

# CSC 112: Computer Operating Systems

## Lecture 2

### Processes and Threads

Department of Computer Science,  
Hofstra University

# Overview

---

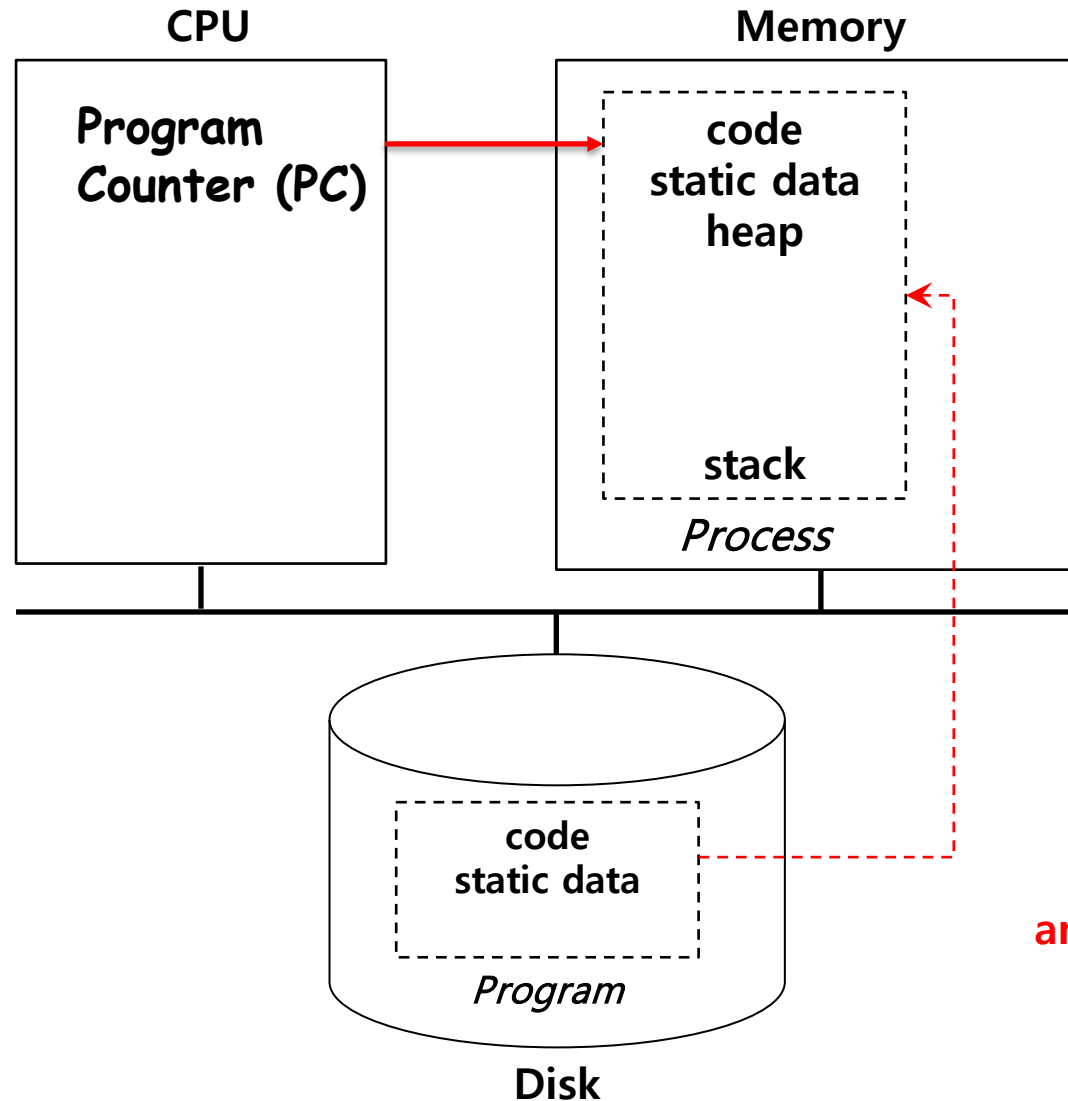
- Process concept
- Process state
- Process API (creation, wait)
- Process tree

# 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**

**Loading:**  
Takes on-disk  
program  
and reads it into the  
address space of  
process

# Process

## Process: a running program

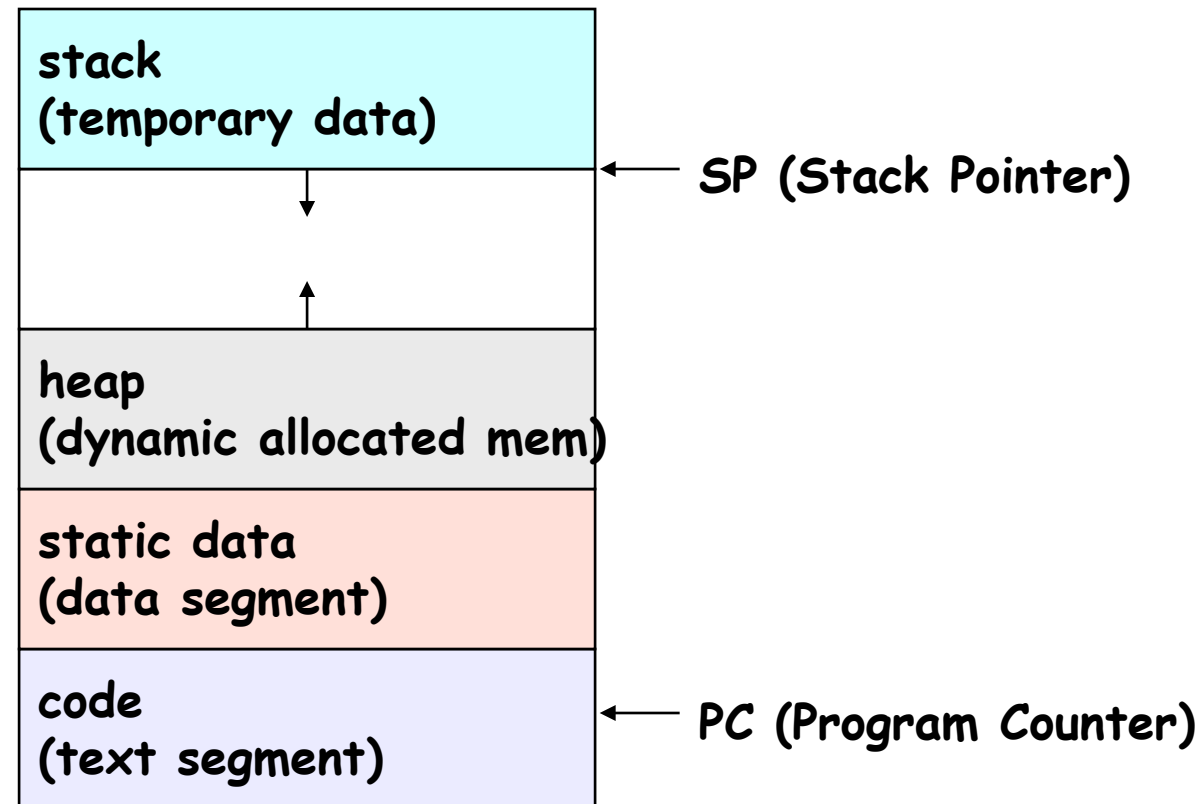
- **Consists of:**

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

0xFFFFFFFF

address space

0x00000000



# Process

```
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**

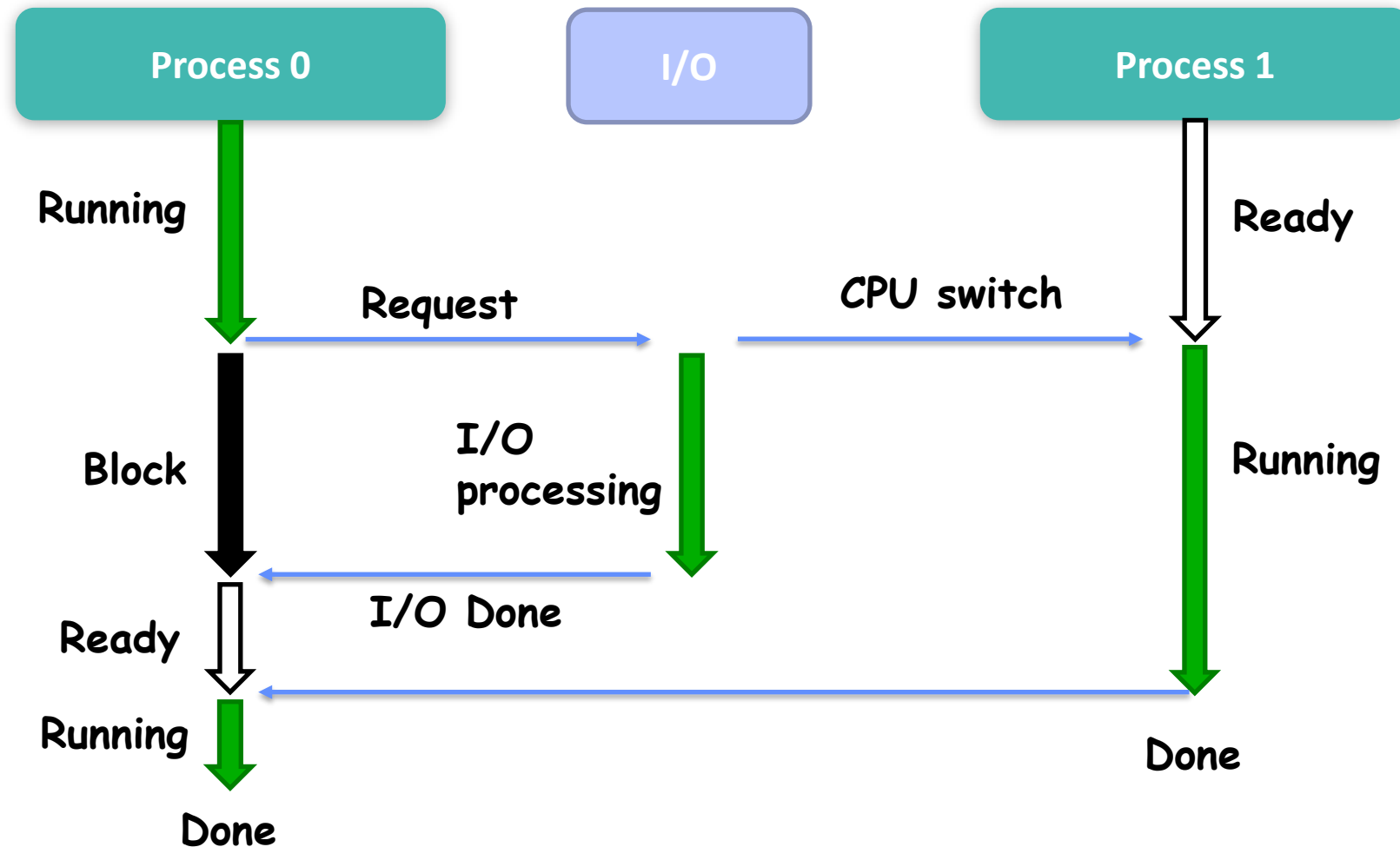
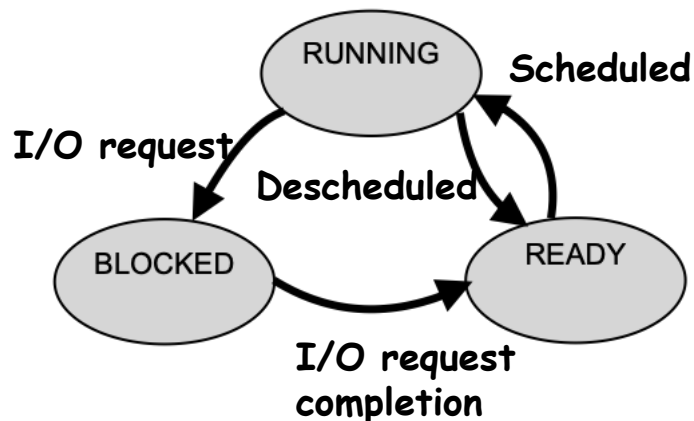
- » Ready to run and pending for running

- **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=vLwMI9qK4T8>

# 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



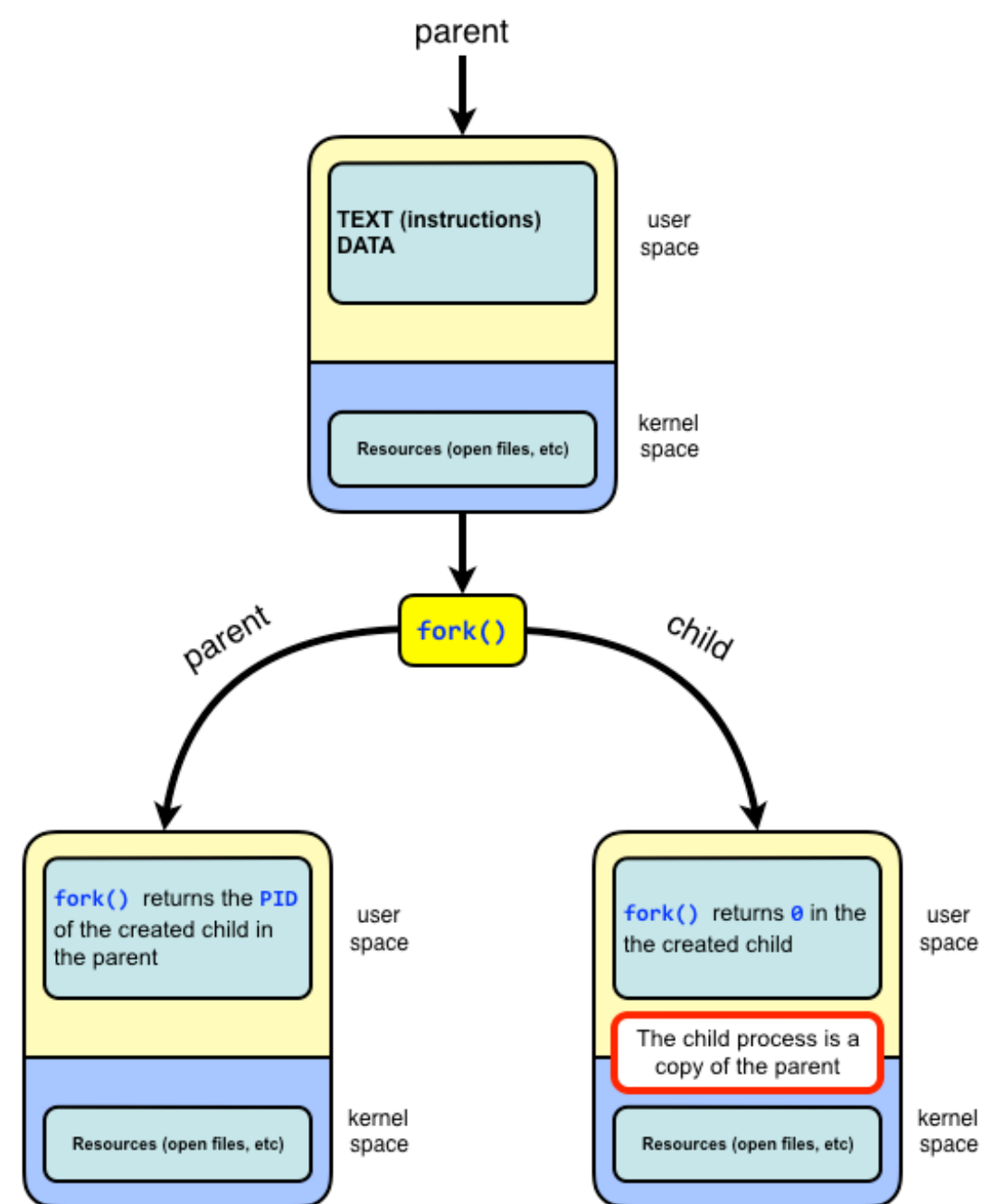
# Process Creation

---

- 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
  - `pid = 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**)
  - **Non-0 pid** (pid of child process) : return value of the **parent** process,
  - **0** : return value of the new **child** process
  - **-1** : an error or failure occurs when creating new process
- 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 rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n"); exit(1);
    } else if (rc == 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", rc, (int) getpid());
    }
    return 0;
}
```



## Output

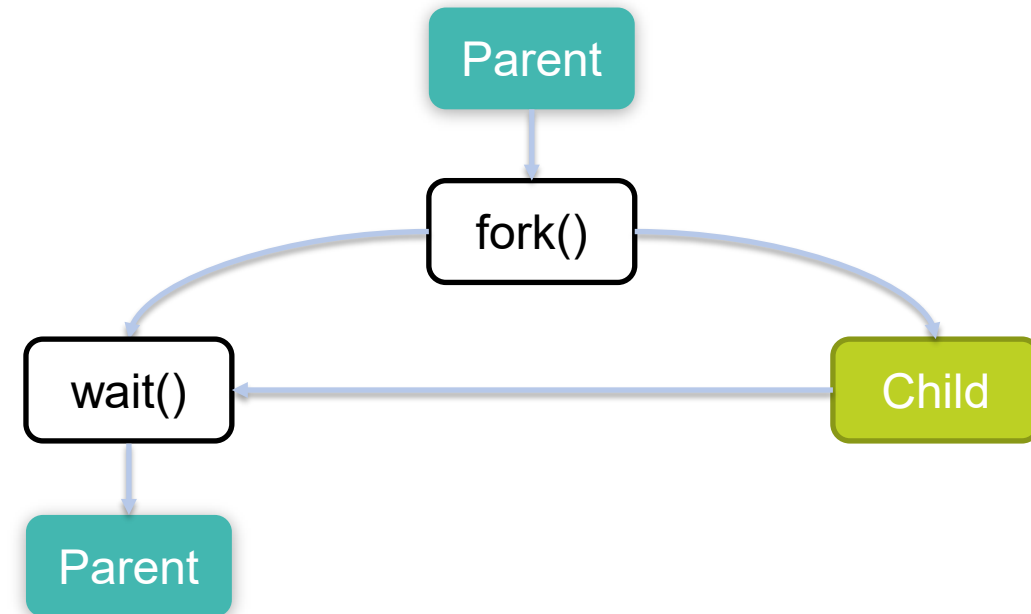
```
hello world (pid:96744)
hello, I am parent of 96745 (pid:96744)
hello, I am child (pid:96745)
```

**Child Process**

**Parent Process**

## 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 process (preventing it from becoming a zombie process). If the parent does not call **wait()** to collect the child's exit status, the child remains in the zombie state, which means its PCB persists in the process table, even though it is no longer running.
  - 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 (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 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 stores pid of the child process that is waited for
        int wc = 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", rc, wc, (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

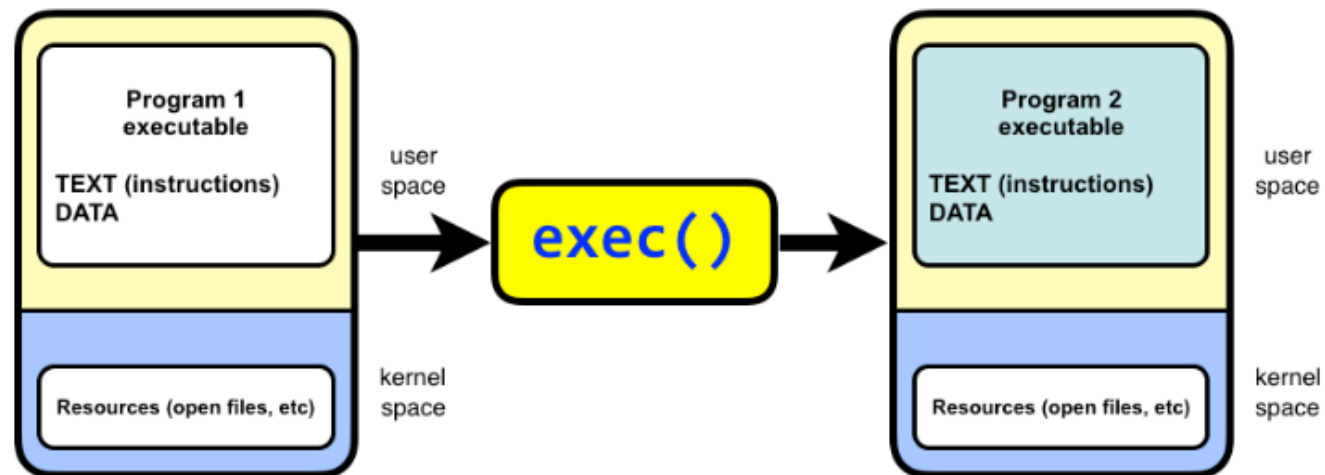
```
hello world (pid:96744)
hello, I am parent of 96745 (pid:96744)
hello, I am child (pid:96745)
```

- **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)
```

## 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[]);`

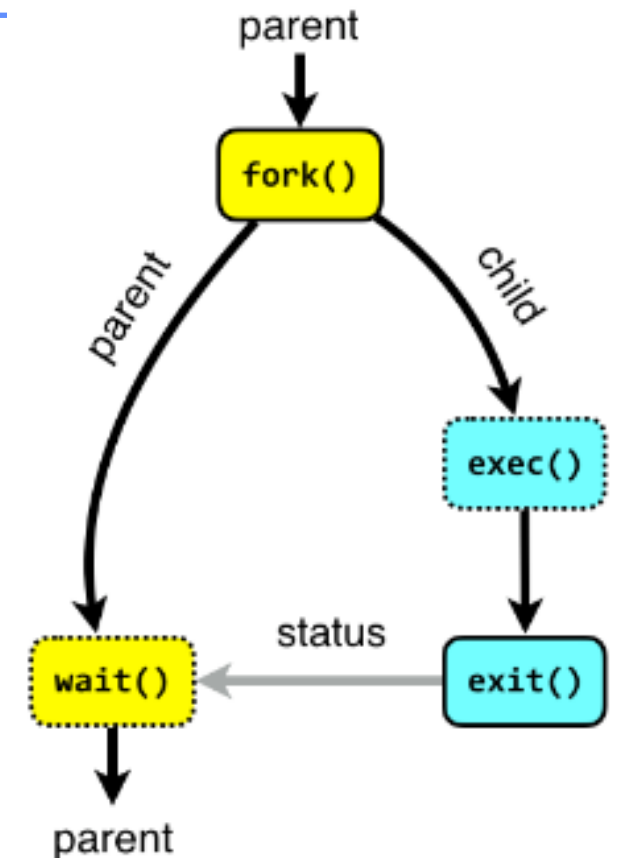


# exec() Example

```
int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n"); exit(1);
    } else if (rc == 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 will be replaced, so not printed out");
    } 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;
}
```

wc: counts Lines, Words, and Bytes in a File:  
Output format: [lines] [words] [bytes] [filename]

```
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)
```





## 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**.

- IO redirection: output of the left command redirected to be written to the file on the right

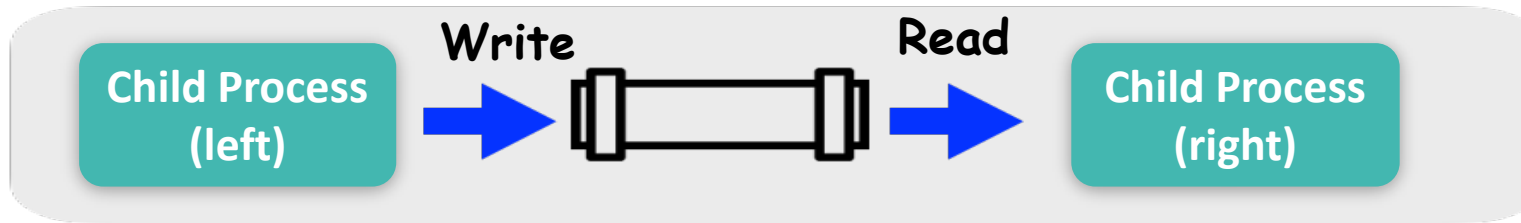
```
% cat w3.c > newfile.txt
```

- 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”



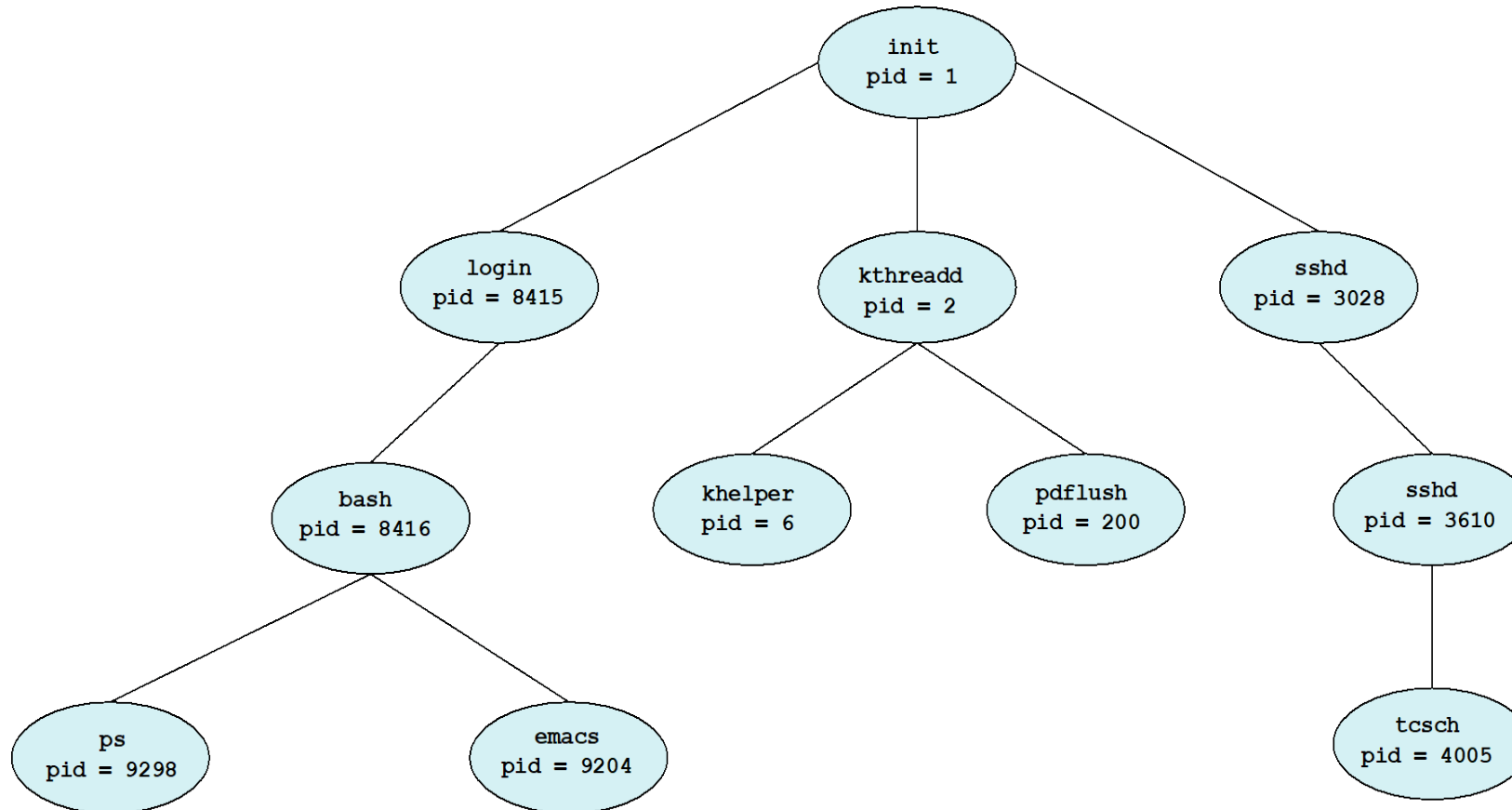
```
(base) dliu@dhcp-10-24-18-121 my_code % cat hello.c |grep printf
```

```
    printf("Hello World!\n");
```

```
(base) dliu@dhcp-10-24-18-121 my_code % █
```

# Process Tree

---



# 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)

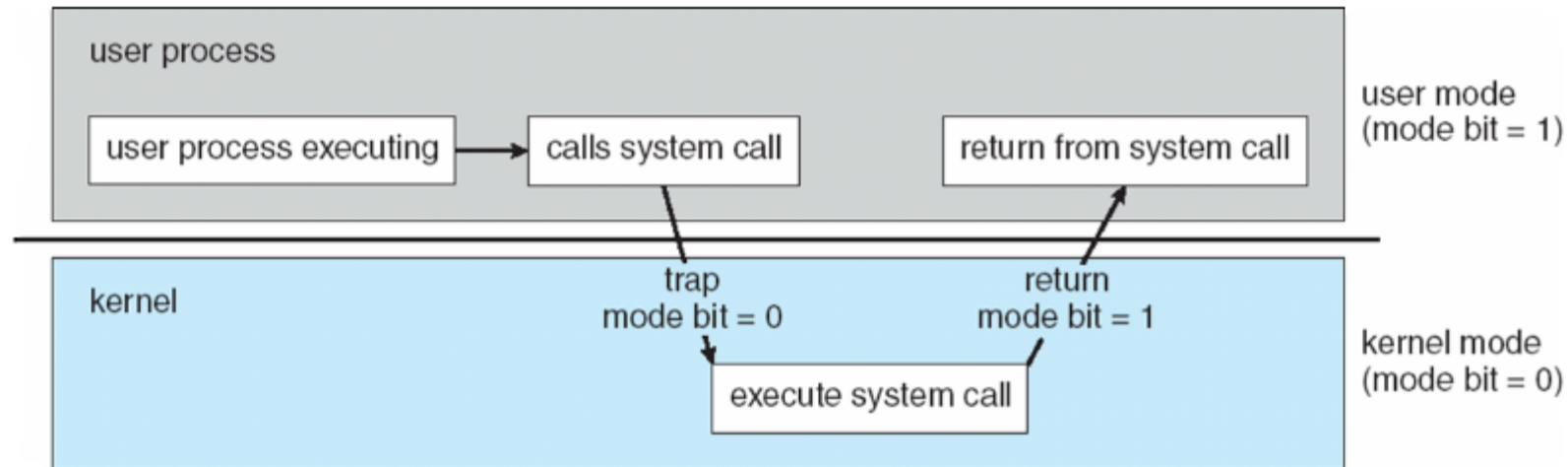
# User/Kernel Mode Separation

---

- **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
  - Non-cooperative approach
- **Cooperative approach**
  - Trust process to relinquish CPU to OS through traps
    - » System calls
    - » Illegal operations, e.g., divided by zero
  - **Issue: if no system call**
- **Non-cooperative approach**
  - The OS takes control
  - OS obtains control periodically, e.g., timer interrupter

## 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
  - 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 “physical 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

# What's needed?

---

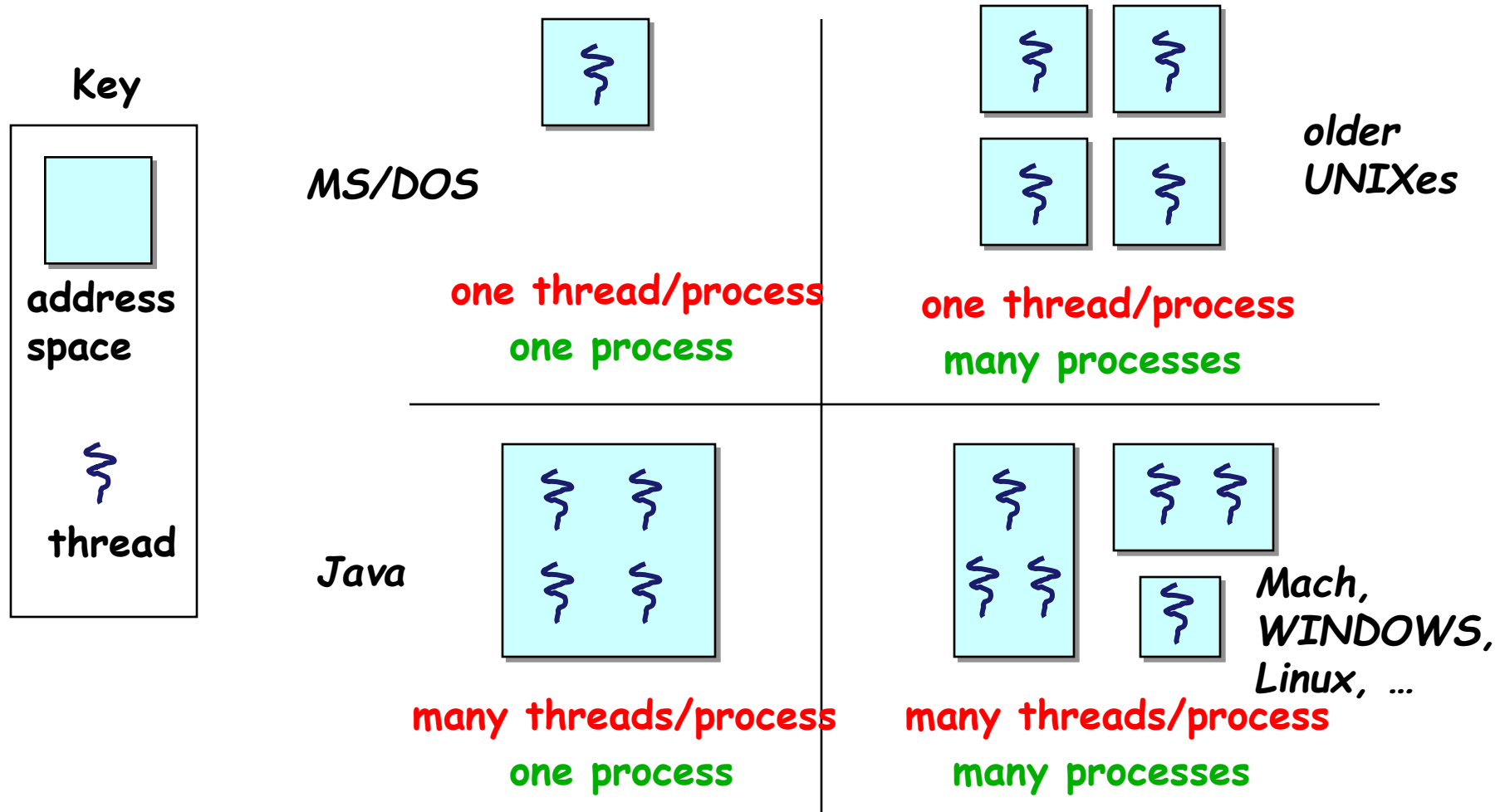
- In each of these examples of concurrency (web server, web client, parallel program):
  - Everybody wants to run the same code
  - Everybody wants to access the same data
  - Everybody has the same privileges
  - Everybody uses the same resources (open files, network connections, etc.)
- But you'd like 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 processor registers and their values
- 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.)
- This execution state is usually called a **thread**, or sometimes, a **lightweight process**

# Processes and Threads

---

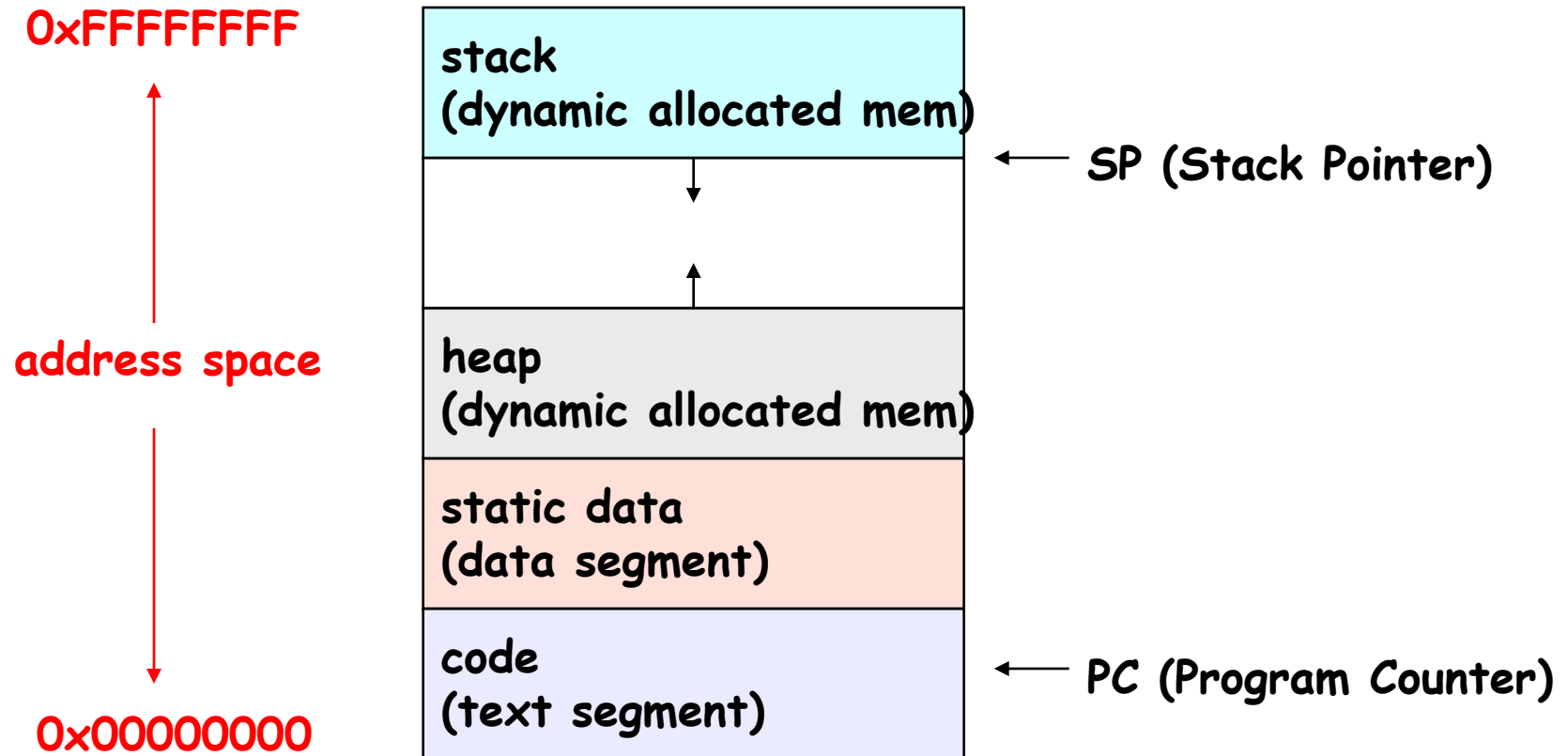
- Modern OSes support two entities:
  - the **process**, which defines the address space and general process attributes (such as open files, etc.)
  - the **thread**, which defines a sequential execution stream within a process
- A thread is bound to a single process / address space
  - address spaces,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

# The design space

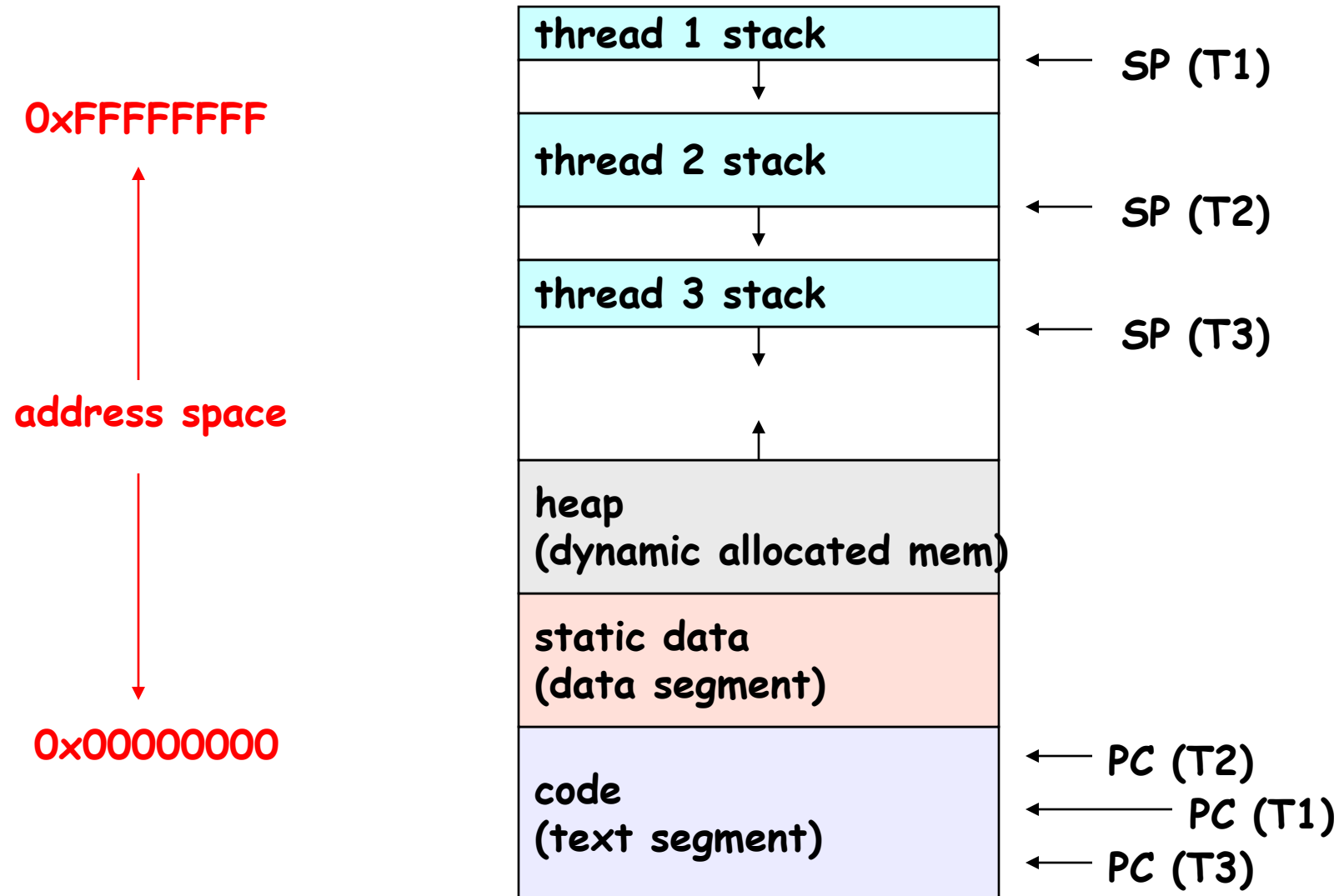


# (old) Process address space

---



## (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 (the Java argument)
- Multithreading is useful even on a uniprocessor
  - even though only one thread can run at a time
- 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”



# “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

---

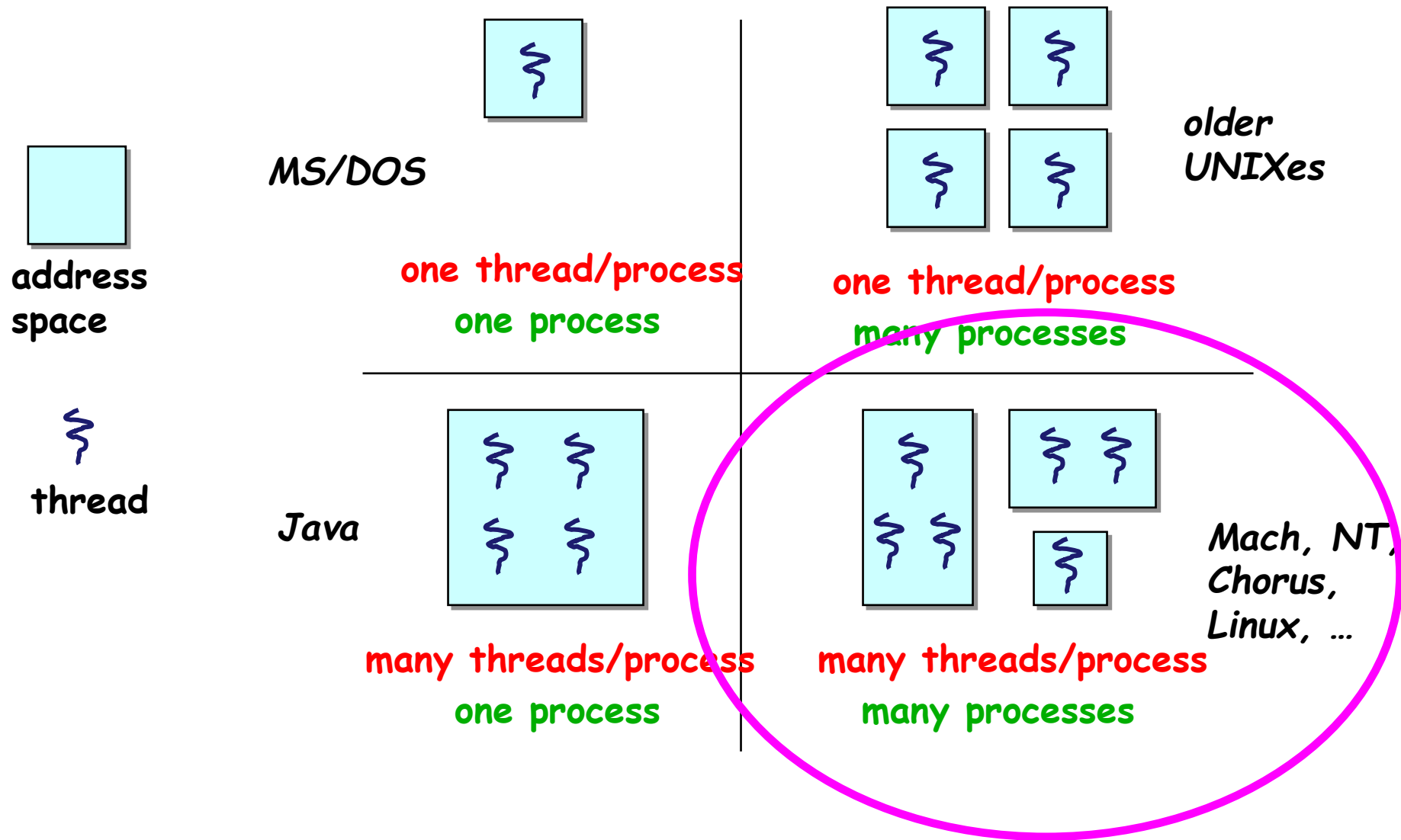
- OS now manages threads *and* processes
  - all thread operations are implemented in the kernel
  - OS schedules all of the threads in a system
    - » if one thread in a process blocks (e.g., on I/O), the OS knows about it, and can run other threads from that process
    - » possible to overlap I/O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

# User-level threads

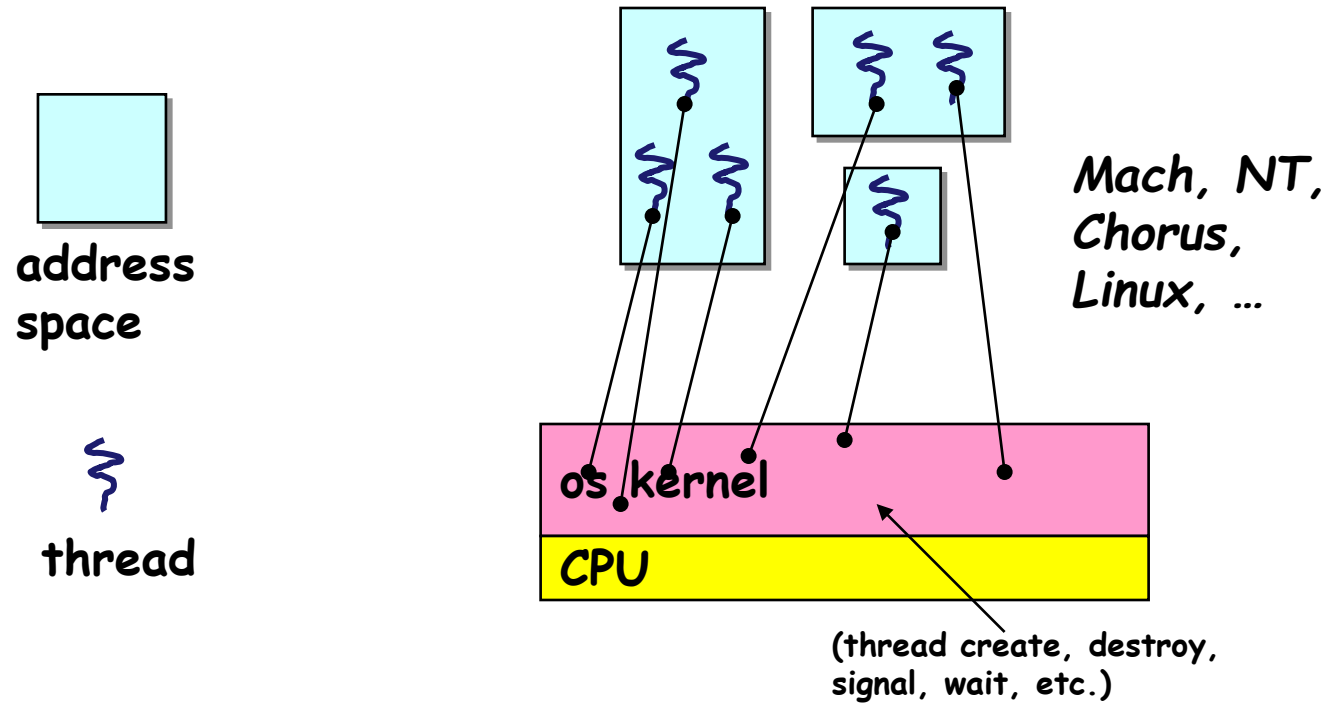
---

- 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

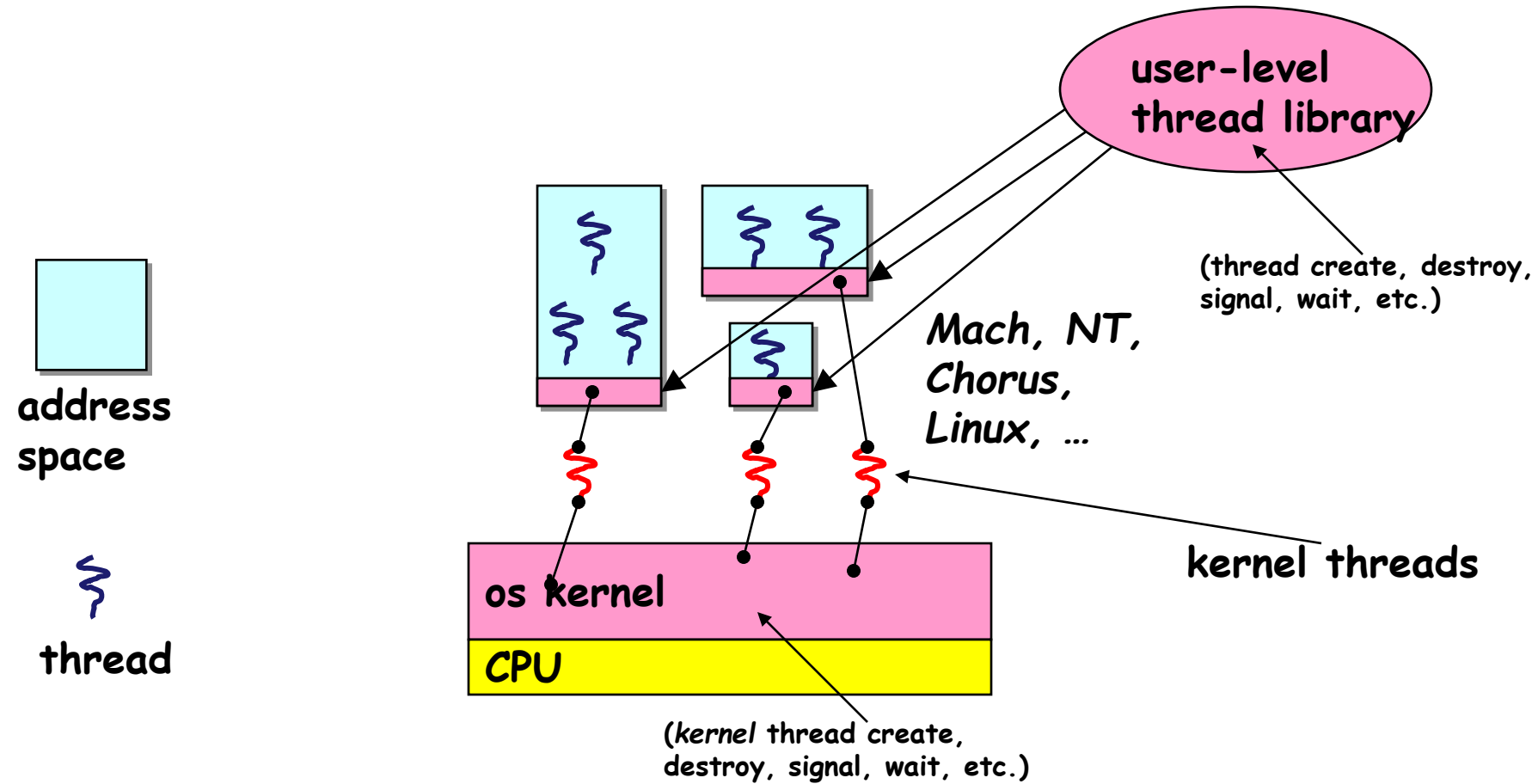
# The design space



# Kernel threads



# User-level threads



# User-level thread implementation

---

- The kernel believes the user-level process is just a normal process running code
  - But, this code includes the thread support library and its associated thread scheduler
- The thread scheduler determines when a thread runs
  - it uses queues to keep track of what threads are doing: run, ready, wait
    - » just like the OS and processes
    - » but, implemented at user-level as a library
- Example implementations of user-level threads
  - Fibers, co-routines



# Thread interface

---

- The POSIX pthreads API:
  - `t = pthread_create(attributes, start_procedure)`
    - » creates a new thread of control
    - » new thread begins executing at `start_procedure`
  - `pthread_cond_wait(condition_variable)`
    - » the calling thread blocks, sometimes called `thread_block()`
  - `pthread_signal(condition_variable)`
    - » starts the thread waiting on the condition variable
  - `pthread_exit()`
    - » terminates the calling thread
  - `pthread_wait(t)`
    - » waits for the named thread to terminate

# How to prevent a user-level thread from hogging the CPU?

---

- Strategy 1: force everyone to cooperate
  - a thread willingly gives up the CPU by calling **yield()**
  - **yield()** calls into the scheduler, which context switches to another ready thread
  - what happens if a thread never calls **yield()**?
- Strategy 2: use preemption
  - scheduler requests that a timer interrupt be delivered by the OS periodically
    - » usually delivered as a UNIX signal (`man signal`)
    - » signals are just like software interrupts, but delivered to user-level by the OS instead of delivered to OS by hardware
  - at each timer interrupt, scheduler gains control and context switches as appropriate

# Thread context switch

---

- Very simple for user-level threads:
  - save context of currently running thread
    - » push machine state onto thread stack
  - restore context of the next thread
    - » pop machine state from next thread's stack
  - return as the new thread
    - » execution resumes at PC of next thread
- This is all done by assembly language
  - it works at the level of the procedure calling convention
    - » thus, it cannot be implemented using procedure calls
    - » e.g., a thread might be preempted (and then resumed) in the middle of a procedure call

## What if a thread tries to do I/O?

---

- The kernel thread is lost for the duration of the (synchronous) I/O operation!
- Could have one kernel thread for each user-level thread
  - no real difference from kernel threads – “common case” operations (e.g., synchronization) would be quick
- Could have a limited-size “pool” of kernel threads “powering” all the user-level threads in the address space
  - the kernel will be scheduling these threads, oblivious to what’s going on at user-level

# Summary

---

- We want multiple threads per address space
- Kernel threads are much more efficient than processes, but they're still not cheap
  - all operations require a kernel call and parameter verification
- User-level threads are:
  - fast
  - great for common-case operations
    - » creation, synchronization, destruction
  - can suffer in uncommon cases due to kernel obliviousness
    - » I/O
    - » preemption of a lock-holder