CIS 415: Operating Systems

Processes

Prof. Kevin Butler
Spring 2013
• Last class:
  ‣ Operating system structure

• Today:
  ‣ More basics, system calls, Process Management
• Lab sections: everyone should know where you’re going
  ‣ this week: programming with system calls and signals

• Assignment 1: due April 23

• Project 1: out today, due April 25

• Manage your time wisely!
Process Address Space

- All locations addressable by the process
- Can restrict use of addresses (RW)
- Restrictions enforced by OS
- Every running program can have its own private address space
  - How?
Virtual Memory

• Provide the illusion of infinite memory
• OS loads pages from disk as needed
  ‣ Page: Fixed sized block of data
• Many benefits
  ‣ Allows the execution of programs that may not fit entirely in memory (think MS Office)
• OS needs to maintain mapping between physical and virtual memory
  ‣ Page tables stored in memory
Translation Lookaside Buffer

- Initial virtual memory systems used to do translation in software
  - Meaning the OS did it
  - An additional memory access for each memory access!
    - S.l.o.w.!!!

- Modern CPUs contain hardware to do this: the TLB
  - Fast cache
  - Modern workloads are TLB-miss dominated
  - Good things often come in small sizes
    - We have seen other instances of this
Operating System Layers

- Timed execution service (cron)
- User application programs
- System call interface
- Network services
- UNIX kernel
- Device drivers
- Print queue
- Electronic mail
- System accessories
- Database systems
- Other utility programs

- Shells

Diagram showing the layers of an operating system, starting from the user application programs and moving inward to the UNIX kernel, with intermediate levels including system call interface, device drivers, and system accessories.
System Layers

- Application
- Libraries (in application process)
- System Services
- OS API
- Operating system kernel
- Hardware
Applications to Libraries

• Application Programming Interface
  ‣ Library functions (e.g., libc)

• Examples
  ‣ printf of stdio.h

• All within the process’s address space
  ‣ Static and Dynamic linking
Applications to Services

• Provide syntactic sugar for using resources
  ‣ Printing, program mgmt, network mgmt, file mgmt, etc.
  ‣ E.g., chmod

• Provide special functions beyond OS
  ‣ E.g., cron

• UNIX man pages, sections 1 and 8
Libraries to System

• System call interface
  ‣ UNIX man pages, section 2
  ‣ Examples
    • open, read, write – defined in unistd.h
  ‣ Call these via libraries? fopen vs. open

• Special files
  ‣ Drivers, /proc, sysfs
System to Hardware

- **Software-hardware** interface
- OS kernel functions
  - Concepts == Managers -- Hardware
  - Files == filesystems – drivers/devices
  - Address space == virtual memory -- memory
  - Instruction Set == process model -- CPU
- OS provides abstractions of devices and hardware objects (files)
System Calls: Overview

User Space

User App

getpid(void)

return

C Library

Kernel Space

Kernel

load args, eax=NR_getpid, transition to kernel (int 0x80)

system call

call system_call_table[eax]

syscall_exit

return

Syscall

return_userspace

return
Figure 3-7
System service exceptions
System Call Handling

Procedure call in user process
Initial work in user mode
Trap instruction to invoke kernel
Preparation
I/O command
Wait
Completion
Return-from-interrupt instruction
Final work in user mode
Ordinary return instruction

(libc)
(int 0x80)
(e.g., sys_read, mmap2)
(read from disk)
(disk is slow)
(interrupt handling)

(libc)
A more accurate picture:

- consider a typical Linux process
- its thread of execution can be several places
  - in your program’s code
  - in **glibc**, a shared library containing the C standard library, POSIX support, and more
  - in the Linux architecture-independent code
  - in Linux x86-32/x86-64 code
• Some routines your program invokes may be entirely handled by glibc
  ‣ without involving the kernel
    • e.g., `strcmp()` from stdio.h
  ‣ some initial overhead when invoking functions in dynamically linked libraries
  ‣ but, after symbols are resolved, invoking glibc routines is nearly as fast as a function call within your program itself
Details on x86 / Linux

- Some routines may be handled by glibc, but they in turn invoke Linux system calls
  - e.g., POSIX wrappers around Linux syscalls
    - POSIX `readdir()` invokes the underlying Linux `readdir()`
  - e.g., C stdio functions that read and write from files
    - `fopen()`, `fclose()`, `fprintf()` invoke underlying Linux `open()`, `read()`, `write()`, `close()`, etc.

![Diagram showing the relationship between your program, glibc, C standard library, and POSIX](image)
• Your program can choose to directly invoke Linux system calls as well
  ‣ nothing forces you to link with glibc and use it
  ‣ but, relying on directly invoked Linux system calls may make your program less portable across UNIX varieties
File Interface

• Goal: Provide a uniform abstraction for accessing the OS and its resources

• Abstraction: File
  ‣ Use file system calls to access OS services
  ‣ Devices, sockets, pipes, etc.
  ‣ And OS in general
I/O with System Calls

• Much I/O is based on a streaming model
  ‣ sequence of bytes

• write() sends a stream of bytes somewhere

• read() blocks until a stream of input is ready

• Annoying details:
  ‣ might fail, can block for a while
  ‣ file descriptors...
  ‣ arguments are pointers to character buffers
  ‣ see the read() and write() man pages
File Descriptors

• A process might have several different I/O streams in use at any given time

• These are specified by a kernel data structure called a file descriptor
  ‣ each process has its own table of file descriptors

• open() associates a file descriptor with a file

• close() destroys a file descriptor

• Standard input and standard output are usually associated with a terminal
  ‣ more on that later
Regular File

• File has a pathname: /tmp/foo

• Can open the file
  ‣ int fd = open("/tmp/foo", O_RDWR )
  ‣ For reading and writing

• Can read from and write to the file
  ‣ bytes = read( fd, buf, max ); /* buf get output */
  ‣ bytes = write( fd, buf, len ); /* buf has input */
Socket File

• File has a pathname: /tmp/bar
  ‣ Files provide a persistence for a communication channel
  ‣ Usually used for local communication (UNIX domain sockets)

• Open, read, and write via socket operations
  ‣ sockfd = socket(AF_UNIX, TCP_STREAM, 0);
  ‣ local.path is set to /tmp/bar
  ‣ bind ( sockfd, &local, len )
  ‣ Use sock operations to read and write
Device File

- Files for interacting with physical devices
  - /dev/null (do nothing)
  - /dev/cdrom (CD-drive)
- Use file system operations, but are handled in device-specific ways
  - open, read, write correspond to device-specific functions
    - Function pointers!
  - Also, use ioctl (I/O control) to interact (later)
Sysfs File and `/proc` Files

- These files enable reading from and writing to kernel
- `/proc` files
  - enable reading of kernel state for a process
- Sysfs files
  - Provide functions that update kernel data
    - File’s `write` function updates kernel based on input data
Other System Calls

• It’s possible to hook the output of one program into the input of another: `pipe()`

• It’s possible to block until one of several file descriptor streams is ready: `select()`

• Special calls for dealing with network
  ‣ `AF_INET` sockets, etc.

• Send a message to other (or all) processes: `signal()`

• Most of these in section 2 of manual
  ‣ e.g., `man 2 select`
SySCALL FUNCTIONALITY

• System calls are the main interface between processes and the OS
  ‣ like an extended “instruction set” for user programs that hide many details
  ‣ first Unix system had a couple dozen system calls
  ‣ current systems have many more (>300 in Linux, >500 in FreeBSD)
    ‣ Understanding the system call interface of a given OS lets you write useful programs under it

• Natural questions to ask:
  ‣ is this the right interface? how to evaluate?
  ‣ how can these system calls be implemented?
• We have programs, so why do we need processes?
Overview

- Questions that we explore
  - How are processes created?
    - From binary program to process
  - How is a process represented and managed?
    - Process creation, process control block
  - How does the OS manage multiple processes?
    - Process state, ownership, scheduling
  - How can processes communicate?
    - Interprocess communication, concurrency, deadlock
Supervisor and User Modes

• OS runs in supervisor mode
  ‣ Has access to protected instructions only available in that mode (ring 0)
  ‣ Can manage the entire system

• OS loads processes into user mode
  ‣ Many processes can run in user mode

• How does OS get programs loaded into processes in user mode and keep them straight?
Process

- Address space + threads + resources
- Address space contains code and data of a process
- Threads are individual execution contexts
- Resources are physical support necessary to run the process (memory, disk, …)
Process Address Space

- Program (Text)
- Global Data (Data)
- Dynamic Data (Heap)
- Thread-local Data (Stack)
- Each thread has its own stack
Process Address Space

```c
int value = 5;        // Global

int main()
{
    int *p;            // Stack

    p = (int *)malloc(sizeof(int));      // Heap

    if (p == 0) {
        printf("ERROR: Out of memory\n");
        return 1;
    }

    *p = value;
    printf("%d\n", *p);
    free(p);
    return 0;
}
```
Heap + stack

```c
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(
        size * sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(
        size * sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
Heap + stack

```
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(size * sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```
Heap + stack

```c
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(
               size * sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(
        size * sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(
        size * sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
Heap + stack

```c
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;
    
    a2 = malloc(
        size * sizeof(int));
    if (a2 == NULL)
        return NULL;
    
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```
Heap + stack

```c
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(
              size * sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```
Heap + stack

```c
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(size * sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```
Heap + stack

```
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(  
        size * sizeof(int));
    if (a2 == NULL)  
        return NULL;

    for (i = 0; i < size; i++)  
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```
Heap + stack

```c
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(
        size * sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```
Heap + stack

```c
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size * sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
```
#include <stdlib.h>

int *copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(
        size * sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(...) {
    int nums[4] = {2, 4, 6, 8};
    int *ncopy = copy(nums, 4);
    // ... do stuff ...
    free(ncopy);
    return 0;
}
Process Creation

• Parent process create children processes,
  ‣ which, in turn create other processes, forming a tree of processes

• Resource sharing options
  ‣ Parent and children share all resources
  ‣ Children share subset of parent’s resources
  ‣ Parent and child share no resources

• Execution options
  ‣ Parent and children execute concurrently
  ‣ Parent waits until children terminate
Process Creation

• Address space
  ‣ Child duplicate of parent
  ‣ Child has a program loaded into it

• UNIX examples
  ‣ `fork` system call creates new process
  ‣ `exec` system call used after a fork to replace the process’s memory space with a new program
Process Creation

• What happens?
  ‣ New process object in kernel
    • Build process data structures
  ‣ Allocate address space (abstract resource)
    • Later, allocate memory (physical resource)
  ‣ Add to execution queue
    • Runnable?
Process Creation
1. PCB with new id created
2. Memory allocated for child
   Initialized by copying over from the parent
3. If parent had called `wait`,
   it is moved to a waiting queue
4. If child had called `exec`,
   its memory overwritten with new code & data
5. Child added to ready queue,
   all set to go now!
```c
int main( )
{
    pid_t pid;
    /* fork another process */
    pid = fork( );
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    } else if (pid == 0) { /* child process */
        execvp("/bin/ls", "ls", NULL);
    } else { /* parent process */
        /* parent will wait for the child to complete */
        wait (NULL);
        printf ("Child Complete");
        exit(0);
    }
}
```
Graphically
Graphically
Graphically
Graphically

client

server

fork() child

server
Graphically

client → server
server → server
fork() → grandchild
Graphically

child exit( )’s / parent wait( )’s
Graphically

Client → Server

Parent closes its client connection
Graphically
Graphically

```
client  ── server
     |           ┌─ fork() child ── server
     |                      │            ┌─ fork() grandchild ── server
client  ── server
```

```
exit()
```
Graphically
Graphically

client -> server

server

client -> server

client -> server

client -> server

client -> server

client -> server
Program Creation

• Design Choices
  ‣ Resource Sharing
    • What resources of parent should the child share?
    • What about after `exec`?
  ‣ Execution
    • Should parent wait for child?
  ‣ What is the relationship between parent and child?
    • Hierarchical or grouped or …?
Program Creation

• `fork` -- copy address space and all threads
• `forkl` -- copy address space and only calling thread
• `vfork` -- do not copy address space; shared between parent and child
• `exec` -- load new program; replace address space
  ‣ Some resources may be transferred (open file descriptors)
  ‣ Specified by arguments
A tree of processes on a typical system
Process Termination

• Process executes last statement and asks the operating system to delete it (exit)
  ‣ Output data from child to parent (via wait)
  ‣ Process’ resources are deallocated by operating system

• Parent may terminate execution of children processes (abort)
  ‣ Child has exceeded allocated resources
  ‣ Task assigned to child is no longer required
  ‣ If parent is exiting
    • Some operating system do not allow child to continue if parent terminates
    • All children terminated - cascading termination
Executing a Process

• What to execute?
  ‣ Program status word
  ‣ Register that stores the program counter
    • Next instruction to be executed

• Registers store state of execution in CPU
  ‣ Stack pointer
  ‣ Data registers

• Thread of execution
  ‣ Has its own stack
Executing a Process

• Thread executes over the process’s address space
  ‣ Usually the text segment

• Until a trap or interrupt...
  ‣ Time slice expires (timer interrupt)
  ‣ Another event (e.g., interrupt from other device)
  ‣ Exception (oops)
  ‣ **System call** (switch to kernel mode)
Details on x86 / Linux

• Let’s walk through how a Linux system call actually works
  ‣ we’ll assume 32-bit x86 using the modern SYSENTER / SYSEXIT x86 instructions
Details on x86 / Linux

• Remember our process address space picture
  ‣ let’s add some details

0xFFFFFFFF

0x00000000

linux-gate.so

Linux kernel

kernel stack

stack

shared libraries

heap (malloc/free)

read/write segment
.data, .bss

read-only segment
.text, .rodata

your program

C standard library

POSIX

glibc

architecture-independent code

architecture-dependent code

Linux kernel

CPU
Details on x86 / Linux

- Your program
- glibc
- Linux kernel
- C standard library
- POSIX

Process is executing your program code

- Stack
- Shared libraries
- Heap (malloc/free)
- Read/write segment .data, .bss
- Read-only segment .text, .rodata

Architecture-independent code

Architecture-dependent code

CPU

Unpriv

0x00000000

0xFFFFFFFF

Linux-gate.so

Kernel stack
glibc begins the process of invoking a Linux system call

- glibc’s `fopen()` likely invokes Linux’s `open()` system call
- puts the system call # and arguments into registers
- uses the `call x86` instruction to call into the routine `__kernel_vsyscall` located in `linux-gate.so`
Details on x86 / Linux

- **linux-gate.so** is a **vdso**
  - a virtual dynamically linked shared object
  - is a kernel-provided shared library, i.e., is not associated with a .so file, but rather is conjured up by the kernel and plunked into a process’s address space
  - provides the intricate machine code needed to trigger a system call

- Your program
  - C standard library
  - POSIX
  - glibc

- Linux kernel
  - architecture-independent code
  - architecture-dependent code

- IP

- SP

- 0xFFFFF000

- 0x00000000

- shared libraries
- heap (malloc/free)
- read/write segment .data, .bss
- read-only segment .text, .rodata

- kernel stack
Details on x86 / Linux

- SYSENTER is x86’s “fast system call” instruction
- it has several side-effects
  - causes the CPU to raise its privilege level
  - traps into the Linux kernel by changing the SP, IP to a previously determined location
  - changes some segmentation related registers

linux-gate.so eventually invokes the SYSENTER x86 instruction
The kernel begins executing code at the SYSENTER entry point:

- is in the architecture-dependent part of Linux
- its job is to:
  - look up the system call number in a system call dispatch table
  - call into the address stored in that table entry; this is Linux’s system call handler
  - for open(), the handler is named sys_open, and is system call #5
Details on x86 / Linux

The system call handler executes

- what it does is system-call specific, of course
- it may take a long time to execute, especially if it has to interact with hardware
  - Linux may choose to context switch the CPU to a different runnable process

- The Linux kernel
  - stack
  - shared libraries
  - heap (malloc/free)
  - read/write segment .data, .bss
  - read-only segment .text, .rodata

- glibc
  - C standard library
  - POSIX

- Linux kernel
  - architecture-independent code
  - architecture-dependent code

- CPU
  - priv
Eventually, the system call handler finishes

- returns back to the system call entry point
  - places the system call’s return value in the appropriate register
  - calls SYSEXIT to return to the user-level code
SYSEXIT transitions the processor back to user-mode code

- has several side-effects
  - restores the IP, SP to user-land values
  - sets the CPU back to unprivileged mode
  - changes some segmentation related registers
- returns the processor back to glibc
Details on x86 / Linux

glibc continues to execute

- might execute more system calls
- eventually returns back to your program code

Details on x86 / Linux

- glibc
- architecture-dependent code
- architecture-independent code
- Linux kernel
- stack
- shared libraries
- heap (malloc/free)
- read/write segment .data, .bss
- read-only segment .text, .rodata
- 0x00000000
- 0xFFFFFFFF
- your program
- C standard library
- POSIX
- linux-gate.so
- SP
- IP
- unpriv
- CPU
Relocatable Memory

• Mechanism that enables the OS to place a program in an arbitrary location in memory
  ‣ Gives the programmer the impression that they own the processor

• Program is loaded into memory at program-specific locations
  ‣ Need virtual memory to do this

• Also, may need to share program code across processes
Process State

• What do we need to track about a process?
  ‣ how many processes?
  ‣ what’s the state of each of them?

• Process table: kernel data structure tracking processes on system

• Process control block: structure for tracking process context
Scheduling Processes

- Processes transition among **execution states**

```
new  admitted  interrupt  exit  terminated

ready  scheduler dispatch

waiting

I/O or event completion

I/O or event wait
```
Process States

- **Running**
  - Running == in processor and in memory with all resources
- **Ready**
  - Ready == in memory with all resources, waiting for dispatch
- **Waiting**
  - Waiting == waiting for some event to occur
State Transitions

• New Process ==> Ready
  ‣ Allocate resources
  ‣ End of process queue

• Ready ==> Running
  ‣ Head of process queue
  ‣ Scheduled

• Running ==> Ready
  ‣ Interrupt (Timer)
  ‣ Back to end of process queue
State Transitions: Page Fault Handling

• Running ==> Waiting
  ‣ Page fault exception (similar for syscall or I/O interrupt)
  ‣ Wait for event

• Waiting ==> Ready
  ‣ Event has occurred (page fault serviced)
  ‣ End of process queue (or head?)

• Ready ==> Running
  ‣ As before…
State Transitions: Other Issues

• Priorities
  ‣ Can provide policy indicating which process should run next
    • More when we discuss scheduling…

• Yield
  ‣ System call to give up processor
  ‣ For a specific amount of time (sleep)

• Exit
  ‣ Terminating signal (Ctrl-C)
Process Control Block

- State of running process
- Linked list of process control information
Per Process Control Info

• Process state
  ‣ Ready, running, waiting (momentarily)

• Links to other processes
  ‣ Children

• Memory Management
  ‣ Segments and page tables

• Resources
  ‣ Open files

• And Much More…
/proc File System

- Linux and Solaris
  - `ls /proc`
  - A directory for each process
- Various process information
  - `/proc/<pid>/io` -- I/O statistics
  - `/proc/<pid>/environ` -- Environment variables (in binary)
  - `/proc/<pid>/stat` -- process status and info
Context Switch

• OS switches from one execution context to another
  ‣ One process to another process
  ‣ Interrupt handling
  ‣ Process to kernel (mode transition, not context switch)

• Current Process to New Process
  ‣ Save the state of the current process
    • Process control block: describes the state of the process in the CPU
  ‣ Load the saved context for the new process
    • Load the new process’s process control block into OS and registers
  ‣ Start the new process

• Does this differ if we are running an interrupt handler?
Context Switch

```
<table>
<thead>
<tr>
<th>process $P_0$</th>
<th>operating system</th>
<th>process $P_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>executing</td>
<td>interrupt or system call</td>
<td>idle</td>
</tr>
<tr>
<td></td>
<td>save state into PCB$_0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>reload state from PCB$_1$</td>
<td>executing</td>
</tr>
<tr>
<td></td>
<td>interrupt or system call</td>
<td>idle</td>
</tr>
<tr>
<td></td>
<td>save state into PCB$_1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>reload state from PCB$_0$</td>
<td>idle</td>
</tr>
<tr>
<td>executing</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Context Switch

- No useful work is being done during a context switch
  - Speed it up and limit system calls to things that can’t be done in user mode

- Hardware support
  - Multiple register sets (Sun UltraSPARC)

- However, hardware optimization may conflict
  - TLB flush is necessary
  - Different virtual to physical mappings on different processes
Next class

• IPC