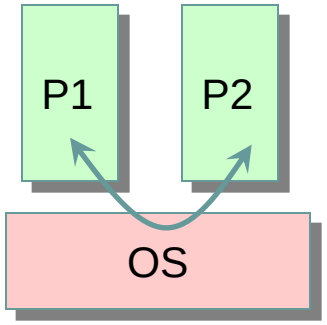
UNIVERSITY OF CALIFORNIA
UC RIVERSIDE

CS 202
Advanced Operating Systems

Thread

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

- 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

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

- 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
  - 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 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.

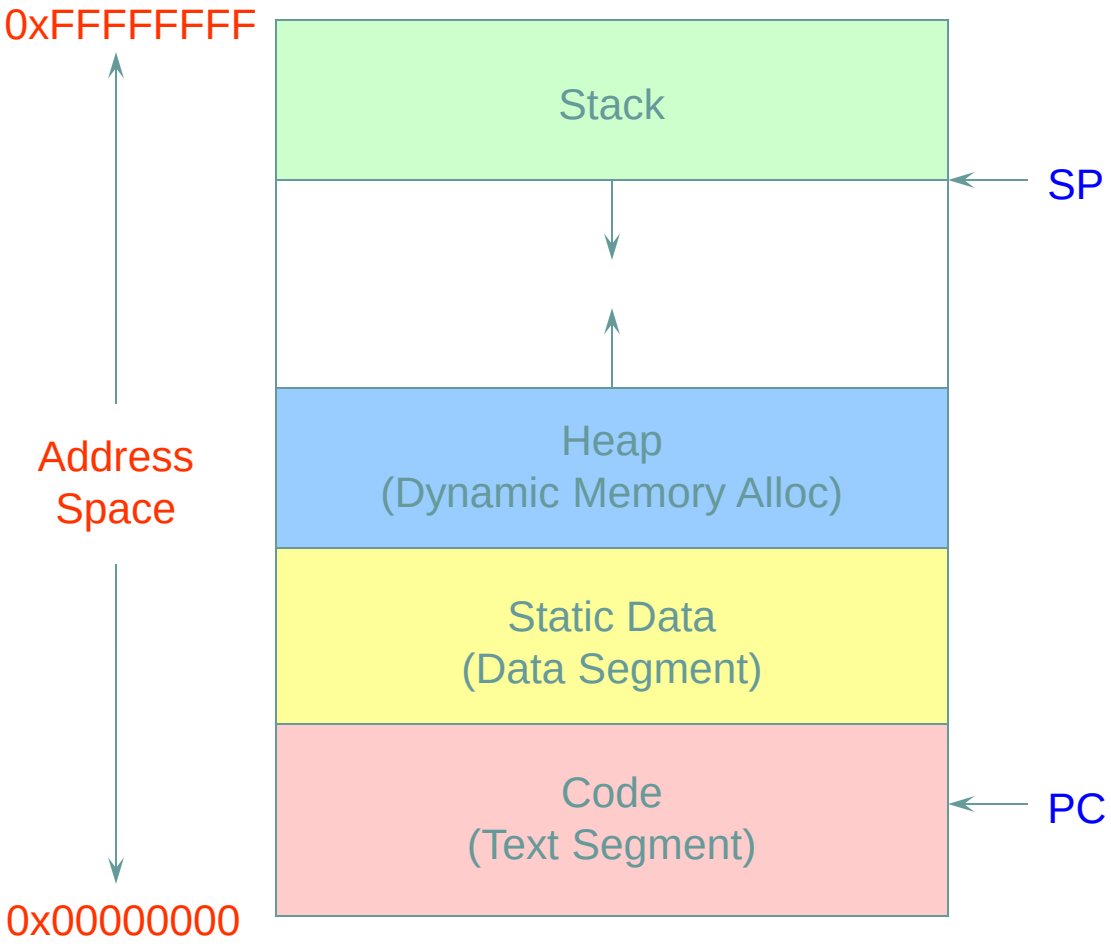
Per-Thread State:
- An execution stack encapsulating the state of procedure calls
- The program counter (PC) indicating the next instruction
- A set of general-purpose registers with current values
- Current execution state (Ready/Running/Waiting)

Threads

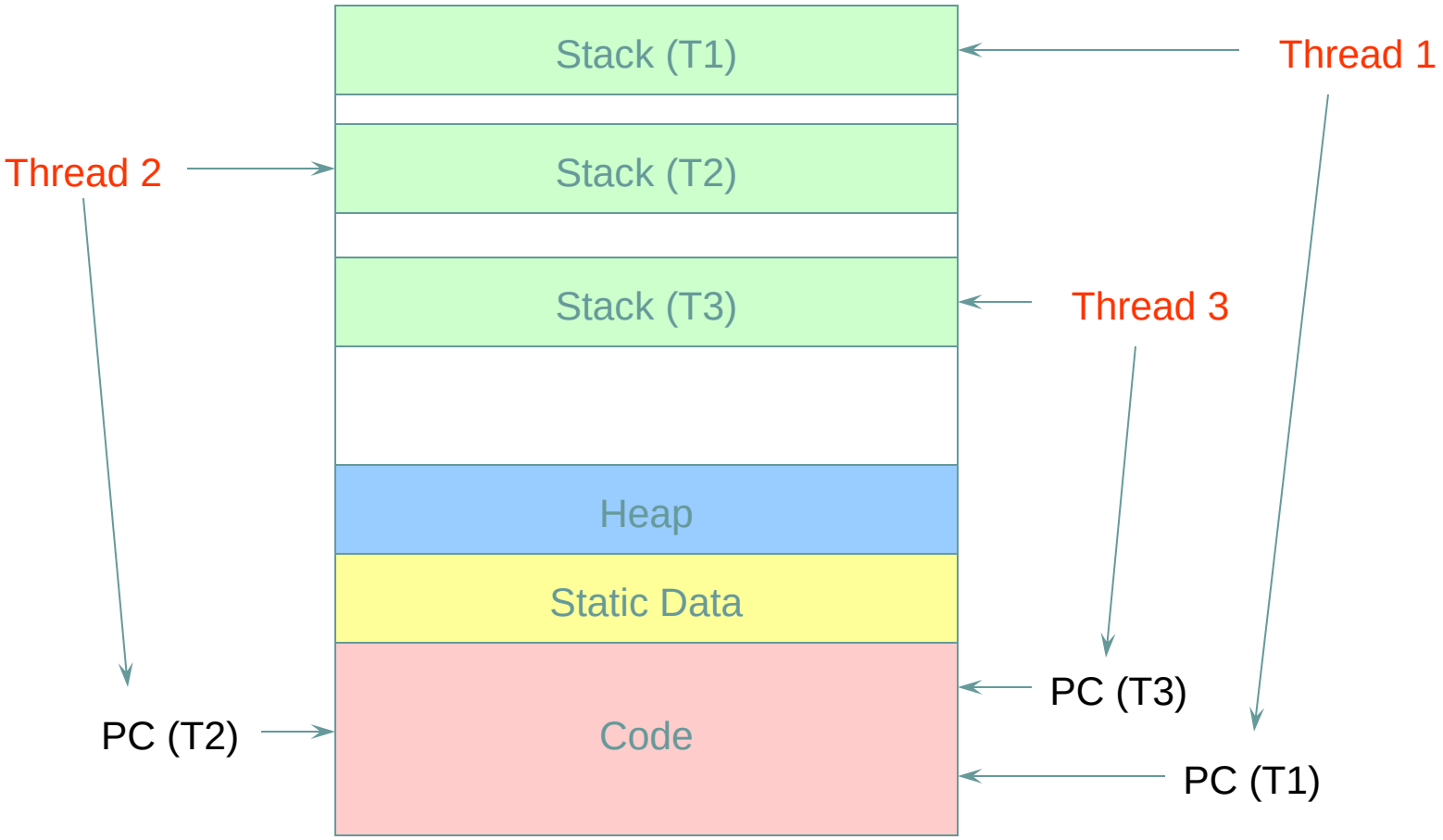
- Separate execution and resource container roles
  - The thread defines a sequential execution stream within a process (PC, SP, registers)
  - The process defines the address space, resources, and general process attributes (everything but threads)

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

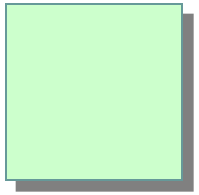
Recap: Process Address Space



Threads in a Process

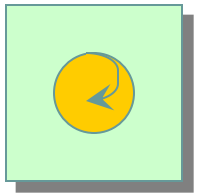


Thread Design Space

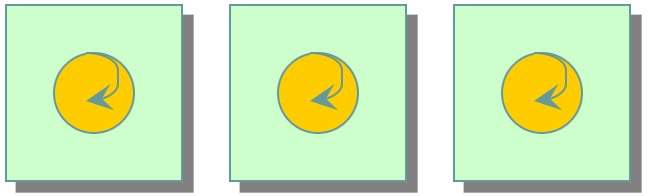


Address Space

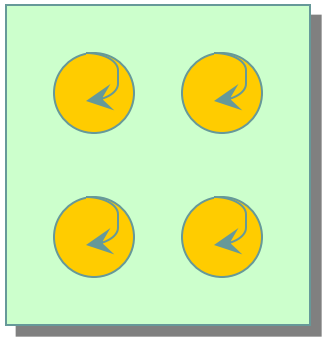
Thread



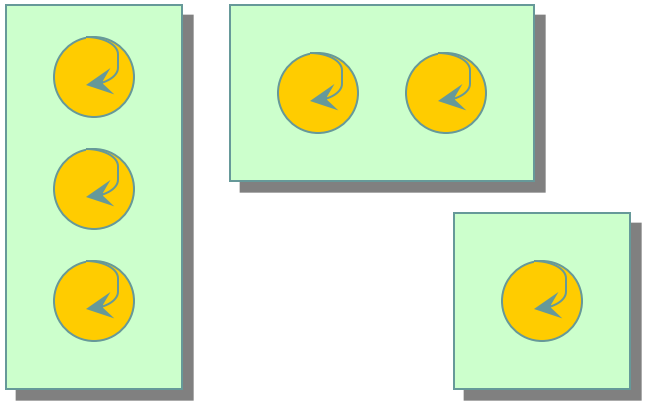
One Thread/Process
One Address Space (MSDOS)



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);
}
```

Thread Implementations

- User-level thread
- Kernel-level thread

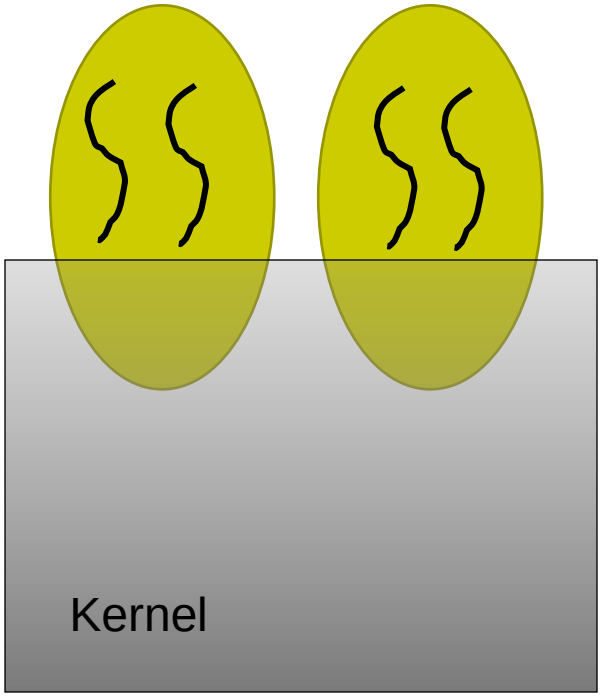
User-Level Threads

- Managed entirely by a user-level thread library
  - Creation, scheduling, etc.
  - No kernel intervention

- ULTs are small and fast
  - A thread is 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 (not available in Linux)

Context switching user-level threads

- The thread library switches threads within process
- OS kernel manages context switch across processes



User-level Thread Limitations

- What happens if a thread invokes a syscall?
  - A blocking syscall blocks the whole process!
- User-level threads are invisible to the OS
  - They are not well integrated with the OS
- As a result, the OS can make poor decisions
  - Scheduling a process with idle threads
  - Blocking a process, in which the current thread initiates an I/O, while many other threads are ready
  - Unscheduling a process with a thread holing a lock

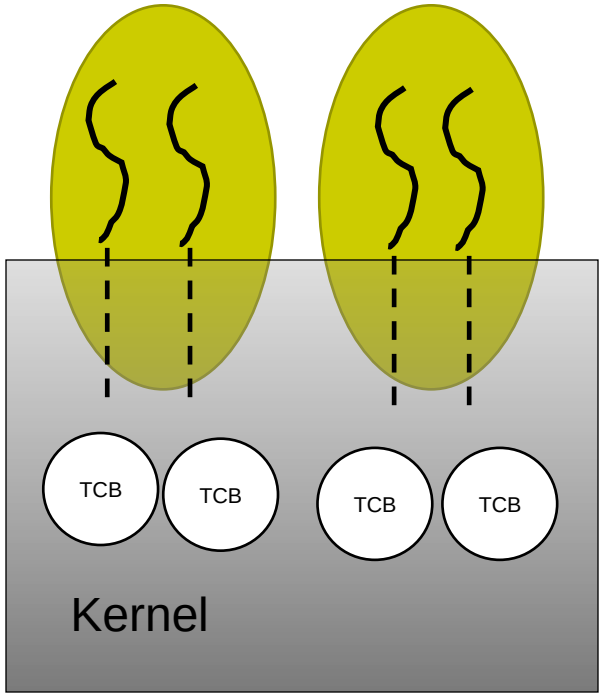
Kernel-Level Thread Implementation

- OS kernel manages threads and processes
  - All thread operations are implemented in the kernel
  - The OS schedules all the threads in the system
  - Process is no longer a unit for scheduling!

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

Kernel-Level Thread Implementation

- Each user thread maps to a kernel thread
- Slow to create and manipulate
  - Must go through syscalls
- Integrated with OS well
  - A blocking syscall will not block the whole 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

- pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);
  - Create a new thread
- pthread_exit(void *status);
  - Terminate the calling thread
- pthread_join(pthread_t thread, void **value_ptr);
  - Wait for a thread to complete
- pthread_yield()
  - Voluntarily give up the processor

An example

```
1  typedef struct { int a; int b; } myarg_t;
2  typedef struct { int x; int y; } myret_t;
3
4  void *mythread(void *arg) {
5      myret_t *rvals = Malloc(sizeof(myret_t));
6      rvals->x = 1;
7      rvals->y = 2;
8      return (void *) rvals;
9  }
10
11 int main(int argc, char *argv[]) {
12     pthread_t p;
13     myret_t *rvals;
14     myarg_t args = { 10, 20 };
15     Pthread_create(&p, NULL, mythread, &args);
16     Pthread_join(p, (void **) &rvals);
17     printf("returned %d %d\n", rvals->x, rvals->y);
18     free(rvals);
19     return 0;
20 }
```

Non-Preemptive Scheduling

- Threads voluntarily give up the CPU with pthread_yield

Ping Thread

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

Pong Thread

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

- What is the output of running these two threads?

pthread_yield()

- The semantics of pthread_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 pthread_yield to return?

- Execution trace of ping/pong
  - printf("ping\n");
  - thread_yield();
  - printf("pong\n");
  - thread_yield();
  - …

An illustrative Implementation of pthread_yield()


```
pthread_yield() {
   thread_t old_thread = current_thread;
   current_thread = get_next_thread();
   append_to_queue(ready_queue, old_thread);
   context_switch(old_thread, current_thread);
   return;
}
```

As old thread

As new thread

- The magic step is invoking context_switch()
- Why do we need to call append_to_queue()?

Thread Context Switch

- The context switch routine does all the magic
  - Saves context of the currently running thread (old_thread)
    - Push all machine state onto its stack (*not* its TCB)
  - Restores context of the next thread
    - Pop all machine state from the next thread's stack
  - The next thread becomes the current thread
  - Return to caller as new thread

- This is all done in assembly language
  - It works at the level of the procedure calling convention, so it cannot be implemented using procedure calls

Preemptive Scheduling

- Non-preemptive threads must voluntarily give up CPU
  - A long-running thread will take over the machine
  - Only voluntary calls to pthread_yield(), pthread_join(), or thread_exit() causes a context switch

- Preemptive scheduling causes an involuntary context switch
  - Need to regain control of processor asynchronously
  - Use timer interrupt (How do you do this?)
  - Timer interrupt handler forces current thread to "call" 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…