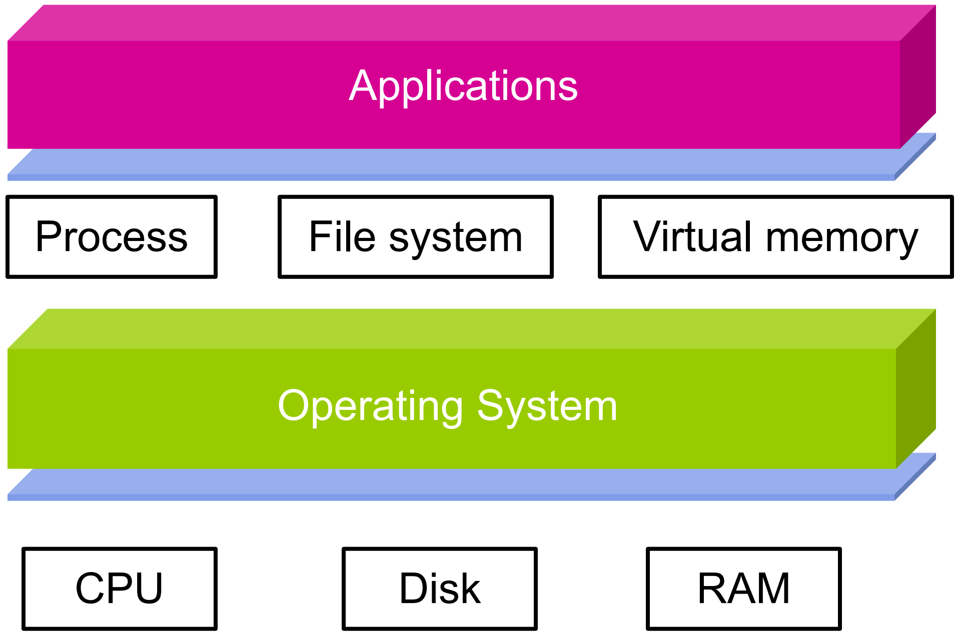
Processes and Threads

CS 202: Advanced Operating Systems

UNIVERSITY OF CALIFORNIA, RIVERSIDE

OS Abstractions



Today, we start discussing the first abstraction that enables us to virtualize (i.e., share) the CPU – processes!

Program

- Program is a file describing a computation
  - Executable code (machine instructions)
  - Data (info manipulated by the instructions

Process

- The process is the OS abstraction for execution
  - It is the unit of execution
  - It is the unit of scheduling
- A process is a program in execution
  - Programs are static entities with the potential for execution
  - Process is the dynamic/active entity of a program
    - Includes dynamic state
    - As the representative of the running program, it is the “owner” of other resources (memory, files, sockets, …)

Process = Program ???

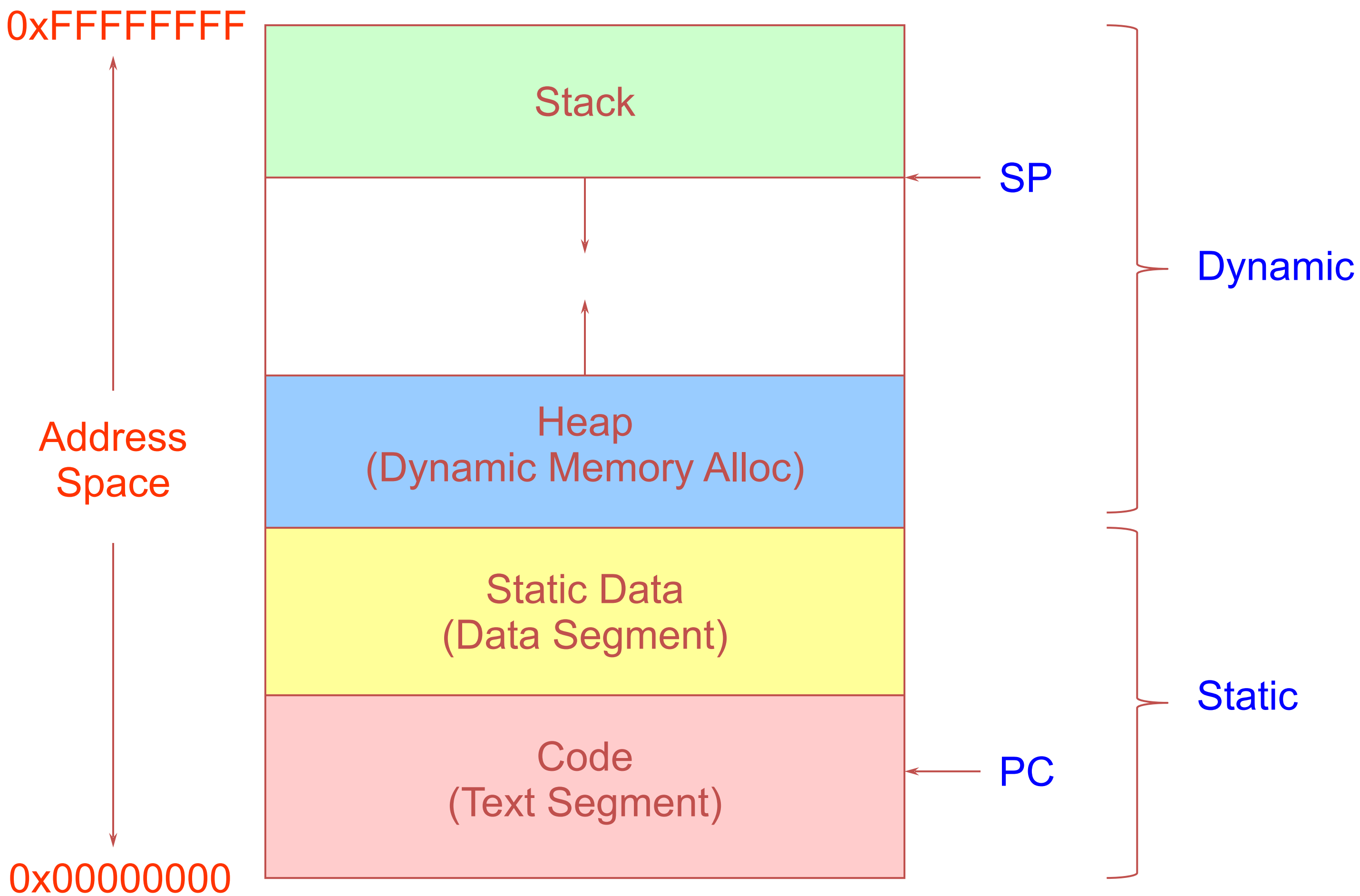
- A program is passive
  - Code + data
- A process is alive:
  - Code + data + stack +registers + PC…
- Same program can be run simultaneously
  - 2 processes
- Why processes?

Process Components

- A process contains all the state for a program in execution
  - An address space containing
    - Static memory:
      - The code and input data for the executing program
    - Dynamic memory:
      - The memory allocated by the executing program
      - An execution stack encapsulating the state of procedure calls
  - Control registers such as the program counter (PC)
  - A set of general-purpose registers with current values
  - A set of operating system resources
    - Open files, network connections, etc.

- A process is named using its process ID (PID)

Address Space (memory abstraction)



Process Execution State

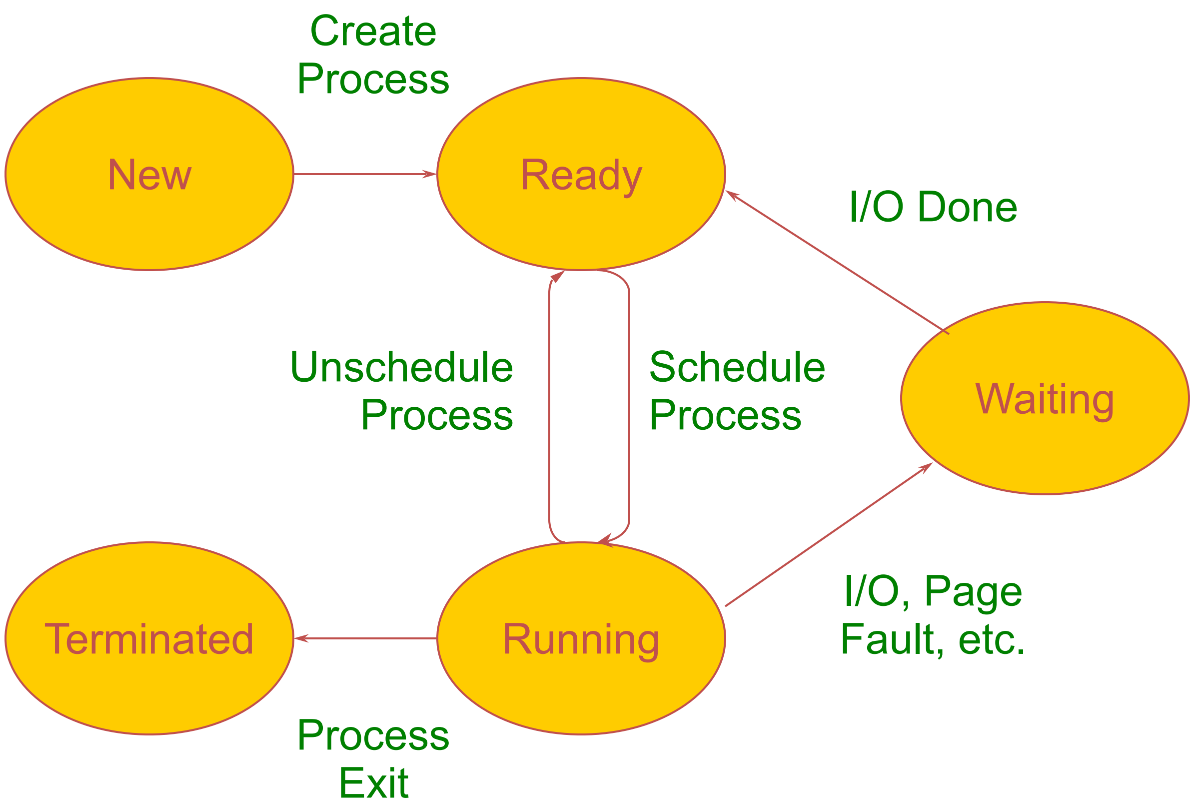
- A process is born, executes for a while, and then dies
- The process execution state that indicates what it is currently doing
  - Running: Executing instructions on the CPU
    - It is the process that has control of the CPU
    - How many processes can be in the running state simultaneously?
  - Ready: Waiting to be assigned to the CPU
    - Ready to execute, but another process is executing on the CPU
  - Waiting: Waiting for an event, e.g., I/O completion
    - It cannot make progress until event is signaled (disk completes)

Execution state (cont'd)

- As a process executes, it moves from state to state
  - Unix: "ps -x": STAT column indicates execution state
  - What state do you think a process is in most of the time?
  - How many processes can a system support?

```
PROCESS STATE CODES
       Here are the different values that the s, stat and state output specifiers (header "S
       D    uninterruptible sleep (usually IO)
       R    running or runnable (on run queue)
       S    interruptible sleep (waiting for an event to complete)
       T    stopped, either by a job control signal or because it is being traced.
       W    paging (not valid since the 2.6.xx kernel)
       X    dead (should never be seen)
       Z    defunct ("zombie") process, terminated but not reaped by its parent.

       For BSD formats and when the stat keyword is used, additional characters may be displ
       <    high-priority (not nice to other users)
       N    low-priority (nice to other users)
       L    has pages locked into memory (for real-time and custom IO)
       s    is a session leader
       l    is multi-threaded (using CLONE_THREAD, like NPTL pthreads do)
       +    is in the foreground process group.
```

Execution State Graph



How does the OS support this model?

We will discuss three issues:

1. How does the OS represent a process in the kernel?
   - The OS data structure representing each process is called the Process Control Block (PCB)

2. How do we pause and restart processes?
   - We must be able to *save* and *restore* the full machine state

3. How do we keep track of all the processes in the system?
   - A lot of queues!

PCB Data Structure

- PCB is also where OS keeps all of a process' hardware execution state when the process is not running
    - Process ID (PID)
    - Execution state
    - Hardware state: PC, SP, regs
    - Location in memory
    - Scheduling info
    - User info
    - Pointers for state queues
    - Etc.
- This state is everything that is needed to restore the hardware to the same configuration it was in when the process was switched out of the hardware

xv6/proc.h: struct proc

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  char *mem;                    // Start of process memory (kernel address)
  uint sz;                      // Size of process memory (bytes)
  char *kstack;                 // Bottom of kernel stack for this process
  enum procstate state;         // Process state
  volatile int pid;             // Process ID
  struct proc *parent;          // Parent process
  struct trapframe *tf;         // Trap frame for current syscall
  struct context *context;      // Switch here to run process
  void *chan;                   // If non-zero, sleeping on chan
  int killed;                   // If non-zero, have been killed
  struct file *ofile[NOFILE];   // Open files
  struct inode *cwd;            // Current directory
  char name[16];                // Process name (debugging)
};
```

How to pause/resume a process?

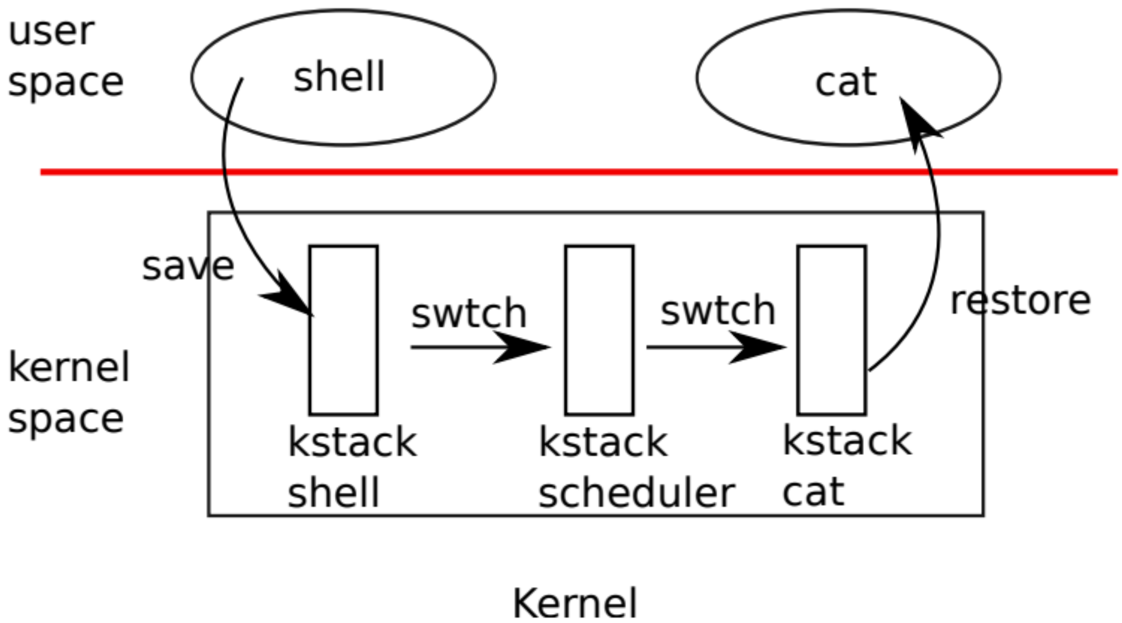
- When a process is running, its dynamic state is in memory and some hardware registers
  - Hardware registers include program counter, stack pointer, control registers, data registers, …
  - To stop and restart a process, we need to completely restore this state

- When the OS stops running a process, it saves the current values of the registers (usually in PCB)
- When the OS resumes 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!

xv6: context switching

- Switching from one user process to another (shell to cat)
  - First moves to the kernel space, then calls swtch() for context switching



- In xv6, each CPU has a scheduler thread (scheduler() in proc.c)

xv6: PCB and context

```c
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  char *mem;
  uint sz;
  char *kstack;
  enum procstate state;
  volatile int pid;
  struct proc *parent;
  struct trapframe *tf;
  struct context *context;
  void *chan;
  int killed;
  struct file *ofile[NOFILE];
  struct inode *cwd;
  char name[16];
};
```

```c
// Saved registers for kernel context switches.
struct context {
  uint64 ra;
  uint64 sp;

  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};
```

xv6: swtch()

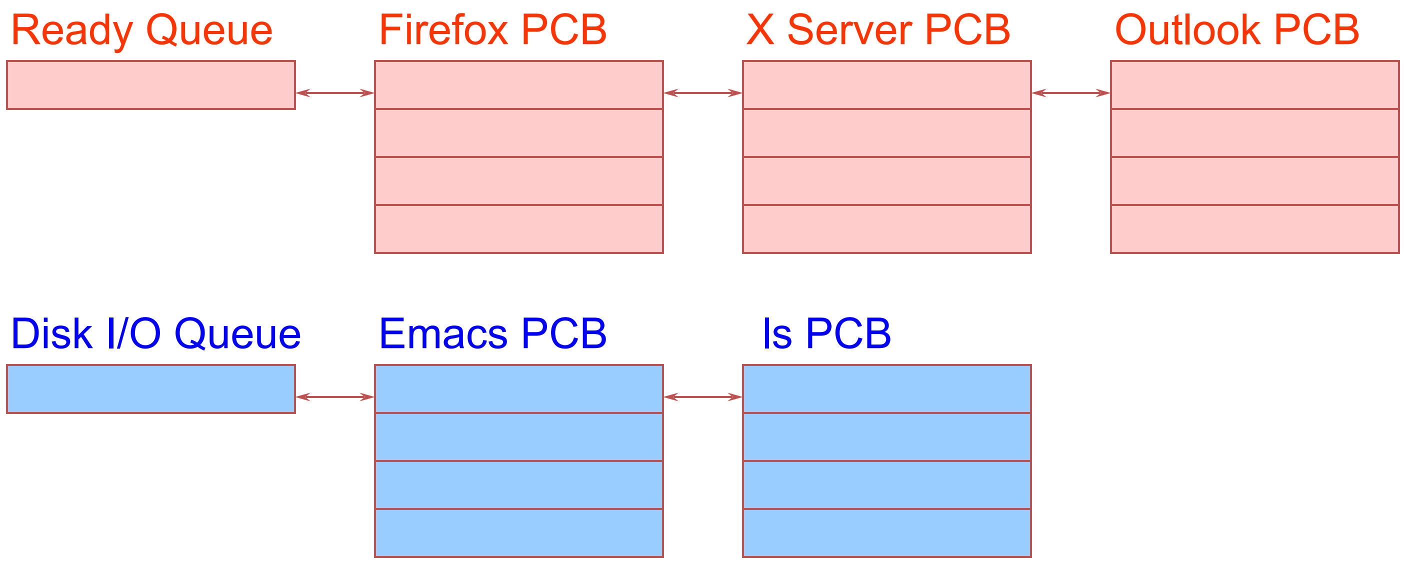
- Kernel/swtch.S

```
swtch:
        sd ra, 0(a0)
        sd sp, 8(a0)
        sd s0, 16(a0)
        sd s1, 24(a0)
        sd s2, 32(a0)
        sd s3, 40(a0)
        sd s4, 48(a0)
        sd s5, 56(a0)
        sd s6, 64(a0)
        sd s7, 72(a0)
        sd s8, 80(a0)
        sd s9, 88(a0)
        sd s10, 96(a0)
        sd s11, 104(a0)

        ld ra, 0(a1)
        ld sp, 8(a1)
        ld s0, 16(a1)
        ld s1, 24(a1)
        ld s2, 32(a1)
        ld s3, 40(a1)
        ld s4, 48(a1)
        ld s5, 56(a1)
        ld s6, 64(a1)
        ld s7, 72(a1)
        ld s8, 80(a1)
        ld s9, 88(a1)
        ld s10, 96(a1)
        ld s11, 104(a1)

        ret
```

How does the OS track process states?

- The OS maintains a collection of queues that represent the state of all processes in the system
- Typically, the OS maintains at least one queue for each state
  - Ready, waiting, etc.
- Each process (namely 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



Console Queue

Sleep Queue

.

.

.

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

Process System Call APIs

- 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
  - Parent is creator, child is created (Unix: ps “PPID” field)
  - What creates the first process (Unix: init (PID 1))?
- After creating a child, the parent may either wait for it to finish its task or continue in parallel (or both)

Process Creation: Windows

- The system call on Windows for creating a process is called, surprisingly enough, CreateProcess:

  `BOOL CreateProcess(char *prog, char *args)` (simplified)

- CreateProcess
  - Creates and initializes a new PCB
  - Creates and initializes a new address space
  - Loads the program specified by “prog” into the address space
  - Copies “args” into memory allocated in address space
  - Initializes the saved hardware context to start execution at main (or wherever specified in the file)
  - Places the PCB on the ready queue

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
    - Child has the same memory and registers: same SP & PC
  - 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
  - Child and parent resume execution from this point: they have same PC

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?

Example Output

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

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()?

- Simple approach to enable concurrent execution
- 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: Unix (2)

- 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?

Process Termination

- All good processes must come to an end. But how?
  - Unix: exit(int status), Windows: ExitProcess(int status)
- Essentially, free resources and terminate
  - Terminate all threads (next lecture)
  - Close open files, network connections
  - Allocated memory (and VM pages out on disk)
  - Remove PCB from kernel data structures, delete
- Note that a process does not need to clean up itself
  - OS will handle this on its behalf

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
  - Returns the process ID of the terminated child
  - waitpid() suspends until the specified child process ends
- Unix: Every process must be reaped by a parent
  - Basically, reading its child’s exit status using wait()
  - What happens if a parent process exits before a child?
    - Orphan process; adopted by init (PID 1)
  - “zombie” process: undead; a process that has died but has not yet been reaped by the parent

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);
   }
}
```

Child (new prog)

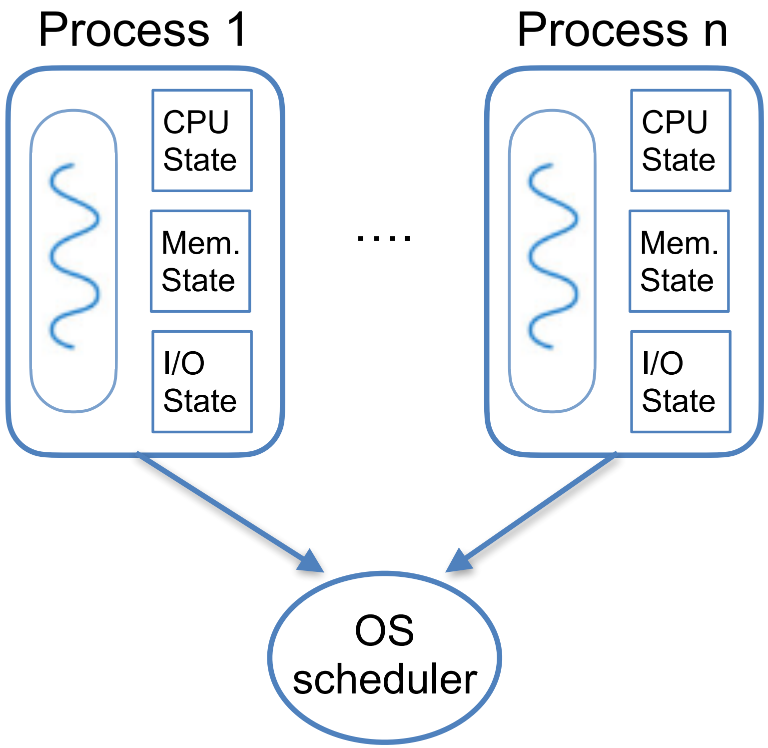
Parent (shell)

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
  - Each process has a **separate address space**
  - Inter Process Communication (IPC) – we will discuss later
  - Overhead of system calls and copying data

Process-associated overheads

- Context switch: **high**
  - CPU state: **low**
  - Memory / IO state: **high**
- Process creation: **high**
- Protection
  - CPU: **yes**
  - Memory / IO: **yes**
- Sharing overhead: **high**
  - At least one context switch



Parallel Programs

- 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 could be very inefficient
  - Space: PCB, page tables, etc.
  - Time: create data structures, fork and copy address space, etc.

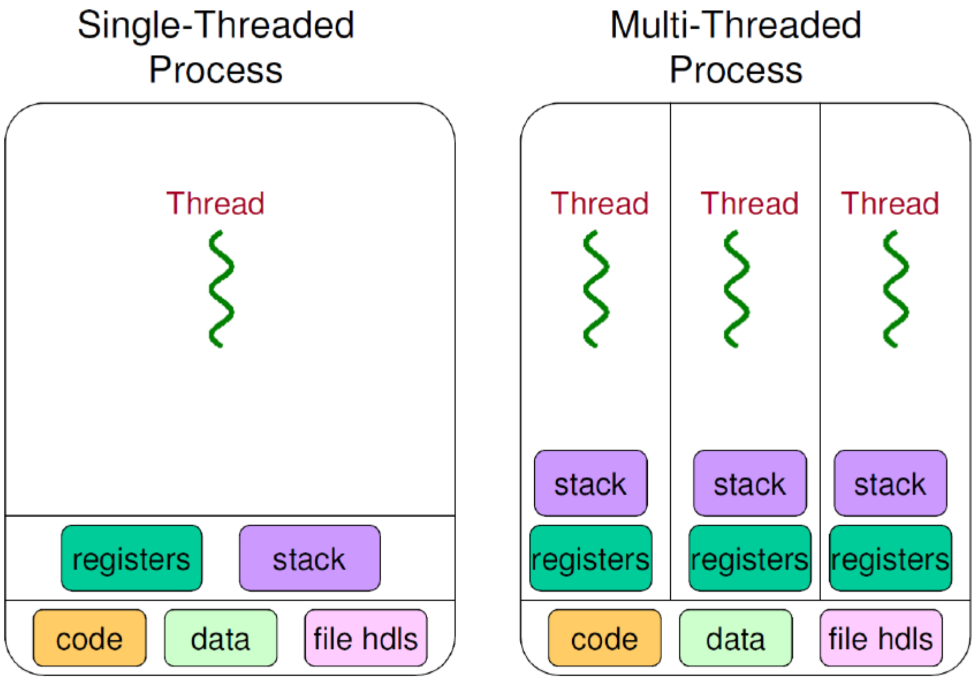
Rethinking Processes

- 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.)
- Why don't they share resources?
  - While each keeps its own execution state: PC, SP, and registers
- Key idea: Separate *resources* from *execution state*

Threads

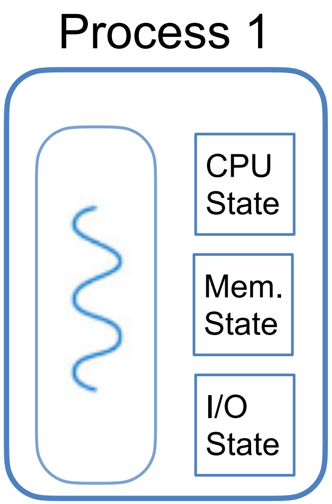
- Separate *execution* and *resource container* roles
  - The thread defines a *sequential execution stream* within a process (PC, SP, registers)
  - The process defines the address space, resources, and general process attributes (everything but threads)

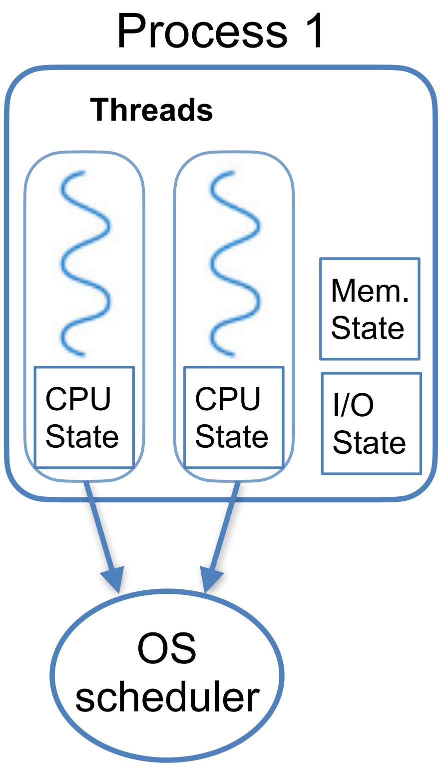
- Threads become the unit of scheduling
  - Processes are now the containers in which threads execute
  - Processes become static, threads are dynamic schedulable entities



Thread-associated overheads

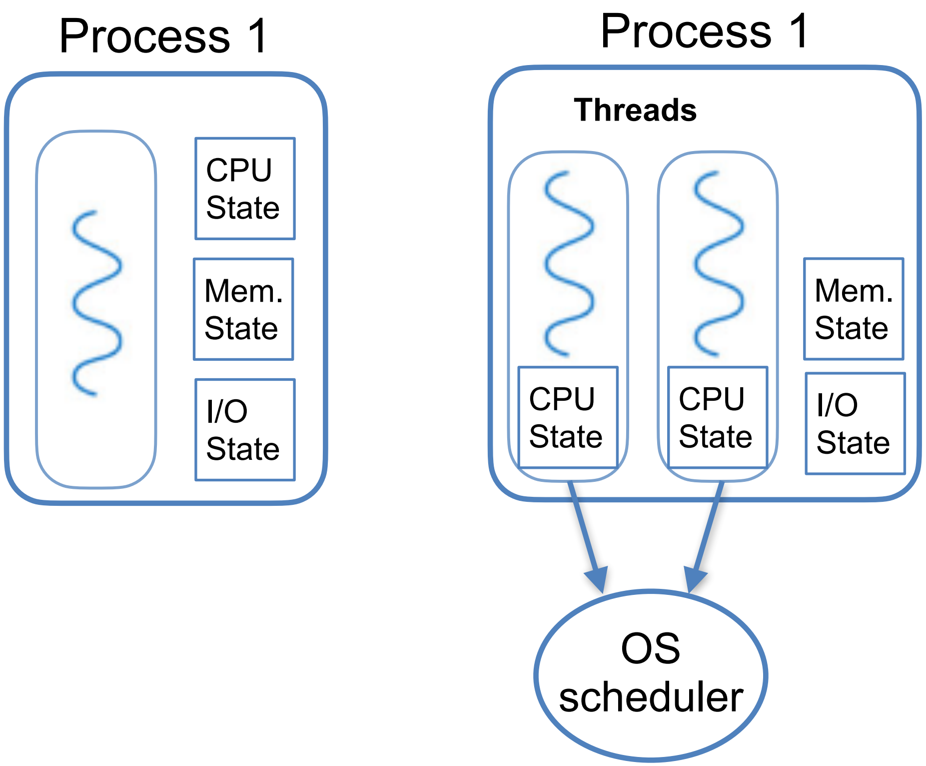
- Context switch: **low**
  - Only CPU state
  - No more memory / IO state
- Thread creation: **low**
- Protection
  - CPU: **yes**
  - Memory / IO: **no**
- Sharing overhead: **low**
  - Due to low thread switch





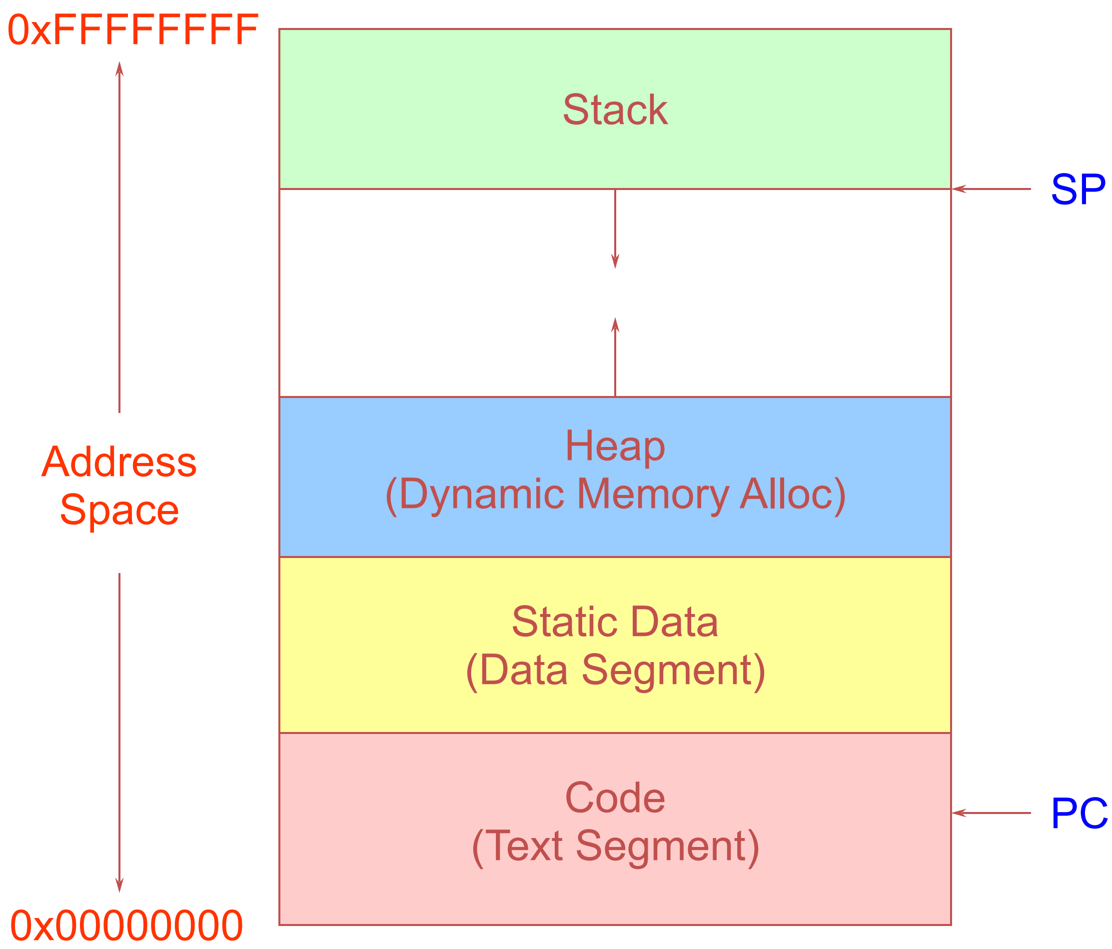
Thread-associated overheads

- Context switch: **low**
  - Only CPU state
  - No more memory / IO state
- Thread creation: **low**
- Protection
  - CPU: **yes**
  - Memory / IO: **no**
- Sharing overhead: **low**
  - Due to low thread switch

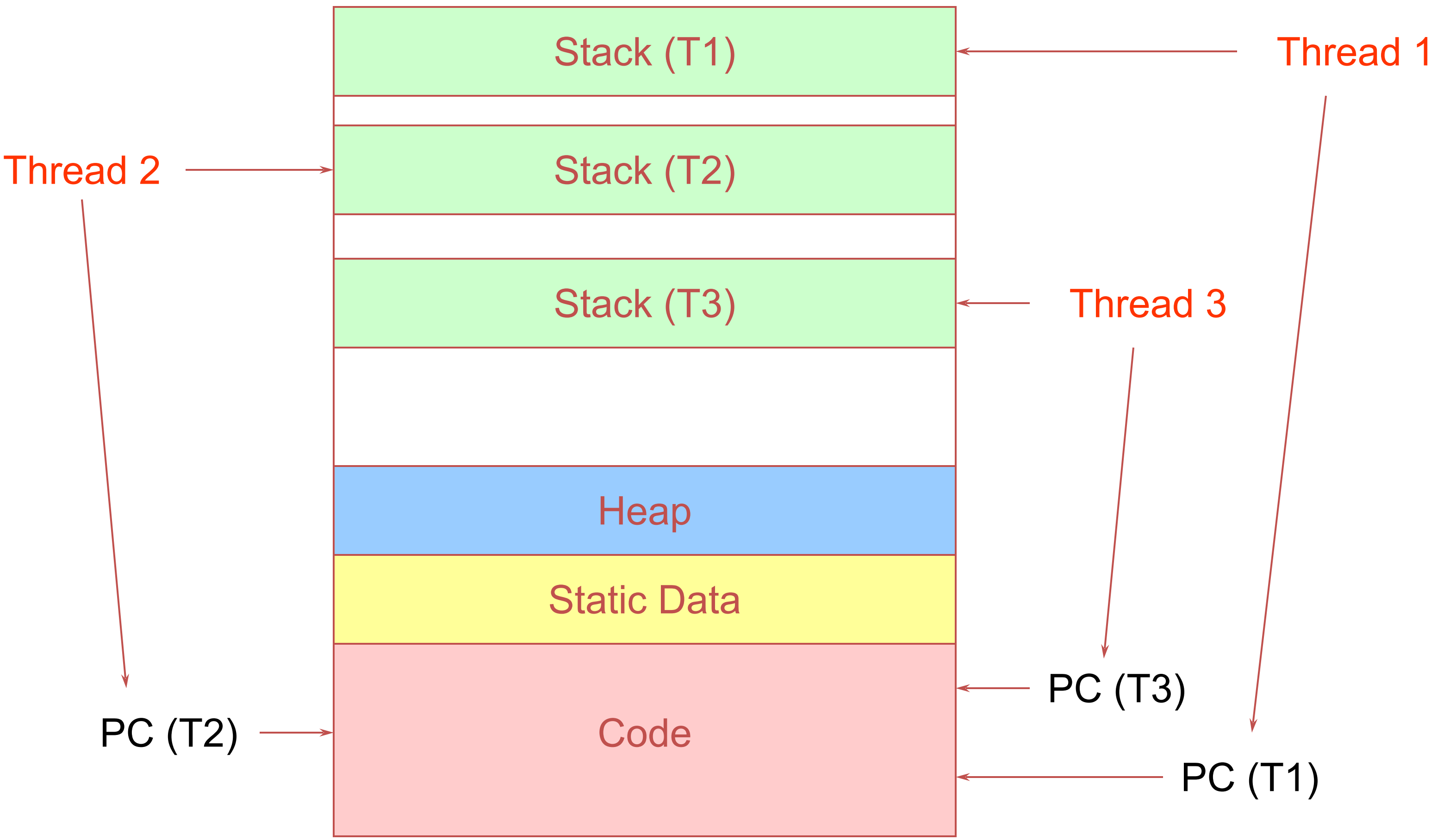


Context switching mainly depends on the process or thread's hunger for memory (i.e., working set size)

Recap: Process Address Space



Threads in a Process



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);
}
```

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

Implementing threads

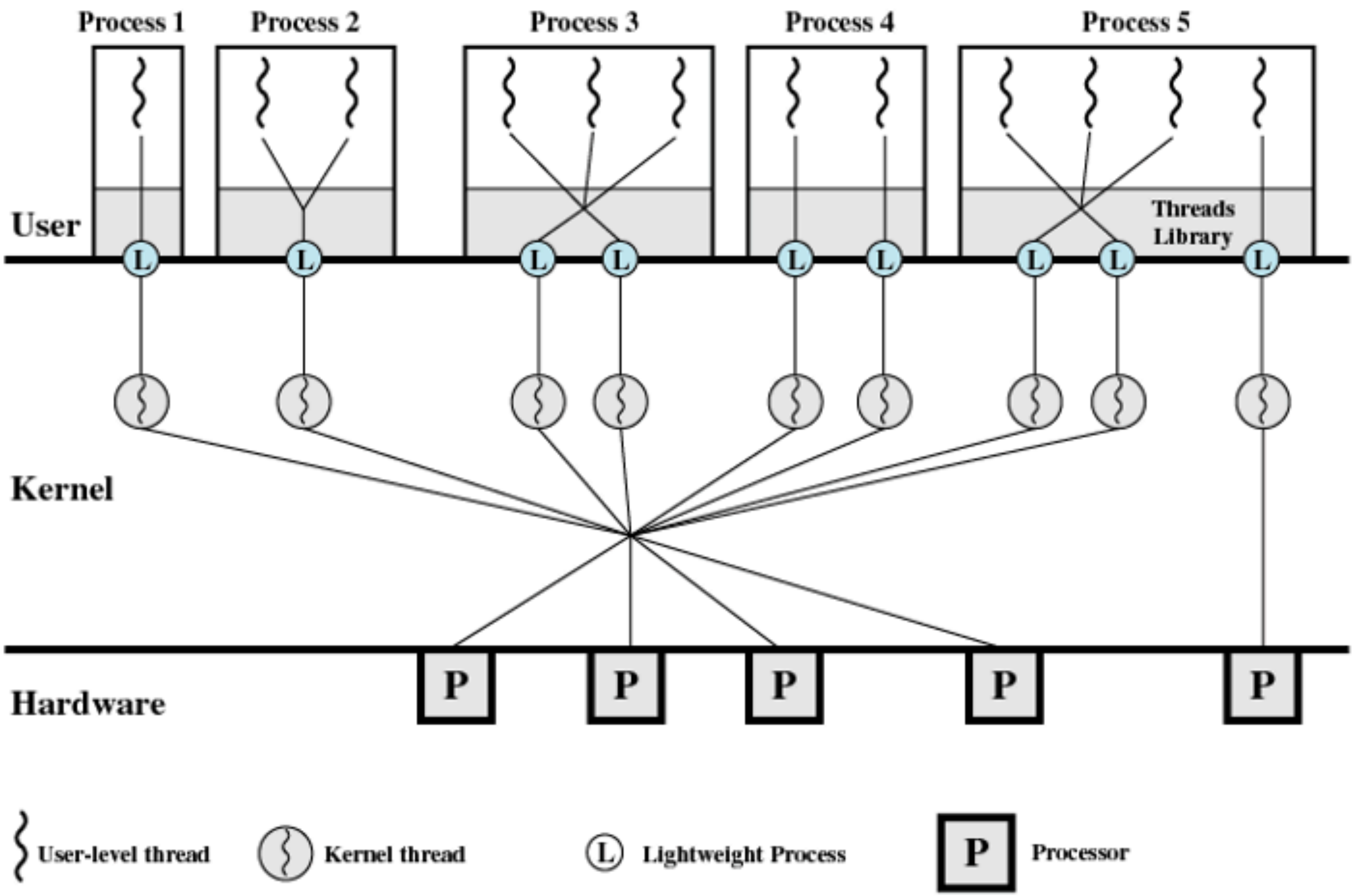
- Kernel-Level Threads
  - All thread operations are implemented in the kernel
  - The OS schedules all of the threads in the system
    - Requires a full thread control block (TCB) for each thread
  - Don’t have to separate from processes

- 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 and manage threads entirely at user level
  - Kernel knows nothing about user-level threads
- 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 user-level procedure call
    - No kernel involvement; No mode switching
  - User-level thread operations can be 100x faster than kernel threads
  - ULTs are required to use non-blocking system calls. (Why?)
  - pthreads: PTHREAD_SCOPE_PROCESS

KLT and ULT combined



(Operating Systems, Stallings)

Figure 4.15 Solaris Multithreaded Architecture Example

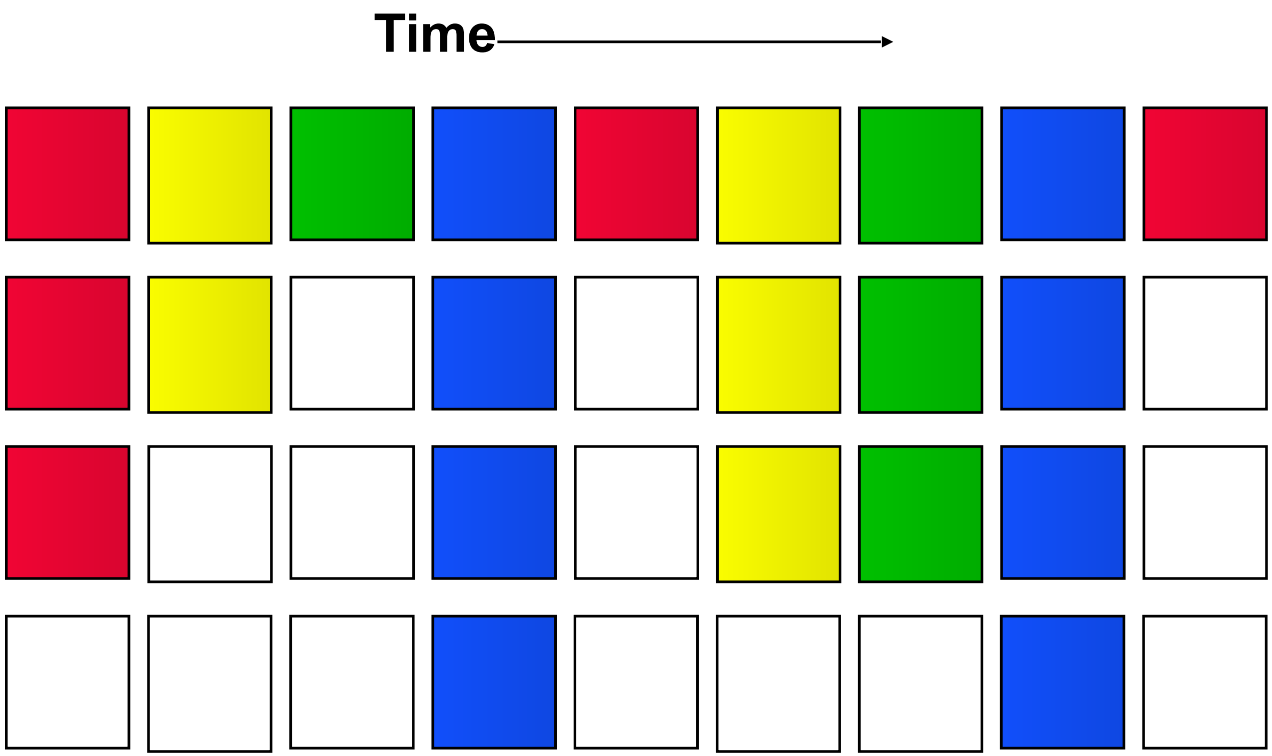
Summary KLT vs. ULT

- 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 fundamental systems knowledge

Simultaneous multithreading & hyperthreading

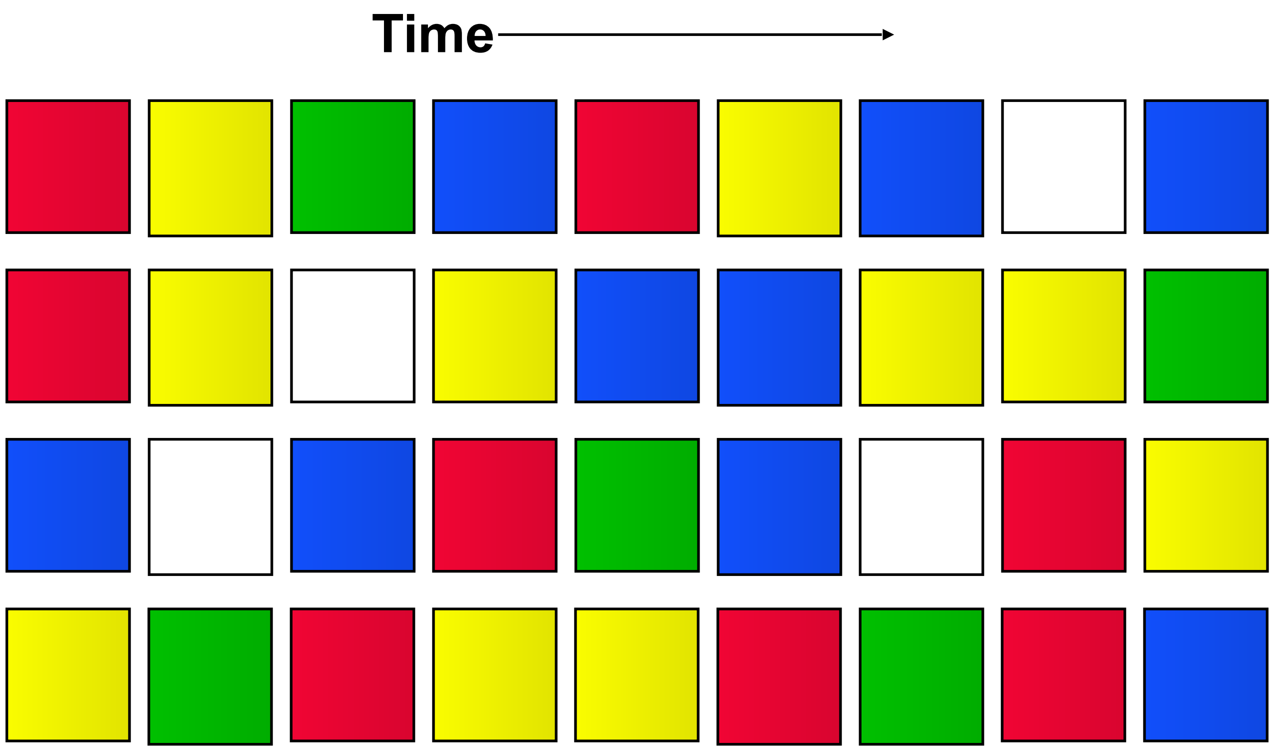
- Traditional multithreading can start execution of instructions from **only** a single thread at a given cycle
  - Low utilization if not enough instructions from a thread to dispatch in one cycle
  - Bad for machines with multiple execution units (i.e., superscalar architecture)
- Idea: dispatch instructions from multiple threads in the same cycle to keep multiple execution units utilized
  - Hirata et al., "An elementary processor architecture with simultaneous instruction issuing from multiple threads," ISCA 1992
  - Tullsen et al., "Simultaneous Multithreading: maximizing on-chip parallelism," ISCA 1995

Multithreading



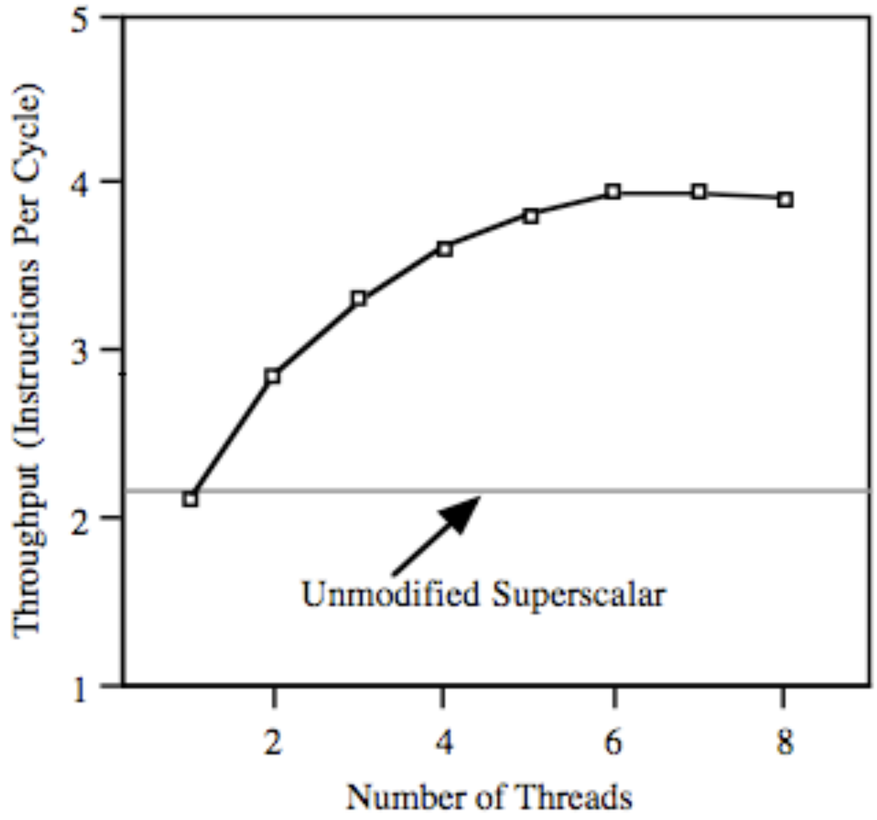
- Intra-thread dependencies
- Single thread performance suffers

SMT



- Utilize functional units with independent operations from multiple threads

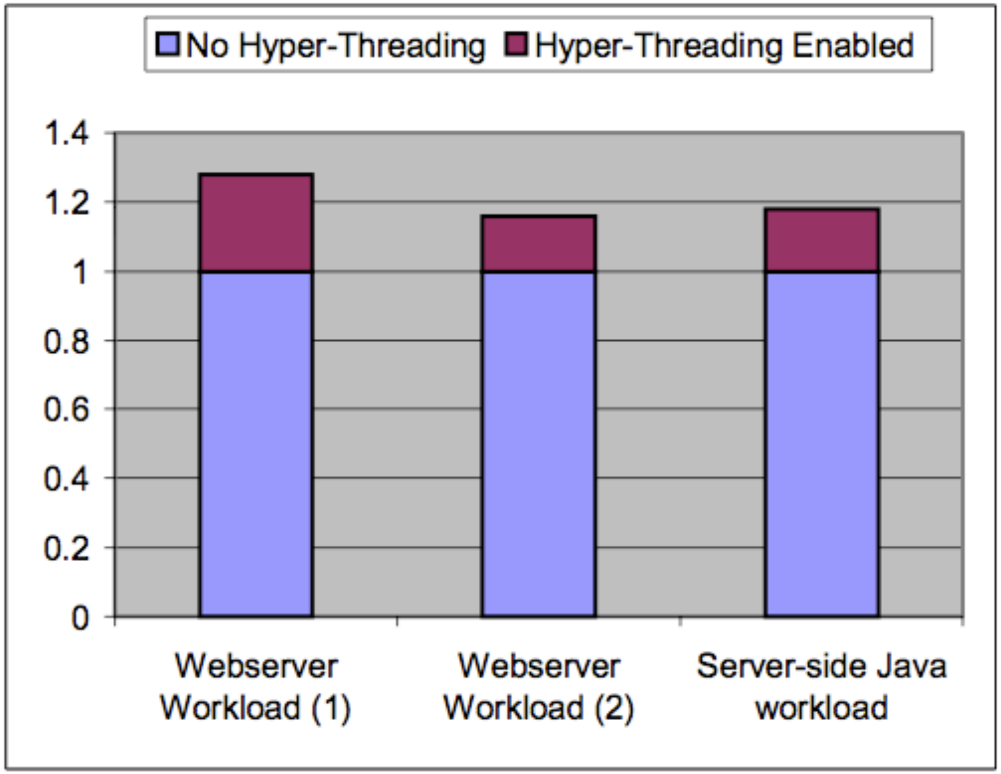
SMT scalability



Commercial SMT implementations

- Intel Pentium 4 (Hyperthreading)
- IBM POWER5
- Intel Nehalem

Intel Pentium 4 HT



- Marr et al., “Hyper-threading technology architecture and microarchitecture,” Intel technology journal, 2002

Summary: why processes & threads?

- Goals
  - Multiprogramming: run multiple applications concurrently
  - Protection: don't want a bad application to crash system
- Solution
  - **Process**: given process/program illusion that it owns hardware resources
- Challenges
  - Process creation & switching is expensive
  - Concurrency within a same application
- Solution
  - **Thread**: decouple allocation and execution
  - Multiple threads within same process with less context switch overhead (only CPU state)