UNIVERSITY OF CALIFORNIA
UCRIVERSIDE

Advanced Operating Systems (CS 202)

Processes (continued)

How to pause/restart processes?

- When a process is running, its dynamic state is in memory and some hardware registers
  - Hardware registers include Program counter, stack pointer, control registers, data registers, …
  - To be able to stop and restart a process, we need to completely restore this state
- When the OS stops running a process, it saves the current values of the registers (usually in PCB)
- When the OS restarts executing a process, it loads the hardware registers from the stored values in PCB
- Changing CPU hardware state from one process to another is called a context switch
  - This can happen 100s or 1000s of times a second!

How does the OS track processes?

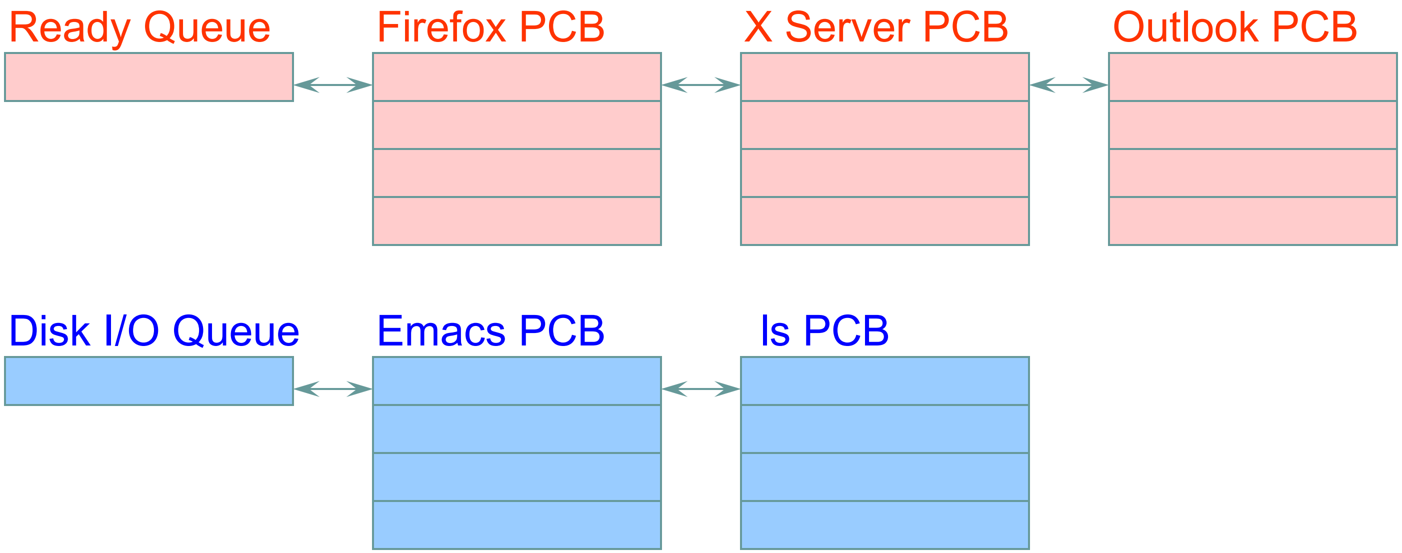
- The OS maintains a collection of queues that represent the state of all processes in the system

- Typically, the OS at least one queue for each state
  - Ready, waiting, etc.

- Each PCB is queued on a state queue according to its current state

- As a process changes state, its PCB is unlinked from one queue and linked into another

State Queues



Ready Queue

Firefox PCB

X Server PCB

Outlook PCB

Disk I/O Queue

Emacs PCB

ls PCB

Console Queue

Sleep Queue

.

.

.

There may be many wait queues, one for each type of wait (disk, console, timer, network, etc.)

Check your understanding

- True or False: a process can move from the running state to the waiting state
  - Yes, when the process asks for a blocking system call
- True or False: There is a separate kernel stack and user stack for each process
  - Yes, its dangerous to allow a process to access an OS page
- Where is process related information stored?
  - In the Process Control Block

Latency Numbers Every Programmer Should Know (2020 Version)

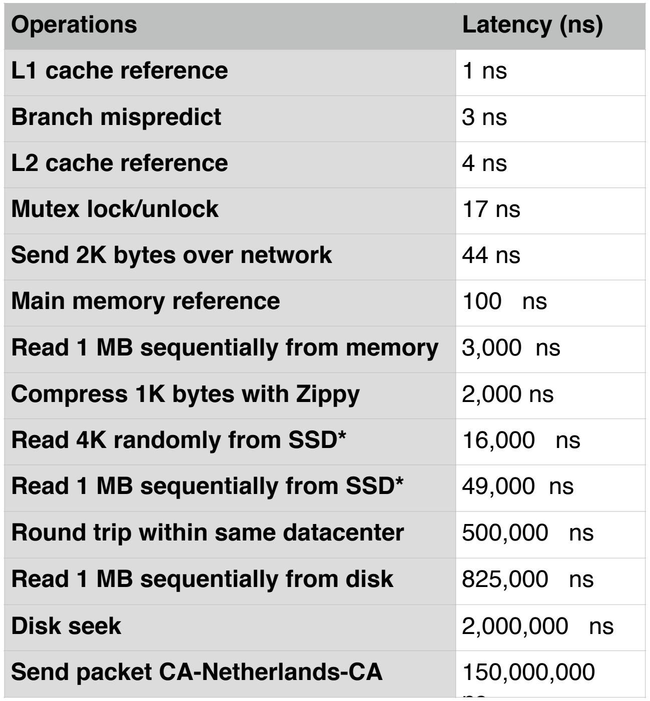
Operations	Latency (ns)	Latency (us)	Latency (ms)	
L1 cache reference	0.5 ns			~ 1 CPU cycle
Branch mispredict	3 ns			
L2 cache reference	4 ns			14x L1 cache
Mutex lock/unlock	17 ns			
Send 2K bytes over network	44 ns			
Main memory reference	100 ns			20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy	2,000 ns	2 us		
Read 1 MB sequentially from memory	3,000 ns	3 us		
Read 4K randomly from SSD*	16,000 ns	16 us		
Read 1 MB sequentially from SSD*	49,000 ns	49 us		
Round trip within same datacenter	500,000 ns	500 us		
Read 1 MB sequentially from disk	825,000 ns	825 us		
Disk seek	2,000,000 ns	2,000 us	2 ms	4x datacenter roundtrip
Send packet CA-Netherlands-CA	150,000,000 ns	150,000 us	150 ms	

https://colin-scott.github.io/personal_website/research/interactive_latency.html

X

The overhead of kernel switches/system calls

- On a 3.7GHz intel Core i5-9600K Processor, please make a guess of the overhead of switching from user-mode to kernel mode.
  A. a single digit of nanoseconds
  B. tens of nanoseconds
  C. hundreds of nanoseconds
  D. a single digit of microseconds
  E. tens of microseconds



Operations	Latency (ns)
L1 cache reference	1 ns
Branch mispredict	3 ns
L2 cache reference	4 ns
Mutex lock/unlock	17 ns
Send 2K bytes over network	44 ns
Main memory reference	100 ns
Read 1 MB sequentially from memory	3,000 ns
Compress 1K bytes with Zippy	2,000 ns
Read 4K randomly from SSD*	16,000 ns
Read 1 MB sequentially from SSD*	49,000 ns
Round trip within same datacenter	500,000 ns
Read 1 MB sequentially from disk	825,000 ns
Disk seek	2,000,000 ns
Send packet CA-Netherlands-CA	150,000,000

x

The overhead of kernel switches/system calls

- On a 3.7GHz intel Core i5-9600K Processor, please make a guess of the overhead of switching from user-mode to kernel mode.
  A. a single digit of nanoseconds
  B. tens of nanoseconds
  C. hundreds of nanoseconds
  D. a single digit of microseconds
  E. tens of microseconds

Operations	Latency (ns)
L1 cache reference	1 ns
Branch mispredict	3 ns
L2 cache reference	4 ns
Mutex lock/unlock	17 ns
Send 2K bytes over network	44 ns
Main memory reference	100 ns
Read 1 MB sequentially from memory	3,000 ns
Compress 1K bytes with Zippy	2,000 ns
Read 4K randomly from SSD*	16,000 ns
Read 1 MB sequentially from SSD*	49,000 ns
Round trip within same datacenter	500,000 ns
Read 1 MB sequentially from disk	825,000 ns
Disk seek	2,000,000 ns
Send packet CA-Netherlands-CA	150,000,000

x

Process system call API

- Process creation: how to create a new process?
- Process termination: how to terminate and clean up a process
- Coordination between processes
  - Wait, waitpid, signal, inter-process communication, synchronization
- Other
  - E.g., set quotas or priorities, examine usage, …

Process Creation

- A process is created by another process
  - Why is this the case?
  - Parent is creator, child is created (Unix: ps “PPID” field)
  - What creates the first process (Unix: init (PID 0 or 1))?
- In some systems, the parent defines (or donates) resources and privileges for its children
  - Unix: Process User ID is inherited – children of your shell execute with your privileges
- After creating a child, the parent may either wait for it to finish its task or continue in parallel (or both)

Process Creation: Unix

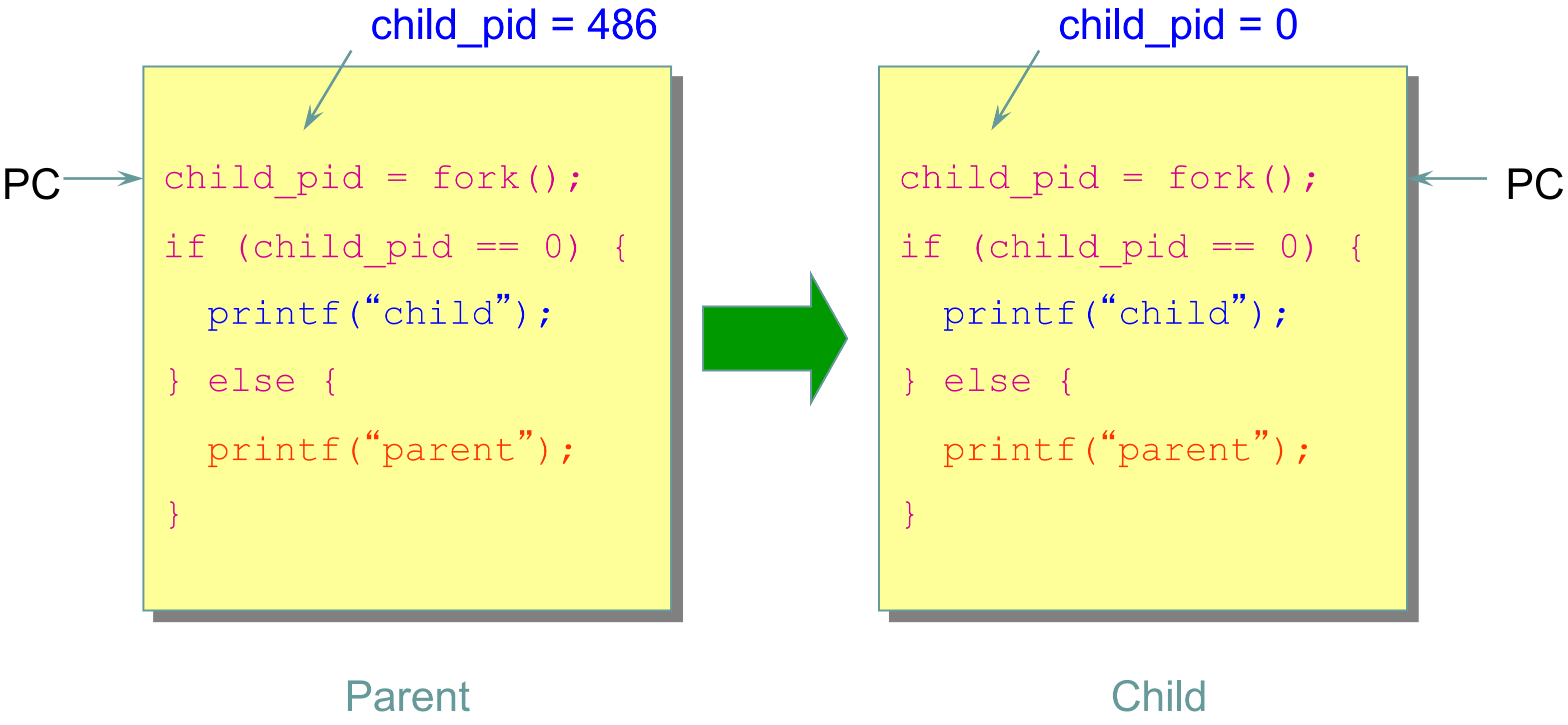
- In Unix, processes are created using fork()
  `int fork()`
- fork()
  - Creates and initializes a new PCB
  - Creates a new address space
  - Initializes the address space with a **copy** of the entire contents of the address space of the parent
  - Initializes the kernel resources to point to the resources used by parent (e.g., open files)
  - Places the PCB on the ready queue
- Fork returns twice
  - Returns the child's PID to the parent, "0" to the child

fork()

```
int main(int argc, char *argv[])
{
   char *name = argv[0];
   int child_pid = fork();
   if (child_pid == 0) {
        printf(“Child of %s is %d\n”, name, getpid());
        return 0;
   } else {
        printf(“My child is %d\n”, child_pid);
        return 0;
   }
}
```

What does this program print?

Duplicating Address Spaces



child_pid = 486

PC

```
child_pid = fork();
if (child_pid == 0) {
  printf(“child”);
} else {
  printf(“parent”);
}
```

Parent

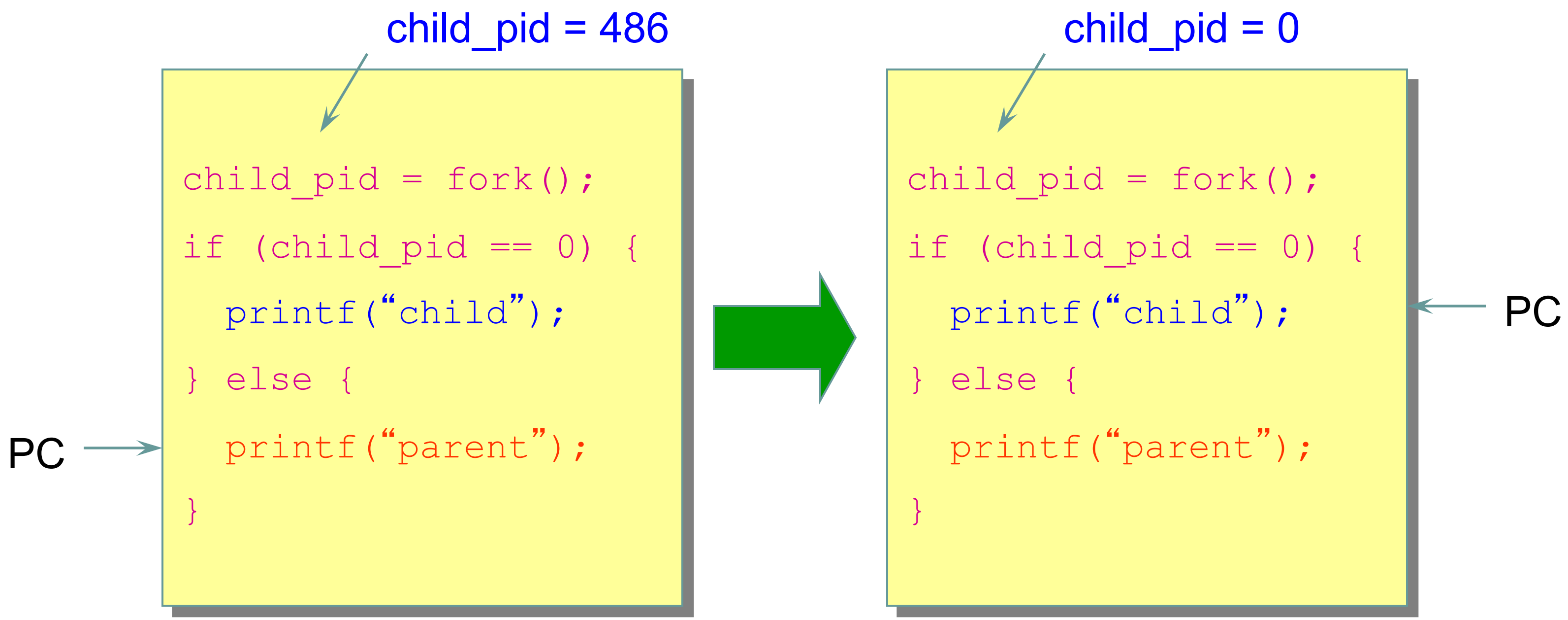
child_pid = 0

PC

```
child_pid = fork();
if (child_pid == 0) {
  printf(“child”);
} else {
  printf(“parent”);
}
```

Child

Divergence



child_pid = 486

```
child_pid = fork();
if (child_pid == 0) {
  printf(“child”);
} else {
  printf(“parent”);
}
```

PC

Parent

child_pid = 0

```
child_pid = fork();
if (child_pid == 0) {
  printf(“child”);
} else {
  printf(“parent”);
}
```

PC

Child

Example Continued

```
[well ~]$ gcc t.c
[well ~]$ ./a.out
My child is 486
Child of a.out is 486
[well ~]$ ./a.out
Child of a.out is 498
My child is 498
```

Why is the output in a different order?

Why fork()?

- Very useful when the child…
  - Is cooperating with the parent
  - Relies upon the parent's data to accomplish its task
- Example: Web server

```
while (1) {
  int sock = accept();
  if ((child_pid = fork()) == 0) {
    Handle client request
  } else {
    Close socket
  }
}
```

Process Creation (2): Unix

- Wait a second. How do we actually start a new program?

  `int exec(char *prog, char *argv[])`
- exec()
  - Stops the current process
  - Loads the program "prog" into the process' address space
  - Initializes hardware context and args for the new program
  - Places the PCB onto the ready queue
  - Note: It **does not** create a new process
- What does it mean for exec to return?
- What does it mean for exec to return with an error?

wait() a second…

- Often it is convenient to pause until a child process has finished
  - Think of executing commands in a shell
- Use wait() (WaitForSingleObject)
  - Suspends the current process until a child process ends
  - waitpid() suspends until the specified child process ends
- Wait has a return value…what is it?
- Unix: Every process must be reaped by a parent
  - What happens if a parent process exits before a child?
  - What do you think is a "zombie" process?

Unix Shells

```
while (1) {
   char *cmd = read_command();
   int child_pid = fork();
   if (child_pid == 0) {
       Manipulate STDIN/OUT/ERR file descriptors for pipes, redirection, etc.
       exec(cmd);
       panic(“exec failed”);
   } else {
       if (!(run_in_background))
             waitpid(child_pid);
   }
}
```

Some issues with processes

- Creating a new process is costly because of new address space and data structures that must be allocated and initialized
  - Recall struct proc in xv6 or Solaris

- Communicating between processes is costly because most communication goes through the OS
  - Inter Process Communication (IPC) – we will discuss later
  - Overhead of system calls and copying data

Parallel Programs

- Also recall our Web server example that forks off copies of itself to handle multiple simultaneous requests

- To execute these programs we need to
  - Create several processes that execute in parallel
  - Cause each to map to the same address space to share data
    - They are all part of the same computation
  - Have the OS schedule these processes in parallel

- This situation is inefficient (Copy on Write helps)
  - Space: Duplicate memory, PCB, page tables, etc.
  - Time: create data structures, fork and copy addr space, etc.

Rethinking Processes

- What is similar in these cooperating processes?
  - They all share the same code and data (address space)
  - They all share the same privileges
  - They all share the same resources (files, sockets, etc.)
- What don’t they share?
  - Each has its own execution state: PC, SP, and registers

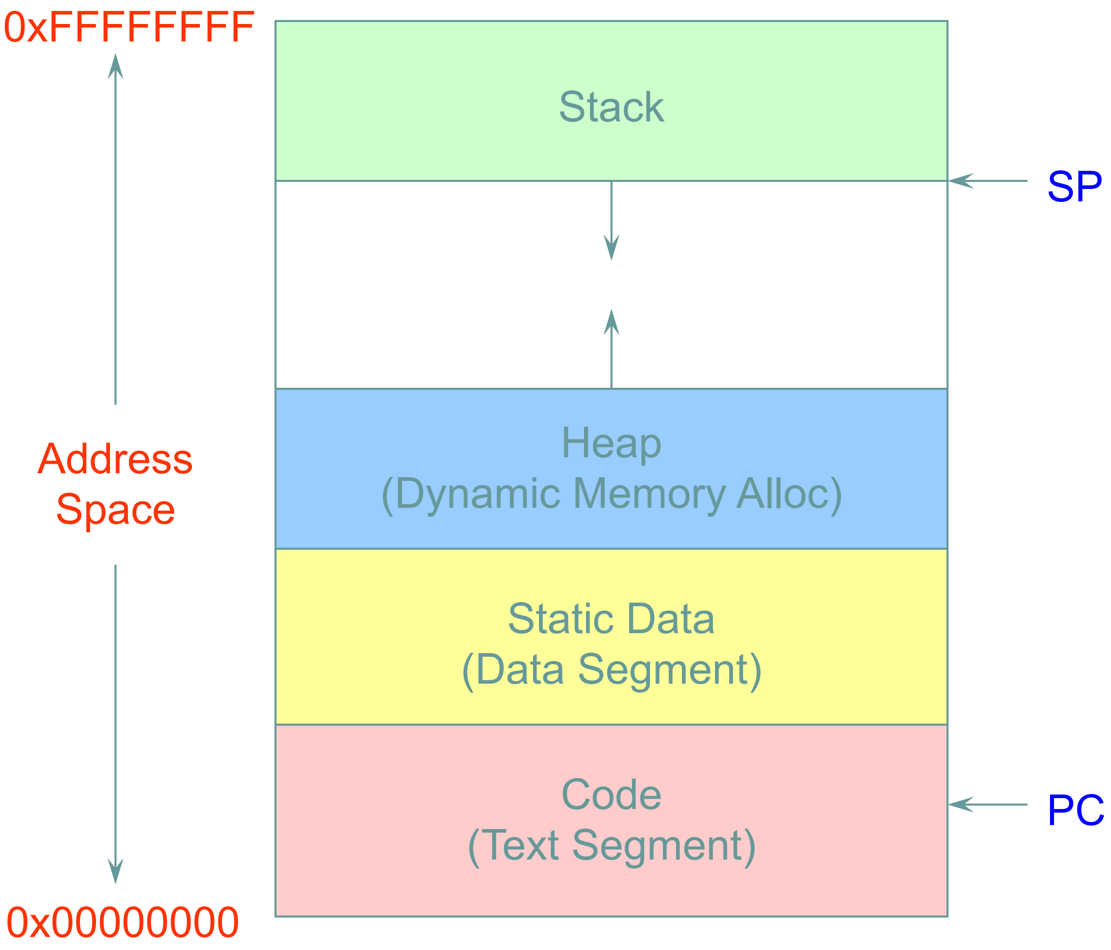
- Key idea: Separate resources from execution state
- Exec state also called thread of control, or thread

Threads

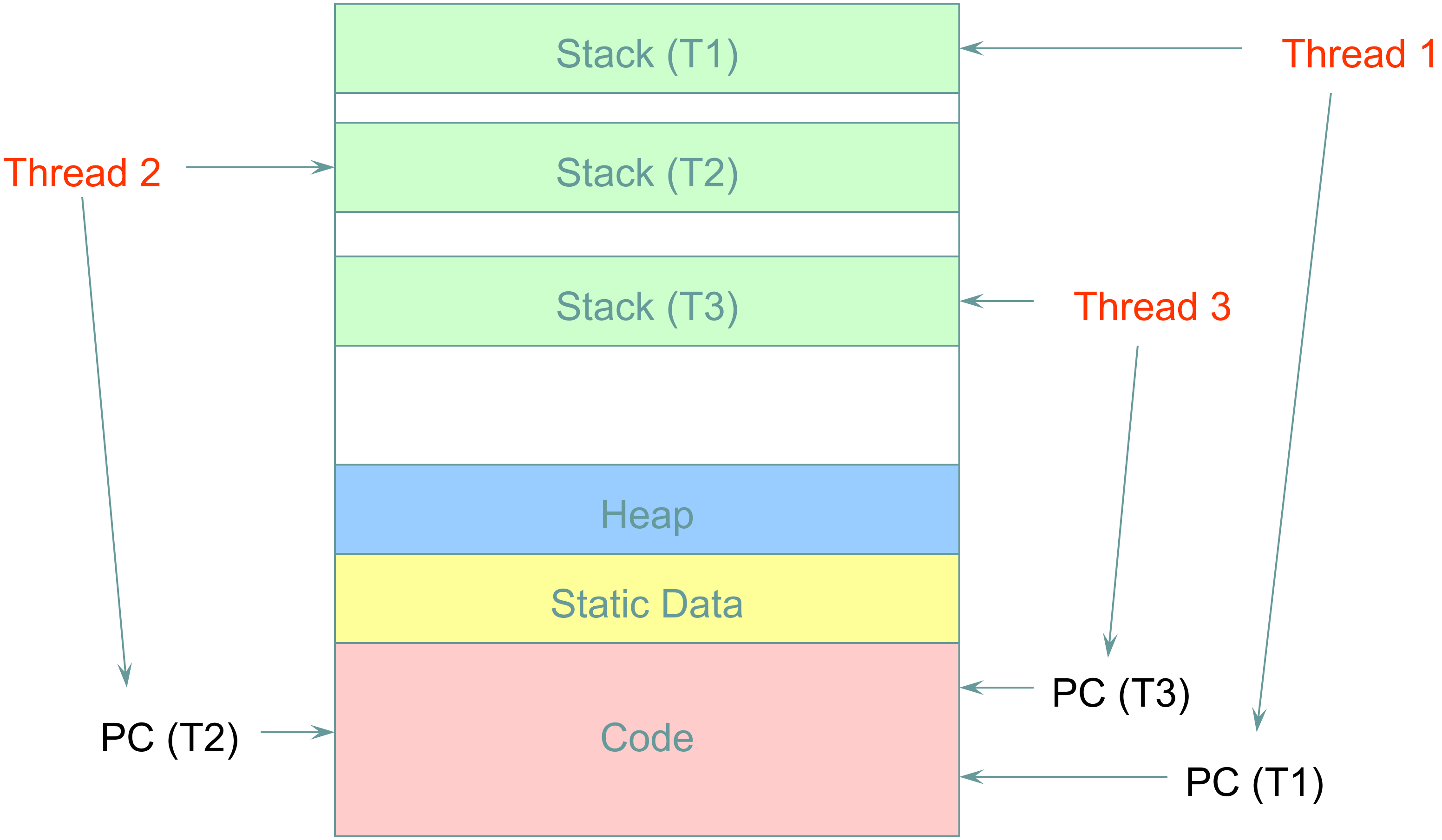
- Separate execution and resource container roles
  - The thread defines a sequential execution stream within a process (PC, SP, registers)
  - The process defines the address space, resources, and general process attributes (everything but threads)

- Threads become the unit of scheduling
  - Processes are now the containers in which threads execute
  - Processes become static, threads are the dynamic entities

Recap: Process Address Space



Threads in a Process



Which of these are needed for each thread

- ☑ Stack pointer
- ☑ Register states
- ☒ Open file descriptors
- ☑ Program Counter
- ☒ Page table (or memory management information)

Which of these are needed for each thread

- Stack pointer
- Register states
- Open file descriptors
- Program Counter
- Page table (or memory management information)

Threads: Concurrent Servers

- Using fork() to create new processes to handle requests in parallel is overkill for such a simple task
- Recall our forking Web server:

```
while (1) {
   int sock = accept();
   if ((child_pid = fork()) == 0) {
     Handle client request
     Close socket and exit
   } else {
     Close socket
   }
}
```

Threads: Concurrent Servers

- Instead, we can create a new thread for each request

```
web_server() {
   while (1) {
     int sock = accept();
     thread_fork(handle_request, sock);
   }
}

handle_request(int sock) {
     Process request
     close(sock);
}
```

Implementing threads

- Kernel Level Threads
    - All thread operations are implemented in the kernel
    - The OS schedules all of the threads in the system
    - Don’t have to separate from processes

- OS-managed threads are called kernel-level threads or lightweight processes
    - Windows: threads
    - Solaris: lightweight processes (LWP)
    - POSIX Threads (pthreads): PTHREAD_SCOPE_SYSTEM

Kernel Thread (KLT) Limitations

- KLTs make concurrency cheaper than processes
  - Much less state to allocate and initialize

- However, there are a couple of issues
  - Issue 1: KLT overhead still high
    - Thread operations still require system calls
    - Ideally, want thread operations to be as fast as a procedure call
  - Issue 2: KLTs are general; unaware of application needs

- Alternative: User-level threads (ULT)

Alternative: User-Level Threads

- Implement threads using user-level library
- ULTs are small and fast
  - A thread is simply represented by a PC, registers, stack, and small thread control block (TCB)
  - Creating a new thread, switching between threads, and synchronizing threads are done via procedure call
    - No kernel involvement
  - User-level thread operations 100x faster than kernel threads
  - pthreads: PTHREAD_SCOPE_PROCESS

Summary KLT vs. ULT

- Kernel-level threads
  - Integrated with OS (informed scheduling)
  - Slow to create, manipulate, synchronize
- User-level threads
  - Fast to create, manipulate, synchronize
  - Not integrated with OS (uninformed scheduling)
- Understanding the differences between kernel and user-level threads is important
  - For programming (correctness, performance)
  - For test-taking ☺

Sample Thread Interface

- thread_fork(procedure_t)
  - Create a new thread of control
  - Also thread_create(), thread_setstate()
- thread_stop()
  - Stop the calling thread; also thread_block
- thread_start(thread_t)
  - Start the given thread
- thread_yield()
  - Voluntarily give up the processor
- thread_exit()
  - Terminate the calling thread; also thread_destroy

Looking ahead

- OS Model
  - We have assumed monolithic kernel
    - Are there disadvantages to that?
    - What alternatives are there?
- Scheduling
  - How do we decide which thread to run next?
- Concurrency and synchronization
  - We have to manage concurrency for correctness
    - But also for performance/scalability
      - Both OS and general multi-threaded programming problem
    - Multi-core->many-core