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

<table>
<thead>
<tr>
<th>Per-Process State</th>
</tr>
</thead>
<tbody>
<tr>
<td>- An address space</td>
</tr>
<tr>
<td>- The code for the executing program</td>
</tr>
<tr>
<td>- The data for the executing program</td>
</tr>
<tr>
<td>- A set of operating system resources</td>
</tr>
<tr>
<td>» Open files, network connections, etc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Per-Thread State</th>
</tr>
</thead>
<tbody>
<tr>
<td>- An execution stack encapsulating the state of procedure calls</td>
</tr>
<tr>
<td>- The program counter (PC) indicating the next instruction</td>
</tr>
<tr>
<td>- A set of general-purpose registers with current values</td>
</tr>
<tr>
<td>- Current execution state (Ready/Running/Waiting)</td>
</tr>
</tbody>
</table>
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

- **Stack**
  - Stack Pointer (SP)
- **Heap** (Dynamic Memory Alloc)
- **Static Data** (Data Segment)
- **Code** (Text Segment)

Address Space:
- **0x00000000**
- **0xFFFFFFFF**
Threads in a Process

The diagram illustrates the structure of threads within a process. It shows the following components:

- **Stack (T1)**
- **Stack (T2)**
- **Stack (T3)**
- **Heap**
- **Static Data**
- **Code**

Each thread has its own stack and program counter (PC). The diagram also shows the relationship between the threads:

- **Thread 1**
- **Thread 2**
- **Thread 3**

The program counter (PC) for each thread is also indicated:

- **PC (T1)**
- **PC (T2)**
- **PC (T3)**

This diagram helps to understand how threads are managed and how they share or have their own resources within a process.
Thread Design Space

- **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:

```c
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

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

```c
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
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
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?
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();`
- ...
Implementing thread_yield()

thread_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;
}

● 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 of 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 have to voluntarily give up CPU
  - A long-running thread will take over the machine
  - Only voluntary calls to `thread_yield()`, `thread_stop()`, 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…