
CSC 369

Operating Systems

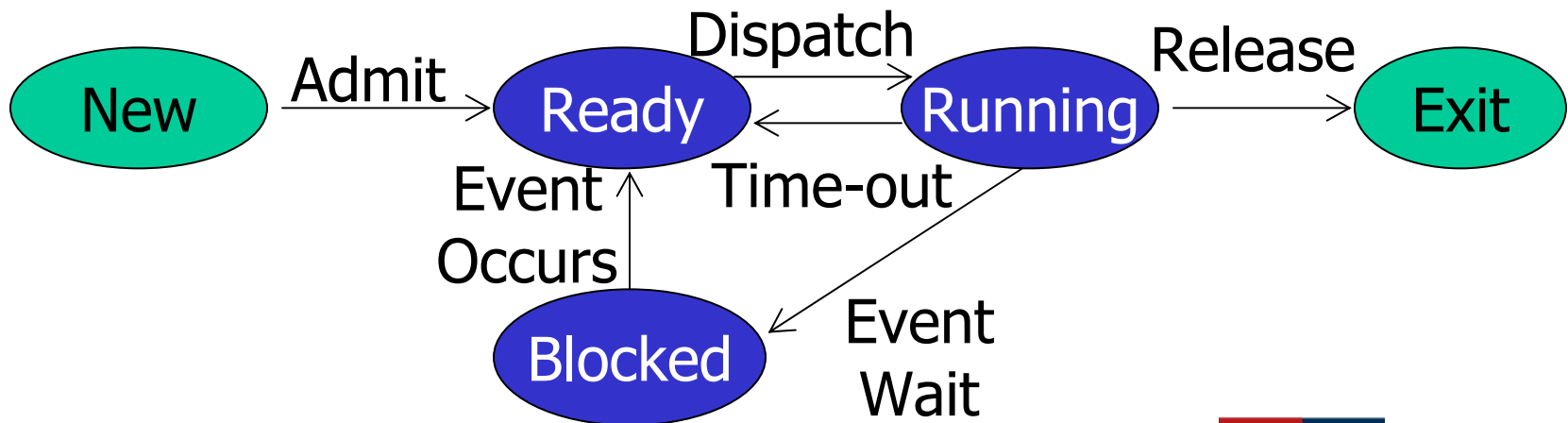
Spring 2007

Week 2: Processes & Threads

Angela Demke Brown

Process Concept

- The process is the OS **abstraction for execution**
 - AKA a **job** or a **task** or a **sequential process**
- Definition: a *process* is a **program in execution**
 - An active entity
 - Programs are static entities with the *potential* for execution
- Managed by process *state* and state changes



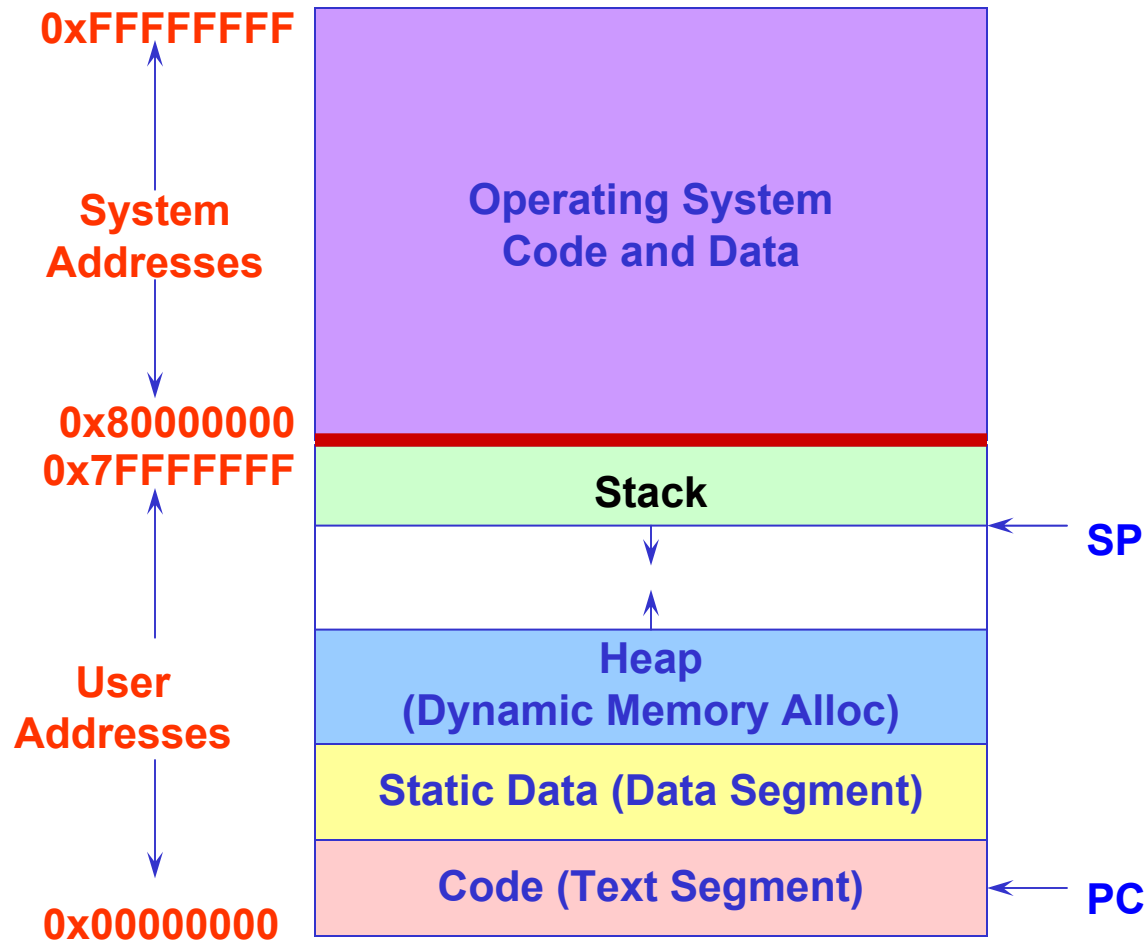
Characterizing a Process

- Ownership of resources
 - Virtual address space
 - Physical memory
 - Persistent Files
 - ...
- Scheduling and Execution
 - Execution state
 - Priority
 - Accounting information (CPU time used, memory used, etc.)

Process Components

- A *process* contains all of the state for a program in execution
 - An address space
 - The code for the executing program
 - The data for the executing program
 - 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
 - A set of operating system resources
 - Open files, network connections, etc.
- A process is named using its process ID (PID)

Process Address Space Diagram



Process Data Structures

How does the OS represent a process in the kernel?

- At any time, there are many processes in the system, each in its own particular state
- The OS data structure representing each process is called the **Process Control Block (PCB)**
 - AKA *process descriptor*
- The PCB contains all of the info about a process
- The PCB also is where the OS keeps all of a process' hardware execution state (PC, SP, regs, etc.) when the process is not running
 - 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 (FreeBSD, 81 fields, 408 bytes)

Process Control Block

- Generally includes:
 - process state (ready, running, blocked ...)
 - program counter: address of the next instruction
 - CPU registers: must be saved at an interrupt
 - CPU scheduling information: process priority
 - memory management info: page tables
 - accounting information: resource use info
 - I/O status information: list of open files

Linux PCB

- Called the *task_struct* in Linux
 - Defined in `/include/linux/sched.h`

```
struct task_struct {
    /* these are hardcoded - don't touch */
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    long counter; long priority; unsigned long signal;
    unsigned long blocked; /* bitmap of masked signals */
    unsigned long flags; /* per process flags, defined below */
    int errno; long debugreg[8]; /* Hardware debugging registers */
    struct exec_domain *exec_domain;
    /* various fields */
    struct linux_binfmt *binfmt;
    struct task_struct *next_task, *prev_task;
    struct task_struct *next_run, *prev_run;
    unsigned long saved_kernel_stack;
    unsigned long kernel_stack_page;
    int exit_code, exit_signal;
    ...
}
```

OS/161 PCB

- Primary OS abstraction is a *thread*
 - In the OS/161 context, threads execute in kernel, processes execute user programs
 - Defined in kern/include/thread.h:

```
struct thread {
    struct pcb t_pcb; /* process control block */
    char *t_name;    /* Name of the thread */
    const void *t_sleepaddr; /* for waiting/blocking */
    char *t_stack;   /* private thread stack area */

    struct addrspace *t_vmspace; /* user address space */
    struct vnode *t_cwd; /* current working directory */
};
```

- Eventually, you will need to add to this structure!

OS/161 PCB (continued)

- Thread struct contains a "struct pcb"
 - Defined in kern/arch/mips/include/pcb.h:

```
struct pcb {
    u_int32_t pcb_switchstack; // Stack saved during ctxt switch
    u_int32_t pcb_kstack;      // Stack to load on entry to kernel
    u_int32_t pcb_ininterrupt; // Are we in an interrupt handler
    pcb_faultfunc pcb_badfaultfunc; // Recovery for fatal
                                    //kernel traps
    jmp_buf pcb_copyjmp; // jump area used by copyin/out etc.
};
```

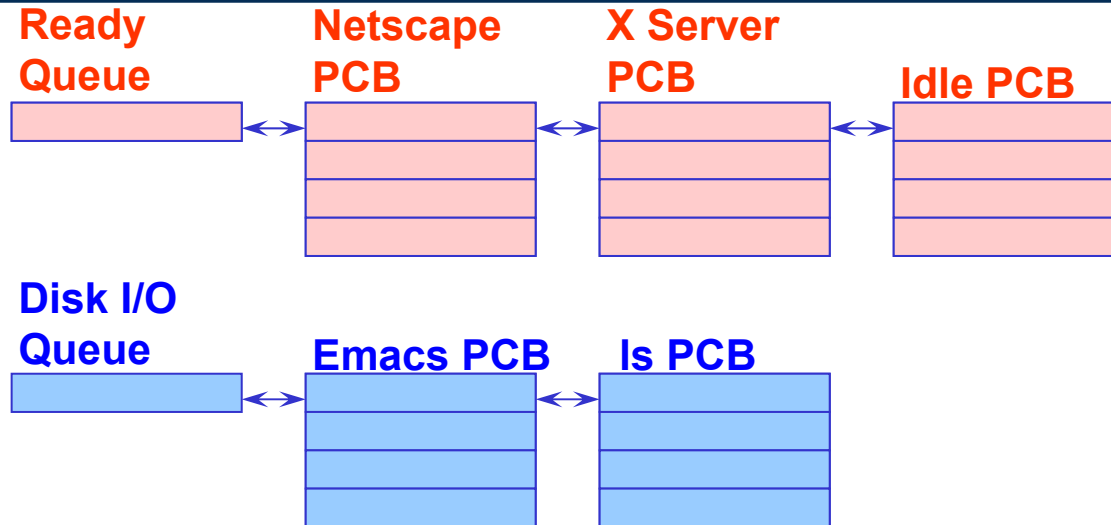
- Machine dependent. You should NOT add to this.
 - Recommend reading all of pcb.h however
 - Note there is no area for registers in thread or pcb struct!
 - OS/161 saves registers on thread stack during switch

State Queues

How does the OS keep track of processes?

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



Console Queue

Sleep Queue

·
·
·

- There may be many wait queues, one for each type of wait (disk, console, timer, network, etc.)
- OS/161 maintains a single wait queue
 - "struct array *sleepers" in kern/thread/thread.c
 - record address of thing thread is waiting for

PCBs and State Queues

- PCBs are data structures dynamically allocated in OS memory
- When a process is created, the OS allocates a PCB for it, initializes it, and places it on the *Ready* queue
- As the process computes, does I/O, etc., its PCB moves from one queue to another
- When the process terminates, its PCB is deallocated

What is a Context Switch?

- *context switch*: switch the CPU to another process, saving the state of the old process and loading the saved state for the new process
- context switch time is pure overhead, so some systems offer specific hardware support
- a performance bottleneck, so new structures (threads) are being used to avoid it

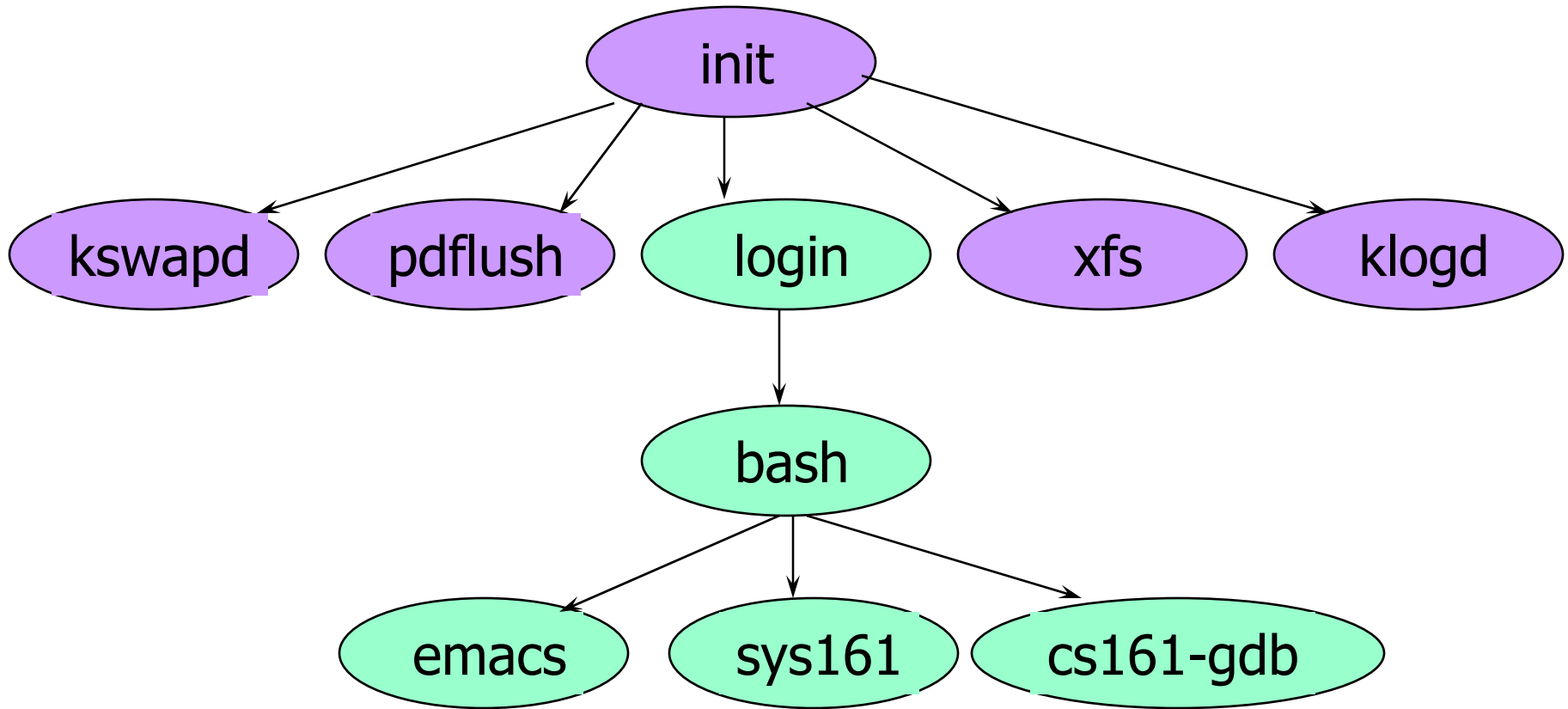
Operations on Processes

- processes execute concurrently and must be created and deleted dynamically
- **process creation**: parent, child, tree of processes
- **process termination**: when a process finishes executing its last statement or when a parent causes the termination of a child process, perhaps because the child exceeded resource requirements

Process Creation

- A process is created by another process
 - Parent is creator, child is created
 - In Linux, the parent is the "PPID" field of "ps -f"
 - 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)

Linux Process Tree



Process Creation: NT

- The system call on XP 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 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**
 - 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
 - Huh?

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

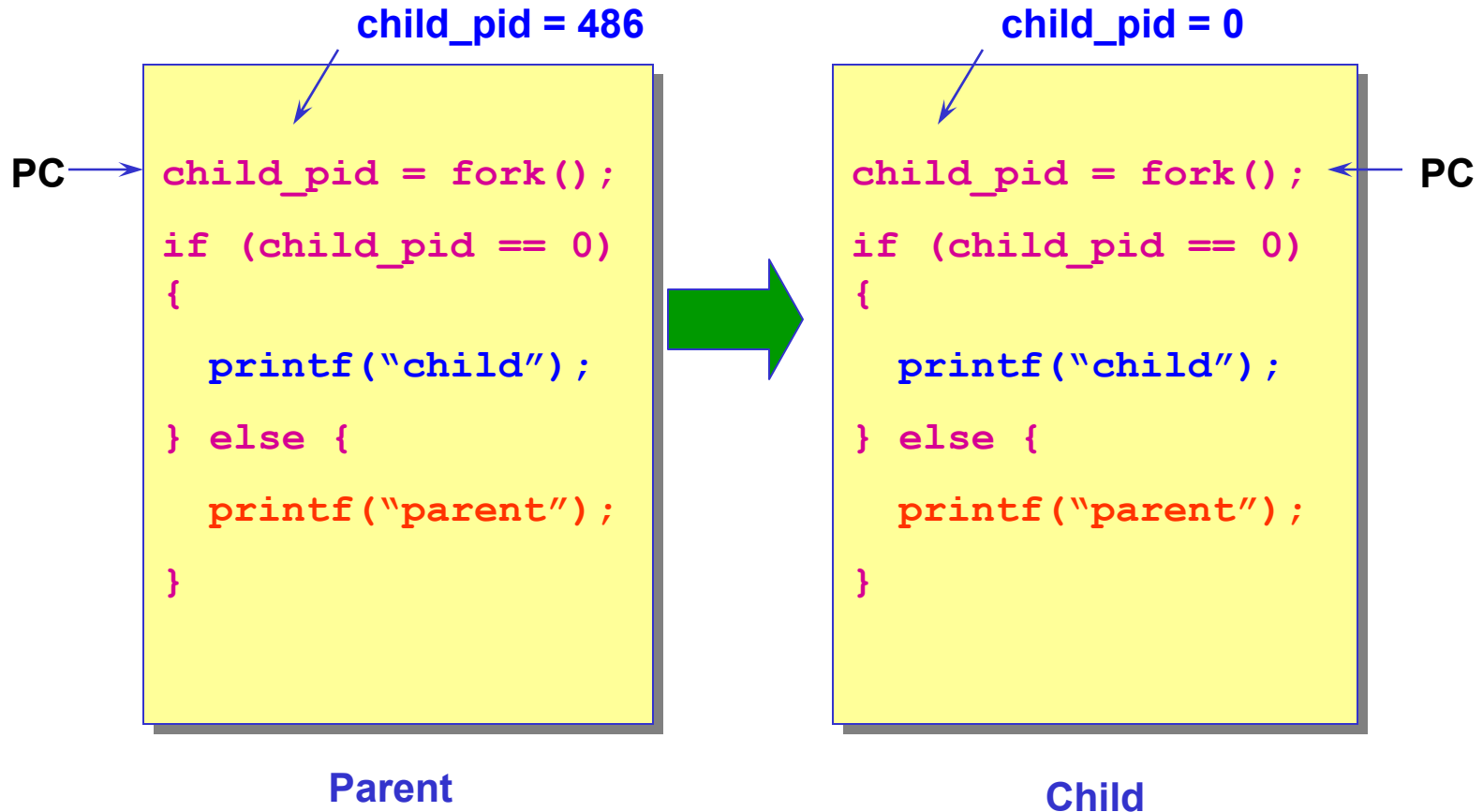
```
skywolf% cc t.c
```

```
skywolf% ./a.out
```

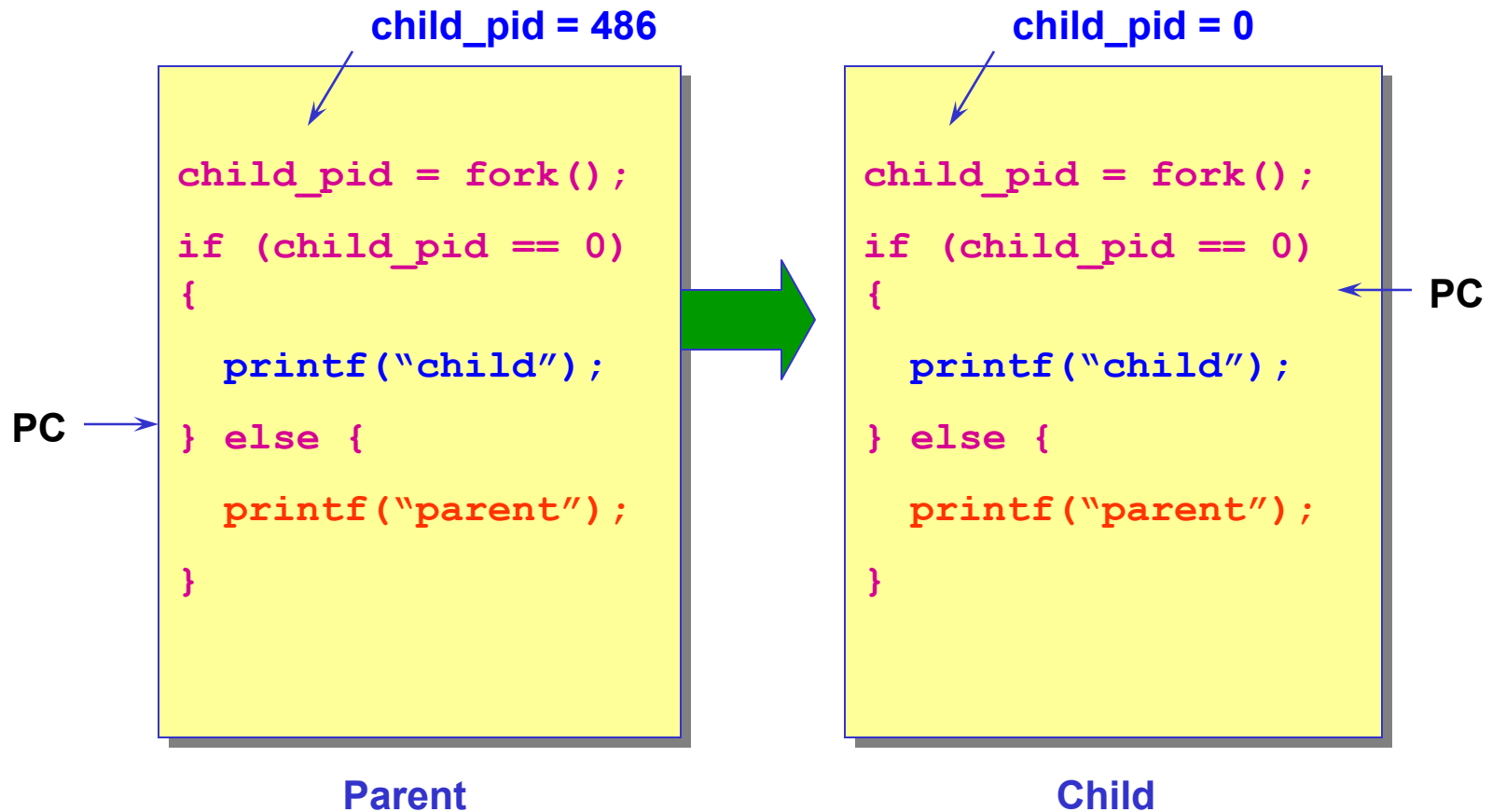
```
My child is 486
```

```
Child of a.out is 486
```

Duplicating Address Spaces



Divergence



Example Continued

```
skywolf% cc t.c
```

```
skywolf% a.out
```

```
My child is 486
```

```
Child of a.out is 486
```

```
skywolf% 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: Unix (2)

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

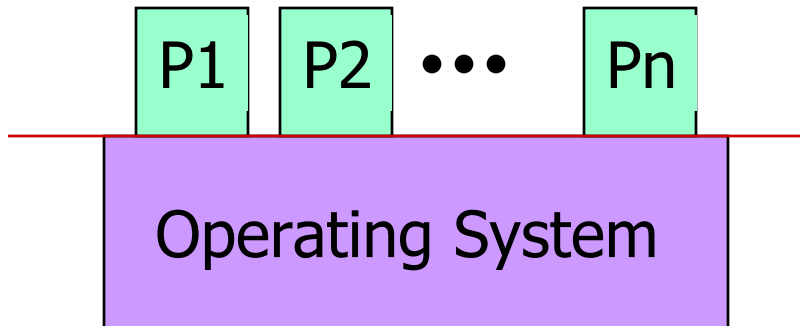
```
int exec(char *prog, char *argv[])
```

- `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?

Unix Shells

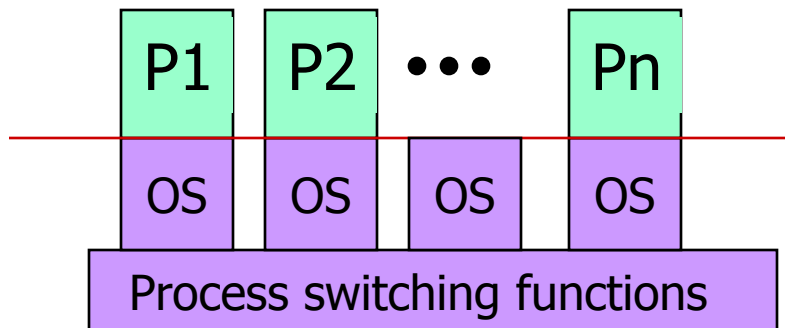
```
while (1) {  
    char *cmd = read_command();  
    int child_pid = fork();  
    if (child_pid == 0) {  
        exec(cmd);  
        panic("exec failed");  
    } else {  
        wait(child_pid);  
    }  
}
```

Relation of OS & Processes



Option 1: Non-process kernel

- OS code executes in a separate context from any user program, in system (privileged) mode
- mode switch implies context switch



Option 2: Execution in user processes

- most OS code executes in context of a user process, but privileged
- mode switch **DOES NOT** imply context switch! (cheaper)
- some OS code executes as system process (no associated user pgm)

Processes

- Recall that a process includes many things
 - An address space (defining all the code and data pages)
 - OS resources (e.g., open files) and accounting information
 - Execution state (PC, SP, regs, etc.)
- Creating a new process is costly because of all of the data structures that must be allocated and initialized
 - FreeBSD: 81 fields, 408 bytes
 - ...which does not even include page tables, etc.
- Often want multiple processes to work together to accomplish some task

Parallel Programs

- Recall our Web server example that forks off copies of itself to handle multiple simultaneous requests
 - Or any parallel program that executes on a multiprocessor
- 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 (logically or physically)
- This situation is **very inefficient**
 - **Space**: PCB, page tables, etc.
 - **Time**: create data structures, fork and copy addr space, etc.

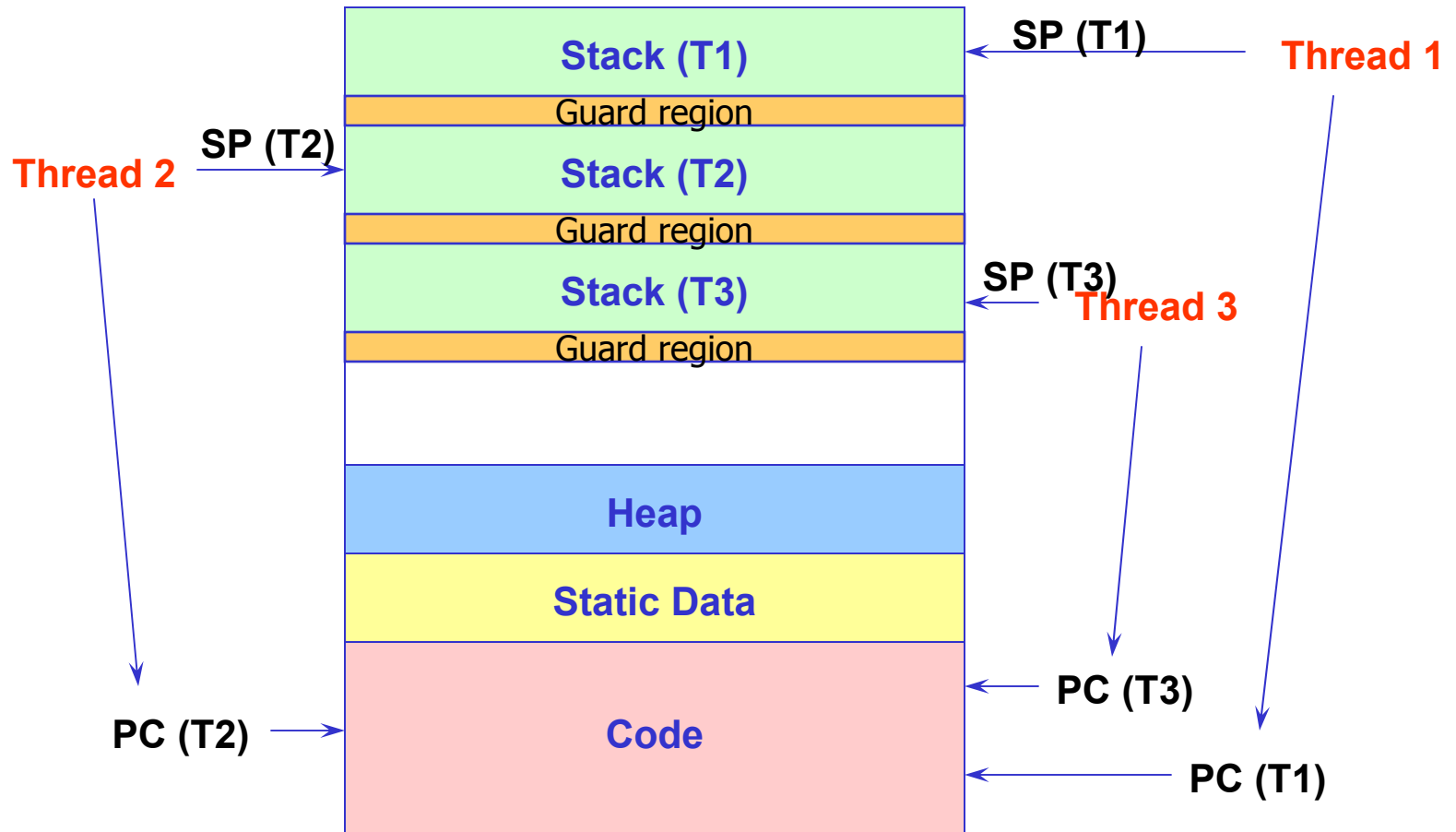
Rethinking Processes

- 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:** Why don't we separate the concept of a process from its execution state?
 - **Process:** address space, privileges, resources, etc.
 - **Execution state:** PC, SP, registers
- Exec state also called **thread of control**, or **thread**

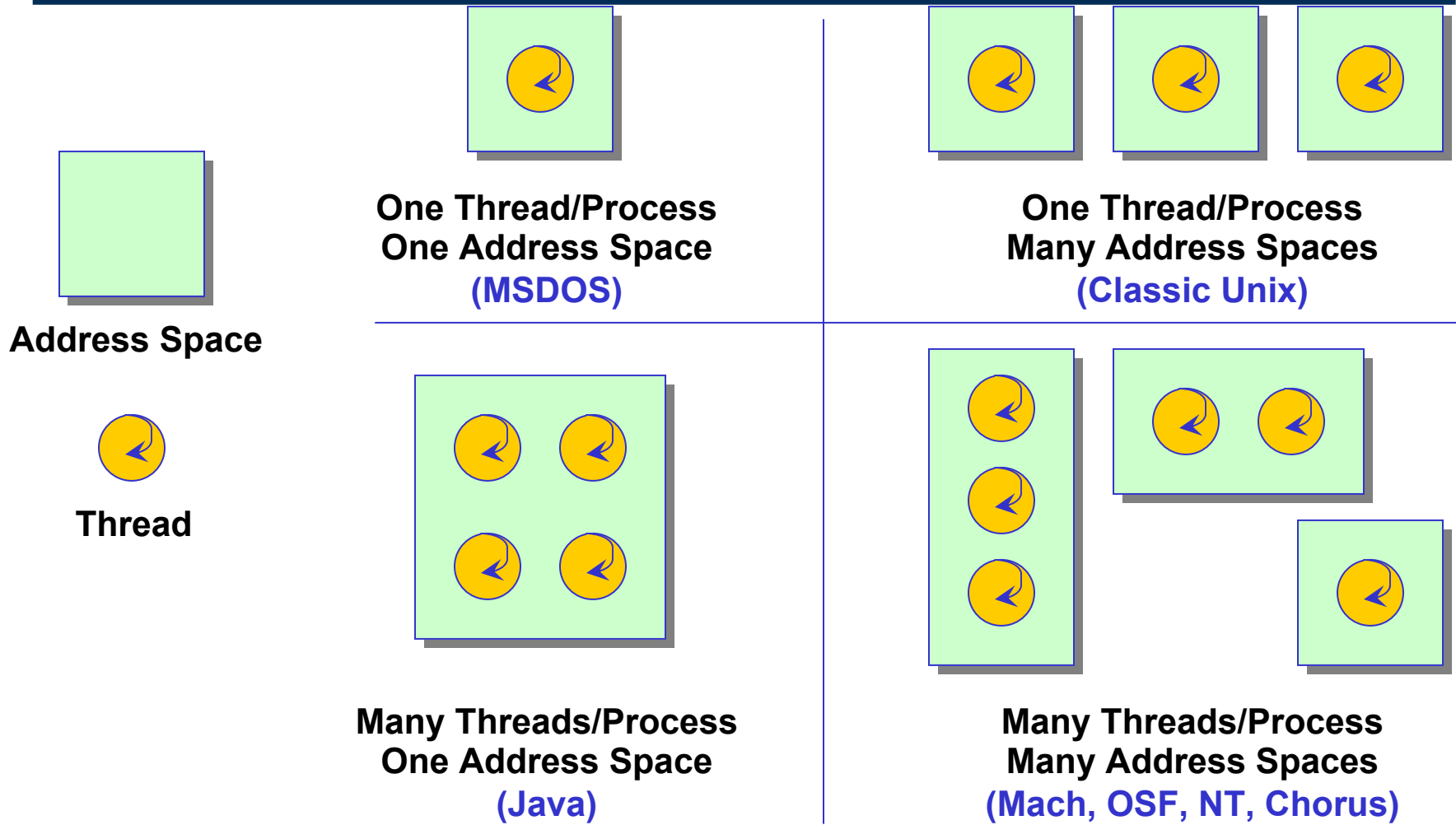
Threads

- Modern OSes (Mach, Chorus, NT, modern Unix) separate the concepts of processes and threads
 - The **thread** defines a sequential execution stream within a process (PC, SP, registers)
 - The **process** defines the address space and general process attributes (everything but threads of execution)
- A thread is bound to a single process
 - Processes, however, can have multiple threads
- Threads become the unit of scheduling
 - Processes are now the **containers** in which threads execute
 - Processes become static, threads are the dynamic entities
 - Every process has at least one thread

Threads in a Process



Thread Design Space



Process/Thread Separation

- Separating threads and processes makes it easier to support parallel/concurrent applications
 - Creating concurrency does not require creating new processes, just more threads
- 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

- Recall our forking Web server:

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

- Using `fork()` to create new processes to handle requests in parallel is overkill for such a simple task

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);  
}
```

Kernel-Level Threads

- We have taken the execution aspect of a process and separated it out into threads
 - To make concurrency cheaper
- As such, the OS now manages threads *and* processes
 - All thread operations are implemented in the kernel
 - The OS schedules all of the threads in the system
- OS-managed threads are called **kernel-level threads** or **lightweight processes**
 - NT: **threads**
 - Solaris: **lightweight processes (LWP)**

Kernel Thread Limitations

- Kernel-level threads make concurrency much cheaper than processes
 - Much less state to allocate and initialize
- However, for fine-grained concurrency, kernel-level threads still suffer from too much overhead
 - Thread operations still require system calls
 - Ideally, want thread operations to be **as fast as a procedure call**
 - Kernel-level threads have to be general to support the needs of all programmers, languages, runtimes, etc.
- For fine-grained concurrency, need even “cheaper” threads

User-Level Threads

- To make threads cheap and fast, they need to be implemented at user level
 - Kernel-level threads are managed by the OS
 - User-level threads are managed entirely by the run-time system (user-level library)
- User-level threads 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 are up to 100x faster than kernel threads
 - But this depends on the quality of both implementations!

U/L Thread Limitations

- But, user-level threads are not a perfect solution
 - As with everything else, they are a tradeoff
- 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 whose thread initiated an I/O, even though the process has other threads that can execute
 - Unscheduling a process with a thread holding a lock
- Solving this requires communication between the kernel and the user-level thread manager

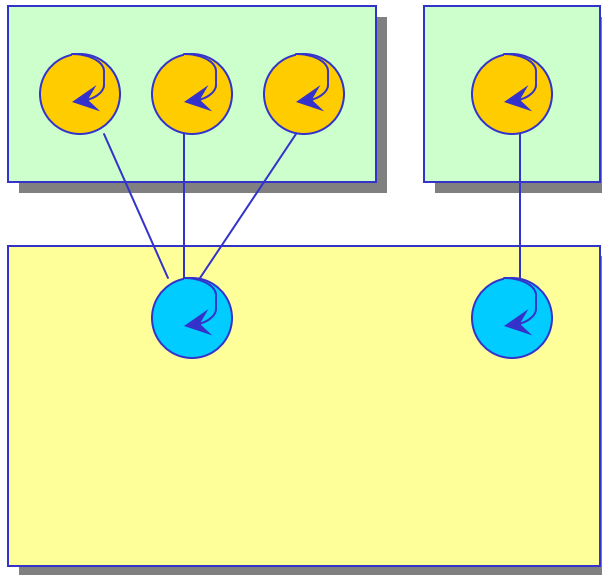
Kernel vs. User Threads

- 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

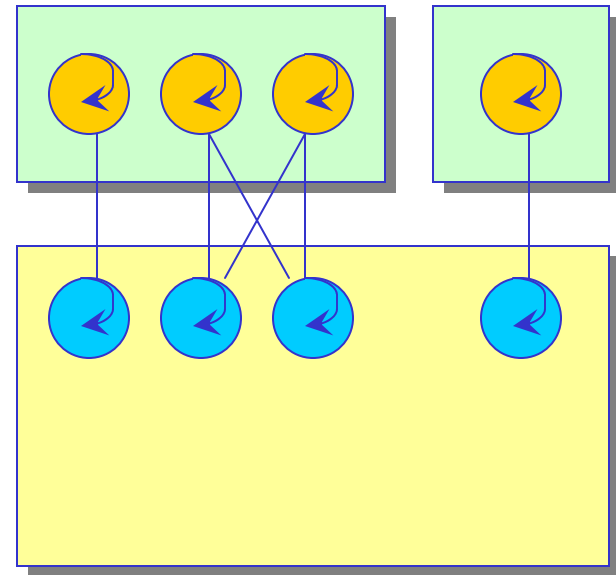
Hybrid Kernel and User Threads

- Another possibility is to use both kernel and user-level threads
 - Can associate a user-level thread with a kernel-level thread
 - Or, multiplex user-level threads on top of kernel-level threads
- Java Virtual Machine (JVM)
 - Java threads are user-level threads (sort of)
 - On older Unix, only one "kernel thread" per process
 - Multiplex all Java threads on this one kernel thread
 - On NT, Solaris, OSF
 - Can multiplex Java threads on multiple kernel threads
 - Can have more Java threads than kernel threads

User and Kernel Threads



**Multiplexing user-level threads
on a single kernel thread for
each process**



**Multiplexing user-level threads
on multiple kernel threads for
each process**

Implementing Threads

- Implementing threads has a number of issues
 - Interface
 - Context switch
 - Preemptive vs. non-preemptive
 - Scheduling
 - Synchronization (next lecture)
- Focus on user-level threads
 - Kernel-level threads are similar to original process management and implementation in the OS (see kern/thread subdirectory in OS/161)

Thread Interface (Pthread API)

- `pthread_create(pthread_t *tid, pthread_attr_t attr, void *(*start_routine)(void *), void *arg)`
 - Create a new thread of control
 - New thread id returned in `tid`, new thread starts executing in `start_routine` with argument `arg`
- `pthread_join(pthread_t tid)`
 - Wait for `tid` to exit
- `pthread_cancel(pthread_t tid)`
 - Destroy `tid`
- `pthread_yield()`
 - Voluntarily give up the processor
- `pthread_exit()`
 - Terminate the calling thread

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)`?

Threads Summary

- The operating system is a large multithreaded program
 - Each process executes as a thread within the OS
- Multithreading is also very useful for applications
 - Efficient multithreading requires fast primitives
 - Processes are too heavyweight
- 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
- Now, how do we get our threads to correctly cooperate with each other?
 - Synchronization... next time!