

CS162

Operating Systems and Systems Programming

Lecture 3

Processes (con't), Fork, System Calls

January 28th, 2020
 Prof. John Kubitowicz
<http://cs162.eecs.Berkeley.edu>

Recall: Four Fundamental OS Concepts

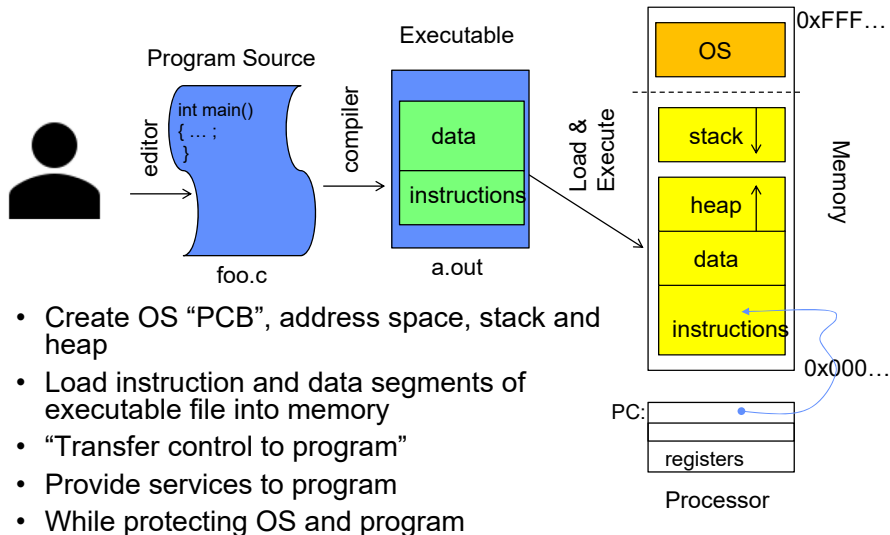
- **Thread: Execution Context**
 - Fully describes program state
 - Program Counter, Registers, Execution Flags, Stack
- **Address space (with or w/o translation)**
 - Set of memory addresses accessible to program (for read or write)
 - May be distinct from memory space of the physical machine (in which case programs operate in a virtual address space)
- **Process: an instance of a running program**
 - Protected Address Space + One or more Threads
- **Dual mode operation / Protection**
 - Only the “system” has the ability to access certain resources
 - Combined with translation, isolates programs from each other and the OS from programs

1/28/20

Kubitowicz CS162 ©UCB Spring 2020

Lec 3.2

Recall: OS Bottom Line: Run Programs



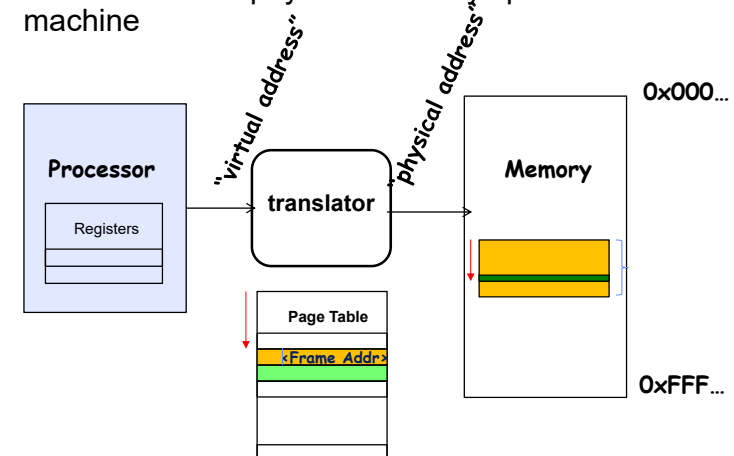
1/28/20

Kubitowicz CS162 ©UCB Spring 2020

Lec 3.3

Recall: Protected Address Space

- Program operates in an address space that is distinct from the physical memory space of the machine

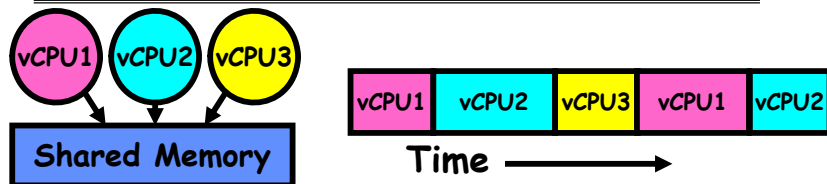


1/28/20

Kubitowicz CS162 ©UCB Spring 2020

Lec 3.4

Recall: give the illusion of multiple processors?



- Assume a single processor. How do we provide the illusion of multiple processors?
 - Multiplex in time!
 - Multiple "virtual CPUs"
- Each virtual "CPU" needs a structure to hold:
 - Program Counter (PC), Stack Pointer (SP)
 - Registers (Integer, Floating point, others...?)
- How switch from one virtual CPU to the next?
 - Save PC, SP, and registers in current state block
 - Load PC, SP, and registers from new state block
- What triggers switch?
 - Timer, voluntary yield, I/O, other things

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.5

Recall: The Process

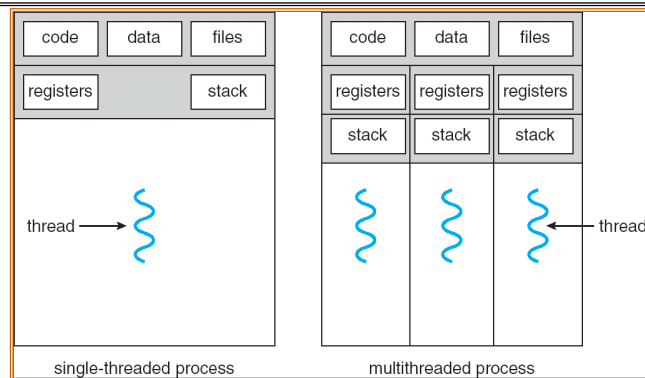
- Definition: **execution environment with restricted rights**
 - Address Space with One or More Threads
 - » *Page table per process!*
 - Owns memory (mapped pages)
 - Owns file descriptors, file system context, ...
 - Encapsulates one or more threads sharing process resources
- Application program executes as a process
 - Complex applications can fork/exec child processes [later]
- Why processes?
 - Protected from each other. OS Protected from them.
 - Execute concurrently [trade-offs with threads? later]
 - Basic unit OS deals with

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.6

Recall: Single and Multithreaded Processes



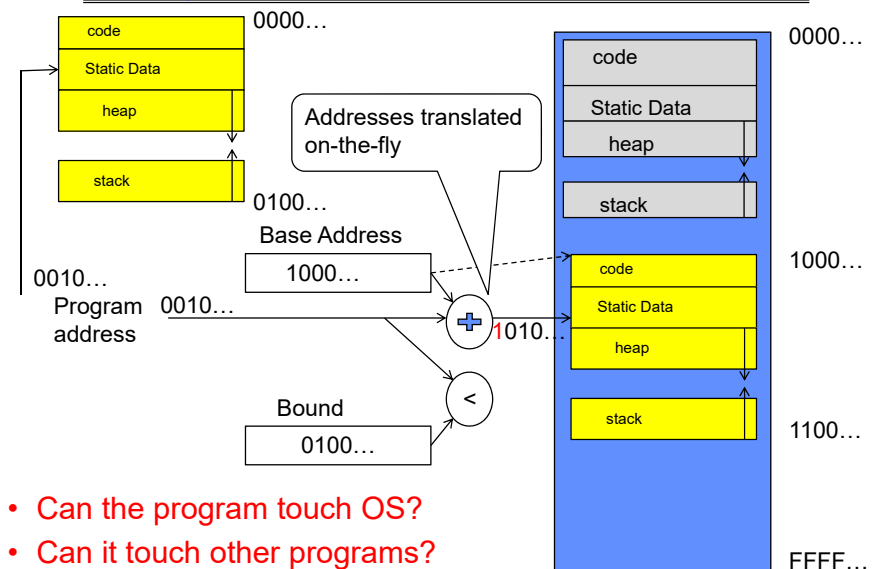
- Threads encapsulate concurrency: "Active" component
- Address spaces encapsulate protection: "Passive" part
 - Keeps buggy program from trashing the system
- Why have multiple threads per address space?

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.7

Recall: Simple address translation with Base and Bound



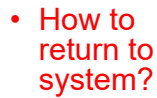
- Can the program touch OS?
- Can it touch other programs?

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.8

Simple B&B: User => Kernel

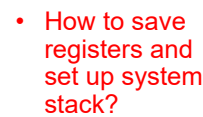


1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.9

Simple B&B: Interrupt

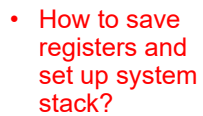


1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.10

Simple B&B: Switch User Process

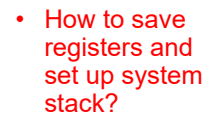


1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.11

Simple B&B: “resume”



1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.12

Is Branch and Bound a Good-Enough Protection Mechanism?

- **NO: Too simplistic for real systems**
- Inflexible/Wasteful:
 - Must dedicate physical memory for *potential* future use
 - (Think stack and heap!)
- Fragmentation:
 - Kernel has to somehow fit whole processes into contiguous block of memory
 - After a while, memory becomes fragmented!
- Sharing:
 - Very hard to share any data between Processes or between Process and Kernel
 - Need to communicate indirectly through the kernel...

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

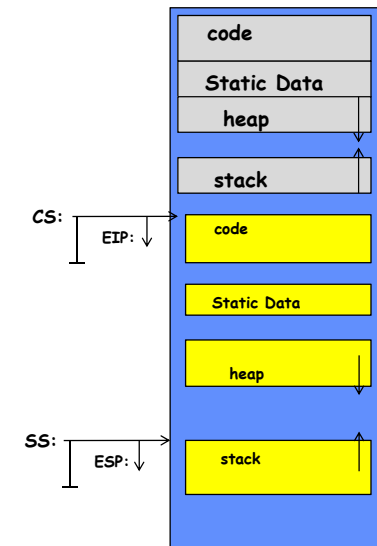
Lec 3.13

Better: x86 – segments and stacks

Processor Registers

CS	EIP
SS	ESP
DS	EAX
ES	EBX
	ECX
	EDX
	ESI
	EDI

Start address, length and access rights associated with each segment

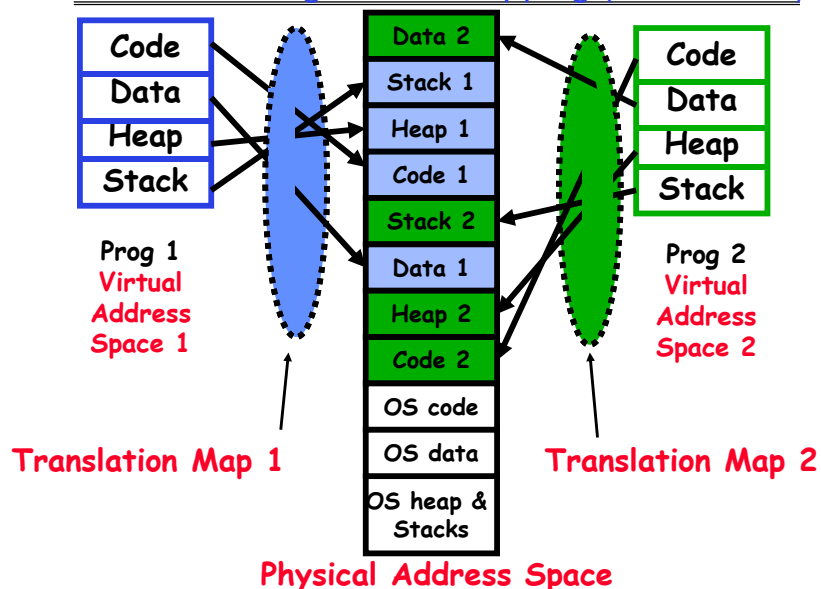


1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.14

Alternative: Page Table Mapping (More soon!)

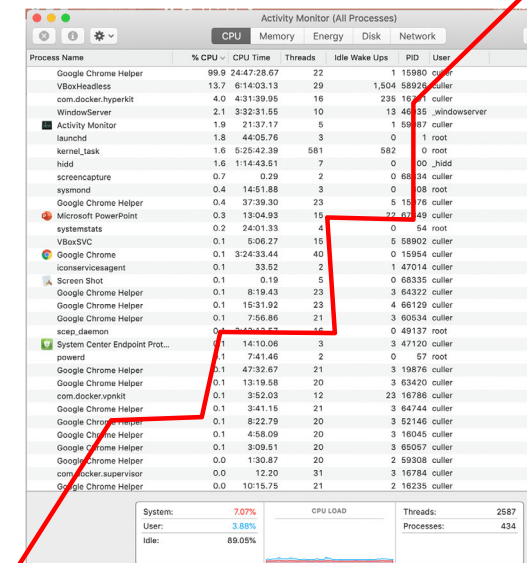


1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.15

What's beneath the Illusion?



1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.16

Today: How does the Operating System create the Process Abstraction?

- What data structures are used?
- What machine structures are employed?
 - Focus on x86, since will use in projects (and everywhere)

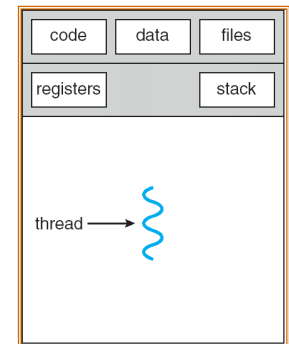
1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.17

Starting Point: Single Threaded Process

- Process: OS abstraction of what is needed to run a single program
 1. Sequential program execution stream
 - » Sequential stream of execution (thread)
 - » State of CPU registers
 2. Protected resources
 - » Contents of Address Space
 - » I/O state (more on this later)



1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.18

Running Many Programs

- We have the basic mechanism to
 - switch between user processes and the kernel,
 - the kernel can switch among user processes,
 - Protect OS from user processes and processes from each other
- Questions ???
 - How do we represent each process in the kernel?
 - How do we decide which user process to run?
 - How do we pack up the process and set it aside?
 - How do we get a stack and heap for the kernel?
 - Aren't we wasting a lot of memory?

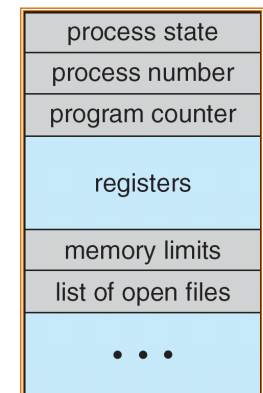
1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.19

Multiplexing Processes: The Process Control Block

- Kernel represents each process as a process control block (PCB)
 - Status (running, ready, blocked, ...)
 - Register state (when not ready)
 - Process ID (PID), User, Executable, Priority, ...
 - Execution time, ...
 - Memory space, translation, ...
- Kernel *Scheduler* maintains a data structure containing the PCBs
 - Give out CPU to different processes
 - This is a Policy Decision
- Give out non-CPU resources
 - Memory/I/O
 - Another policy decision



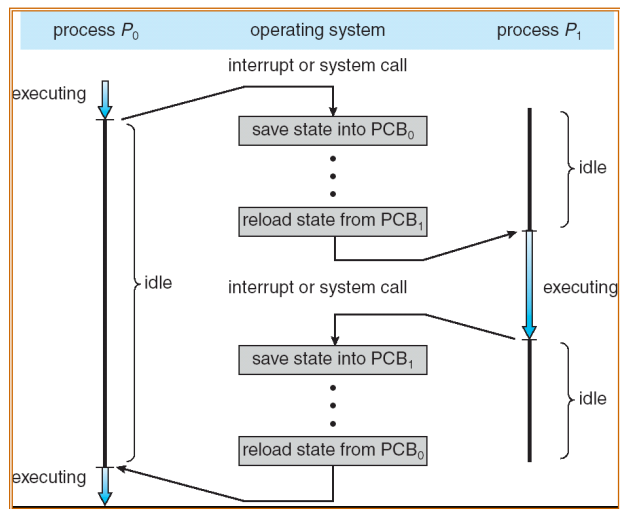
Process
Control
Block

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.20

Context Switch

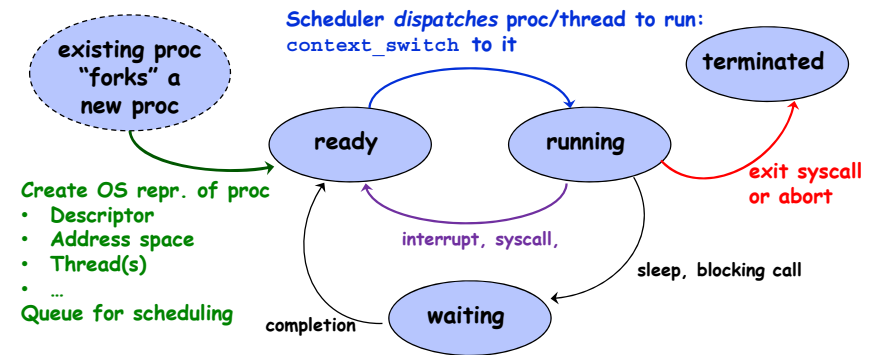


1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.21

Lifecycle of a process / thread



- OS juggles many process/threads using kernel data structures
- Proc's may create other process (fork/exec)
 - All starts with init process at boot

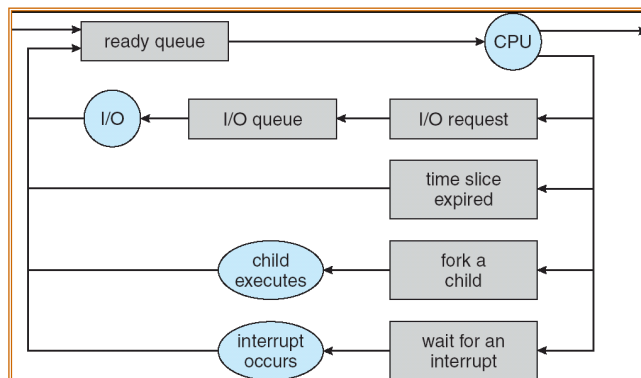
Pintos: process.c

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.22

Scheduling: All About Queues



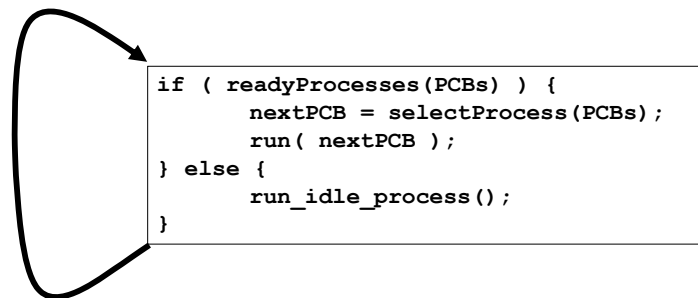
- PCBs move from queue to queue
- Scheduling:** which order to remove from queue
 - Much more on this soon

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.23

Scheduler



- Scheduling: Mechanism for deciding which processes/threads receive the CPU
- Lots of different scheduling policies provide ...
 - Fairness or
 - Realtime guarantees or
 - Latency optimization or ..

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.24

Simultaneous MultiThreading/Hyperthreading

- Hardware scheduling technique

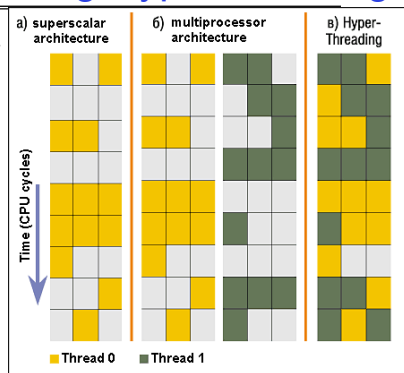
- Superscalar processors can execute multiple instructions that are independent.
- Hyperthreading duplicates register state to make a second "thread," allowing more instructions to run.

- Can schedule each thread as if were separate CPU

- But, sub-linear speedup!

- Original technique called "Simultaneous Multithreading"

- <http://www.cs.washington.edu/research/smt/index.html>
- SPARC, Pentium 4/Xeon ("Hyperthreading"), Power 5



Colored blocks show instructions executed

Also Recall: The World Is Parallel

- Intel Skylake (2017)

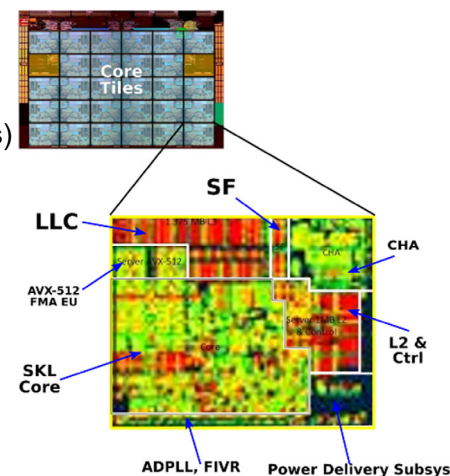
- 28 Cores
- Each core has two hyperthreads!
- So: 54 Program Counters(PCs)

- Scheduling here means:

- Pick which core
- Pick which thread

- Space of possible scheduling much more interesting

- Can afford to dedicate certain cores to housekeeping tasks
- Or, can devote cores to services (e.g. Filesystem)



Administrivia: Getting started

- Kubiatowicz Office Hours:

- 1-2pm, Monday & Thursday

- Homework 0 **Due Friday!**

- Get familiar with the cs162 tools
- configure your VM, submit via git
- Practice finding out information:
 - » How to use GDB? How to understand output of unix tools?
 - » We don't assume that you already know everything!
 - » Learn to use "man" (command line), "help" (in gdb, etc), google

- **Should be going to sections now – Important information there**

- Any section will do until groups assigned

- Class status: All regular students made it!

- Concurrent enrollment will be added as possible. Over half will be admitted already. Perhaps more.

- **THIS Friday is Drop Deadline! HARD TO DROP LATER!**

- If you know you are going to drop, please do so to leave room for others!

Administrivia (Con't)

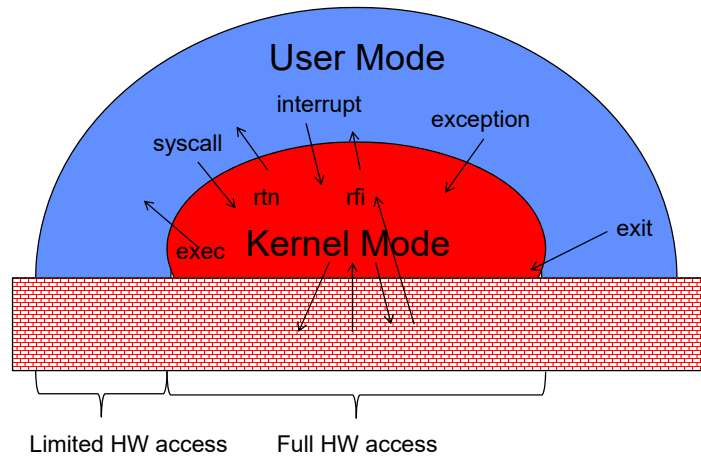
- Group sign up via autograder form next week

- Get finding groups of 4 people ASAP
- Priority for same section; if cannot make this work, keep same TA
- Remember: Your TA needs to see you in section!

- Midterm 1 conflicts

- We will handle these conflicts after have final class roster
- I know about one problem with Midterm 1 scheduling, and it can be dealt with. Have I missed any others?
- Watch for queries by HeadTA to collect information

Recall: User/Kernel (Privileged) Mode



1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.29

Three types of Kernel Mode Transfer

- Syscall
 - Process requests a system service, e.g., exit
 - Like a function call, but “outside” the process
 - Does not have the address of the system function to call
 - Like a Remote Procedure Call (RPC) – for later
 - Marshall the syscall id and args in registers and exec syscall
- Interrupt
 - External asynchronous event triggers context switch
 - eg. Timer, I/O device
 - Independent of user process
- Trap or Exception
 - Internal synchronous event in process triggers context switch
 - e.g., Protection violation (segmentation fault), Divide by zero, ...

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.30

Implementing Safe Kernel Mode Transfers

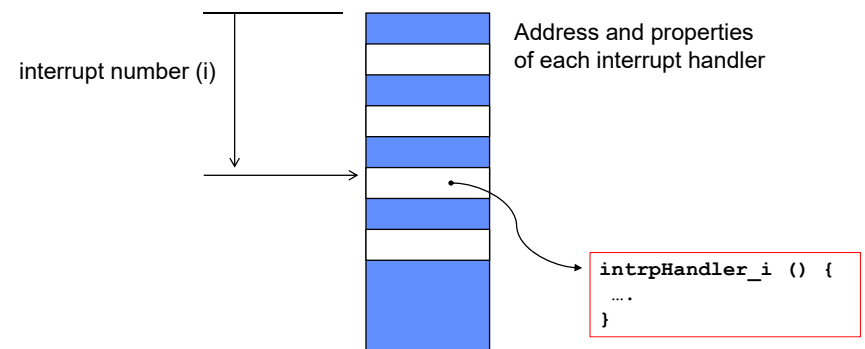
- Important aspects:
 - Controlled transfer into kernel (e.g., syscall table)
 - Separate kernel stack
- Carefully constructed kernel code packs up the user process state and sets it aside
 - Details depend on the machine architecture
- Should be impossible for buggy or malicious user program to cause the kernel to corrupt itself

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.31

Interrupt Vector



- Where else do you see this dispatch pattern?

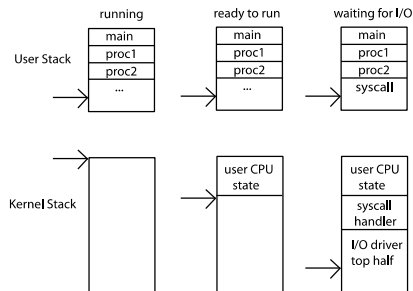
1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.32

Need for Separate Kernel Stacks

- Kernel needs space to work
- Cannot put anything on the user stack (Why?)
- Two-stack model
 - OS thread has interrupt stack (located in kernel memory) plus User stack (located in user memory)
 - Syscall handler copies user args to kernel space before invoking specific function (e.g., open)
 - Interrupts (???)

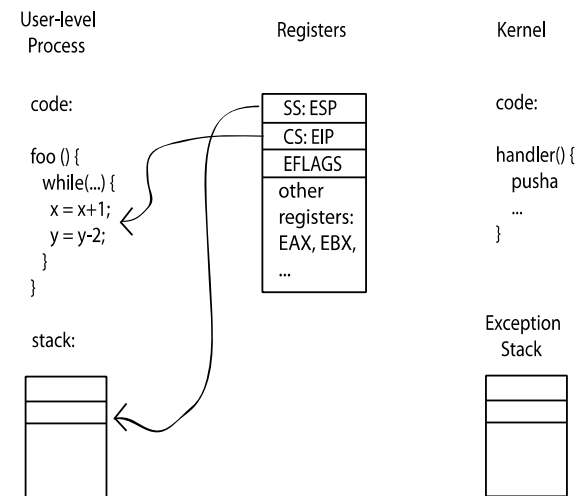


1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.33

Before

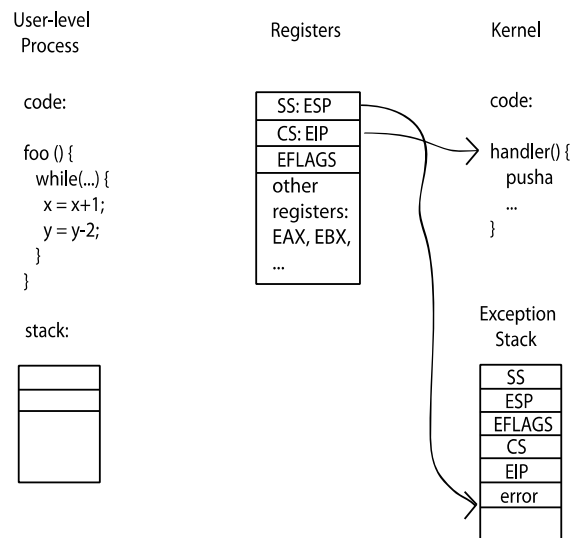


1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.34

During



1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.35

Kernel System Call Handler

- **Vector through well-defined syscall entry points!**
 - Table mapping system call number to handler
- **Locate arguments**
 - In registers or on user (!) stack
- **Copy arguments**
 - From user memory into kernel memory
 - Protect kernel from malicious code evading checks
- **Validate arguments**
 - Protect kernel from errors in user code
- **Copy results back**
 - Into user memory

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.36

Hardware support: Interrupt Control

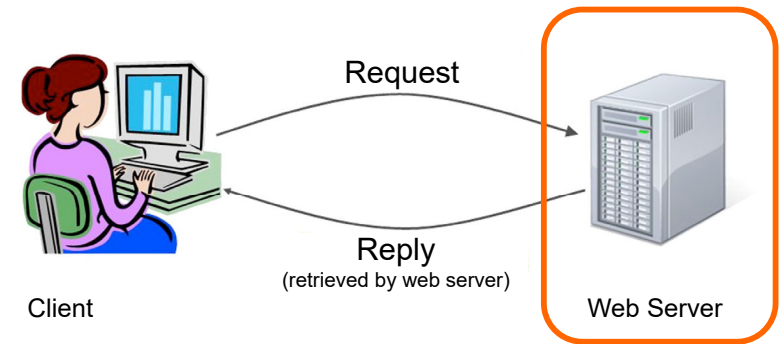
- Interrupt processing not visible to the user process:
 - Occurs between instructions, restarted transparently
 - No change to process state
 - What can be observed even with perfect interrupt processing?
- Interrupt Handler invoked with interrupts 'disabled'
 - Re-enabled upon completion
 - Non-blocking (run to completion, no waits)
 - Pack up in a queue and pass off to an OS thread for hard work
 - » wake up an existing OS thread

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.37

Putting it together: web server

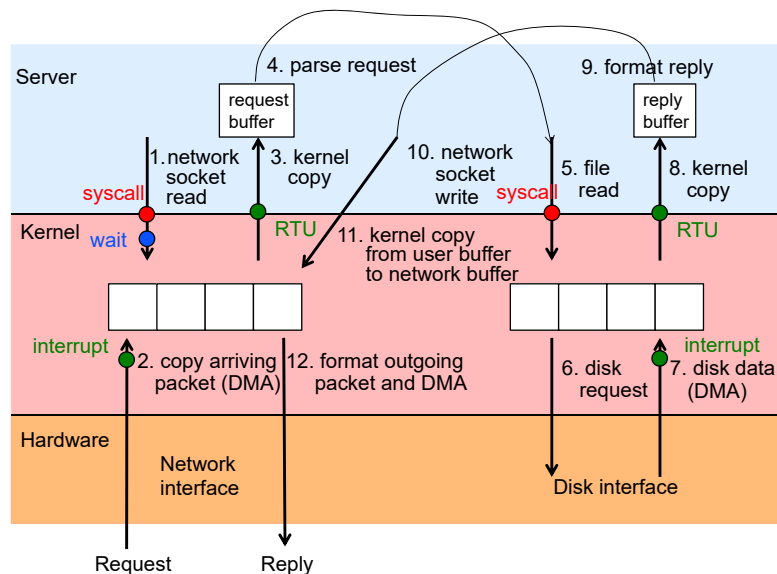


1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.38

Putting it together: web server



1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.39

A DEEP Dive into Pintos
and
Where we are Going!

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.40

Project 1: Processes

- Allocate and initialize Process object
- Allocate and initialize kernel thread mini-stack and associated Thread object
- Allocate and initialize page table for process
- Load code and static data into user pages
- Build initial User Stack
 - Initial register contents
- Schedule (post) process thread for execution
- ...
- Eventually switch to user thread ...
- Several lists of various types

Project 1

Pintos: `process.c`, `thread.c`

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.41

Understanding "Address Space"

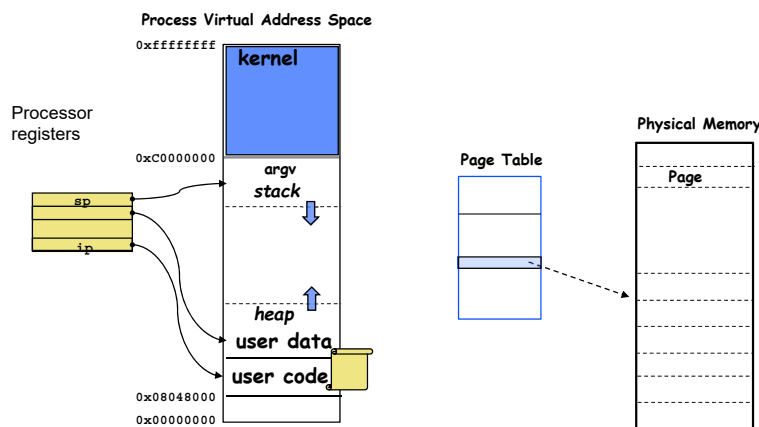
- Page table is the primary mechanism
- Privilege Level determines which regions can be accessed
 - Which entries can be used
- **System (PL=0) can access all, User (PL=3) only part**
- Each process has its own address space
- The "System" part of all of them is the same
 - ⇒ All system threads share the same system address space and same memory
- **This address pattern less (not?) common now after the Meltdown attack was discovered in 2017**
 - More Later in Term!!

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.42

User Process View

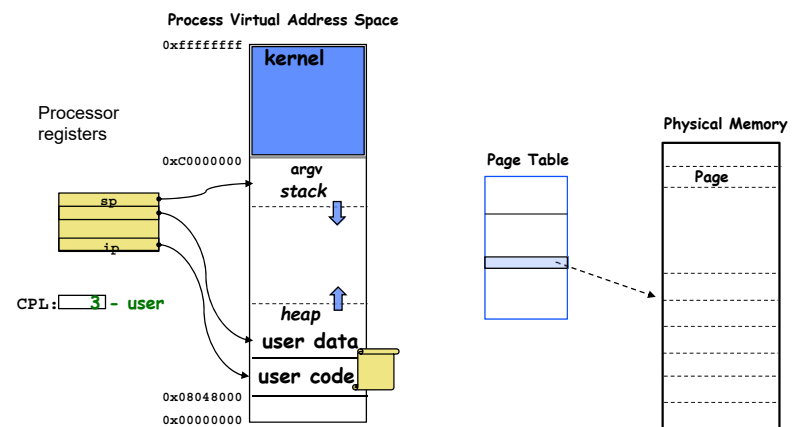


1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.43

Processor Mode (Privilege Level)

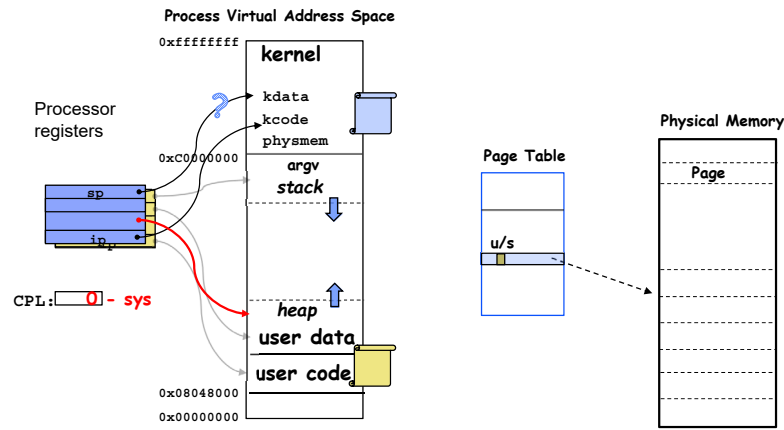


1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.44

User → Kernel: PL = 0

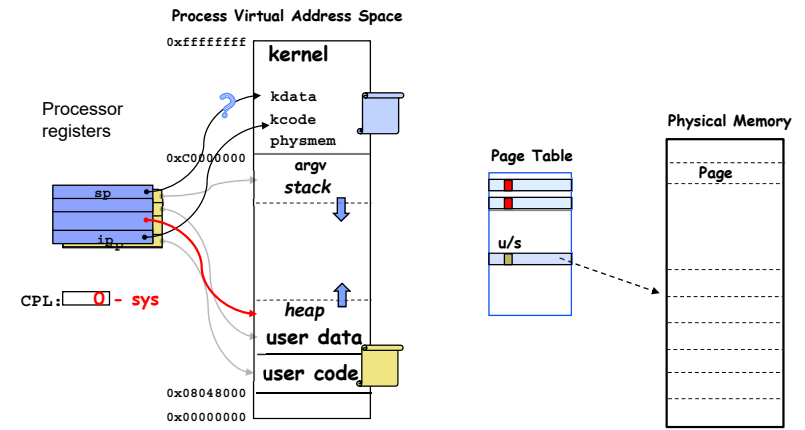


1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.45

Page Table enforces PL

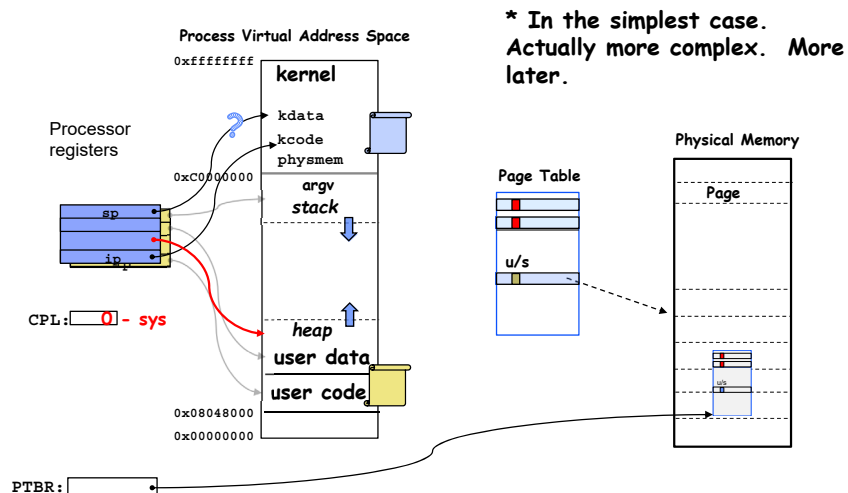


1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.46

Page Table resides in memory*

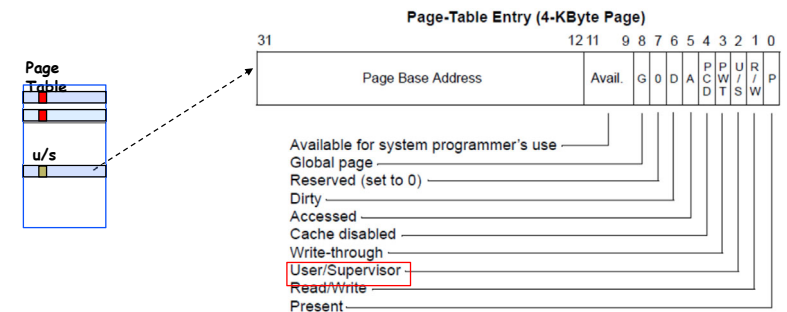


1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.47

x86 (32-bit) Page Table Entry



- Controls many aspects of access
- Later – discuss page table organization
 - For 32 (64?) bit VAS, how large? vs size of memory?
 - Use sparsely. Very very fast HW access

Pintos: [page_dir.c](#)

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.48

Kernel Portion of Address Space

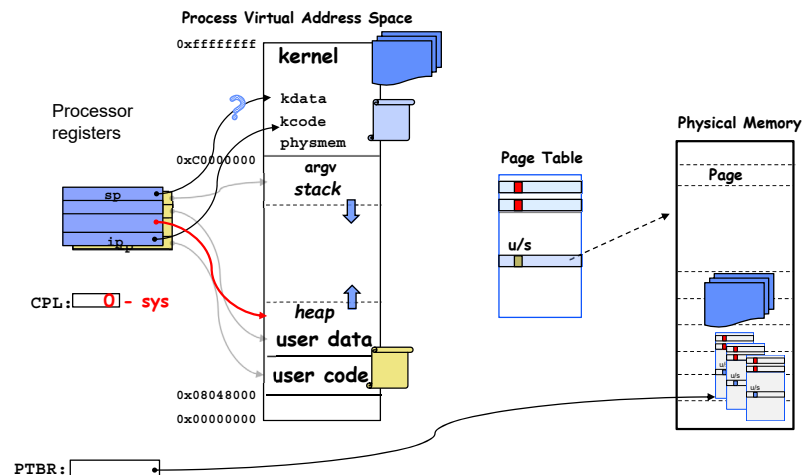
- Contains the kernel code
 - Loaded when the machine booted
- Explicitly mapped to physical memory
 - OS creates the page table
- Used to contain all kernel data structures
 - List of all the processes and threads
 - The page tables for those processes
 - Other system resources (files, sockets, ttys, ...)
- Also contains (little) stacks for “kernel threads”
 - Early OS design serviced all processes on a single execution thread
 - » Event driven programming
 - Today: Each Process Thread supported by (little) Kernel Thread

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.49

1 Kernel Code, many Kernel “stacks”



1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.50

From Machine Structure to OS Data Structures

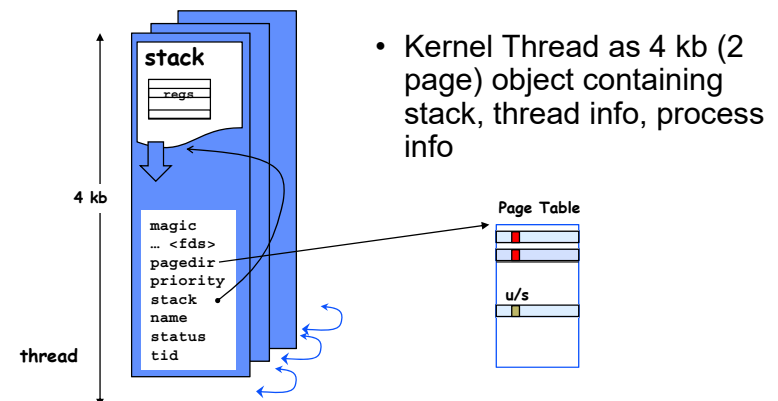
- Traditional Unix, etc. design maintains a Process Control Block (PCB) per process
- Each with a Thread Control Block (TCB) per thread of that process
- Today, assume **single** thread per process
 - PINTOS model
- Linux organized around threads with “groups of threads” associated with a process

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.51

PINTOS Thread



- Kernel Thread as 4 kb (2 page) object containing stack, thread info, process info

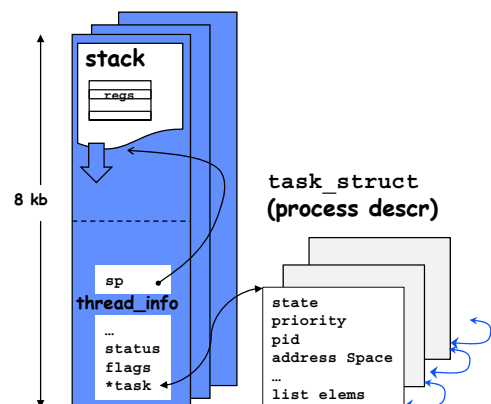
Pintos: thread.c

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.52

Linux “Task”



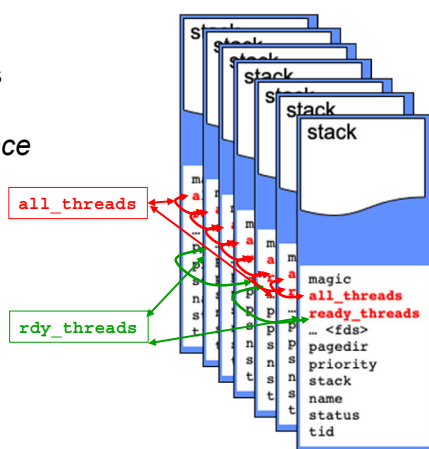
- Kernel Thread as 8 kb (2 page) object containing stack and thread information + process descriptor

Process Creation

- Allocate and initialize Process object
- Allocate and initialize kernel thread mini-stack and associated Thread object
- Allocate and initialize page table for process
 - Referenced by process object
- Load code and static data into user pages
- Build initial User Stack
 - Initial register contents, argv, ...
- Schedule (post) process thread for execution
- ...
- Eventually *switch* to user thread ...
- Several lists of various types

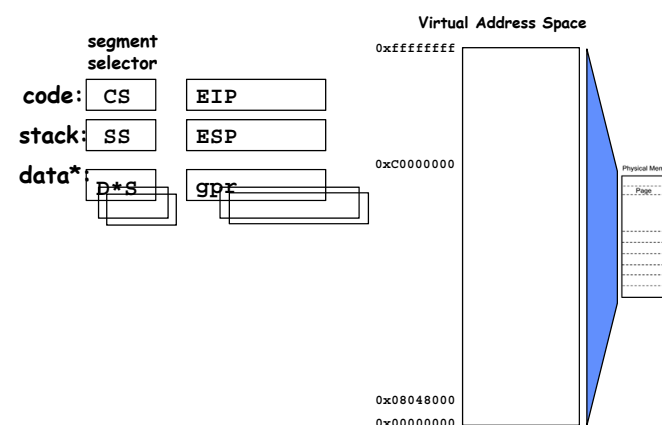
Aside: Polymorphic lists in C

- Many places in the kernel need to maintain a “list of X”
 - This is tricky in C, which has no polymorphism
 - Essentially adding an *interface* to a package (ala Go)
- In Linux and Pintos this is done by embedding a `list_elem` in the struct
 - Macros allow shift of view between object and list
 - You’ll practice in HW1 – before getting into PINTOS

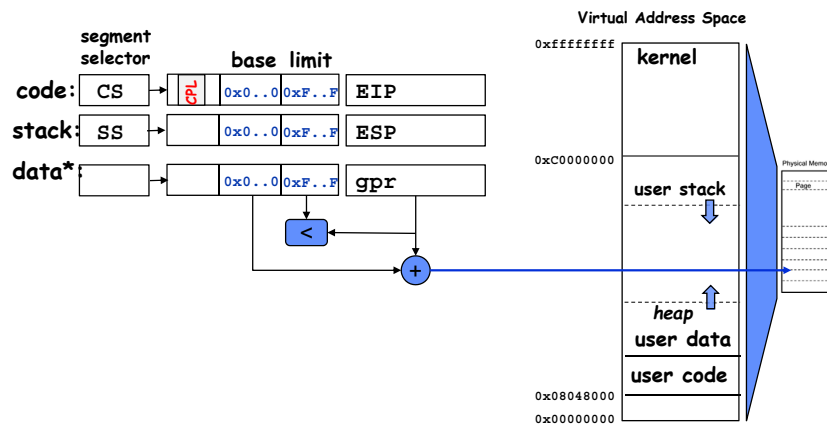


Pintos: list.c

Bit of x86 thread/process/VAS management



Bit of x86 thread/process/VAS management

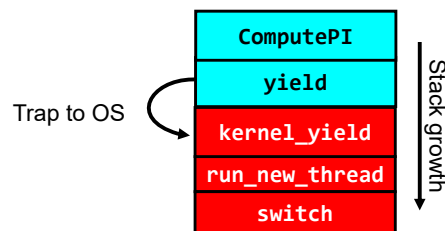


Pintos: loader.h

Recall: 3 types of U→K Mode Transfer

- Syscall
 - Process requests a system service, e.g., exit
 - Like a function call, but “outside” the process
 - Does not have the address of the system function to call
 - Like a Remote Procedure Call (RPC) – for later
 - Marshall the syscall id and args in registers and exec syscall
- Interrupt
 - External asynchronous event triggers context switch
 - eg. Timer, I/O device
 - Independent of user process
- Trap
 - Internal synchronous event in process triggers context switch
 - e.g., Protection violation (segmentation fault), Divide by zero, ...
- All 3 *exceptions* are an UNPROGRAMMED CONTROL TRANSFER
 - Where does it go? (To handler specified in interrupt vector)
 - Are interrupts enabled or disabled when get there?

Stack for Thread Transition

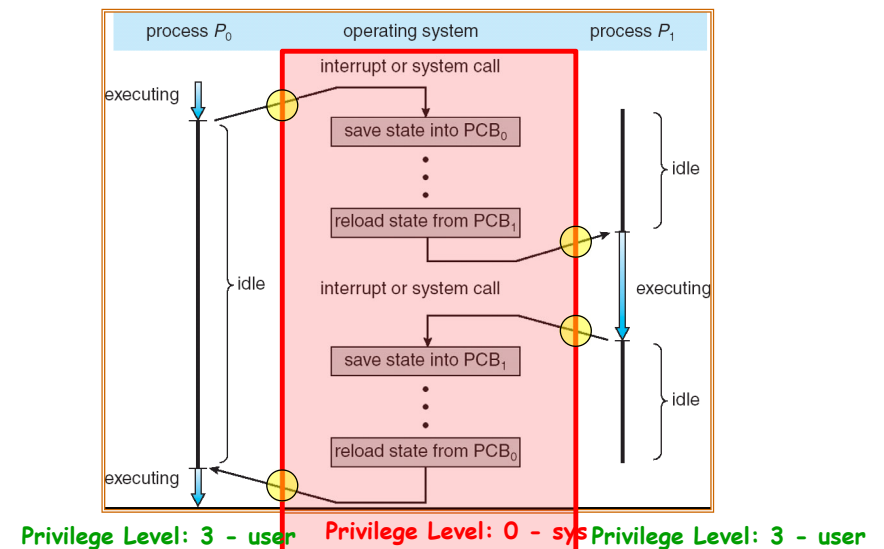


Cyan = User Stack; Red = Kernel Stack

```
run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread)
    ThreadHouseKeeping(); /* Do any cleanup */
}
```

Scheduling: Policy Decision

A Privileged View of the Context Switch



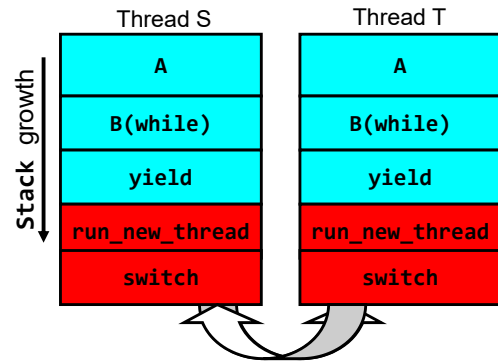
Stacks During Context Switch

- Consider the following code blocks:

```

proc A() {
    B();
}
proc B() {
    while(TRUE) {
        yield();
    }
}
    
```

- Suppose we have 2 threads:
 - Threads S and T



Thread S's switch returns to Thread T's (and vice versa)

Saving/Restoring state (often called "Context Switch")

```

Switch(tCur, tNew) {
    /* Unload old thread */
    TCB[tCur].regs.r7 = CPU.r7;
    ...
    TCB[tCur].regs.r0 = CPU.r0;
    TCB[tCur].regs.sp = CPU.sp;
    TCB[tCur].regs.retpc = CPU.retpc; /*return addr*/

    /* Load and execute new thread */
    CPU.r7 = TCB[tNew].regs.r7;
    ...
    CPU.r0 = TCB[tNew].regs.r0;
    CPU.sp = TCB[tNew].regs.sp;
    CPU.retpc = TCB[tNew].regs.retpc;
    return; /* Return to CPU.retpc */
}
    
```

Hardware context switch support

- Syscall/Intr (U → K)
 - PL 3 → 0;
 - TSS ← EFLAGS, CS:EIP;
 - SS:SP ← k-thread stack (TSS PL 0);
 - push (old) SS:ESP onto (new) k-stack
 - push (old) eflags, cs:eip, <err>
 - CS:EIP ← <k target handler>
- Then
 - Handler then saves other regs, etc
 - Does all its work, possibly choosing other threads, changing PTBR (CR3)
- iret (K → U)
 - PL 0 → 3;
 - Eflags, CS:EIP ← popped off k-stack
 - SS:SP ← user thread stack (TSS PL 3);

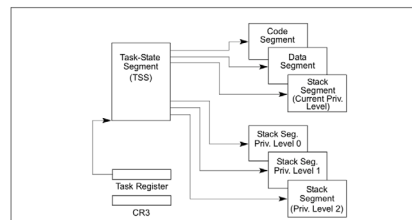
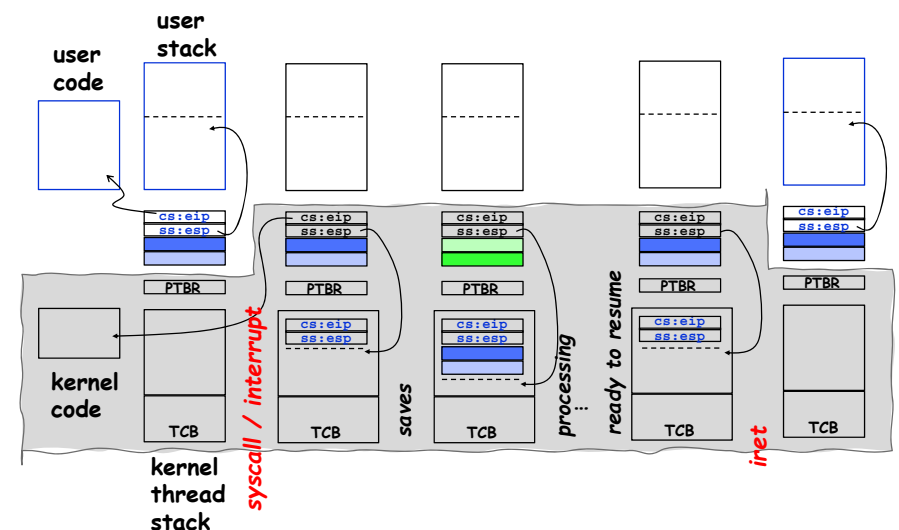
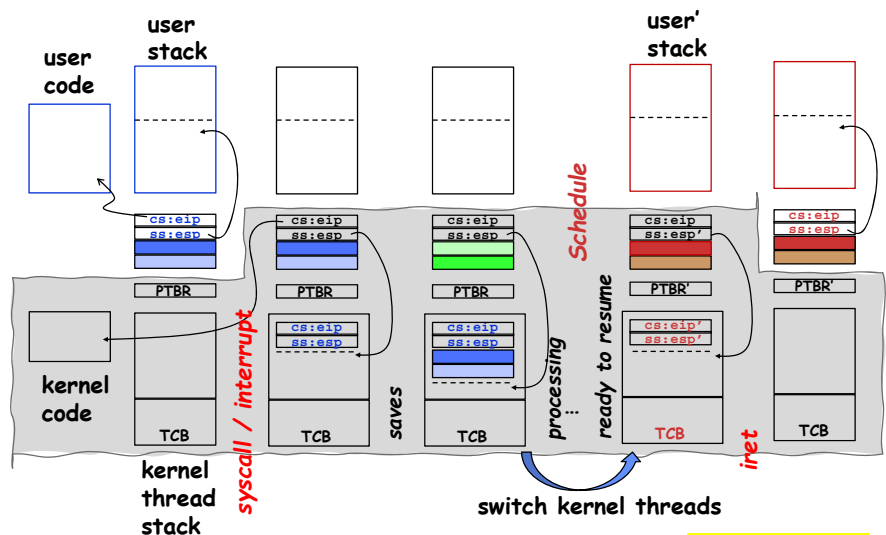


Figure 7-1. Structure of a Task

Context Switch – in pictures



Context Switch – Scheduling



Pintos: switch.S

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.65

Concurrency

- But, ... ???
- With all these threads in the kernel, won't they step on each other?
 - For example, while one is loading a program, other threads should run ...
 - Processes are isolated from each other, but all the threads in the kernel share the kernel address space, memory, data structures
- We will study synchronization soon
- The kernel controls whether hardware interrupts are enabled or not
 - Disabled on entry, selectively enable
 - Atomic operations, ...

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.66

Dispatch Loop

```

Loop {
    RunThread();
    ChooseNextThread();
    SaveStateOfCPU(curTCB);
    LoadStateOfCPU(newTCB);
}
    
```

- Conceptually all the OS executes
- Infinite Loop
 - When would we ever "exit?"
 - Can we assume some thread is always ready?

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.67

Dispatch Loop

```

Loop {
    RunThread();
    ChooseNextThread();
    SaveStateOfCPU(curTCB);
    LoadStateOfCPU(newTCB);
}
    
```

How to run a new thread?

- Load thread's registers into CPU
- Load its environment (address space, if in different process)
- Jump to thread's PC

How does dispatch loop get control again?

- Thread returns control voluntarily – yield, I/O
- External events: thread is preempted

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.68

Thread Operations in Pintos

- `thread_create(name, priority, func, args)`
 - Create a new thread to run `func(args)`
- `thread_yield()`
 - Relinquish processor voluntarily
- `thread_join(thread)`
 - Wait (put in queue) until thread exits, then return
- `thread_exit`
 - Quit thread and clean up, wake up joiner if any

*More later,
incl. synch
ops*

Meta-Question

- Process is an instance of a program executing.
 - The fundamental OS responsibility
- Processes do their work by processing and calling file system operations
- Are there any operations on processes themselves?
- `exit` ?

pid.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
int main(int argc, char *argv[])
{
    pid_t pid = getpid();    /* get current processes PID */

    printf("My pid: %d\n", pid);

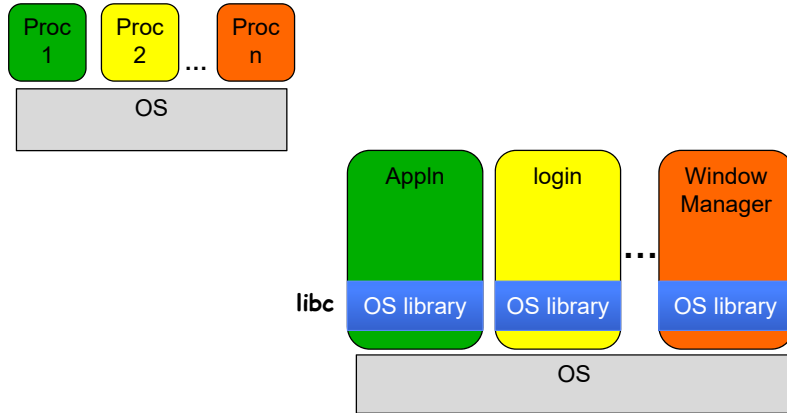
    exit(0);
}
```

ps anyone?

Can a process create a process ?

- Yes
- Fork creates a copy of process
- What about the program you want to run?

OS Run-Time Library

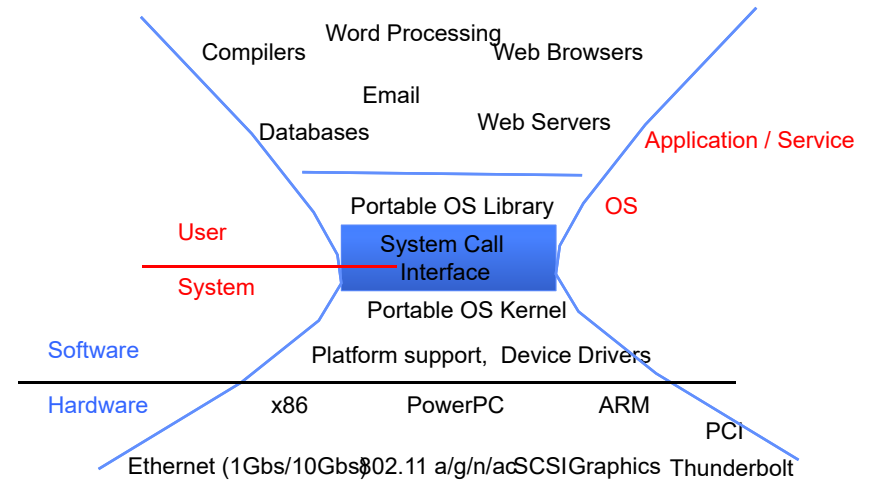


1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.73

A Narrow Waist



1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.74

POSIX/Unix

- **Portable Operating System Interface [X?]**
- Defines “Unix”, derived from AT&T Unix
 - Created to bring order to many Unix-derived OSs
- Interface for **application programmers** (mostly)

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.75

System Calls

Application:

```
fd = open(pathname);
```

Library:

```
File *open(pathname) {
    asm code ... syscall # into ax
    put args into registers bx, ...
    special trap instruction
```

Operating System:

```
get args from regs
dispatch to system func
process, schedule, ...
complete, resume process
```

```
get results from regs
```

```
};
```

Continue with results

Pintos: userprog/syscall.c, lib/user/syscall.c

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.76

SYSCALLs (of over 300)

%eax	Name	Source	%ebx	%ecx	%edx	%esi	%edi
1	sys_exit	kernel/exit.c	int	-	-	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
3	sys_read	fs/read_write.c	unsigned int	char *	size_t	-	-
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t	-	-
5	sys_open	fs/open.c	const char *	int	int	-	-
6	sys_close	fs/open.c	unsigned int	-	-	-	-
7	sys_waitpid	kernel/exit.c	pid_t	unsigned int *	int	-	-
8	sys_creat	fs/open.c	const char *	int	-	-	-
9	sys_link	fs/namei.c	const char *	const char *	-	-	-
10	sys_unlink	fs/namei.c	const char *	-	-	-	-
11	sys_execve	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
12	sys_chdir	fs/open.c	const char *	-	-	-	-
13	sys_time	kernel/time.c	int *	-	-	-	-
14	sys_mknod	fs/namei.c	const char *	int	dev_t	-	-
15	sys_chmod	fs/open.c	const char *	mode_t	-	-	-
16	sys_lchown	fs/open.c	const char *	uid_t	gid_t	-	-
18	sys_stat	fs/stat.c	char *	struct _old_kernel_stat *	-	-	-
19	sys_lseek	fs/read_write.c	unsigned int	off_t	unsigned int	-	-
20	sys_getpid	kernel/sched.c	-	-	-	-	-
21	sys_mount	fs/super.c	char *	char *	char *	-	-
22	sys_oldumount	fs/super.c	char *	-	-	-	-
23	sys_setuid	kernel/sys.c	uid_t	-	-	-	-
24	sys_getuid	kernel/sched.c	-	-	-	-	-
25	sys_stime	kernel/time.c	int *	-	-	-	-
26	sys_ptrace	arch/i386/kernel/ptrace.c	long	long	long	long	-
27	sys_alarm	kernel/sched.c	unsigned int	-	-	-	-
28	sys_fstat	fs/stat.c	unsigned int	struct _old_kernel_stat *	-	-	-
29	sys_pause	arch/i386/kernel/sys_i386.c	-	-	-	-	-
30	sys_ftime	fs/open.c	char *	struct utimbuf *	-	-	-

Pintos: syscall-nr.h

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.77

Recall: Kernel System Call Handler

- Locate arguments
 - In registers or on user(!) stack
- Copy arguments
 - From user memory into kernel memory
 - Protect kernel from malicious code evading checks
- Validate arguments
 - Protect kernel from errors in user code
- Copy results back
 - into user memory

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.78

Process Management

- exit – terminate a process
- **fork – copy the current process**
- exec – change the *program* being run by the current process
- wait – wait for a process to finish
- kill – send a *signal* (interrupt-like notification) to another process
- sigaction – set handlers for signals

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.79

Creating Processes

- pid_t fork(); -- copy the current process
 - New process has different pid
- Return value from **fork()**: pid (like an integer)
 - When > 0:
 - » Running in (original) **Parent** process
 - » return value is **pid** of new child
 - When = 0:
 - » Running in new **Child** process
 - When < 0:
 - » Error! Must handle somehow
 - » Running in original process
- **State of original process duplicated in *both* Parent and Child!**
 - Address Space (Memory), File Descriptors (covered later), etc...

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.80

fork1.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    pid_t cpid, mypid;
    pid_t pid = getpid(); /* get current processes PID */
    printf("Parent pid: %d\n", pid);
    cpid = fork();
    if (cpid > 0) { /* Parent Process */
        mypid = getpid();
        printf("[%d] parent of [%d]\n", mypid, cpid);
    } else if (cpid == 0) { /* Child Process */
        mypid = getpid();
        printf("[%d] child\n", mypid);
    } else {
        perror("Fork failed");
    }
}
```

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.81

fork1.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    pid_t cpid, mypid;
    pid_t pid = getpid(); /* get current processes PID */
    printf("Parent pid: %d\n", pid);
    cpid = fork();
    if (cpid > 0) { /* Parent Process */
        mypid = getpid();
        printf("[%d] parent of [%d]\n", mypid, cpid);
    } else if (cpid == 0) { /* Child Process */
        mypid = getpid();
        printf("[%d] child\n", mypid);
    } else {
        perror("Fork failed");
    }
}
```

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.82

fork1.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    pid_t cpid, mypid;
    pid_t pid = getpid(); /* get current processes PID */
    printf("Parent pid: %d\n", pid);
    cpid = fork();
    if (cpid > 0) { /* Parent Process */
        mypid = getpid();
        printf("[%d] parent of [%d]\n", mypid, cpid);
    } else if (cpid == 0) { /* Child Process */
        mypid = getpid();
        printf("[%d] child\n", mypid);
    } else {
        perror("Fork failed");
    }
}
```

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.83

fork_race.c

```
int i;
cpid = fork();
if (cpid > 0) {
    for (i = 0; i < 10; i++) {
        printf("Parent: %d\n", i);
        // sleep(1);
    }
} else if (cpid == 0) {
    for (i = 0; i > -10; i--) {
        printf("Child: %d\n", i);
        // sleep(1);
    }
}
```

- What does this print?
- Would adding the calls to sleep matter?

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.84

Fork “race”

```
int i;
cpid = fork();
if (cpid > 0) {
    for (i = 0; i < 10; i++) {
        printf("Parent: %d\n", i);
        // sleep(1);
    }
} else if (cpid == 0) {
    for (i = 0; i > -10; i--) {
        printf("Child: %d\n", i);
        // sleep(1);
    }
}
```



Process Management

- fork – copy the current process
- exec – change the *program* being run by the current process
- **wait – wait for a process to finish**
- kill – send a *signal* (interrupt-like notification) to another process
- sigaction – set handlers for signals

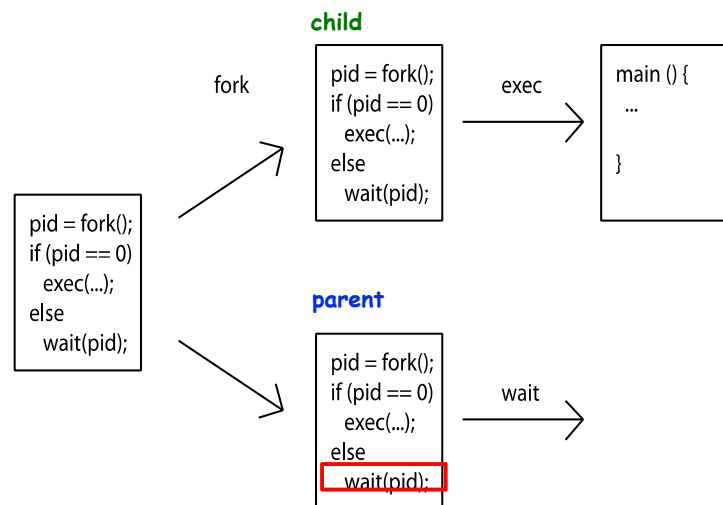
fork2.c – parent waits for child to finish

```
int status;
pid_t tcpid;
...
cpid = fork();
if (cpid > 0) {                /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
    tcpid = wait(&status);
    printf("[%d] bye %d(%d)\n", mypid, tcpid, status);
} else if (cpid == 0) {        /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
}
...
```

Process Management

- fork – copy the current process
- **exec – change the *program* being run by the current process**
- wait – wait for a process to finish
- kill – send a *signal* (interrupt-like notification) to another process
- sigaction – set handlers for signals

Process Management



1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.89

fork3.c

```
...
cpid = fork();
if (cpid > 0) {                /* Parent Process */
    tcpid = wait(&status);
} else if (cpid == 0) {        /* Child Process */
    char *args[] = {"ls", "-l", NULL};
    execv("/bin/ls", args);
    /* execv doesn't return when it works.
       So, if we got here, it failed! */
    perror("execv");
    exit(1);
}
...
```

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.90

Shell

- A shell is a job control system
 - Allows programmer to create and manage a set of programs to do some task
 - Windows, MacOS, Linux all have shells
- Example: to compile a C program

```
cc -c sourcefile1.c
cc -c sourcefile2.c
ln -o program sourcefile1.o sourcefile2.o
./program
```



1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.91

Process Management

- fork – copy the current process
- exec – change the *program* being run by the current process
- wait – wait for a process to finish
- kill – send a *signal* (interrupt-like notification) to another process
- **sigaction – set handlers for signals**

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.92

inf_loop.c

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

void signal_callback_handler(int signum) {
    printf("Caught signal!\n");
    exit(1);
}

int main() {
    struct sigaction sa;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    sa.sa_handler = signal_callback_handler;

    sigaction(SIGINT, &sa, NULL);
    while (1) {}
}
```

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.93

Common POSIX Signals

- SIGINT – control-C
- SIGTERM – default for kill shell command
- SIGSTP – control-Z (default action: stop process)

- SIGKILL, SIGSTOP – terminate/stop process
 - Can't be changed or disabled with sigaction
 - Why?

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.94

Summary

- Process consists of two pieces
 1. Address Space (Memory & Protection)
 2. One or more threads (Concurrency)
- Represented in kernel as
 - Process object (resources associated with process)
 - Thread object + (mini) stack
 - Hardware support critical in U → K → U context switch
 - Different privileges in different modes (CPL, Page Table)
- Variety of process management syscalls
 - fork, exec, wait, kill, sigaction
- Scheduling: Threads move between queues
- Threads: multiple stacks per address space
 - Context switch: Save/Restore registers, "return" from new thread's switch routine
 - So far, we've only seen kernel threads

1/28/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 3.95