

Categorizing Events



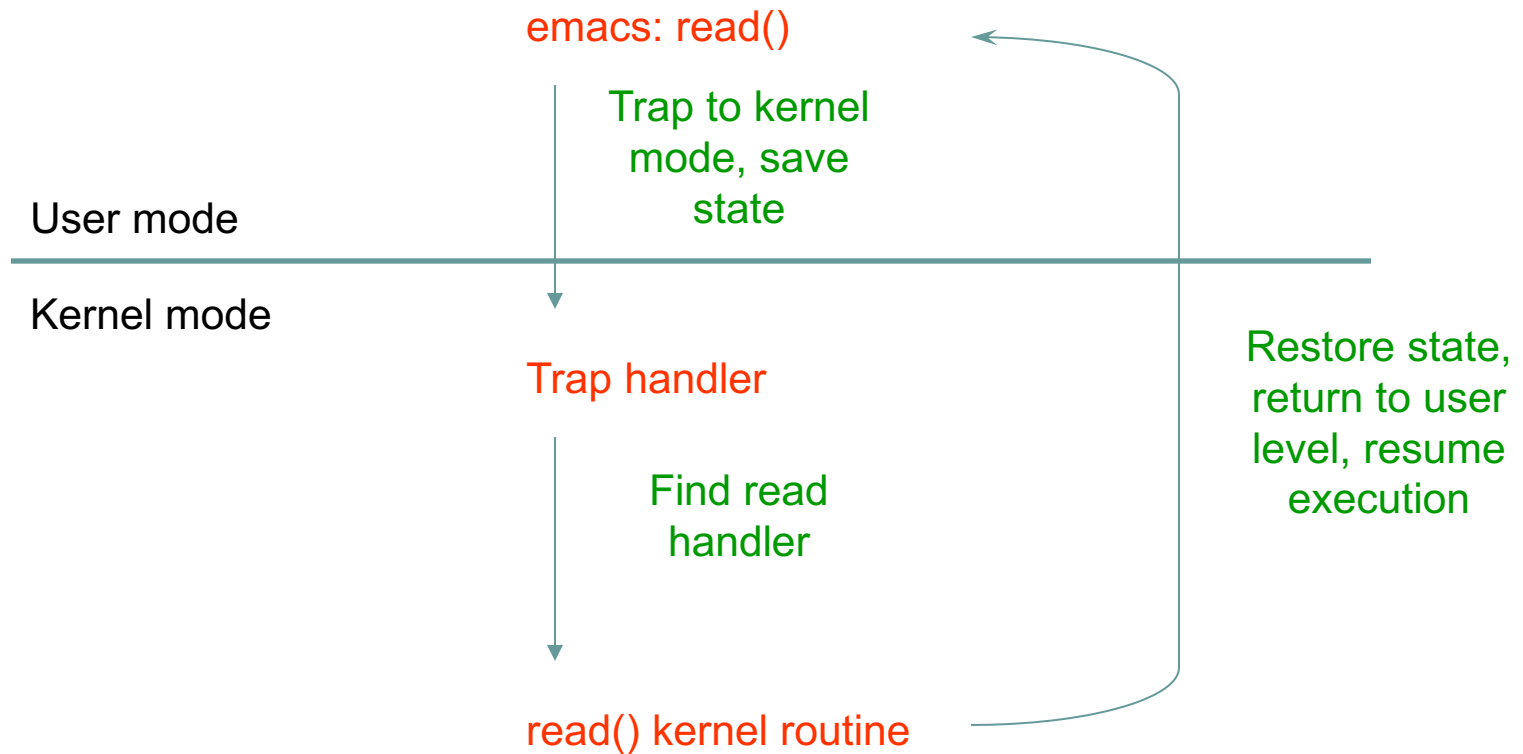
	Unexpected	Deliberate
Synchronous	fault	syscall trap
Asynchronous	interrupt	signal

System Calls



- › For a user program to do something “privileged” (e.g., I/O) it must call an OS procedure
 - › Known as **crossing the protection boundary**, or a **protected procedure call**
- › Hardware provides a **system call** instruction that:
 - › Causes an exception, which invokes a kernel handler
 - › Passes a parameter determining the system routine to call
 - › Saves caller state (PC, regs, mode) so it can be restored
 - › **Why save mode?**
 - › Returning from system call restores this state

System Call



System Call Questions

- ▶ There are hundreds of syscalls. How do we let the kernel know which one we intend to invoke?
 - ▶ Before issuing **int \$0x80** or **sysenter**, set **%eax/%rax** with the syscall number

- ▶ System calls are like function calls, but how to pass parameters?
 - ▶ Just like calling convention in syscalls, typically passed through **%ebx**, **%ecx**, **%edx**, **%esi**, **%edi**, **%ebp**

More questions



- ▶ How to reference kernel objects (e.g., files, sockets)?
 - ▶ Naming problem – an integer mapped to a unique object
 - ▶ `int fd = open("file"); read(fd, buffer);`
 - ▶ Why can't we reference the kernel objects by memory address?

Categorizing Events

	Unexpected	Deliberate
Synchronous	fault	syscall trap
Asynchronous	interrupt	software interrupt

- Interrupts signal asynchronous events
 - ◆ I/O hardware interrupts
 - ◆ Software and hardware timers

Timer



- › The key to a timesharing OS
- › The fallback mechanism by which the OS reclaims control
 - › Timer is set to generate an interrupt after a period of time
 - › Setting timer is a privileged instruction
 - › When timer expires, generates an interrupt
 - › Handled by the OS, forcing a switch from the user program
 - › Basis for OS **scheduler** (*more later...*)
- › Also used for time-based functions (e.g., *sleep()*)

I/O Control

- ▶ I/O issues
 - ▶ Initiating an I/O
 - ▶ Completing an I/O
- ▶ Initiating an I/O
 - ▶ Special instructions
 - ▶ Memory-mapped I/O
 - ▶ Device registers mapped into address space
 - ▶ Writing to address sends data to I/O device

I/O using Interrupts

- ▶ Interrupts are the basis for asynchronous I/O
 - ▶ OS initiates I/O
 - ▶ Device operates independently of rest of machine
 - ▶ Device sends an interrupt signal to CPU when done
 - ▶ OS maintains a vector table containing a list of addresses of kernel routines to handle various events
 - ▶ CPU looks up kernel address indexed by interrupt number, context switches to routine

I/O Example

1. Ethernet receives packet, writes packet into memory
2. Ethernet signals an interrupt
3. CPU stops current operation, switches to kernel mode, saves machine state (PC, mode, etc.) on kernel stack
4. CPU reads address from vector table indexed by interrupt number, branches to address (Ethernet device driver)
5. Ethernet device driver processes packet (reads device registers to find packet in memory)
6. Upon completion, restores saved state from stack

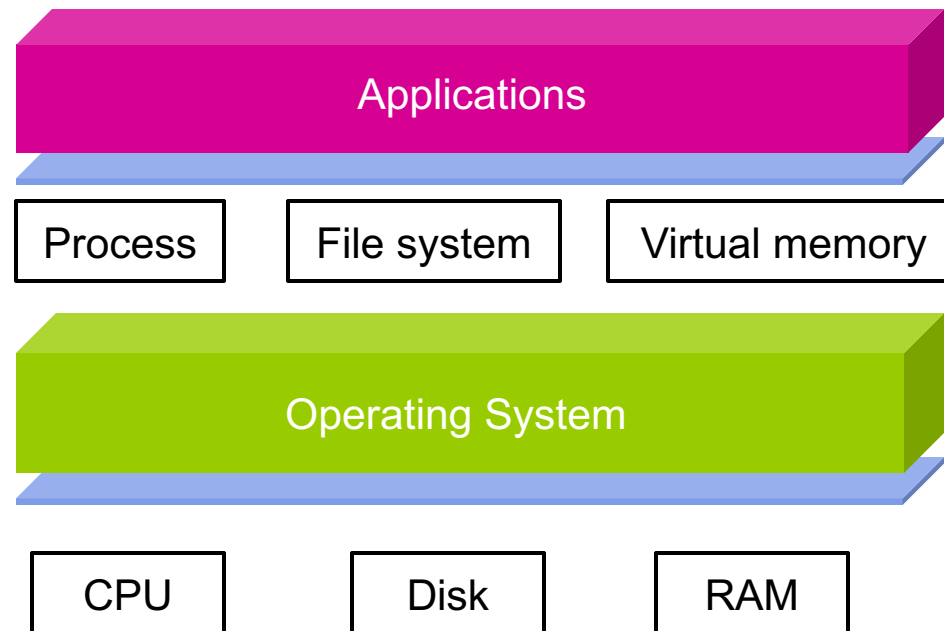
Summary



- ▶ Protection
 - ▶ User/kernel modes
 - ▶ Protected instructions
- ▶ System calls
 - ▶ Used by user-level processes to access OS functions
 - ▶ Access what is “in” the OS
- ▶ Exceptions
 - ▶ Unexpected event during execution (e.g., divide by zero)
- ▶ Interrupts
 - ▶ Timer, I/O

Processes

OS Abstractions



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

The Process

- › The process is the OS **abstraction for execution**
 - › It is the unit of execution
 - › It is the unit of scheduling

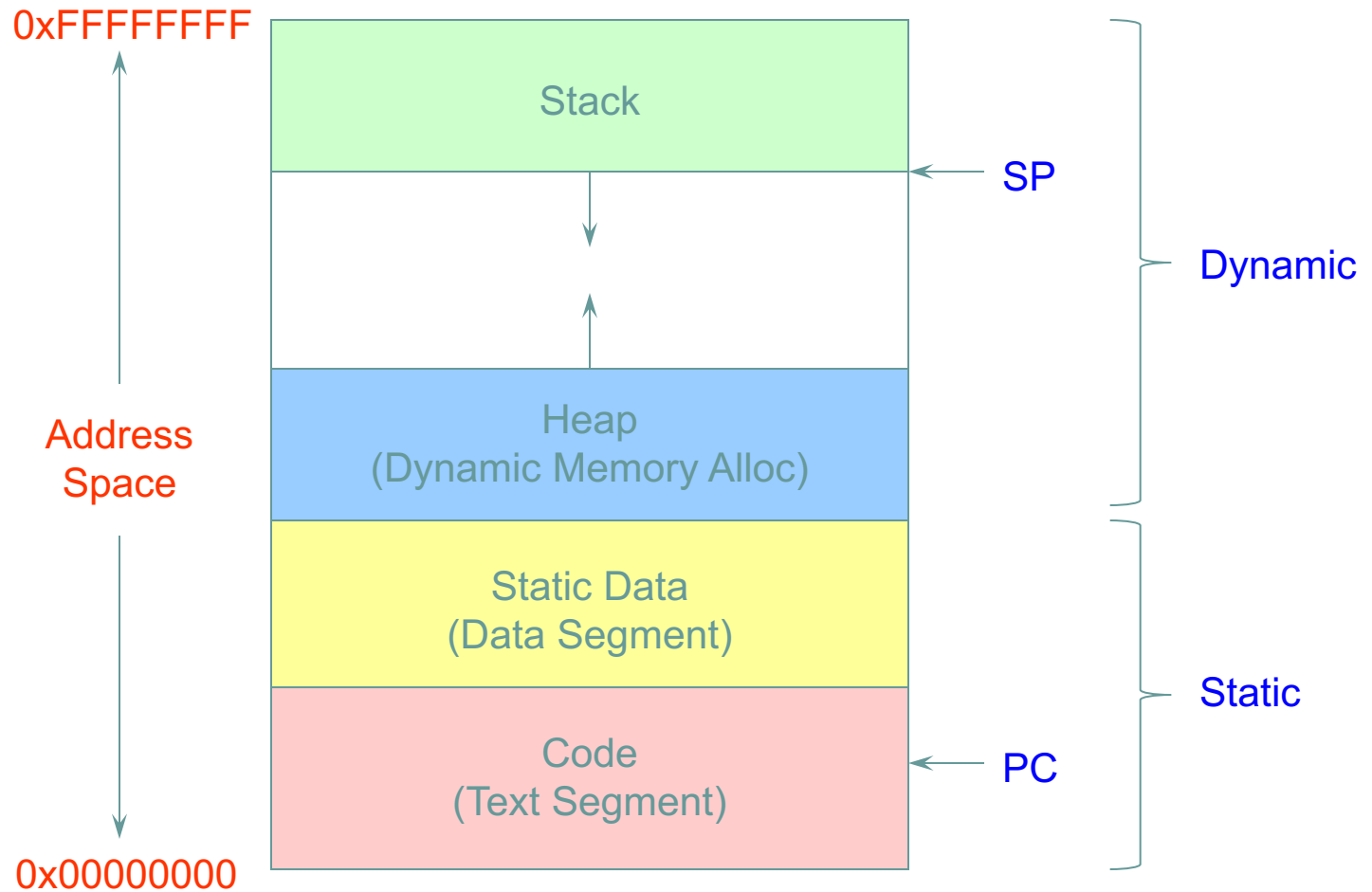
- › A process is a **program in execution**
 - › Programs are static entities with the **potential** for execution
 - › Process is the animated/active program
 - › Starts from the program, but also includes dynamic state
 - › As the representative of the program, it is the “owner” of other resources (memory, files, sockets, ...)

- › How does the OS implement this abstraction?
 - › How does it share the CPU?

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

PROCESS STATE CODES

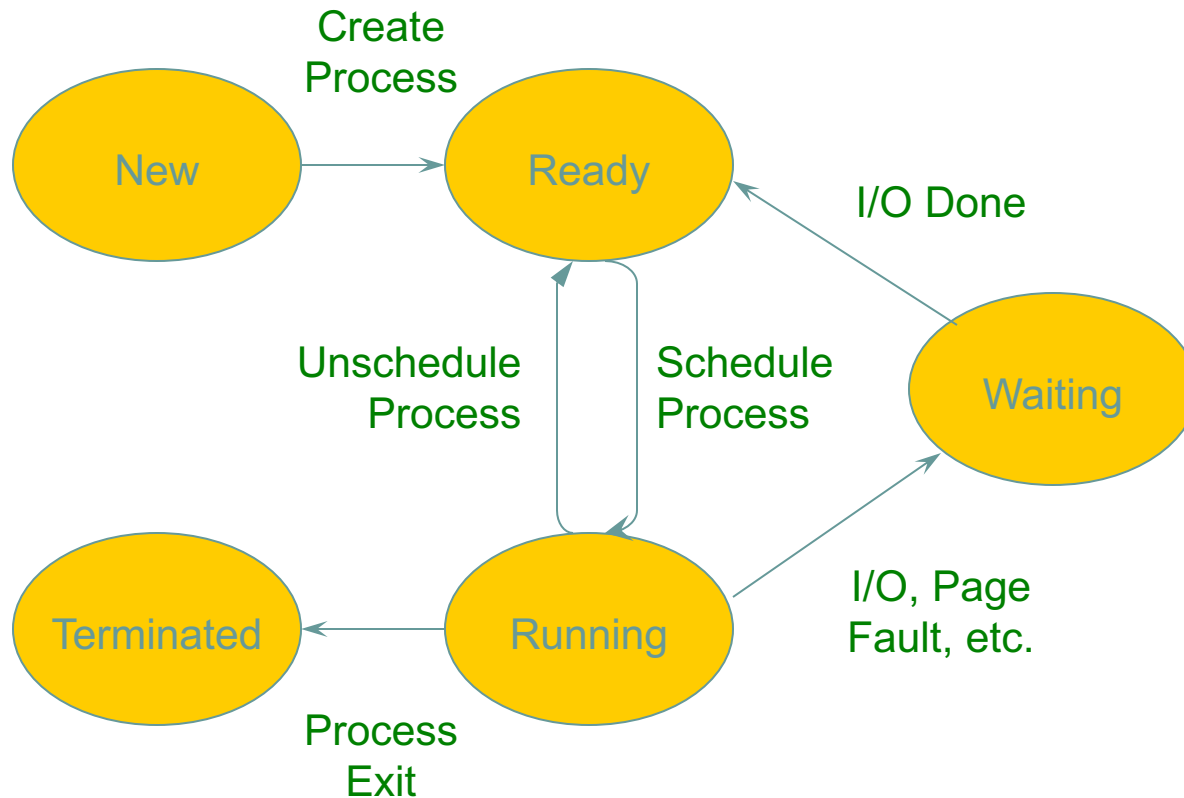
Here are the different values that the s, stat and state output specifiers (header "S

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?
 - u The OS data structure representing each process is called the **Process Control Block** (PCB)
2. How do we pause and restart processes?
 - u 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?
 - u A lot of queues!

PCB Data Structure



- › PCB also is 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
 - › Memory management
 - › Scheduling
 - › Accounting
 - › 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 struct proc

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

// Per-process state
struct proc {
    uint sz; // Size of process memory (bytes)
    pde_t* pgdir; // Linear address of proc's pgdir
    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)
};
```

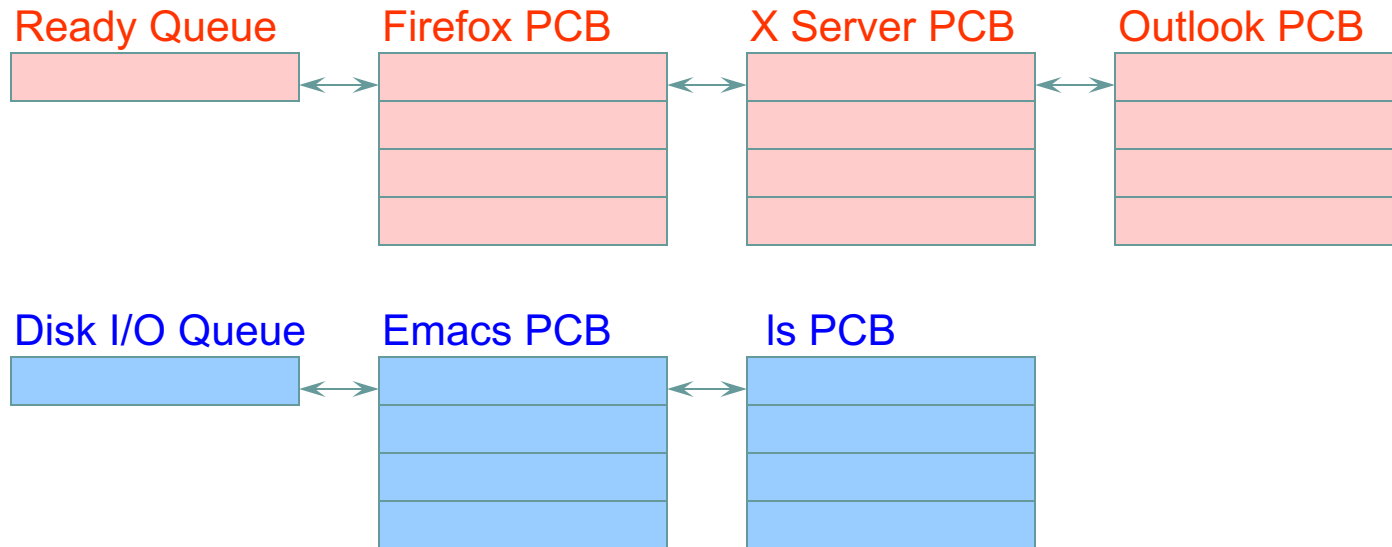
How to pause/restart processes?

- 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 be able 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 restarts executing 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!

How does the OS track processes?

- › The OS maintains a collection of queues that represent the state of all processes in the system
- › Typically, the OS at least 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.)

Advanced Operating Systems (CS 202)

Process (continued)

Process system call API



- › 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
 - › Why is this the case?
 - › Parent is creator, child is created (Unix: ps “PPID” field)
 - › 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)

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**
 - › 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

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



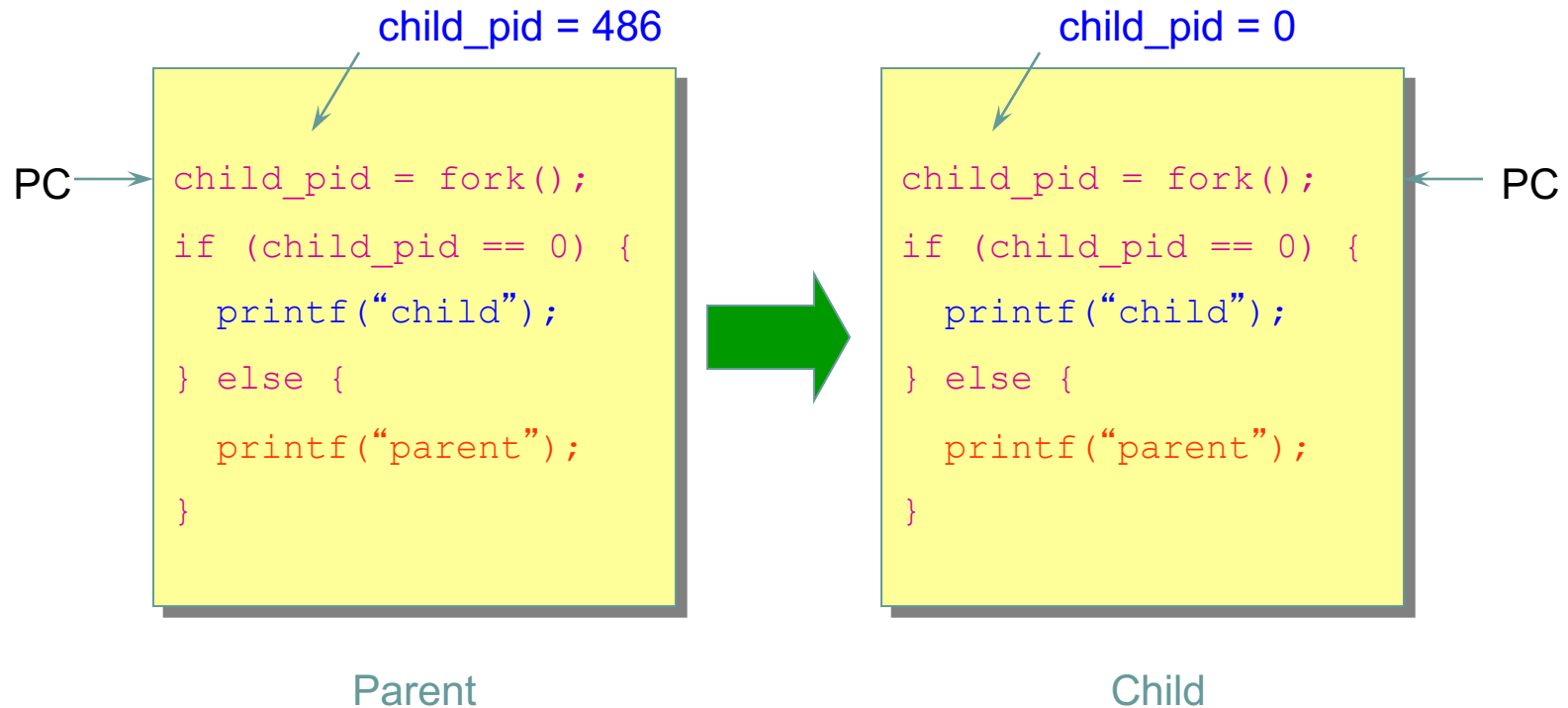
```
[well ~]$ gcc t.c
```

```
[well ~]$ ./a.out
```

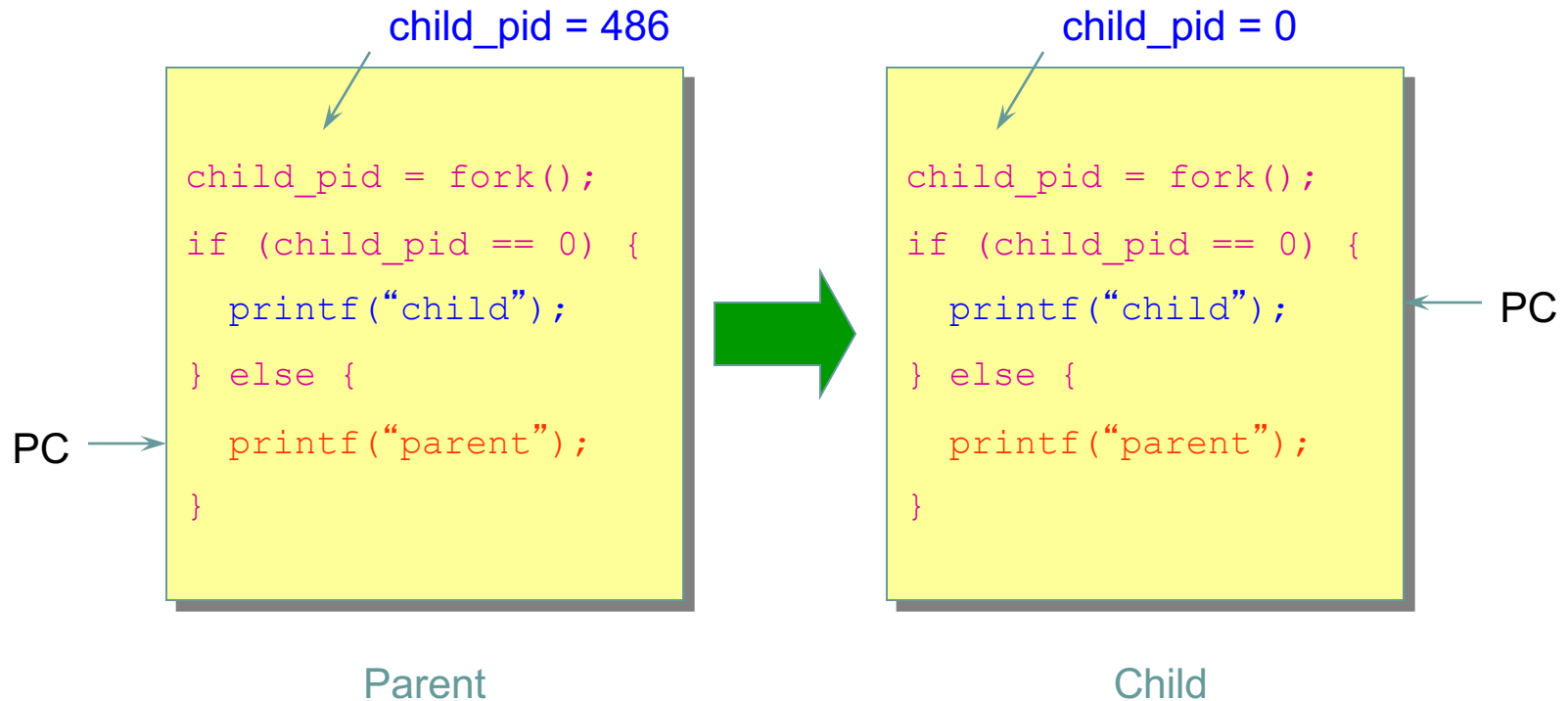
```
My child is 486
```

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


Duplicating Address Spaces



Divergence



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()?

- ▶ 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?
- › What does it mean for exec to return with an error?

Process Termination

- › All good processes must come to an end. But how?
 - › Unix: `exit(int status)`, NT: `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
 - › `waitpid()` suspends until the specified child process ends
- › **Wait has a return value...what is it?**
- › Unix: Every process must be reaped by a parent
 - › **What happens if a parent process exits before a child?**
 - › **What do you think is a “zombie” process?**

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);
    }
}
```


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**

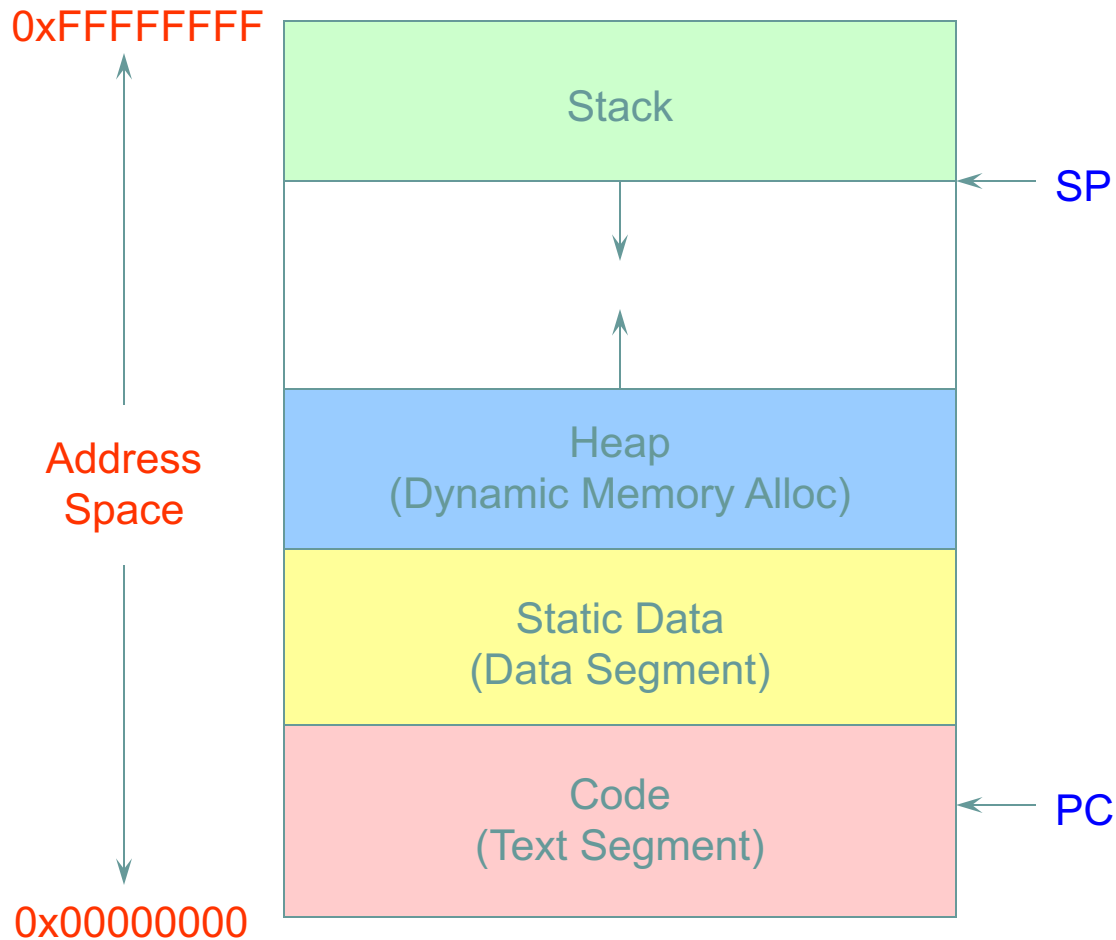
Threads



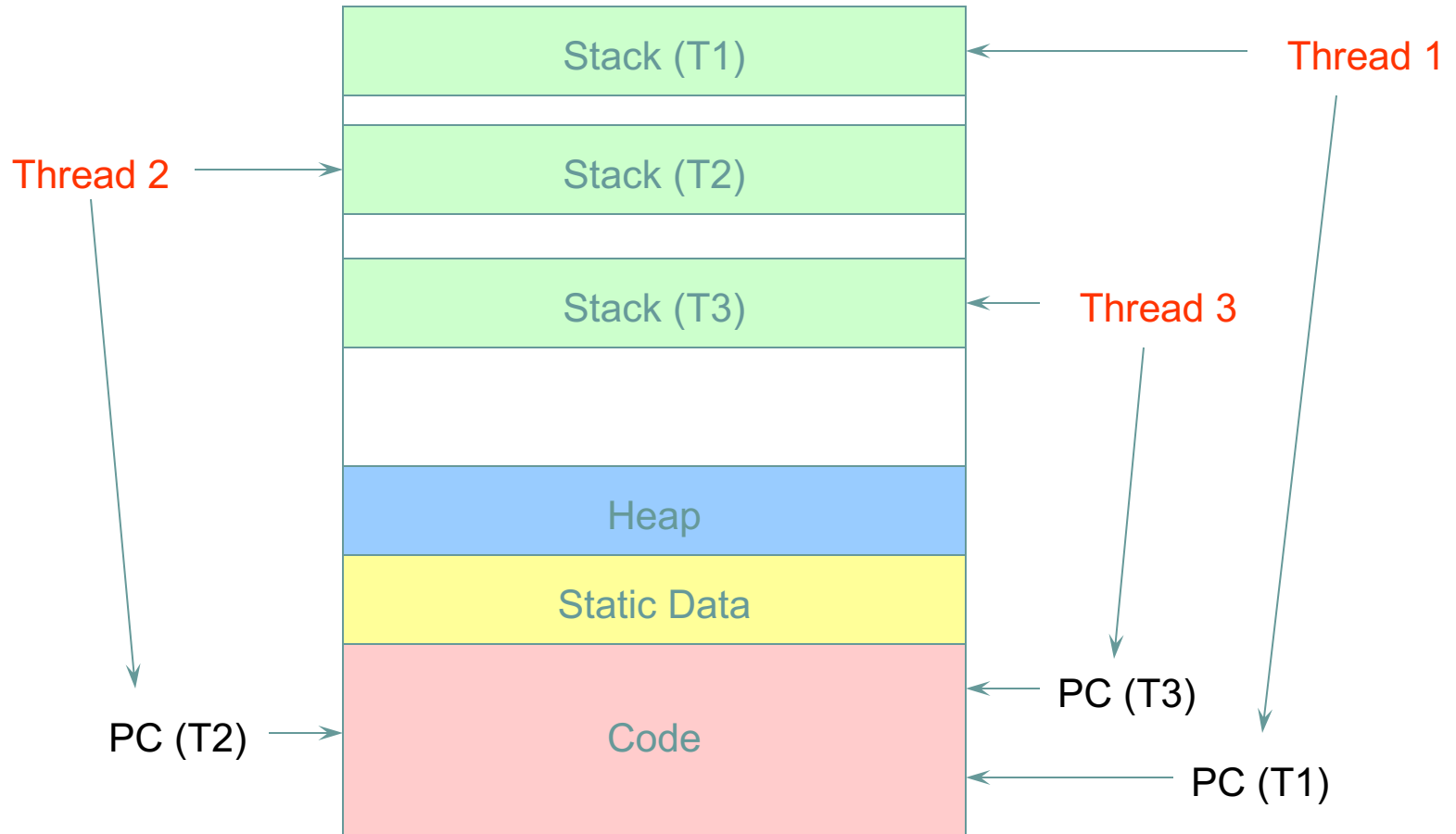
- Separate execution and resource container roles
 - u The **thread** defines a sequential execution stream within a process (PC, SP, registers)
 - u The **process** defines the address space, resources, and general process attributes (everything but threads)

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

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);  
}
```


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**

Kernel Thread (KLT) Limitations

- KLTs make concurrency cheaper than processes
 - u Much less state to allocate and initialize
- However, there are a couple of issues
 - u Issue 1: KLT overhead still high
 - › Thread operations still require system calls
 - › Ideally, want thread operations to be **as fast as a procedure call**
 - u Issue 2: KLTs are general; unaware of application needs
- Alternative: User-level threads (ULT)

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`