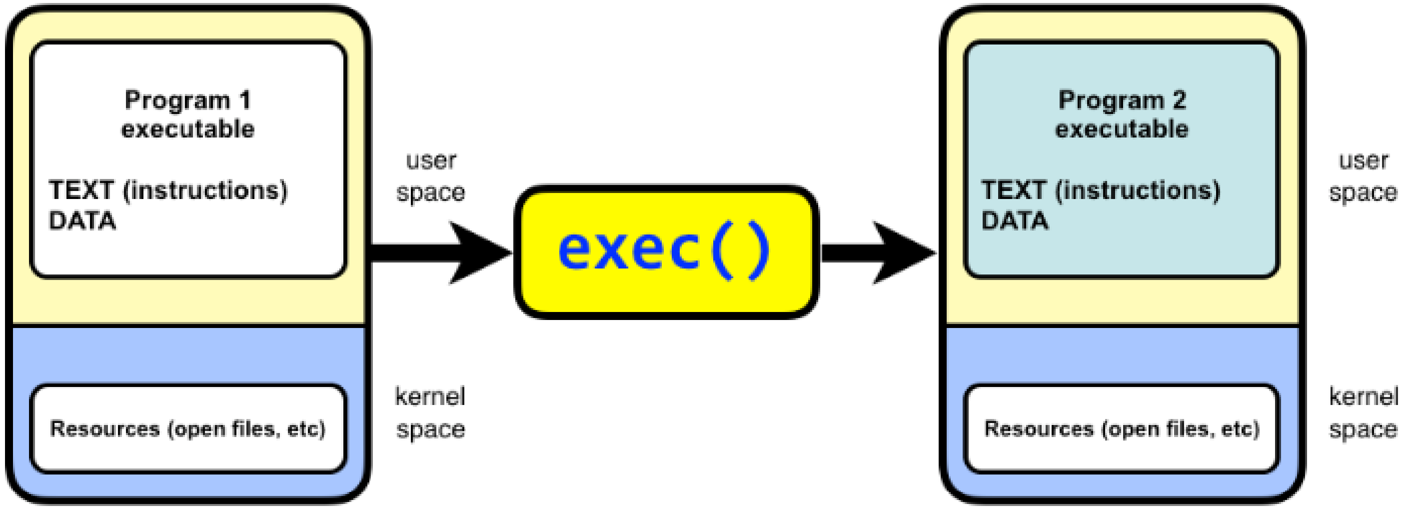
- **With wait():** child runs first, and parents waits for child to finish

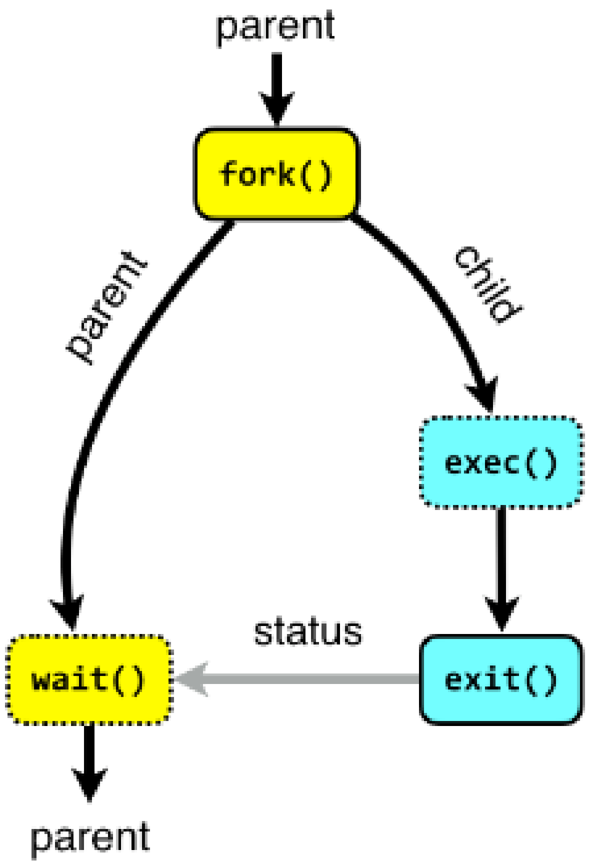
```
hello world (pid:96848)
hello, I am child (pid:96849)
hello, I am parent of 96849 (wc:96849) (pid:96848)
```

Fork() system call tutorial
https://www.youtube.com/watch?v=xVSPv-9x3gk

exec()

- **exec(cmd, argv)** replaces the current process image with a new process image specified by the path to an executable file.
  - It does not return. It starts to execute the new program.
- There is a family of **exec(),** e.g., **execl(), execvp()**
  - **execl()** takes a variable number of arguments that represent the program name and its arguments.
    - » int execl(const char *path, const char *arg, ..., NULL);
  - **execvp()** takes an array of arguments instead of a variable-length argument list
    - » int execvp(const char *file, char *const argv[]);





exec() Example

```
int main(int argc, char *argv[])
{
        printf("hello world (pid:%d)\n", (int) getpid());
        int  pid = fork();
        if (pid < 0) {
                                // fork failed; exit
                                fprintf(stderr, "fork failed\n"); exit(1);
        } else if (pid == 0) {  // child (new process)
                printf("hello, I am child (pid:%d)\n", (int) getpid());
                char *myargs[3];
                myargs[0] = strdup("wc");  // program: "wc" (word count)
                myargs[1] = strdup("p3.c");  // argument: file to count
                myargs[2] = NULL; // marks end of array
                execvp(myargs[0], myargs); // run word count
                printf("This line will never be executed.");
        } else {  // parent
                int rc_wait = wait(NULL);
                printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n", rc, rc_wait, (int)
        getpid());}
        return 0;
}
```

Output:

```
hello world (pid:97511)
hello, I am child (pid:97512)
      32      123      966 p3.c
hello, I am parent of 97512 (wc:97512) (pid:97511)
```

- In the child process (rc == 0), the execvp() function replaces the current process image with the program named “wc“, a program that counts Lines, Words, and Bytes in a file, with output format: [lines] [words] [bytes] [filename].
- The arguments for the program are passed as an array (args[]), where the first element is the program name “wc” and subsequent elements are its arguments. The array must end with NULL.
- After call to execvp(), the whole child process address space is overwritten and replaced by the wc program, so the line “printf(“This line will never be executed.");“ will never be executed.
- Minor point: strdup() allocates memory on the heap and stores a copy of the string there. This is done to ensure that the strings are stored in memory that can be safely modified or freed later if needed. In this program, strdup() is not strictly necessary, and you can pass strings directly to myargs without using `strdup`, since the strings are read only and not modified later.

IO redirection and pipe

- By separating **`fork()`** and **`exec()`**, we can manipulate various settings just before executing a new program and **make the IO redirection and pipe possible**. (details omitted.)
  - IO redirection: output of the left command redirected to be written to the file on the right

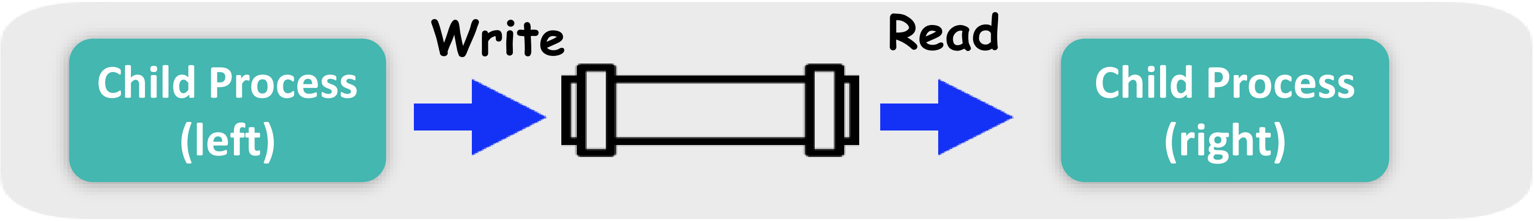
```
% cat w3.c > newfile.txt
```

  - Pipe: output of the left command passed as input to the right command

```
% echo hello world | wc
```

pipe

- A communication method between two processes




```
(base) dliu@dhcp-10-24-18-121 my_code % cat hello.c
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

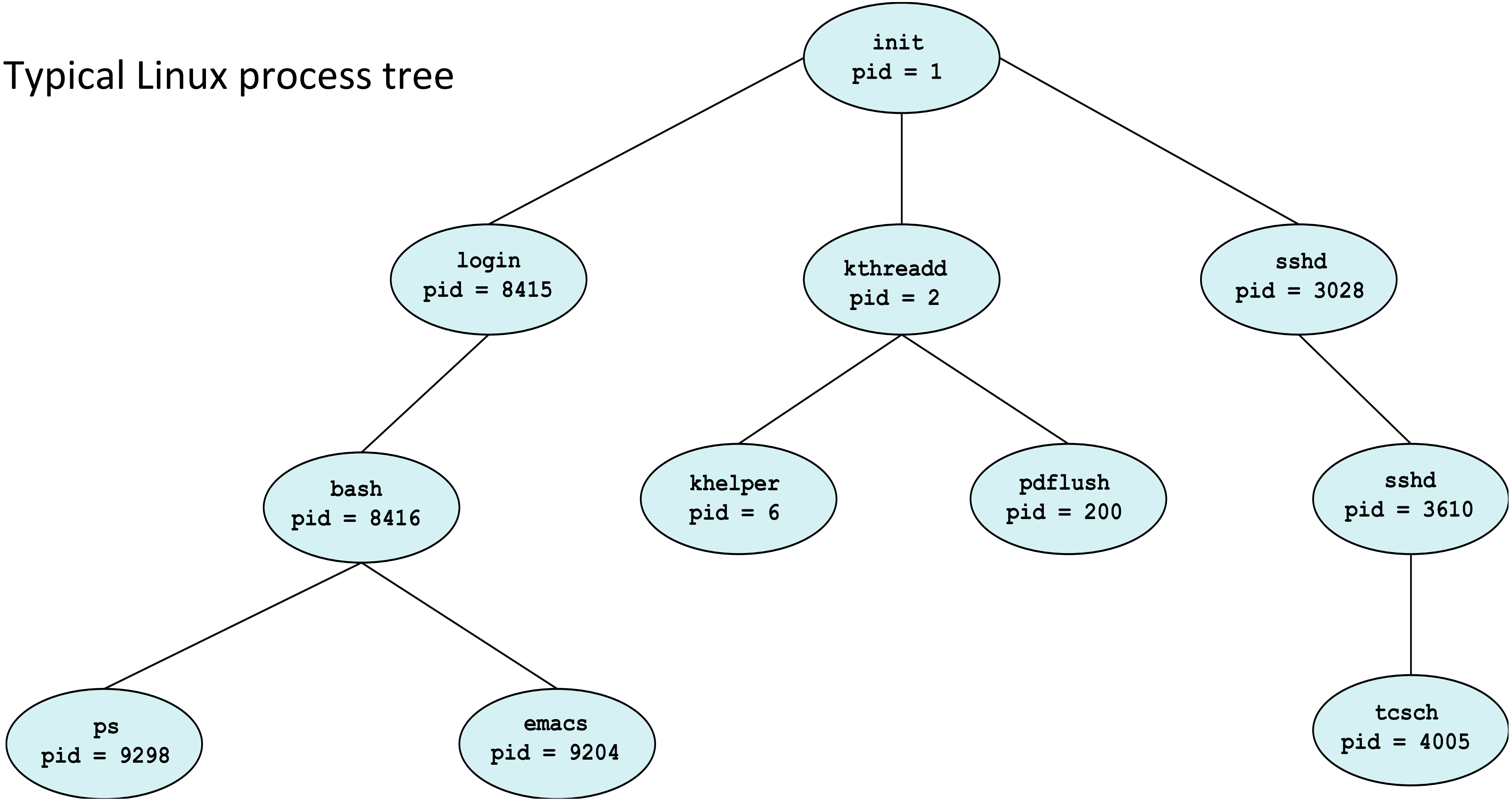
Command “cat” prints out content of hello.c file

Output of “cat” command passed through the pipe to command “grep” to search for any lines that contain “printf”

```
(base) dliu@dhcp-10-24-18-121 my_code % cat hello.c |grep printf
    printf("Hello World!\n");
(base) dliu@dhcp-10-24-18-121 my_code %
```

Process Tree

- Typical Linux process tree



Process Tree

- % pstree (to show the process tree in a hierarchy)

```
(base) dliu@dhcp-10-24-17-236 ~ % pstree
-+= 00001 root /sbin/launchd
 |--= 00322 root /usr/libexec/logd
 |--= 00323 root /usr/libexec/smd
 |--= 00324 root /usr/libexec/UserEventAgent (System)
```

- % ps (to show all processes as a flat list)

```
PID   TT  STAT      TIME COMMAND
  1   ??  Ss    17:57.36 /sbin/launchd
322   ??  Rs     6:29.86 /usr/libexec/logd
323   ??  Ss     0:00.19 /usr/libexec/smd
324   ??  Ss     0:19.58 /usr/libexec/UserEventAgent (System)
```

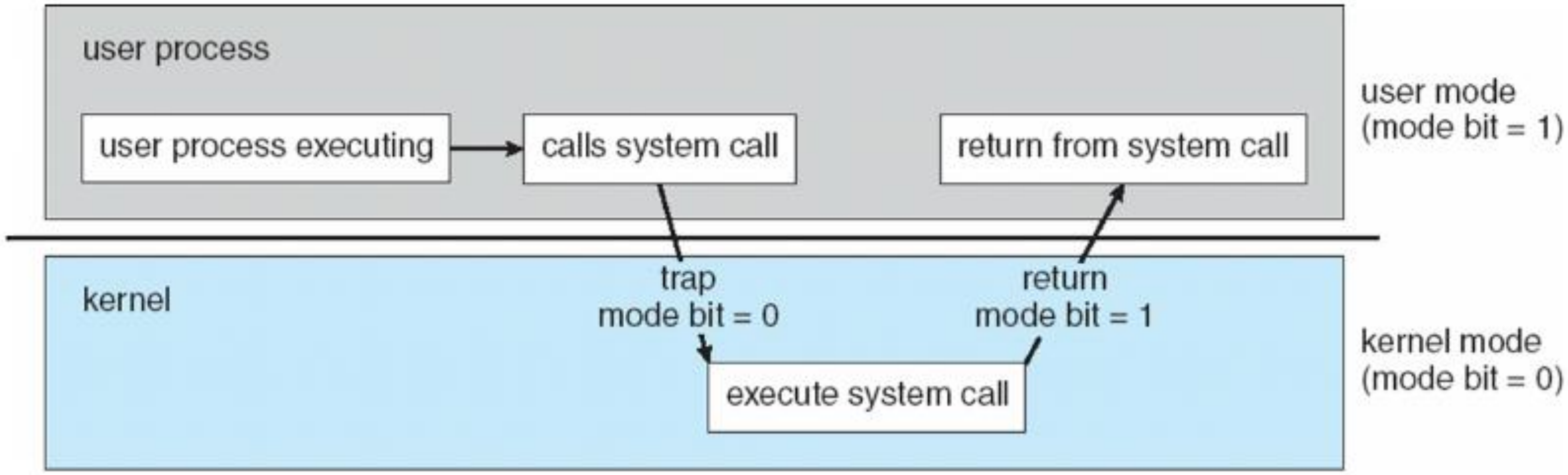
User/Kernel Mode Separation

- **User mode**: restricted, limited operations
  - Processes start in user mode
- **Kernel mode**: privileged, not restricted
  - OS starts in kernel mode

- What if a process wants to perform some restricted operations?
  - **System calls:** Allow the kernel services to provide some functionalities to user programs

User/Kernel Mode Separation

- A process starts in **user mode**
- If it needs to perform a restricted operation, it calls a system call by executing a **trap instruction**
- The state and registers of the calling process are stored, the system enters **kernel mode**, OS completes the syscall work
- **Return from syscall,** restore the states and registers of the process, and resume the execution of the process



Process Scheduling

- **Switching Between Processes**

- **Cooperative approach**
  - **Trust process to relinquish processor time to OS through yield()**

- **Non-cooperative approach**
  - **The OS takes control periodically, e.g., timer interrupter**

Process Summary

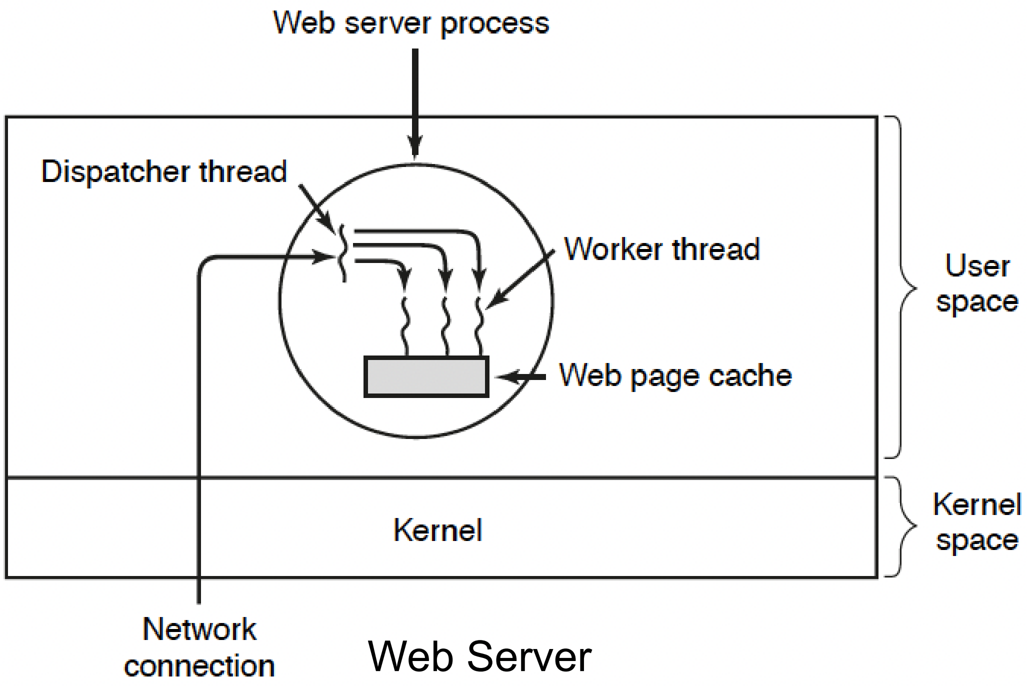
- In OS, process is a running program and has an address space
- We use process API to create and manage processes
- fork() to duplicate a process, exec() to replace the command
- Process scheduling

What’s in a process?

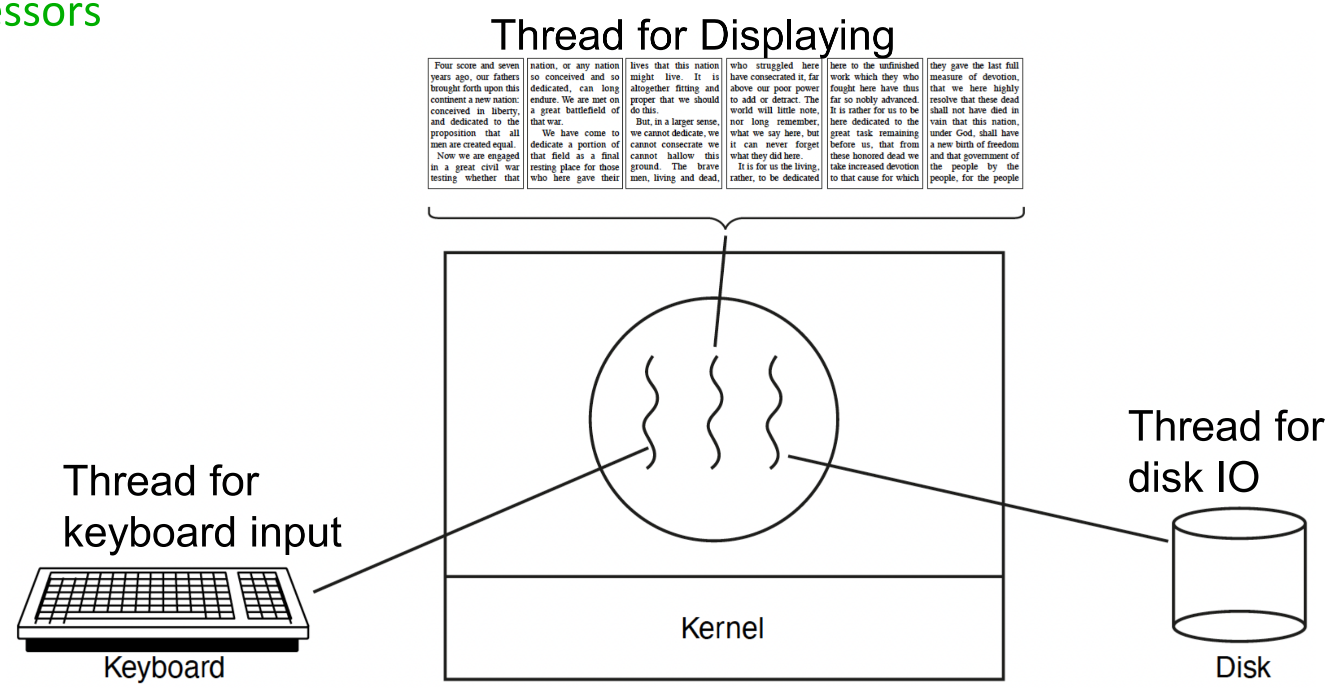
- A process consists of:
  - an address space
  - the code for the running program
  - the data for the running program
  - at least one thread
    - » Registers, IP
    - » Floating point state
    - » Stack and stack pointer
  - a set of OS resources
    - » open files, network connections, sound channels, …
- Today: decompose process from threads of control

Concurrency

- Imagine a web server that handles multiple requests concurrently
  - Multiple worker threads: while waiting for the credit card server to approve a purchase for one client, it could be retrieving the data requested by another client from disk, and assembling the response for a third client from cached information
- Imagine a web client (browser), which might like to initiate multiple requests concurrently
- Imagine a parallel program running on a multiprocessor, which might like to employ parallelism = “true concurrency”
  - For example, multiplying a large matrix – split the output matrix into k regions and compute the entries in each region concurrently using k processors



Web Server



Web Browser

What's needed?

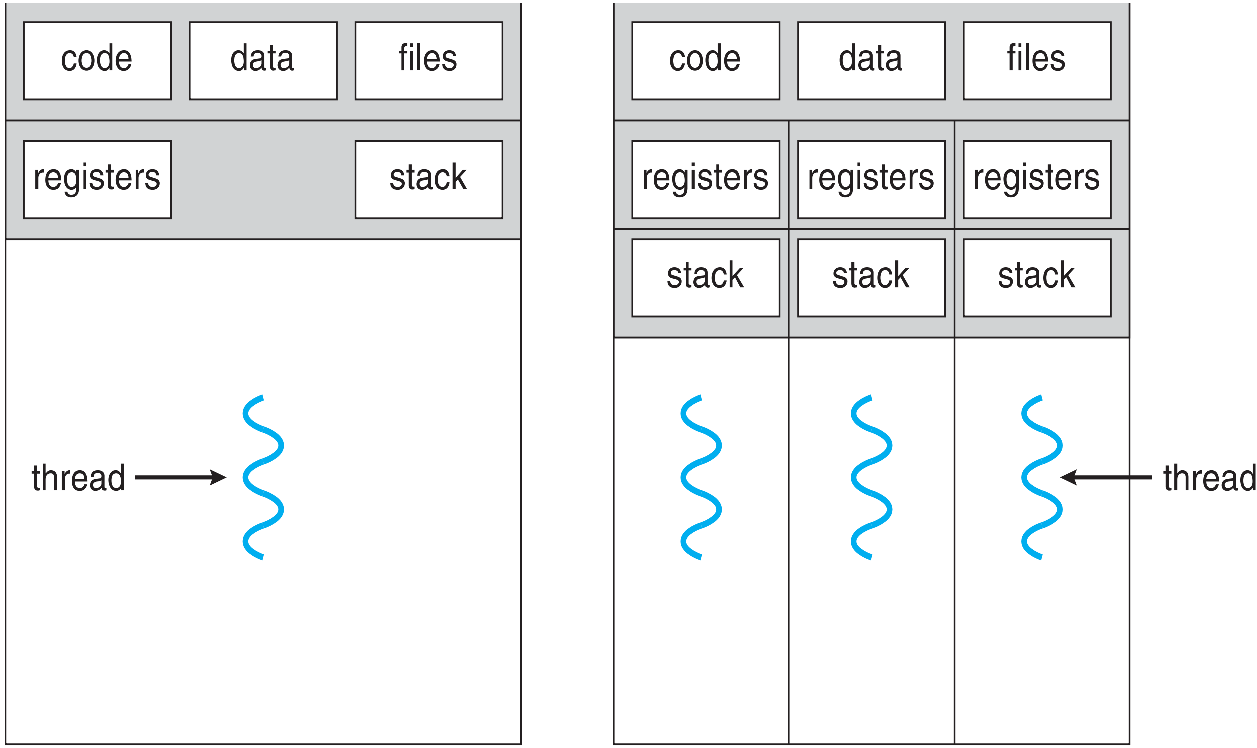
- In each of these examples of concurrency (web server, web client, parallel program):
  - Everybody wants to run the same code, access the same data, has the same privileges, uses the same resources (open files, network connections, etc.)
- But you'd like to have multiple hardware execution states:
  - an execution stack and stack pointer (SP)
    - » traces state of procedure calls made
  - program counter (PC), indicating the next instruction
  - a set of general-purpose registers
- Creating multiple processes is inefficient
- Key idea: separate the concept of a process (address space, etc.) from that of a minimal "thread of control" (execution state: PC, etc.)

Processes and Threads

- Modern OSes support two entities:
  - the process, which defines the address space and general process attributes (such as open files, etc.)
  - the thread, which defines a sequential execution stream within a process
- A thread is bound to a single process / address space
  - address spaces, however, can have multiple threads executing within them
  - threads in the same process share the same address space, making it easy to share data among them
- Threads become the unit of scheduling
  - processes / address spaces are just containers in which threads execute

Processes and Threads

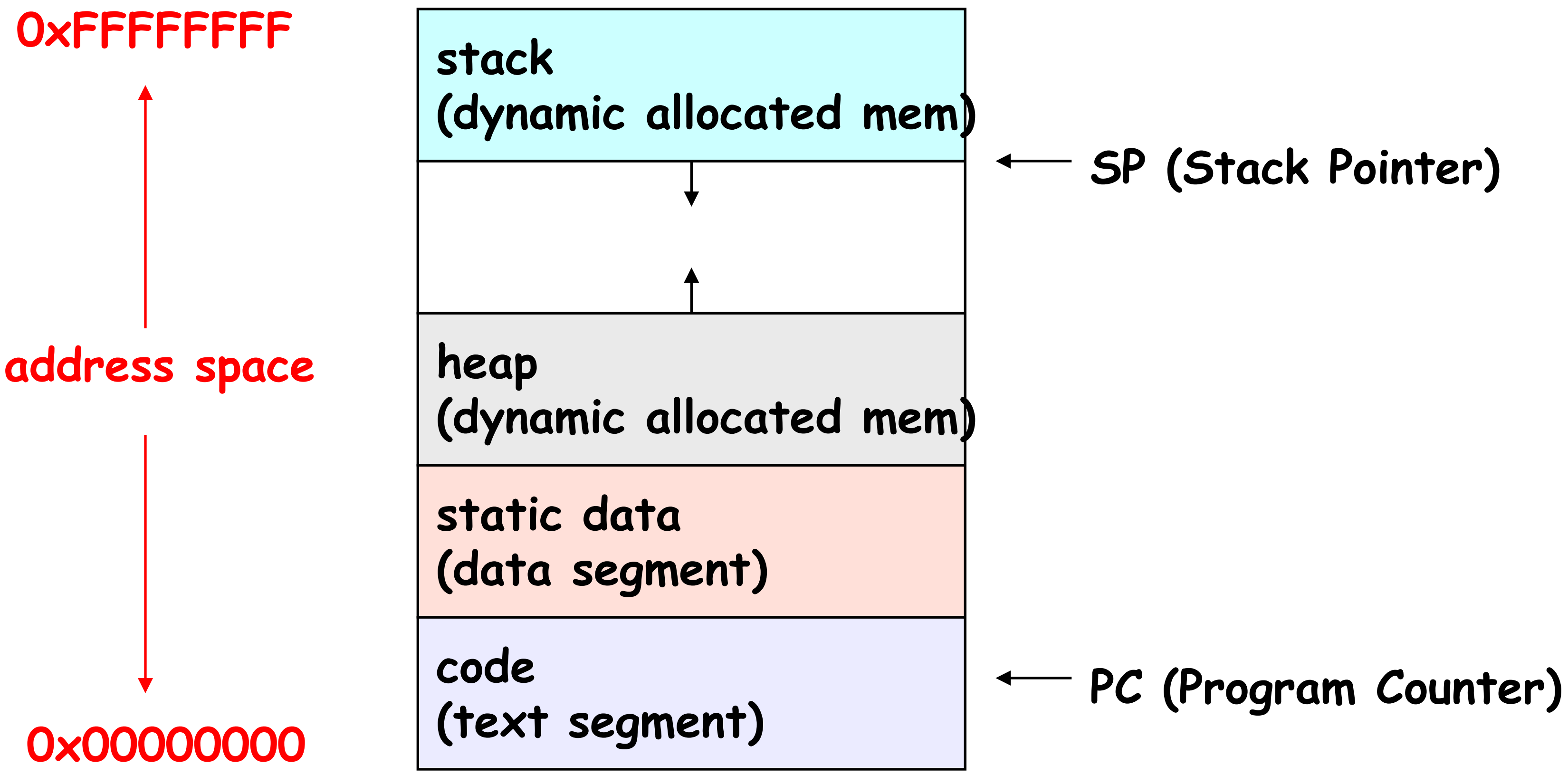
- Multiple threads within a process will share
  - PID
  - The address space: code, most data (heap)
  - Open files (file descriptors)
  - Current working directory
  - Other resources
- Each thread has its own:
  - Thread ID (TID)
  - Set of registers, including Program Counter and Stack Pointer
  - Stack for local variables and return addresses
- Advantages
  - Efficient and fast resource sharing
  - Efficient utilization of many CPU cores with only one process
  - Less context switching overhead



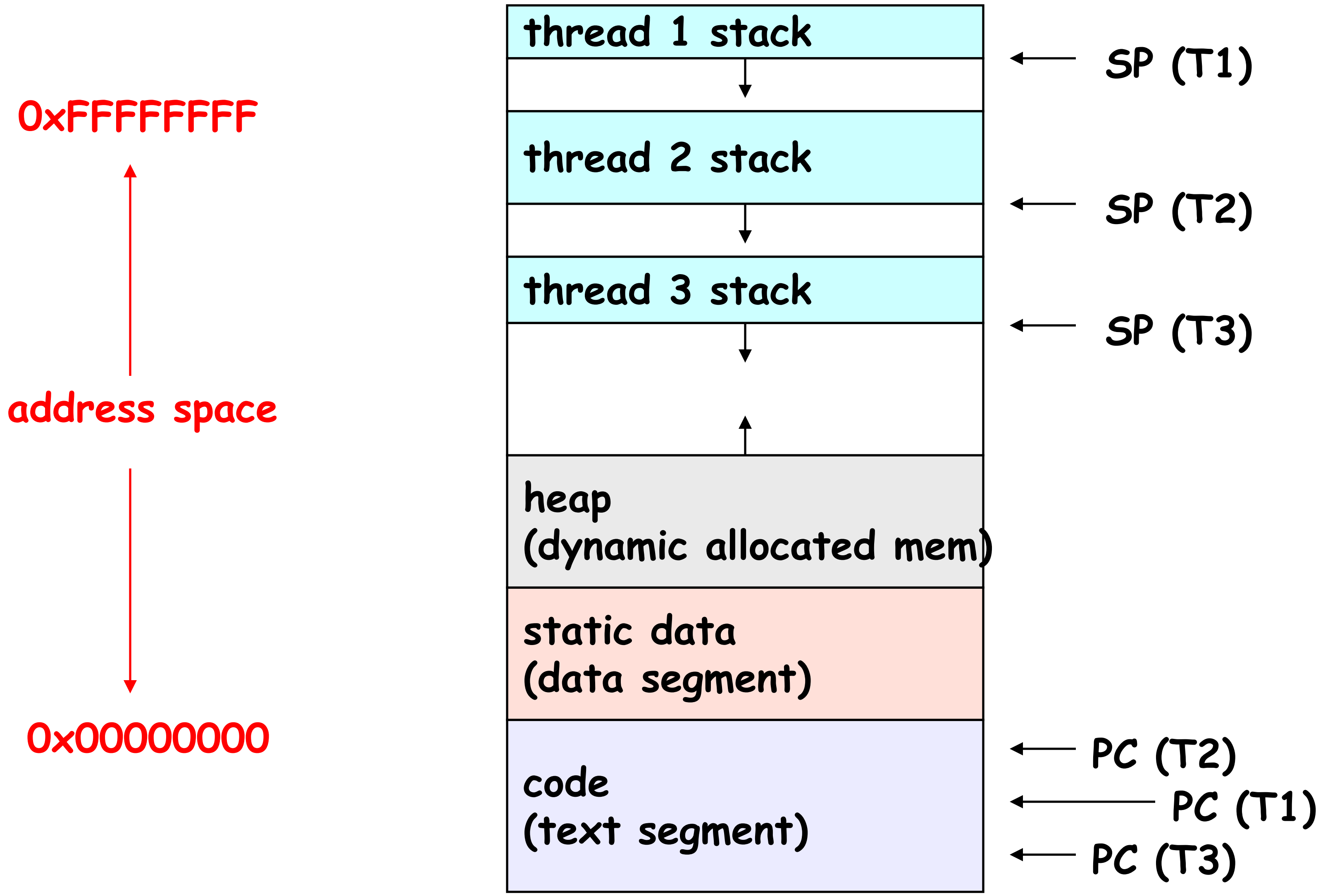
single-threaded process

multithreaded process

(old) Process address space



(new) Process address space with threads



Process/thread separation

- Concurrency (multithreading) is useful for:
  - handling concurrent events (e.g., web servers and clients)
  - building parallel programs (e.g., matrix multiply, ray tracing)
  - improving program structure
- Multithreading is useful even on a uniprocessor
  - even though only one thread can run at a time, multiple threads may be executed in a time-sharing schedule, so they appear to run concurrently
- Supporting multithreading – that is, separating the concept of a process (address space, files, etc.) from that of a minimal thread of control (execution state), is a big win
  - creating concurrency does not require creating new processes
  - faster / better / cheaper

POSIX pthreads API

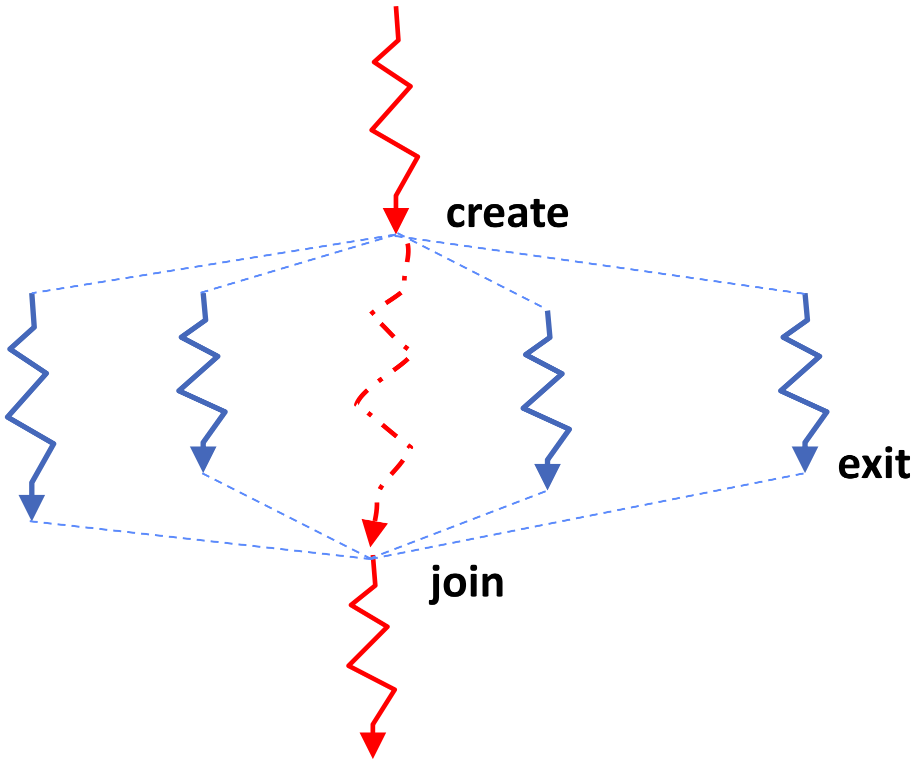
- POSIX thread -> pthread
- A Portable Operating System Interface (POSIX) library (IEEE 1003.1c), written in C language
- Pthread library: 60+ functions, API specifies behavior of the thread library

API	Functionality
pthread_create	Create a new thread in the caller's address space
pthread_exit	Terminate the calling thread
pthread_join	Wait for a thread to terminate
pthread_mutex_lock	Lock a mutex
pthread_mutex_unlock	Unlock a mutex
sem_wait	Wait on a semaphore
sem_post	Signal or post on a semaphore
pthread_cond_wait	Wait on a condition variable
pthread_cond_signal	Wake up one thread waiting on a condition variable
pthread_cond_broadcast	Wake up all threads waiting on a condition variable

Pthread Fork-Join Pattern

```
void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t p1, p2;
  printf("main: begin\n");
  pthread_create(&p1, NULL, mythread, "A");
  pthread_create(&p2, NULL, mythread, "B");
  // join waits for the threads to finish
  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  printf("main: end\n");
}
```



Main thread creates multiple sub-threads, passing them args to work on… then joins with them to collect results.

"Where do threads come from?"

- The kernel is responsible for creating/managing threads
  - for example, the kernel call to create a new thread would
    - » allocate an execution stack within the process address space
    - » create and initialize a Thread Control Block
      - stack pointer, program counter, register values
    - » stick it on the ready queue
  - we call these kernel threads

"Where do threads come from?" (2)

- Threads can also be managed at the user level (that is, entirely from within the process)
  - – a library linked into the program manages the threads
    - » because threads share the same address space, the thread manager doesn't need to manipulate address spaces (which only the kernel can do)
    - » threads differ (roughly) only in hardware contexts (PC, SP, registers), which can be manipulated by user-level code
    - » the Linux thread package multiplexes user-level threads on top of kernel thread(s), which it treats as "virtual processors"
  - – we call these user-level threads

Kernel threads

- OS now manages threads *and* processes
  - all thread operations are implemented in the kernel
  - OS schedules all of the threads in a system
    - » if one thread in a process blocks (e.g., on I/O), the OS knows about it, and can run other threads from that process
    - » possible to overlap I/O and computation inside a process
- Kernel threads are cheaper than processes
  - less state to allocate and initialize
- But they are still quite expensive (e.g., orders of magnitude more expensive than a procedure call)
  - thread context switch involves system calls, since OS must maintain kernel state for each thread

User-level threads

- To make threads cheap and fast, they may be implemented at the user level
  - managed entirely by user-level library, e.g., `libpthreads.a`
- User-level threads are small and fast
  - each thread is represented simply by a PC, registers, a stack, and a small thread control block (user-space TCB)
  - creating a thread, switching between threads, and synchronizing threads are done via procedure calls
    - no kernel involvement is necessary!
  - user-level thread operations can be 10-100x faster than kernel threads as a result
- The OS kernel scheduler schedules the kernel threads; the user-level thread scheduler within each process schedules the user-level threads within the time intervals that the underlying kernel thread runs.
  - it uses queues to keep track of the thread states: run, ready, wait. Just like the OS kernel scheduler, but implemented as a user-level library

Example implementations of user-level threads

- Fibers, co-routines

FANG Interview Question | Process vs Thread
https://www.youtube.com/watch?v=4rLW7zg21gI

Summary

- Processes
  - In OS, process is a running program and has an address space
  - We use process API to create and manage processes
  - fork() to duplicate a process, exec() to replace the command
- Threads:
  - Multiple threads per process / address space
  - Kernel threads are much more efficient than processes, but they’re still not cheap
  - User-level threads are very efficient