

# Operating Systems Principles

## UCLA-CS111-W18

Quentin Truong  
Taught by Professor Reiher

Winter 2018

## Contents

<b>1</b>	<b>L3: Arpaci-Dusseau Chapter 5: Interlude: Process API</b>	<b>2</b>
1.1	The fork() System Call . . . . .	2
1.2	The wait() System Call . . . . .	2
1.3	The exec() System Call . . . . .	2
1.4	Why? Motivating The API . . . . .	2
1.5	Other Parts Of The API . . . . .	2
<b>2</b>	<b>L3: Arpaci-Dusseau Chapter 6: Mechanism: Limited Direct Execution</b>	<b>2</b>
2.1	Basic Technique: Limited Direct Execution . . . . .	2
2.2	Problem 1: Restricted Operations . . . . .	3
2.3	Problem 2: Switching Between Processes . . . . .	3
2.4	Worried About Concurrency? . . . . .	4
2.5	Summary . . . . .	4
<b>3</b>	<b>L3: Linking and Libraries: Object Modules, Linkage Editing, Libraries</b>	<b>4</b>
3.1	Introduction . . . . .	4
3.2	The Software Generation Tool Chain . . . . .	4
3.3	Object Modules . . . . .	4
3.4	Libraries . . . . .	5
3.5	Linkage Editing . . . . .	5
3.6	Load Modules . . . . .	5
3.7	Static vs. Shared Libraries . . . . .	5
3.8	Dynamically Loaded Libraries . . . . .	5
<b>4</b>	<b>L3: Linkage Conventions: Stack Frames and Linkage Conventions</b>	<b>5</b>
4.1	Introduction . . . . .	5
4.2	The Stack Model of Programming Languages . . . . .	5
4.3	Subroutine Linkage Conventions . . . . .	6
4.4	Traps and Interrupts . . . . .	6
<b>5</b>	<b>L4: Arpaci-Dusseau Chapter 7: Scheduling: Introduction</b>	<b>7</b>
5.1	Overview . . . . .	7
<b>6</b>	<b>L4:</b>	<b>7</b>
6.1	Overview . . . . .	7

## 1 L3: Arpaci-Dusseau Chapter 5: Interlude: Process API

### 1.1 The fork() System Call

- Crux: How to create and control processes
- fork()
  - Creates new process; returns child's PID to parent; returns 0 to child;
  - Each has own PC, registers, address space
- Nondeterministic Behavior
  - Scheduler will decide which process to run
  - May lead to problems in multi-threaded programs

### 1.2 The wait() System Call

- wait()
  - Parent calls wait() to wait for child to finish execution

### 1.3 The exec() System Call

- exec()
  - Loads code, overwrites code segment, and reinitializes memory space
  - Takes executable name and arguments
  - Does not create a new process; transform current process

### 1.4 Why? Motivating The API

- Separation
  - Separating fork() and exec() allows code to alter the environment of the about-to-run program
- Example
  - Shell forks a process, execs the program, and waits until finished
  - The separation allows for things such as output to be redirected (closes stdout and opens file)

### 1.5 Other Parts Of The API

- kill()
  - System call sends signal to process to sleep, die, etc

## 2 L3: Arpaci-Dusseau Chapter 6: Mechanism: Limited Direct Execution

### 2.1 Basic Technique: Limited Direct Execution

- Crux: How to efficiently virtualize CPU with control
- Limited Direct Execution
  - OS will create entry for process list, allocate memory for program, load program into memory, setup stack with argc/v, clear registers, execute call to main()
  - Program will run main(), execute return
  - OS will free memory, remove from process list
- LDE good bc fast, but
  - Problem of keeping control
  - Problem of time sharing still

## 2.2 Problem 1: Restricted Operations

- User mode vs. Kernel mode
  - Restricted mode which needs to ask kernel to perform system calls
  - Calls like open() are actually procedure calls with trap to enter kernel and raise privilege
  - Return-from-trap is used to enter user mode from kernel and drop privilege
  - Push counters, flags, registers onto per-process kernel stack when trapping
- Trap table is used to control what code is executed when trapping
  - Trap handler used by hardware to cause interrupts
  - Telling hardware where trap table is is privileged
  - Trap handler actually uses system-call number, rather than specifying an address (another layer of protection)
- Two phases of LDE
  - At boot, kernel initializes trap table and remembers where it is

OS @ boot (kernel mode)	Hardware
<b>initialize trap table</b>	remember addresses of... syscall handler timer handler
<b>start interrupt timer</b>	start timer interrupt CPU in X ms

## 2.3 Problem 2: Switching Between Processes

- How can OS regain control?
  - Because process is running, so OS is not running
- Cooperative Approach
  - System calls include explicit yield system call, transferring control back to OS
- Noncooperative Approach
  - Reboot, Timer Interrupt
- Saving and Restoring Context
  - Scheduler will choose when to switch processes

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...
	<b>timer interrupt</b> save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call switch() routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) <b>return-from-trap (into B)</b>		
	restore regs(B) from k-stack(B) move to user mode jump to B's PC	
		Process B
		...

## 2.4 Worried About Concurrency?

- Interrupt during interrupt?
  - Many complex things to do
  - Could disable interrupts (but this might lose interrupts), or locking schemes, etc

## 2.5 Summary

- Reboot
  - Good technique because restores system to well-tested state
  - OS will 'baby-proof' by only allowing processes to run in restricted mode and with interrupt handlers

# 3 L3: Linking and Libraries: Object Modules, Linkage Editing, Libraries

## 3.1 Introduction

- Process as fundamental; as executing instance of program
  - Program as one or more files (these are not the executables though)
  - Source must be translated

## 3.2 The Software Generation Tool Chain

- Source module
  - Editable text in some language like C
- Relocatable object module
  - Sets of compiled instructions; incomplete programs
- Library
  - Collection of object modules
- Load module
  - Complete programs ready to be loaded into memory
- Compiler
  - Parse source modules; usually generates assembly, may generate pseudo-machine
- Assembler
  - Object module with mostly machine code
  - Memory addresses of functions, variables may not be filled in
- Linkage Editor
  - Find all required object modules and resolve all references
- Program Loader
  - Examines load module, creates virtual space, reads instructions, initializes data values
  - Find and map additional shared libraries

## 3.3 Object Modules

- Code in multiple files
  - Because more understandable if splitting functionality
  - Many functions are reused, so use external libraries
- Relocatable object modules are program fragments
  - Incomplete because make references to code in other modules
  - Even the references to other code are only relative
- ELF format
  - Header section with types, sizes, and location of other sections
  - Code and data section to be loaded contiguously
  - Symbol table of external symbols
  - Relocation entries describing location of field, width/type of field, symbol table entry

### 3.4 Libraries

- Reusable, standard functions in libraries
  - Libraries not always orthogonal and independent
- Build program by combining object modules and resolving external references

### 3.5 Linkage Editing

- Resolution
  - Search libraries to find object modules to resolve external references
- Loading
  - Lay text and data in single virtual address space
- Relocation
  - Ensure references correctly reflect chosen address

### 3.6 Load Modules

- Load module requires no relocation and is complete
- When loading new module
  - Determine required text and data sizes and locations, allocate segments, read contents, create a stack segment with pointer
- Load module has symbol table to help determine where exceptions occurred

### 3.7 Static vs. Shared Libraries

- Static Linking
  - Many copies, so inefficient; also, permanent copy, so don't receive updates
- Shared Libraries
  - Implementations vary, but one way
    - Reserve address for libraries, linkage edit, map with redirection table, etc, more mapping
  - Efficient, but doesn't work for static data because one copy
  - But can be slow to load many libraries, and must know library name at loadtime

### 3.8 Dynamically Loaded Libraries

- DLL loaded once needed
  - Choose and load library, binds, use library, unload
  - Resource efficient because can unload
- Implicitly Loaded Dynamically Loadable Libraries
  - Another implementation of DLL with different pros/cons

## 4 L3: Linkage Conventions: Stack Frames and Linkage Conventions

### 4.1 Introduction

- What is the state of computation and how can it be saved?
- What is the mechanism of requesting and receiving services?

### 4.2 The Stack Model of Programming Languages

- Procedure-local variables
  - Stored on a LIFO stack
  - New call frames pushed onto stack when procedure called; old frames popped when procedure returns
  - Long-lived resources on heap, not stack

### 4.3 Subroutine Linkage Conventions

- X86 Subroutine Linkage
  - Pass parameters to be called by routine
  - Save return address and transfer control to entry
  - Save content of non-volatile registers
  - Allocate space for local variables
- X86 Return Process
  - Return value to where routine expects it
  - Pop local storage
  - Restore registers
  - Subroutine transfer control to return address
- Responsibilities split between caller and callee
- Saving and restoring state of procedure is a matter of stack frame and registers

### 4.4 Traps and Interrupts

- Procedure call vs Trap/Interrupt
  - Procedure requested by running software and expects result; linkage conventions under software control
  - After trap/interrupt, should restore state
- How
  - Number associated with every interrupt/exception, maps to PS/PC
  - Push new program counter and program status (from interrupt/trap vector table) onto CPU stack
  - Resume execution at new PC
  - First level handler
    - Save general registers on stack
    - Choose second level handler based on info from interrupt/trap
  - Second level handler (procedure call)
    - Deal with interrupt/exception
    - Return to first level handler
      - Restore saved registers and return-from-interrupt/trap
  - CPU reloads PC/PS and resumes execution
- Stacking/unstacking interrupt/trap is 100x+ slower than procedure call

## 5 L4: Arpaci-Dusseau Chapter 7: Scheduling: Introduction

### 5.1 Overview

## 6 L4:

### 6.1 Overview