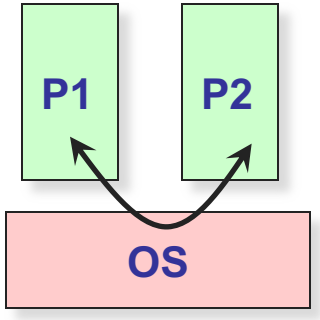
CSE 153
Design of Operating Systems

Winter 2023

Lecture 6: Threads

Processes



- Recall that …
  - A process includes:
    - » An address space (defining all the code and data pages)
    - » OS resources (e.g., open files) and accounting info
    - » Execution state (PC, SP, regs, etc.)
    - » PCB to keep track of everything
  - Processes are completely isolated from each other
- But…

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 very inefficient (CoW helps)
  - Space: 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

Recap: Process Components

- A process is named using its process ID (PID)
- A process contains all of the state for a program in execution

Per-Process State

- An address space
- The code for the executing program
- The data for the executing program
- A set of operating system resources
  - » Open files, network connections, etc.

Per-Thread State

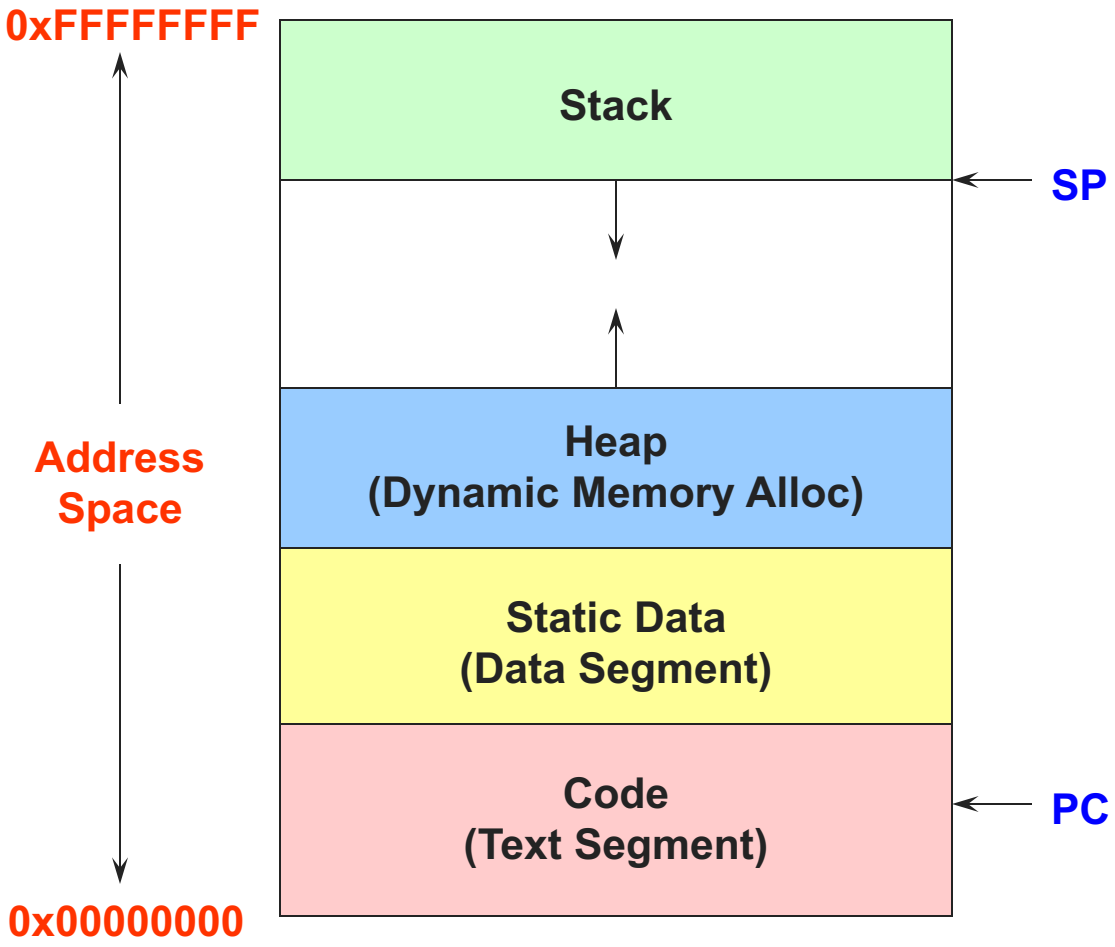
- An execution stack encapsulating the state of procedure calls
- The program counter (PC) indicating the next instruction
- A set of general-purpose registers with current values
- Current execution state (Ready/Running/Waiting)

Threads

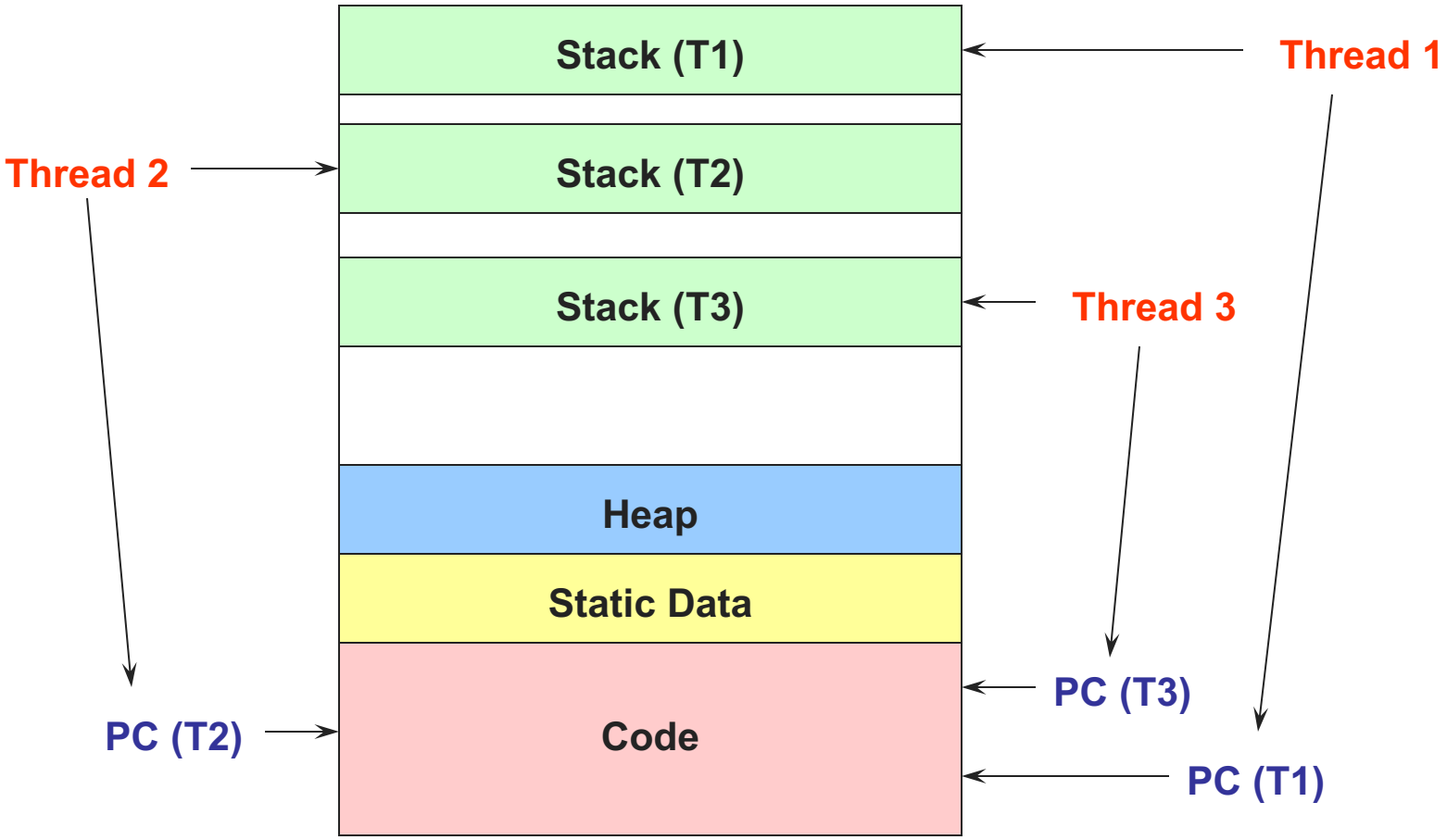
- Separate execution and resource container roles
  - The thread defines a sequential execution stream within a process (PC, SP, registers)
  - The process defines the address space, resources, and general process attributes (everything but threads)

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

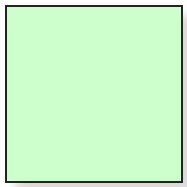
Recap: Process Address Space



Threads in a Process

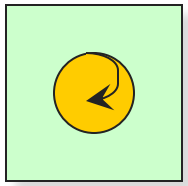


Thread Design Space

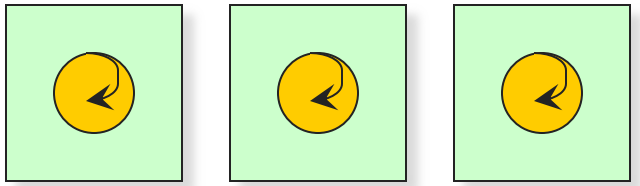


Address Space

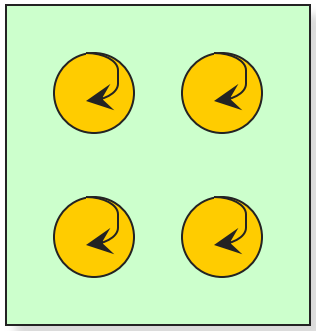
Thread



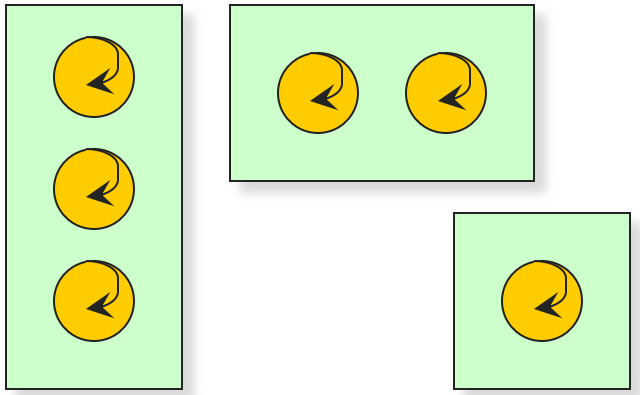
One Thread/Process
One Address Space
(MSDOS)



One Thread/Process
Many Address Spaces
(Early Unix)



Many Threads/Process
One Address Space
(Pilot, Java)



Many Threads/Process
Many Address Spaces
(Mac OS, Unix, Windows)

Process/Thread Separation

- Separating threads and processes makes it easier to support multithreaded applications
  - Concurrency does not require creating new processes

- Concurrency (multithreading) can be very useful
  - Improving program structure
  - Handling concurrent events (e.g., Web requests)
  - Writing parallel programs

- So multithreading is even useful on a uniprocessor

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

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

Thread Scheduling

- The thread scheduler determines when a thread runs
- It uses queues to keep track of what threads are doing
  - Just like the OS and processes
  - But it is implemented at user-level in a library
- Run queue: Threads currently running (usually one)
- Ready queue: Threads ready to run
- Are there wait queues?
  - How would you implement thread_sleep(time)?

Non-Preemptive Scheduling

- Threads voluntarily give up the CPU with thread_yield

Ping Thread

```
while (1) {
    printf("ping\n");
    thread_yield();
}
```

Pong Thread

```
while (1) {
    printf("pong\n");
    thread_yield();
}
```

- What is the output of running these two threads?

thread_yield()

- The semantics of thread_yield are that it gives up the CPU to another thread
  - In other words, it context switches to another thread
- So what does it mean for thread_yield to return?
- Execution trace of ping/pong
  - printf("ping\n");
  - thread_yield();
  - printf("pong\n");
  - thread_yield();
  - …

Threads Summary

- Processes are too heavyweight for multiprocessing
  - Time and space overhead
- Solution is to separate threads from processes
  - Kernel-level threads much better, but still significant overhead
  - User-level threads even better, but not well integrated with OS
- Scheduling of threads can be either preemptive or non-preemptive

- Now, how do we get our threads to correctly cooperate with each other?
  - Synchronization…