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

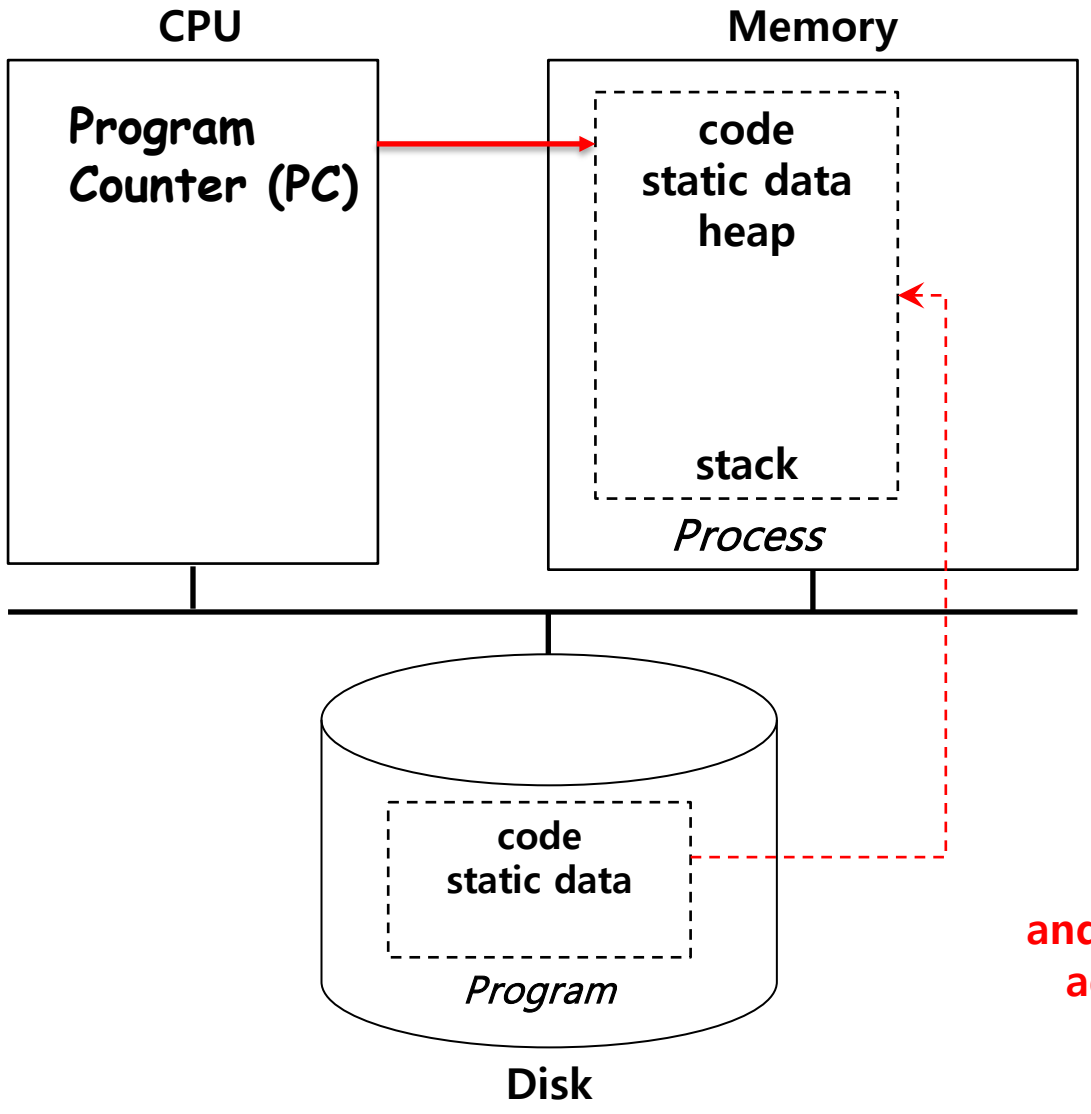
Overview

- 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



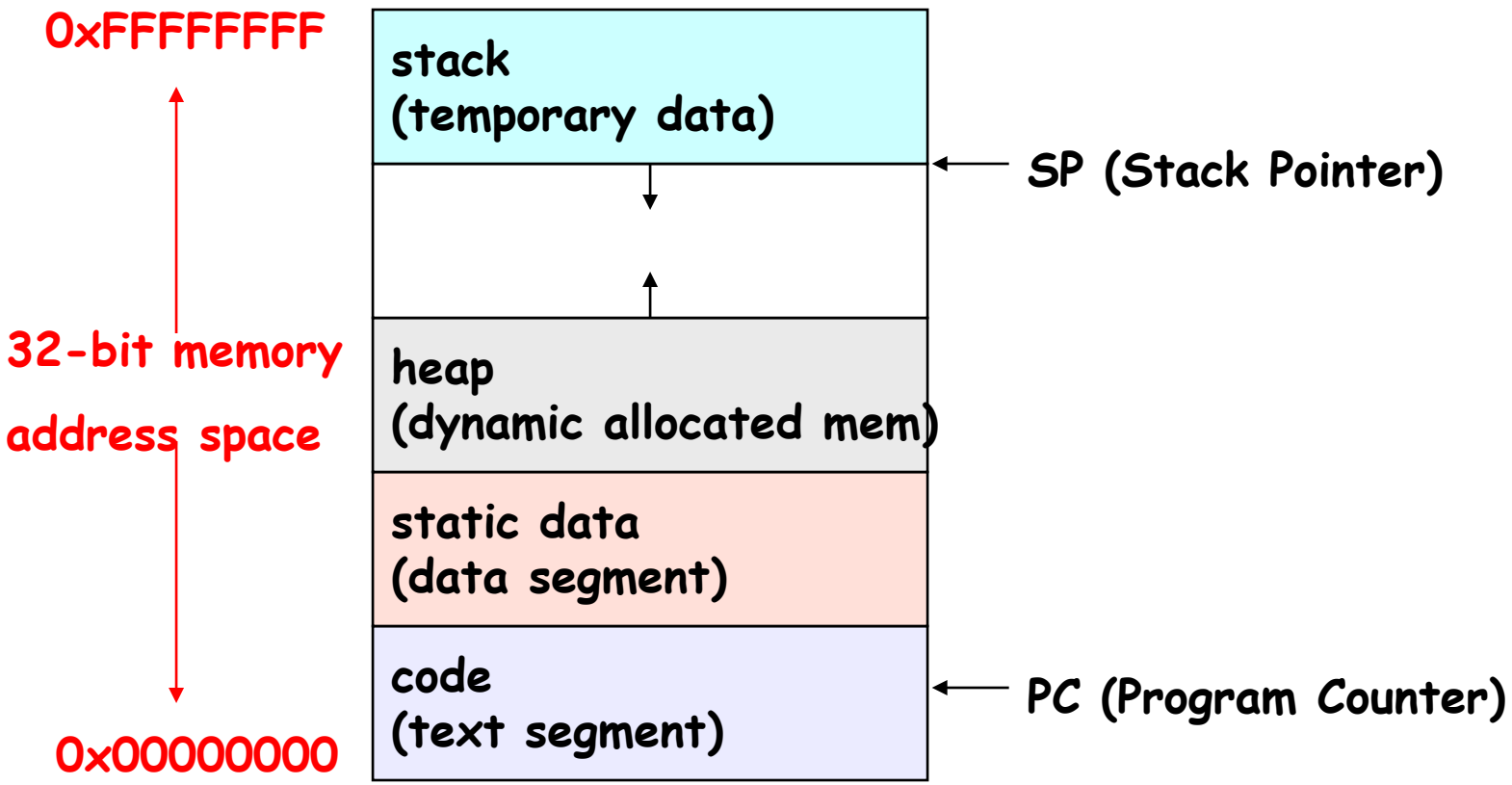
- A program becomes a process when it is selected to execute and loaded into memory.
- A process has an **address space**

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
  - Static data: Global variables
  - Code: Instructions
  - Registers: SP (Stack Pointer), PC (Program counter)



Process

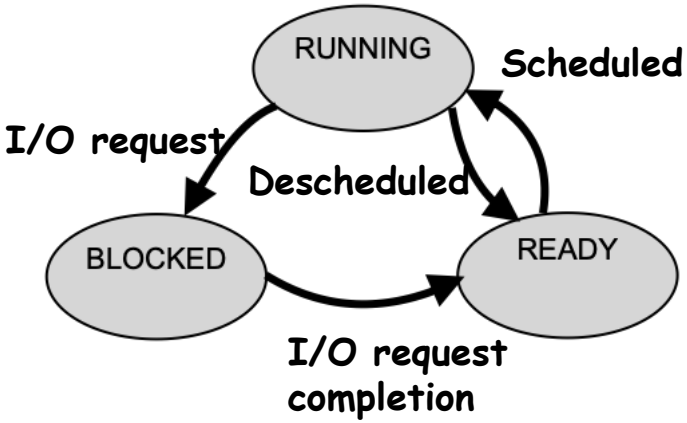
```
struct proc {
        struct spinlock lock; // p->lock must be held when using these:
        enum procstate state; // Process state
        void *chan; // If non-zero, sleeping on chan
        int killed; // If non-zero, have been killed
        int xstate; // Exit status to be returned to parent's 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)
};
```

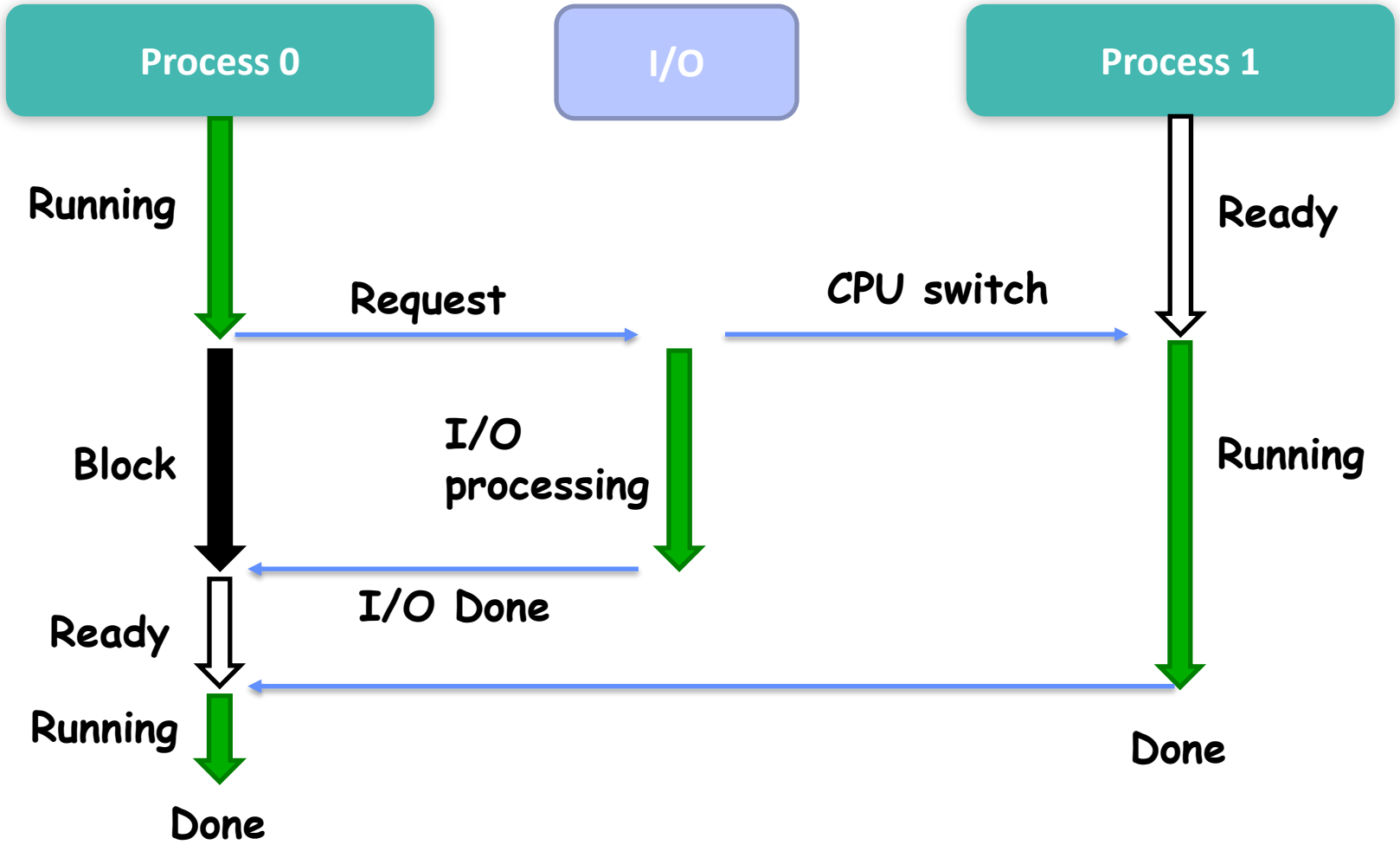
XV6 (proc.h)

- 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=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

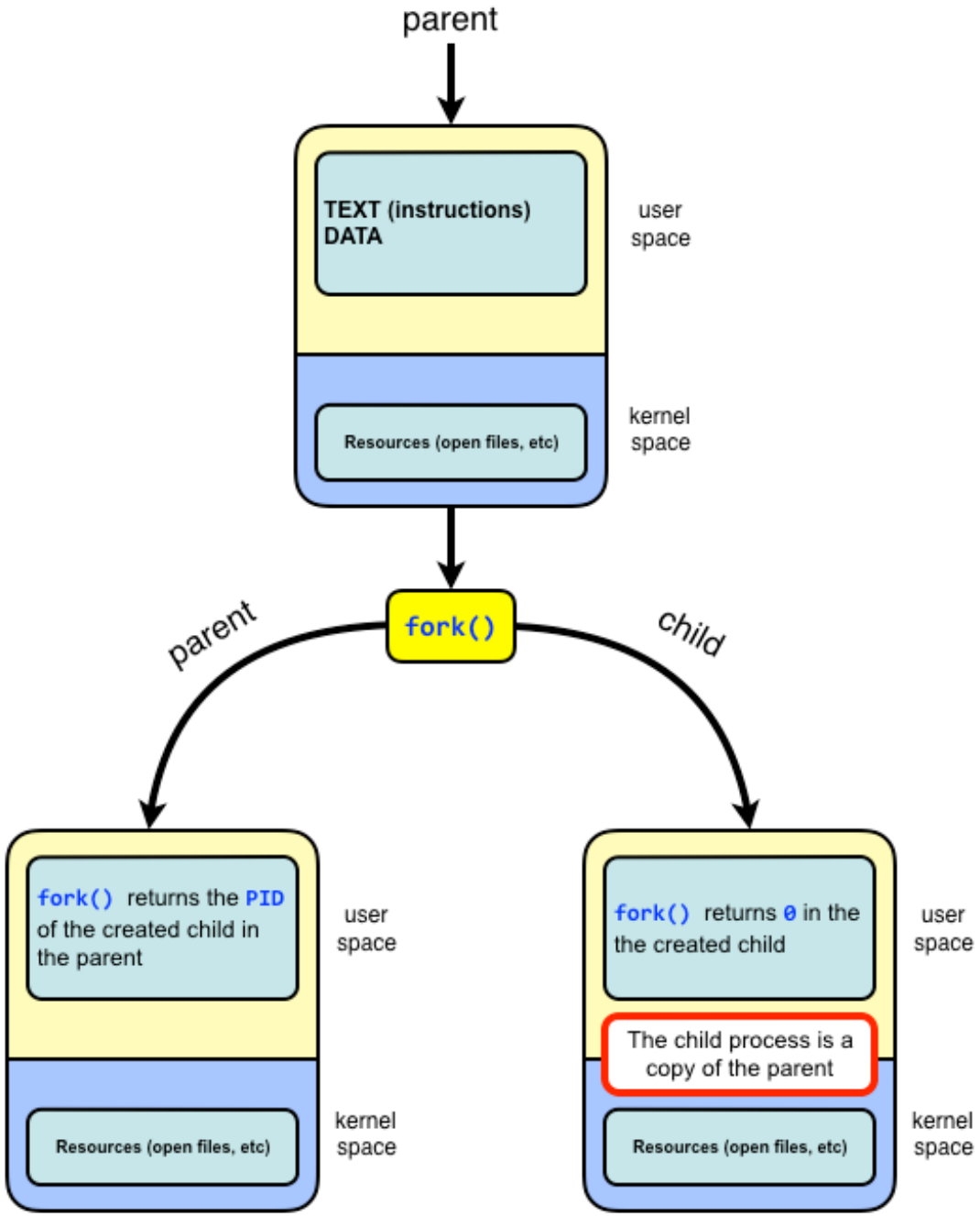
Process Creation

- 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
  - **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

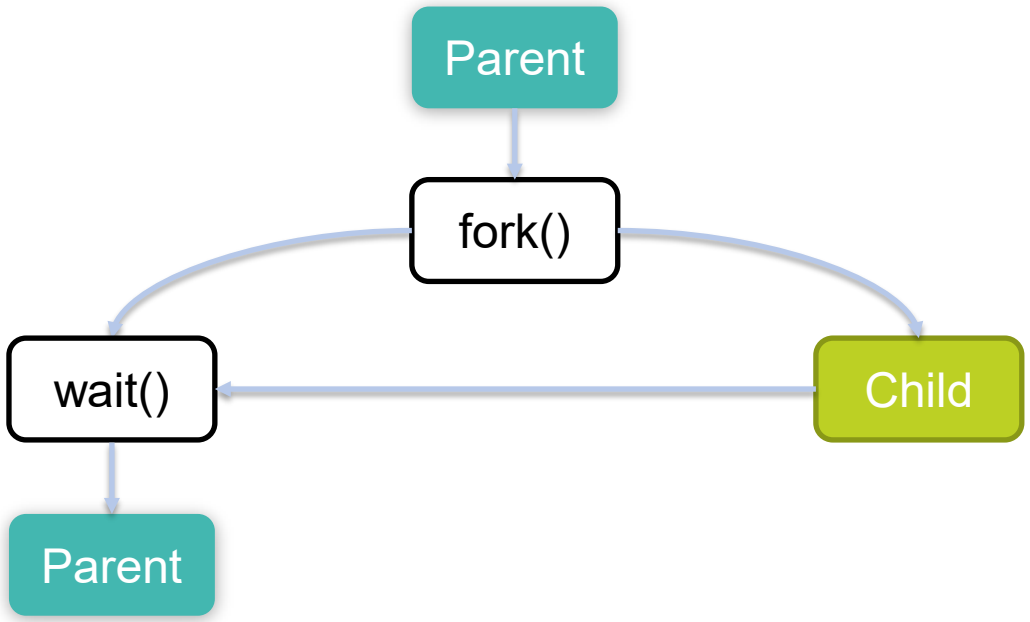
```
hello world (pid:96744)
hello, I am parent of 96745 (pid:96744)
hello, I am child (pid:96745)
```

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. 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 (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)
```

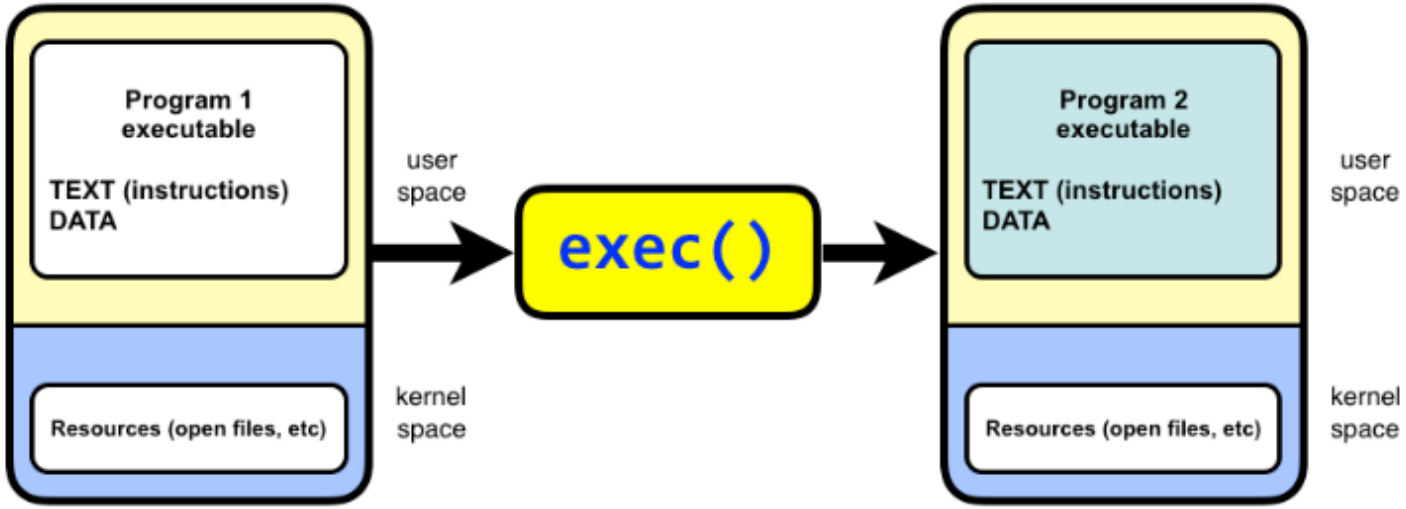
- **With wait():** child runs first, and parents waits for child to finish

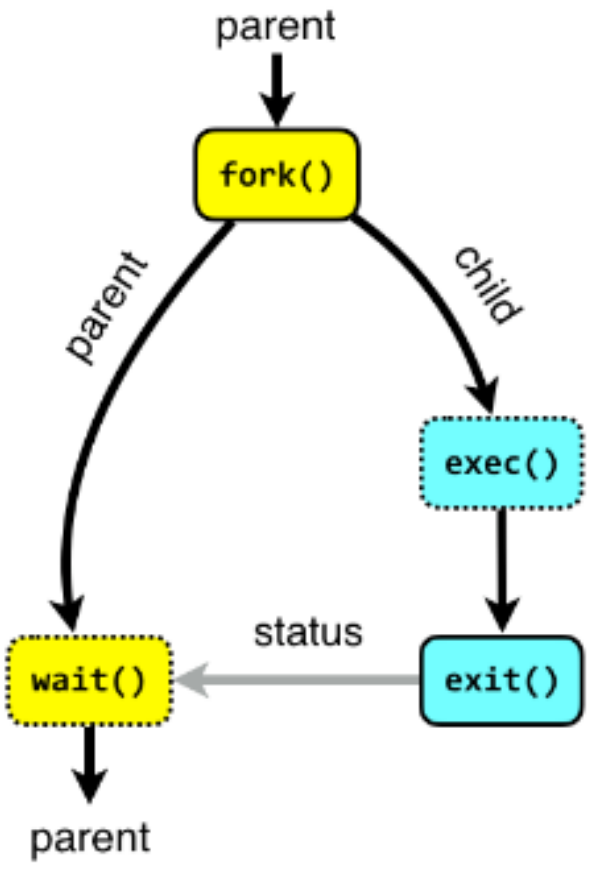
```
hello world (pid:96848)
hello, I am child (pid:96849)
hello, I am parent of 96849 (wc:96849) (pid:96848)
```

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 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 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.
- The strdup() function 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.
- 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.");" is overwritten and will never be called.

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

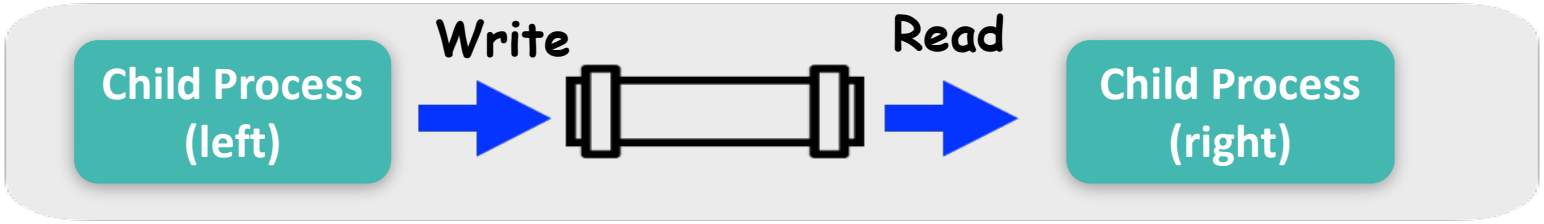
```
% cat w3.c > newfile.txt
```

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

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

pipe

- A communication method between two processes



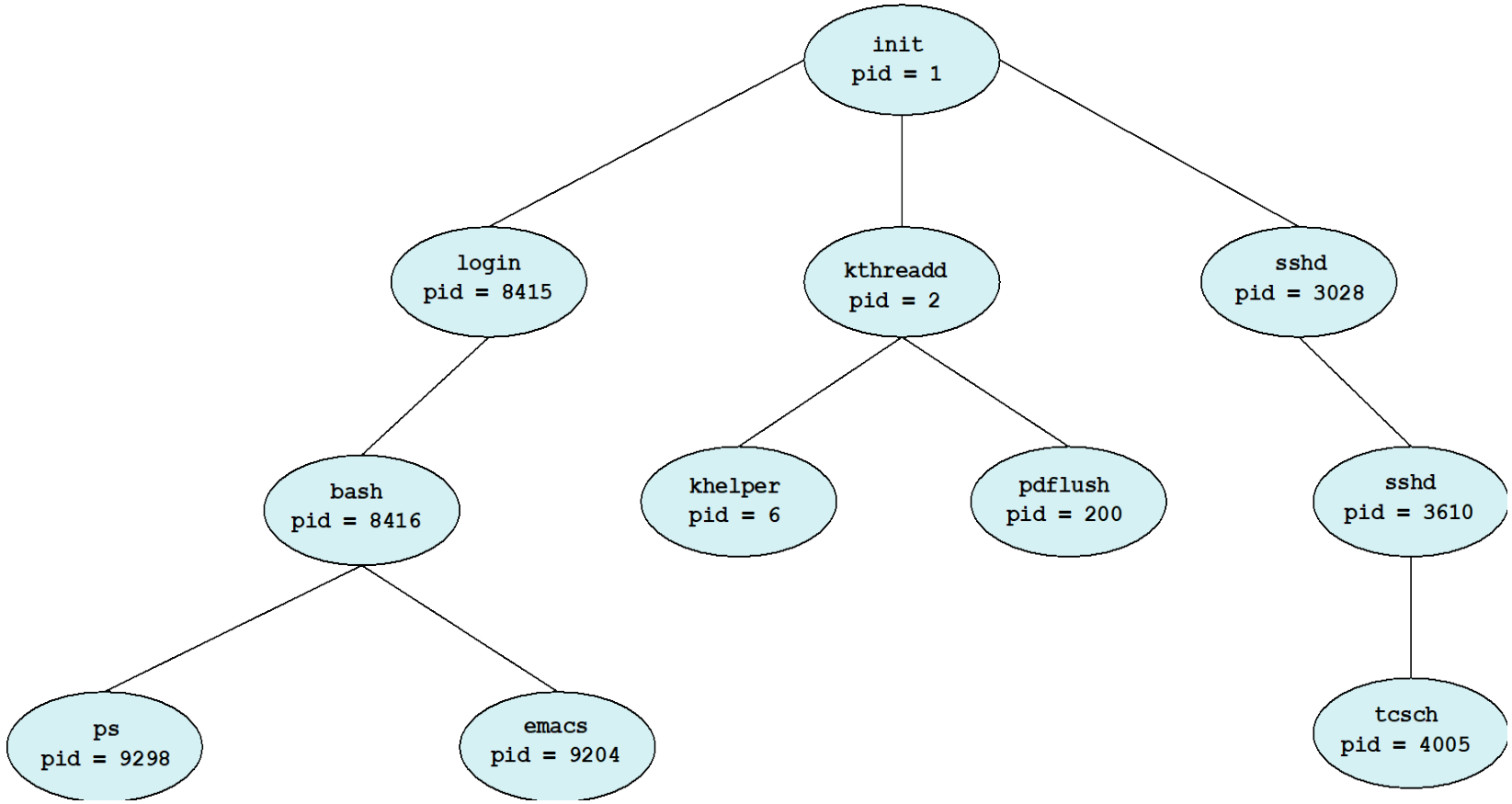

```
(base) dliu@dhcp-10-24-18-121 my_code % cat hello.c
#include <stdio.h>

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



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)
```

Quiz: Fork

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
  pid_t p = fork();
  if(p<0){
      perror("fork fail");
      exit(1);
  }
  printf("Hello world!, process_id(pid) =
%d\n", getpid());
  return 0;
}
```

- Since we do not check for return value of fork(), both child process and parent process run the same code after fork, and print out its own pid. (The pids 32, 33 shown are just examples)

Output:
Hello world!, process_id(pid) = 32
Hello world!, process_id(pid) = 33

https://www.geeksforgeeks.org/fork-system-call/

Quiz: Fork

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
  int x = 1;
  pid_t p = fork();
  if(p<0){
    perror("fork fail");
    exit(1);
  }
  else if (p == 0){
    printf("Child has x = %d\n",
++x); }
  else{
    printf("Parent has x = %d\n",
--x); }
  return 0;
}
```

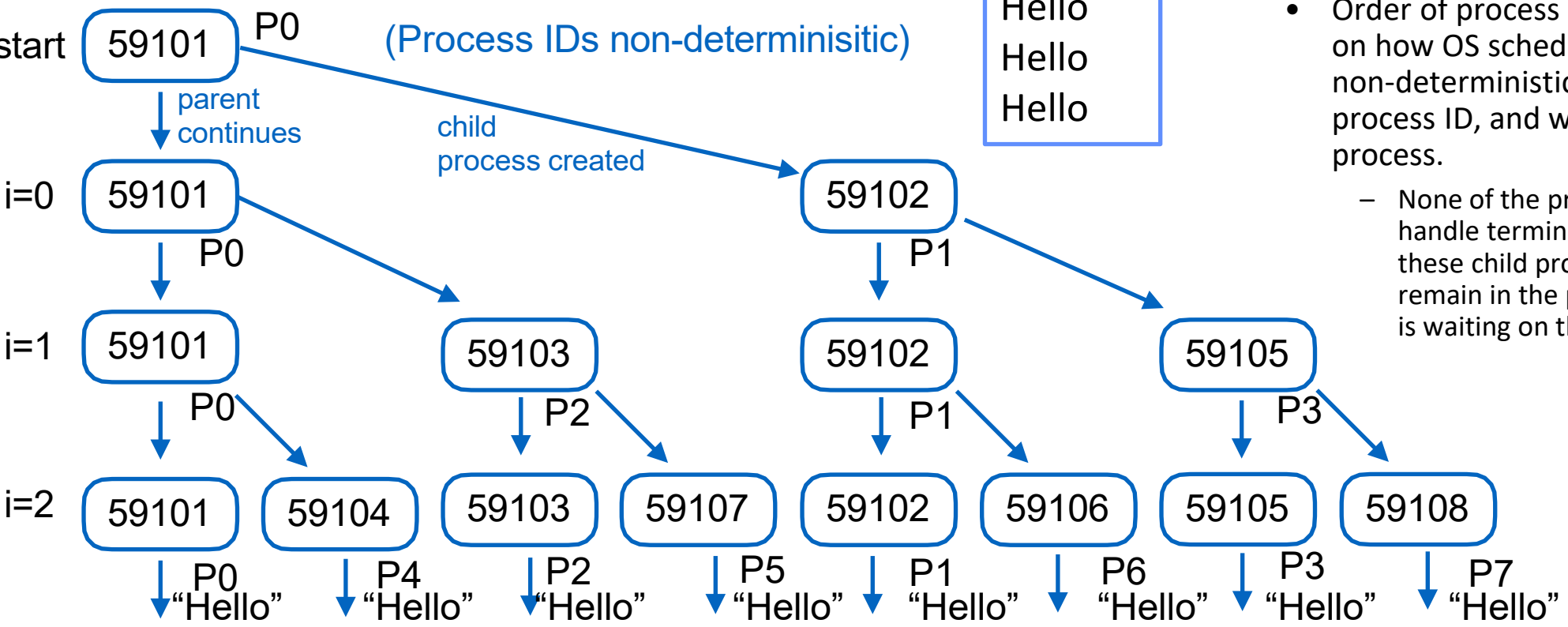
- Updates to global variable x in the parent process and child process do not affect each other because they have separate address spaces, and separate copies of data variables.
- Since parent process and child process run concurrently without wait(), two outputt interleavings are possible.

Output:
Parent has x = 0
Child has x = 2
Or,
Child has x = 2
Parent has x = 0

Quiz: Fork

```
int main() {
  int i;

  for (i = 0; i < 3; i++)
  {fork();}
  printf("Hello\n"); //outside for loop
  return 0;
}
```

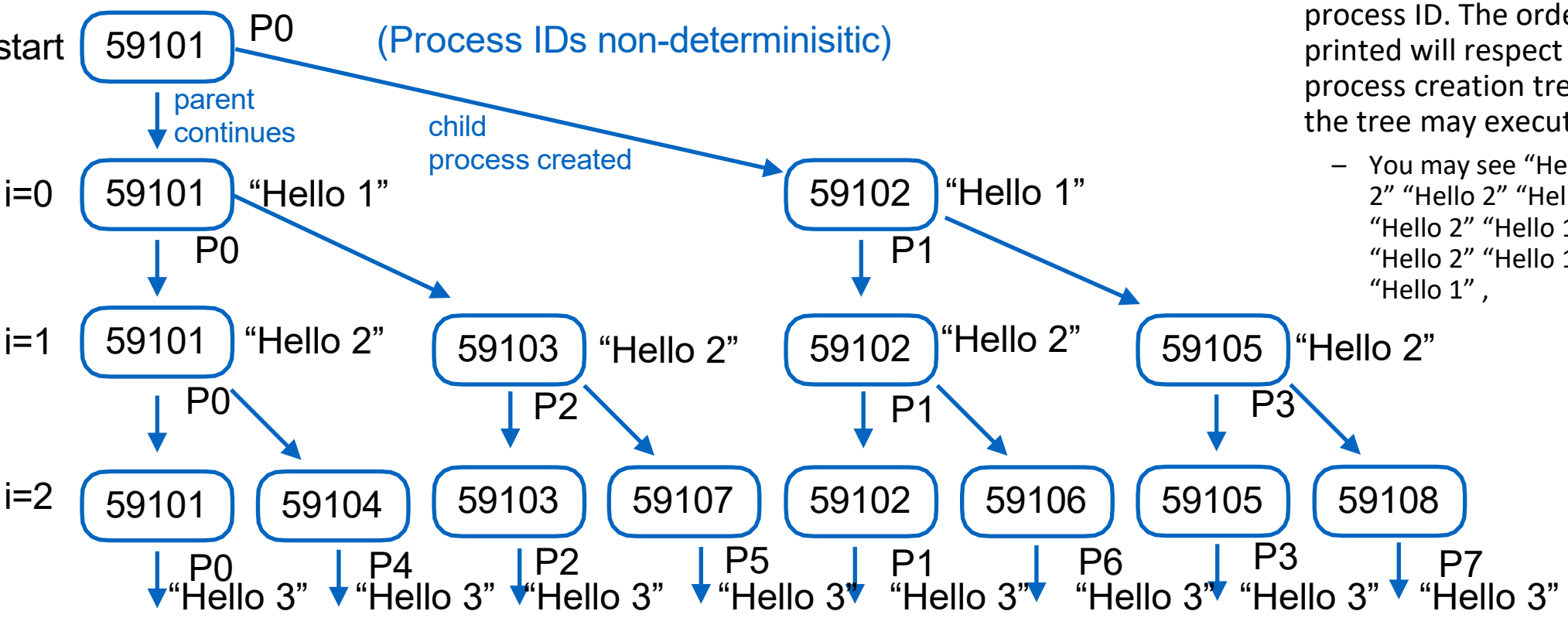
Output:
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello



- In general, "for (i = 0; i < n; i++) fork();" creates 1+2+…+2^(n-1)=(2^n)-1 child processes. Plus the main process P0, we have a total of 2^n processes, hence "Hello" is printed 2^n times. Here n = 3, 2^3 = 8.
  - Main process: P0
  - P0 creates 1 child process by the 1st fork: P1
  - P0, P1 create 2 child processes by the 2nd fork: P2, P3
  - P0, P1, P2, P3 create 4 child processes by the 3rd fork: P4, P5, P6, P7
- Order of process execution may vary depending on how OS schedules these processes, so it is non-deterministic which process gets which process ID, and which Hello is printed by which process.
  - None of the processes include a wait() call to handle terminated child processes. When any of these child processes terminate, their PCBs remain in the process table as no parent process is waiting on them, resulting in zombie processes.

Quiz: Fork

```
int main() {
  int i;

  for (i = 0; i < 3; i++)
  {fork();
  printf("Hello i\n"); } //inside for loop
  return 0;
}
```



- This program will print 14 lines.
  - Main process: P0
  - P0 creates 1 child process by the 1st fork: P1. Then P0 and P1 each prints "Hello 1"
  - P0, P1 create 2 child processes by the 2nd fork: P2, P3. Then P0, P1, P2, P3 each prints "Hello 2"
  - P0, P1, P2, P3 create 4 child processes by the 3rd fork: P4, P5, P6, P7. Then P0 to P7 each prints "Hello 3"
- Order of process execution may vary depending on how OS schedules these processes, so it is non-deterministic which process gets which process ID. The order in which "Hello i" is printed will respect the dependencies in the process creation tree, but parallel branches in the tree may execute in any order
  - You may see "Hello 1" "Hello 1" "Hello 2" "Hello 2" "Hello 2" "Hello 2", or "Hello 1" "Hello 2" "Hello 2" "Hello 1" "Hello 2" "Hello 2", but NOT "Hello 2" "Hello 1" "Hello 2" "Hello 2" "Hello 2" "Hello 1" ,

Quiz: Fork

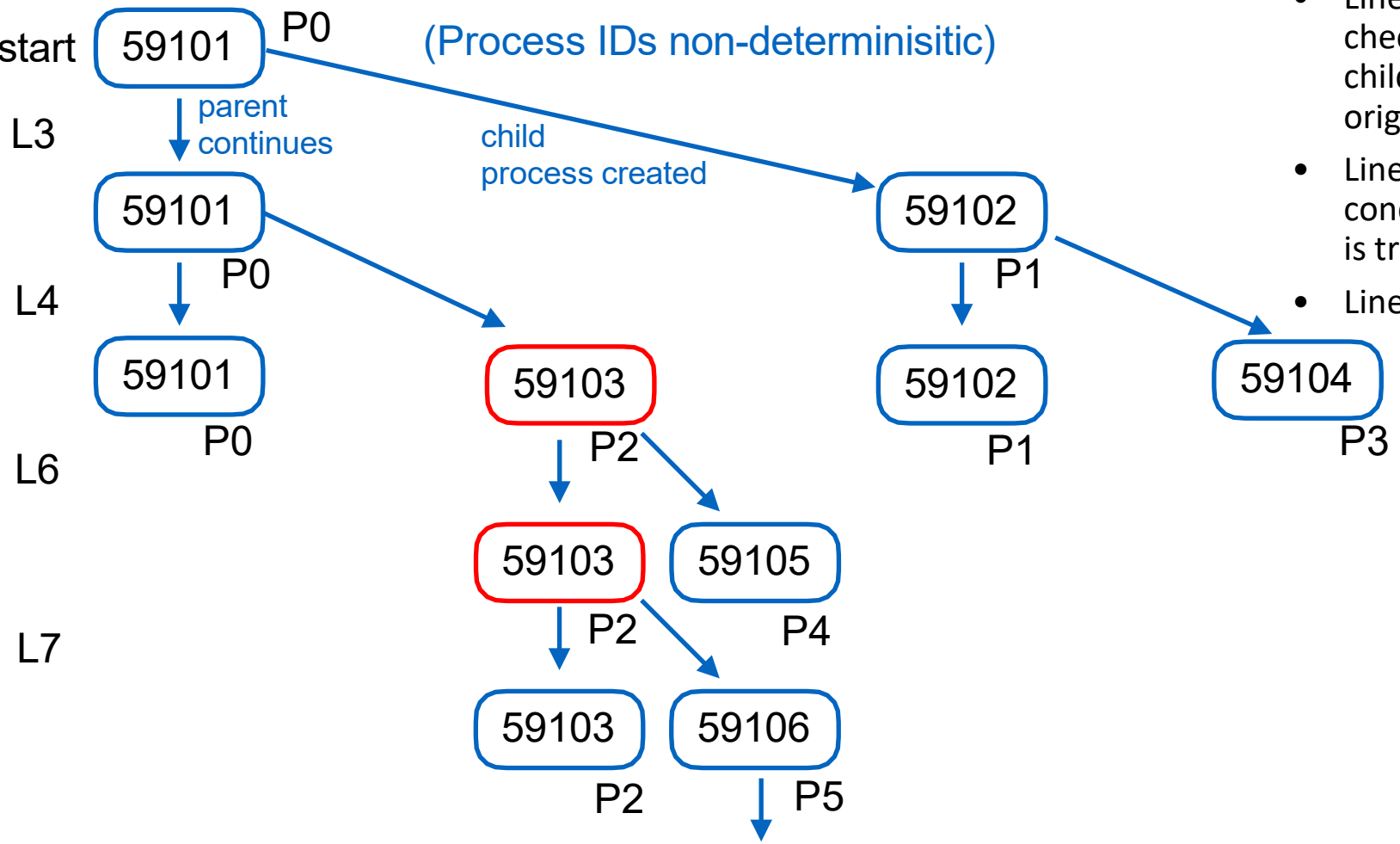
```
int main() {
  While(true) fork();
  return 0;
}
```

```
me@Proton ~ % ulimit -a
-t: cpu time (seconds)              unlimited
-f: file size (blocks)              unlimited
-d: data seg size (kbytes)          unlimited
-s: stack size (kbytes)             8176
-c: core file size (blocks)         0
-v: address space (kbytes)          unlimited
-l: locked-in-memory size (kbytes)  unlimited
-u: processes                       2666
-n: file descriptors                2560
me@Proton ~ %
```

- A fork bomb is a type of denial-of-service (DoS) attack designed to exhaust system resources by creating an exponential number of processes. This is achieved through self-replicating code that repeatedly calls the fork() system call. The result is resource starvation, which can slow down or crash the system.
- Prevention countermeasures:
  - Limit User Processes: Use ulimit in Linux to restrict the number of processes a user can create:
    - » ulimit -u 30 # Limits user to 30 processes
  - Configure /etc/security/limits.conf for persistent limits:
    - » username hard nproc 30

Quiz: Fork

```
1  #include <unistd.h>
2 int main (void) {
3   pid_t pid1 = fork();
4   pid_t pid2 = fork() ;
5   if (pid1>0 && pid2==0){
6    if (fork()>0){
7      fork();}
8   } // end if
9 } // end main
```



- Q: Give the total number of processes resulting from executing the code, including the process executing this code itself.
- A: There are 6 processes in total.
- Line 3: The initial process P0 generates one child process P1.
- Line 4: Two processes P0 and P1 generate additional child processes P2 and P3.
- Line 5: The condition “if (pid1 > 0 && pid2 == 0)” is checked in all four processes, and it is true only for the child process P2 created by the second fork() in the original parent process.
- Line 6: Process P2 generates one child process P4. The condition “if (fork() > 0)” is checked in P2 and P4, and it is true only for P2 (parent process of P4).
- Line 7: Process P2 generates one more child process P5.

Quiz: Fork

```
if (fork() == 0) {
    a = a + 5;
    printf("a=%d, a memory address=%d\n",
a, &a);
}
else {
    a = a -5;
    printf("a=%d, a memory address=%d\n",
a, &a);
}
```

- Let u, v be the values printed by the parent process, and x, y be the values printed by the child process. Which one of the following is TRUE?
- (A) u = x + 10 and v = y
- (B) u = x + 10 and v != y
- (C) u + 10 = x and v = y
- (D) u + 10 = x and v != y

- ANS: (C) u + 10 = x and v = y
- fork() returns 0 in child process and process ID of child process in parent process. In Child (x), a = a + 5 In Parent (u), a = a – 5; Therefore x = u + 10.
- The physical addresses of 'a' in parent and child must be different. But our program accesses virtual addresses (assuming we are running on an OS that uses virtual memory). The child process gets an exact copy of parent process and virtual address of 'a' doesn't change in child process. Therefore, we get same addresses in both parent and child.

Fork Quiz

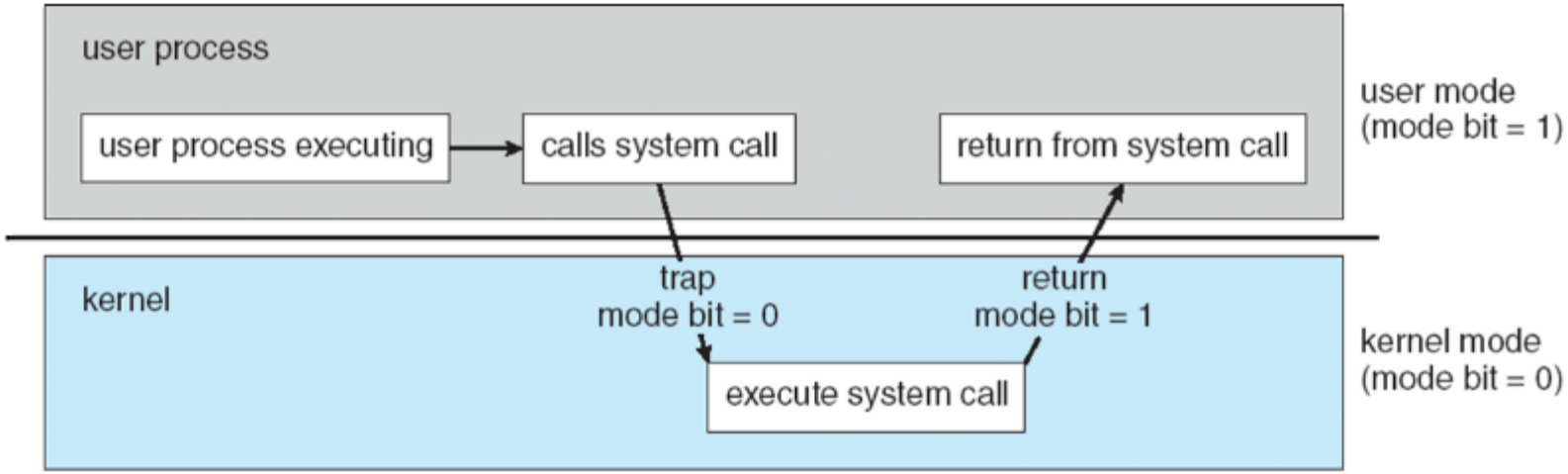
User/Kernel Mode Separation

- **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

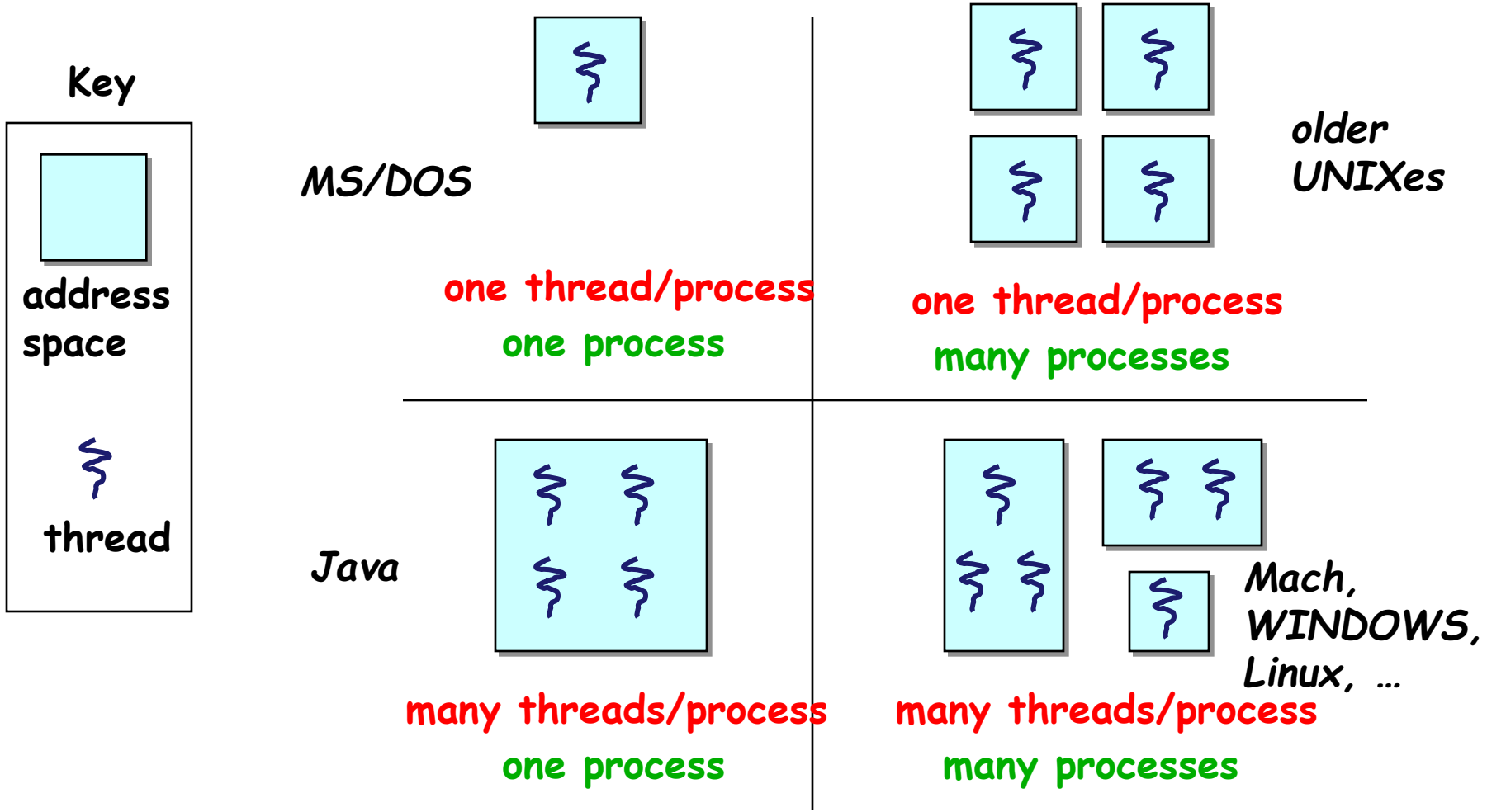
What's needed?

- 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

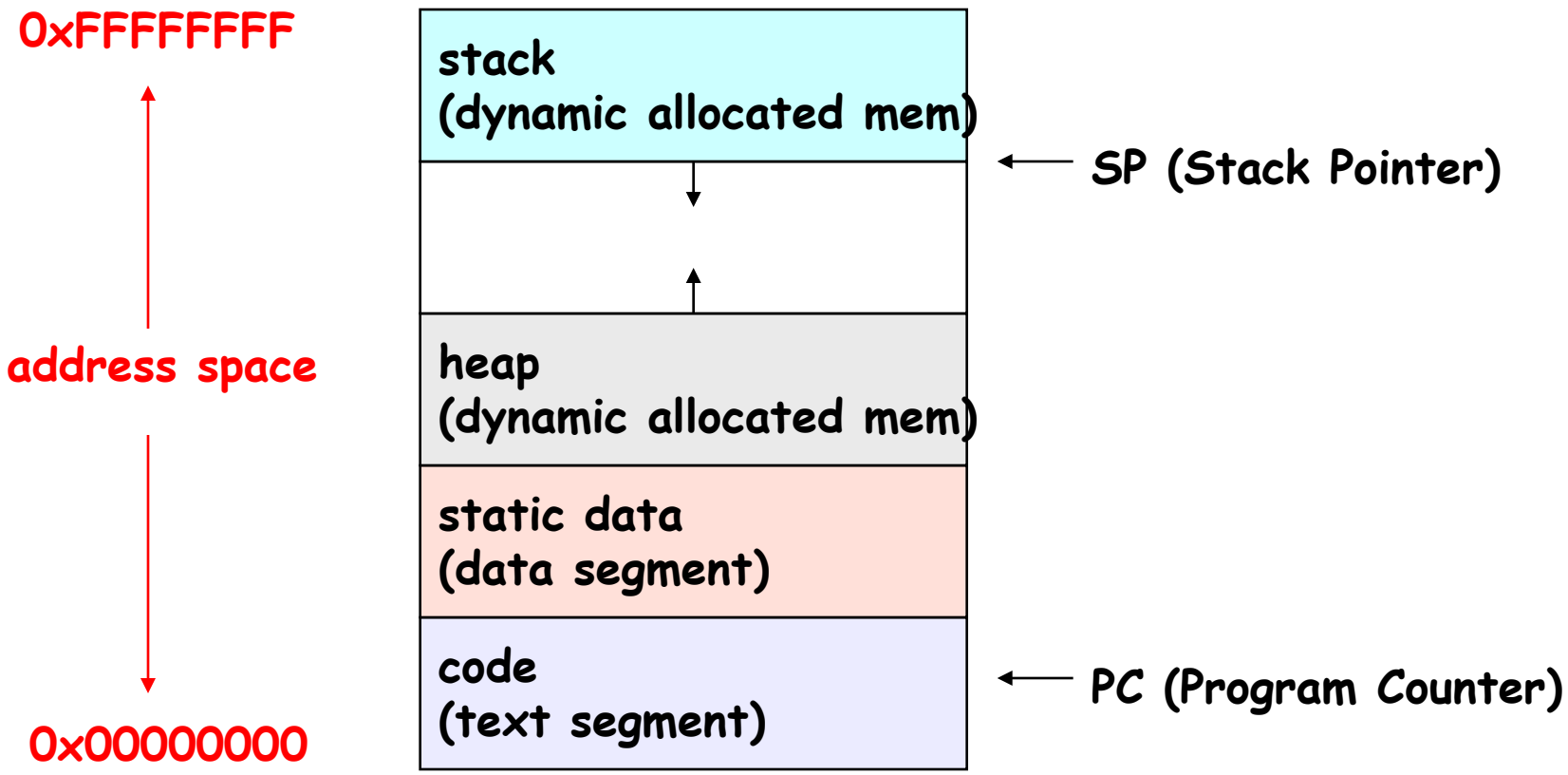
Processes and Threads

- 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

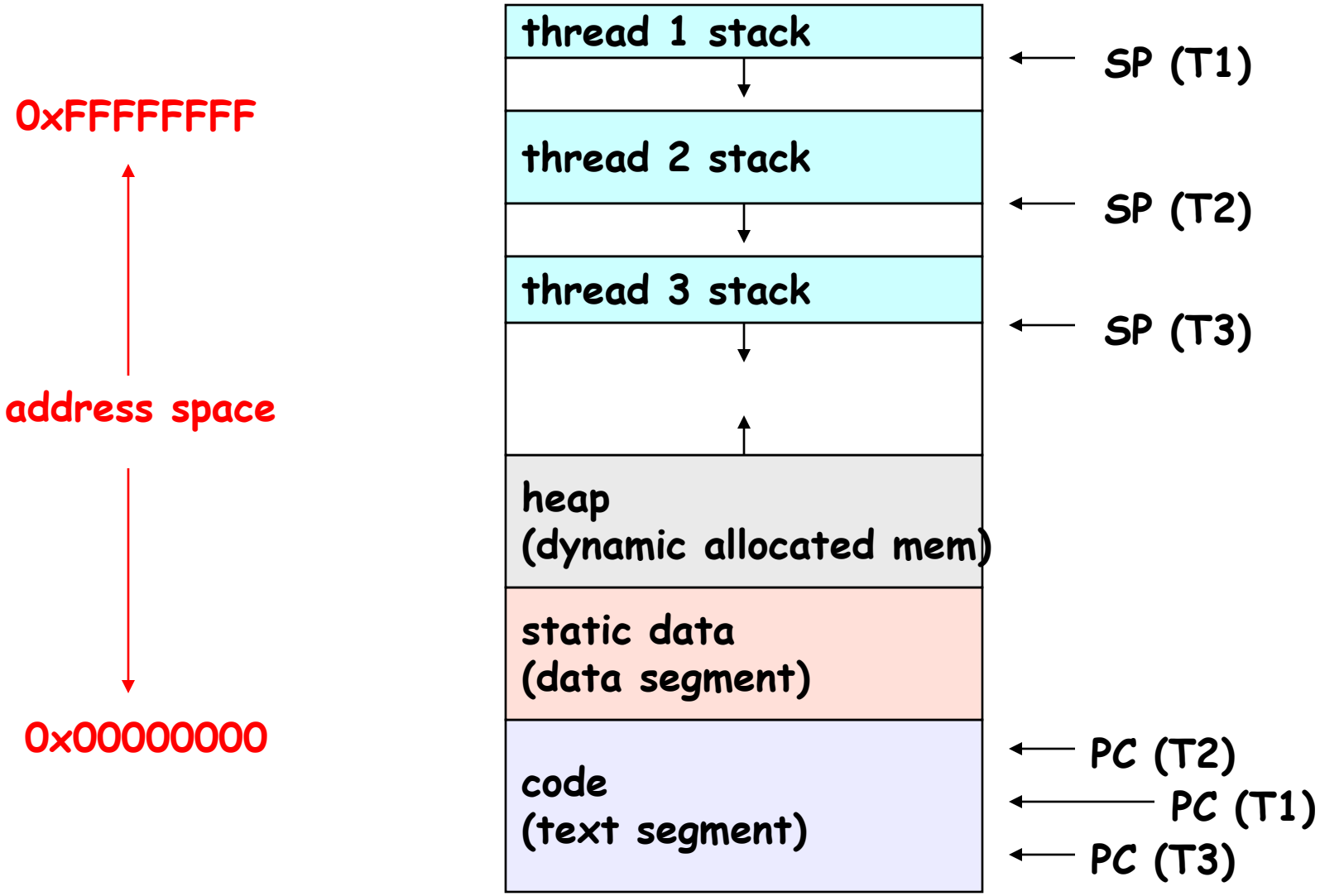
The design space



(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

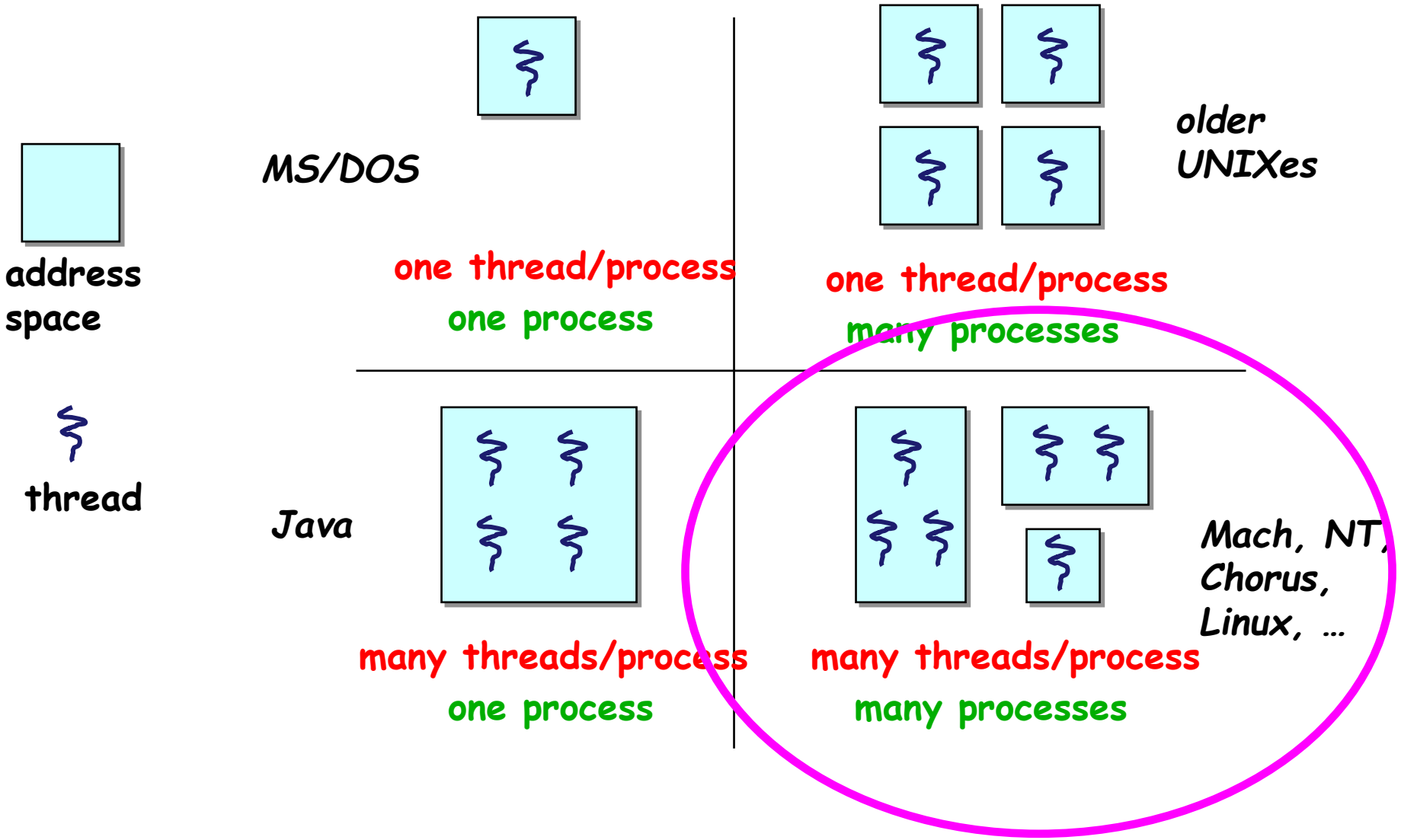
Kernel threads

- 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

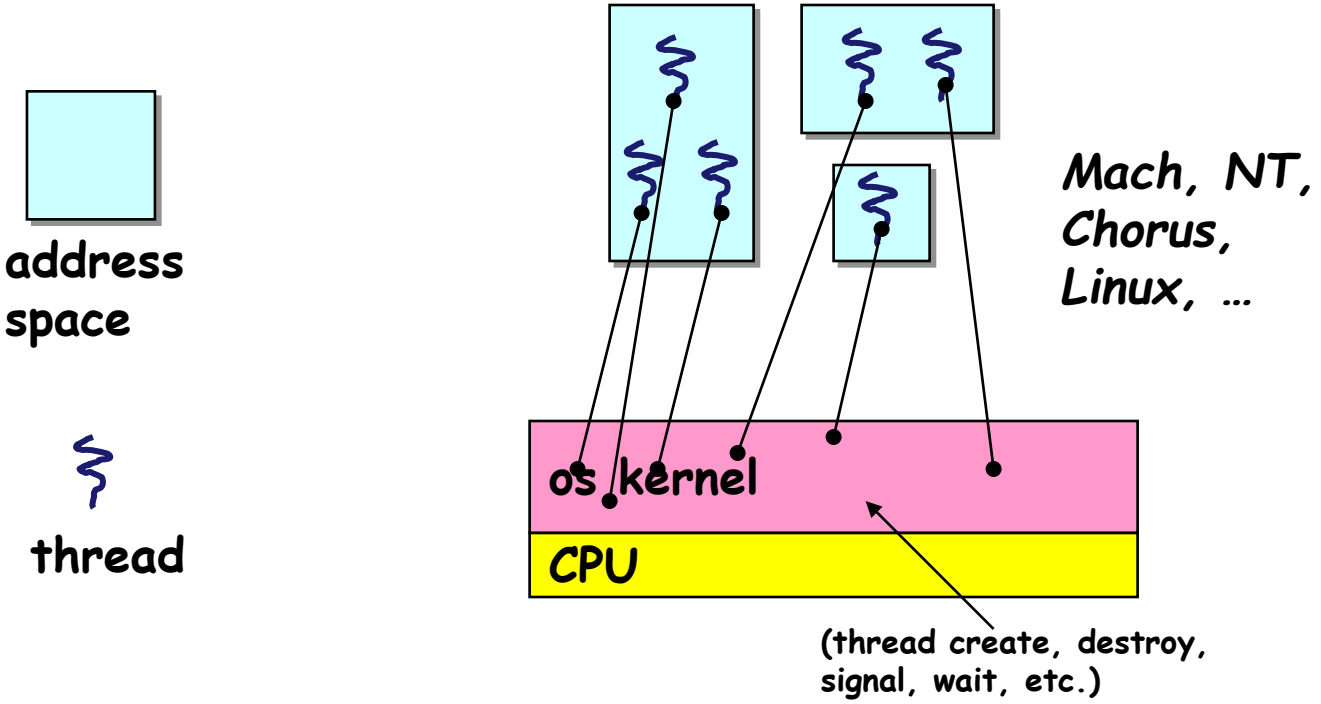
User-level threads

- 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

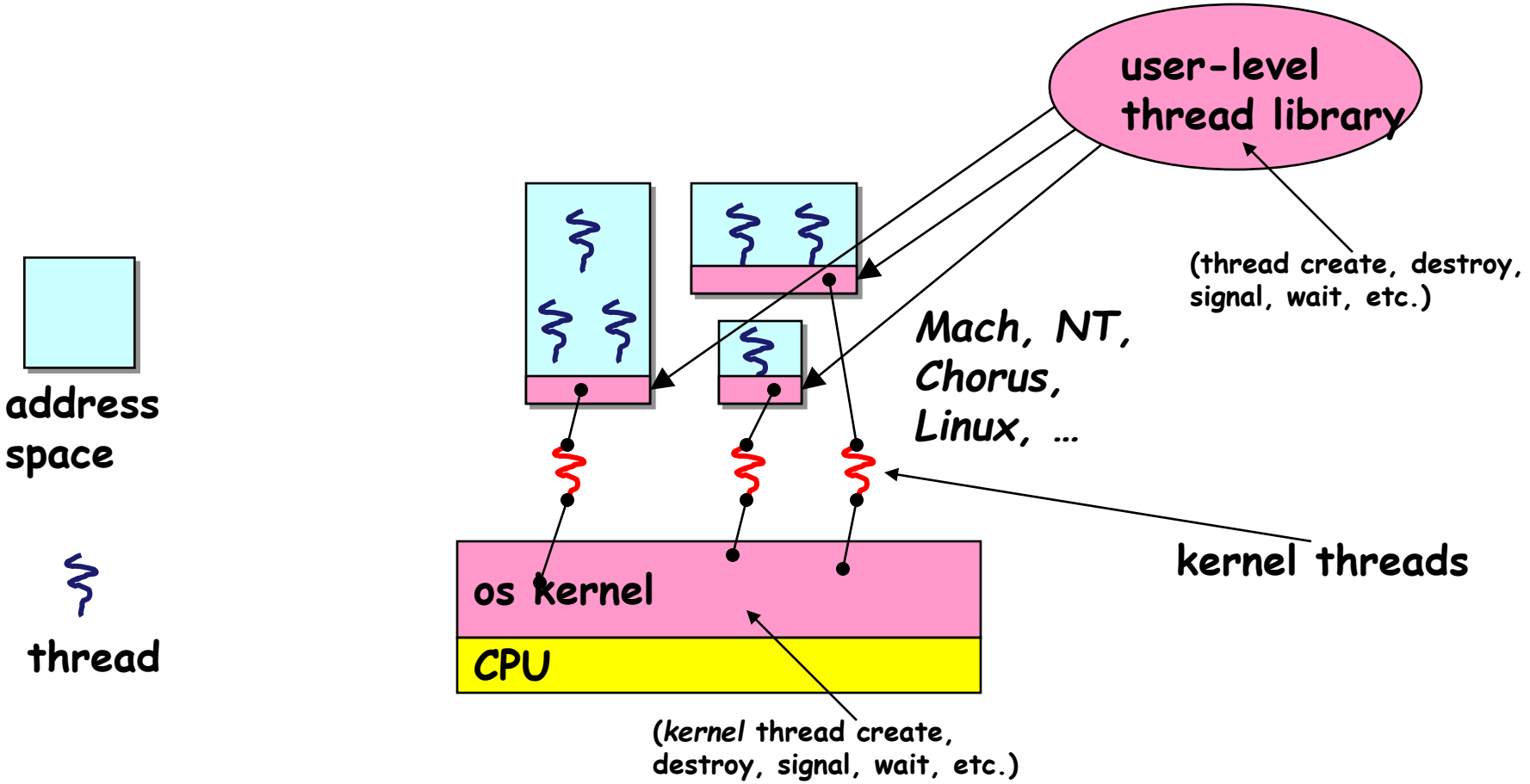
The design space



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

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

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, obliviously 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