

**OPEN SOURCE  
LECTURE NOTES**  
**Operating System Design and  
Implementation**

**Anthony Aaby**

DRAFT Version:  $\alpha$ 1.0  
EDITED: November 10, 2006

This work is licensed © 2006 by

Anthony A. Aaby  
Walla Walla College  
204 S. College Ave.  
College Place, WA 99324  
E-mail: aabyan@wwc.edu

under the Creative Commons Attribution-NonCommercial License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/1.0>, or send a letter to Creative Commons, 559 Nathan Abbot Way, Stanford, California 94305, USA.

This book is distributed in the hope it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose. You may copy, modify, and distribute this book for the cost of reproduction provided the above creative commons notice remains intact. No explicit permission is required from the author for reproduction of this book in any medium, physical or electronic.

The most current version of this text and L<sup>A</sup>T<sub>E</sub>Xsource is available at:

Source: <http://www.cs.wwc.edu/~aabyan/LN/OS/osnotes.tar.gz>  
Postscript: <http://www.cs.wwc.edu/~aabyan/LN/OS/osln.ps>  
DVI: <http://www.cs.wwc.edu/~aabyan/LN/OS/osln.dvi>  
PDF: <http://www.cs.wwc.edu/~aabyan/LN/OS/osln.pdf>  
HTML: <http://www.cs.wwc.edu/~aabyan/LN/OS/osln/index.html>

# Contents

<b>Preface</b>	<b>11</b>
<b>Assignments</b>	<b>13</b>
<b>I Introduction</b>	<b>15</b>
<b>1 OS1: Overview of Operating Systems</b>	<b>17</b>
1.1 A Computer System . . . . .	18
1.2 Role and Purpose of the OS . . . . .	18
1.2.1 Provide a service for clients - a virtual machine (top-down view) . . . . .	19
1.2.2 Resource manager (bottom-up view) . . . . .	19
1.2.3 The general functions of an operating system . . . . .	20
1.3 History of OS Development . . . . .	21
1.4 Functionality of a Typical OS . . . . .	21
1.5 Mechanisms to Support Client-Server Models, Hand-held Devices	22
1.6 Design Issues . . . . .	23
1.7 Influences of Security, Networking, Multimedia, Windows . . . .	24
<b>2 OS2: Operating System Principles</b>	<b>25</b>
2.1 Elements . . . . .	26
2.2 Structuring Methods . . . . .	26
2.2.1 Monolithic systems . . . . .	27
2.2.2 Layered systems . . . . .	27
2.2.3 Virtual machines . . . . .	27
2.2.4 Exokernels . . . . .	27
2.2.5 Client-server . . . . .	28
2.2.6 OS Research . . . . .	28
2.3 Abstractions, processes, and resources . . . . .	28
2.3.1 Resources . . . . .	28
2.3.2 Processes . . . . .	29
2.3.3 Threads . . . . .	29
2.3.4 Objects . . . . .	29

2.4	Concepts of APIs . . . . .	29
2.5	Application needs and evolution of techniques . . . . .	29
2.6	Device Organization . . . . .	29
2.7	Interrupts: methods and implementations . . . . .	29
2.8	User and System State . . . . .	29
<b>II</b>	<b>Processes</b>	<b>31</b>
<b>3</b>	<b>OS3: Concurrency</b>	<b>33</b>
<b>4</b>	<b>Processes</b>	<b>35</b>
4.1	Process Concept . . . . .	35
4.1.1	Definition . . . . .	35
4.1.2	Goal/Rationale . . . . .	36
4.1.3	Design . . . . .	36
4.1.4	Implementation . . . . .	37
4.2	Threads . . . . .	39
4.2.1	Goal/Rationale . . . . .	39
4.2.2	Thread Structure . . . . .	39
4.2.3	Design . . . . .	40
4.2.4	Implementation . . . . .	40
4.2.5	Solaris 2 . . . . .	40
4.3	Exercises . . . . .	40
<b>5</b>	<b>Deadlock</b>	<b>41</b>
5.1	Basic Concepts . . . . .	41
5.1.1	Resource-Allocation Graph . . . . .	42
5.2	Methods for handling deadlocks . . . . .	42
5.3	Ignore/Ostrich Algorithm . . . . .	42
5.4	Deadlock Prevention . . . . .	42
5.5	Deadlock Avoidance . . . . .	42
5.5.1	Safe State . . . . .	42
5.5.2	Resource-Allocation Graph Algorithm . . . . .	43
5.5.3	Banker's Algorithm . . . . .	43
5.6	Deadlock Detection and Recovery . . . . .	43
5.7	Combined Approach to Deadlock Handling . . . . .	43
5.8	Exercises . . . . .	43
<b>6</b>	<b>Synchronization and Communication</b>	<b>45</b>
6.1	Basic Concepts . . . . .	45
6.2	Software Solutions . . . . .	46
6.2.1	Two-Process "Solutions" . . . . .	46
6.2.2	Multiple-Process Solutions . . . . .	48
6.3	Hardware Solutions . . . . .	48
6.4	Semaphores . . . . .	49

6.4.1	Definition . . . . .	49
6.4.2	Goal/Rationale . . . . .	50
6.4.3	Design . . . . .	50
6.4.4	Implementation . . . . .	50
6.4.5	Usage . . . . .	50
6.4.6	Deadlocks and starvation . . . . .	50
6.4.7	Binary Semaphores . . . . .	50
6.5	Classical Problems of Synchronization . . . . .	51
6.5.1	Bounded Buffer . . . . .	51
6.5.2	Dining Philosophers . . . . .	51
6.5.3	Readers and Writers . . . . .	51
6.5.4	Sleeping Barber . . . . .	51
6.6	Critical Regions . . . . .	51
6.7	Monitors . . . . .	52
6.7.1	Example: Solaris 2 . . . . .	52
6.8	Message Passing . . . . .	52
6.9	Atomic Transactions . . . . .	54
6.10	Exercises . . . . .	54
<b>7</b>	<b>OS4 - Scheduling and dispatch</b>	<b>55</b>
<b>8</b>	<b>CPU Scheduling</b>	<b>57</b>
8.1	Ontology: Basic Concepts . . . . .	57
8.2	Values - Scheduling Criteria (Goals) . . . . .	60
8.3	Methods - Scheduling Algorithms . . . . .	60
8.3.1	First-Come, First-Served (FCFS) . . . . .	60
8.3.2	Shortest-Job-First (SJF)- shortest process next (SPN) . . . . .	60
8.3.3	Priority Scheduling . . . . .	61
8.3.4	Round-Robin (RR) . . . . .	61
8.3.5	Multilevel Queue Scheduling . . . . .	62
8.3.6	Multilevel Feedback Queue Scheduling . . . . .	62
8.4	Multiple-Processor Scheduling . . . . .	64
8.5	Real-time Scheduling . . . . .	64
8.6	Algorithm Evaluation . . . . .	64
8.6.1	Deterministic modeling . . . . .	64
8.6.2	Queueing models . . . . .	65
8.6.3	Simulations . . . . .	65
8.6.4	Implementation . . . . .	65
8.7	Exercises . . . . .	65
<b>III</b>	<b>I/O</b>	<b>67</b>
<b>9</b>	<b>OS6 - Device management</b>	<b>69</b>

<b>10 I/O Devices and Systems</b>	<b>71</b>
10.1 Devices as files . . . . .	71
10.2 I/O Devices . . . . .	71
10.2.1 Disk Devices . . . . .	71
10.3 I/O Systems . . . . .	72
10.3.1 Software . . . . .	72
10.3.2 Disk Scheduling . . . . .	72
10.3.3 Disk Management . . . . .	73
10.3.4 Swap Space Management . . . . .	74
10.3.5 Disk Reliability . . . . .	74
10.3.6 Stable-Storage Implementation . . . . .	74
 <b>IV Memory Management</b>	 <b>75</b>
<b>11 OS5 - Memory Management</b>	<b>77</b>
<b>12 Memory Management</b>	<b>79</b>
12.1 Background . . . . .	79
12.1.1 Modeling CPU Utilization . . . . .	79
12.1.2 Basic memory layout . . . . .	79
12.1.3 Address Binding . . . . .	79
12.1.4 Dynamic Loading . . . . .	80
12.1.5 Static Linking . . . . .	80
12.1.6 Dynamic Linking . . . . .	80
12.1.7 Overlays . . . . .	80
12.1.8 Logical versus Physical Address Space . . . . .	80
12.1.9 Implementation of Protection & Relocation . . . . .	80
12.1.10 Supervisor mode/User mode . . . . .	81
12.1.11 Swapping . . . . .	81
12.2 Contiguous Allocation . . . . .	81
12.2.1 Single-Partition Allocation . . . . .	81
12.2.2 Multiple-Partition Allocation . . . . .	81
12.3 Swap Space Management . . . . .	82
12.4 Paging (non-contiguous allocation) . . . . .	82
12.4.1 Basic Method . . . . .	82
12.4.2 Structure of the Page Table . . . . .	82
12.4.3 Multilevel Paging . . . . .	83
12.4.4 Inverted Page Table . . . . .	83
12.4.5 Shared Pages . . . . .	83
12.5 Segmentation . . . . .	83
12.5.1 Basic Method . . . . .	83
12.5.2 Hardware . . . . .	83
12.5.3 Implementation of Segment Tables . . . . .	83
12.5.4 Protection and Sharing . . . . .	83
12.5.5 Fragmentation . . . . .	83

12.6 Segmentation with Paging . . . . .	83
12.6.1 Multics . . . . .	83
12.6.2 OS/2 32-Bit Version . . . . .	83
12.7 Exercises . . . . .	83
<b>13 Virtual Memory</b>	<b>85</b>
13.1 Background . . . . .	85
13.2 Demand Paging . . . . .	85
13.3 Performance of Demand Paging . . . . .	86
13.4 Page Replacement . . . . .	86
13.5 Page Replacement Algorithms . . . . .	87
13.5.1 FIFO . . . . .	87
13.5.2 Optimal Algorithm . . . . .	87
13.5.3 LRU Algorithm . . . . .	87
13.5.4 LRU Approximation Algorithm . . . . .	87
13.6 Allocation of Frames . . . . .	88
13.6.1 Minimum number of frames . . . . .	88
13.6.2 Allocation Algorithms . . . . .	88
13.6.3 Global vs Local Allocation . . . . .	88
13.7 Thrashing . . . . .	88
13.7.1 Cause of thrashing . . . . .	88
13.7.2 Working-Set Model . . . . .	88
13.8 Other Considerations . . . . .	88
13.8.1 Prepaging . . . . .	88
13.8.2 Page Size . . . . .	88
13.8.3 Program Structure . . . . .	88
13.9 Demand Segmentation . . . . .	88
<b>V File Management</b>	<b>89</b>
<b>14 OS8 - File systems</b>	<b>91</b>
<b>15 File System Interface</b>	<b>93</b>
15.1 The File Concept . . . . .	93
15.2 The Directory Concept . . . . .	95
15.2.1 Unix . . . . .	95
15.3 Consistency Semantics . . . . .	95
15.3.1 Unix Semantics . . . . .	95
15.3.2 Session Semantics . . . . .	95
15.3.3 Immutable-Shared-Files Semantics . . . . .	95
<b>16 File System Implementation</b>	<b>97</b>
16.1 File-System Structure . . . . .	97
16.1.1 File System Organization . . . . .	97
16.1.2 File-System Mounting . . . . .	98

16.2 Allocation Methods . . . . .	98
16.3 Free-Space Management . . . . .	98
16.4 Directory Implementation . . . . .	98
16.5 Efficiency and Performance . . . . .	99
 <b>VI Other</b>	 <b>101</b>
<b>17 OS7 - Security and protection</b>	<b>103</b>
17.1 Overview of system security . . . . .	104
17.2 Policy/mechanism separation . . . . .	104
17.3 Security methods and devices . . . . .	104
17.4 Protection, access, and authentication . . . . .	104
17.5 Models of protection . . . . .	105
17.6 Memory protection . . . . .	106
17.7 Encryption . . . . .	106
17.8 Recovery management . . . . .	106
17.9 Trusted systems . . . . .	107
17.9.1 Covert channels . . . . .	107
17.10References . . . . .	107
 <b>18 OS9 - Real-time and embedded systems</b>	 <b>109</b>
 <b>19 OS10 Fault tolerance</b>	 <b>111</b>
 <b>20 OS11 System performance evaluation</b>	 <b>113</b>
 <b>21 OS12 Scripting</b>	 <b>115</b>
 <b>VII Computer Organization</b>	 <b>117</b>
<b>22 System Organization</b>	<b>119</b>
22.1 The von Neumann Architecture . . . . .	119
22.2 The Central Processing Unit (CPU) . . . . .	119
22.2.1 Basic Structure . . . . .	119
22.2.2 Interrupts . . . . .	121
22.2.3 Processor Modes . . . . .	122
22.3 Memory . . . . .	123
22.3.1 Memory Hierarchy . . . . .	123
22.3.2 Protected memory . . . . .	123
22.3.3 Paged Memory . . . . .	124
22.3.4 Virtual Memory . . . . .	124
22.4 Devices . . . . .	124
22.4.1 Disk Devices . . . . .	125
22.4.2 Addressing . . . . .	125



<b>23 Assembly Programming in GNU/Linux</b>	<b>129</b>
23.1 x86 Architecture and Assembly Instructions . . . . .	129
23.1.1 Programming Model . . . . .	129
23.1.2 AT & T Style Syntax (GNU C/C++ compiler and GAS)	130
23.1.3 Subroutines . . . . .	131
23.1.4 Data . . . . .	132
23.1.5 Data Transfer Instructions . . . . .	133
23.1.6 Arithmetic Instructions . . . . .	134
23.1.7 Logic Instructions . . . . .	135
23.1.8 Shift and Rotate Instructions . . . . .	135
23.1.9 Control Transfer Instructions . . . . .	136
23.1.10 String Instructions . . . . .	138
23.1.11 Miscellaneous Instructions . . . . .	139
23.1.12 Floating Point Instructions . . . . .	139
23.1.13 MMX Instructions . . . . .	139
23.1.14 System Instructions . . . . .	139
23.1.15 References . . . . .	140
23.2 x86 Assembly Programming . . . . .	140
23.2.1 Assumptions . . . . .	140
23.2.2 The G++ options . . . . .	140
23.2.3 Using GAS the GNU assembler . . . . .	140
23.2.4 Inline Assembly . . . . .	140
23.2.5 References . . . . .	143
 <b>VIII Minix Laboratory Projects for CE, CS, or SE</b>	 <b>145</b>
<b>24 Minix Project Information</b>	<b>147</b>
24.1 Design & documentation . . . . .	147
24.2 Packaging Your Software - <b>tar</b> . . . . .	148
24.3 Controlling Recompilation - <b>make</b> . . . . .	149
24.4 Version Control . . . . .	150
24.5 Alternate Projects . . . . .	150
 <b>25 Programming Project #1</b>	 <b>151</b>
25.1 Purpose . . . . .	151
25.2 Basics . . . . .	151
25.3 Details . . . . .	152
25.4 Deliverables . . . . .	153
25.5 Hints . . . . .	153
25.6 Project groups . . . . .	154
25.7 shell.l . . . . .	155
25.8 myshell.c . . . . .	155

<b>26 Programming Project #2</b>	<b>157</b>
26.1 Purpose . . . . .	157
26.2 Basics . . . . .	157
26.3 Details . . . . .	158
26.3.1 Lottery Scheduling . . . . .	159
26.3.2 Dual Round-Robin Queues . . . . .	160
26.4 Deliverables . . . . .	161
26.5 Hints . . . . .	161
26.6 Project groups . . . . .	163
26.7 longrun.c . . . . .	163
<b>27 Programming Project #3</b>	<b>165</b>
27.1 Purpose . . . . .	165
27.2 Basics . . . . .	165
27.3 Details . . . . .	166
27.3.1 Deliverables . . . . .	167
27.4 Hints . . . . .	167
27.5 Project groups . . . . .	168
<b>28 Programming Project #4</b>	<b>169</b>
28.1 Purpose . . . . .	169
28.2 Basics . . . . .	169
28.3 Deliverables . . . . .	170
28.4 Hints . . . . .	171
28.5 Project groups . . . . .	172
<b>IX Simulated System Laboratory Projects for CE, CS, or SE</b>	<b>173</b>
<b>29 Operating System Project &amp; Labs</b>	<b>175</b>
29.1 Lab 0: EWD and SM . . . . .	176
29.2 Lab 1: Modify the SM to support the following: . . . . .	176
29.3 LAB 2: Create a simple nucleus of an operating system for SM . . . . .	177
29.4 LAB 3: Hard Drive and I/O . . . . .	178
29.5 LAB 4: File System, Memory Manager, and Swap System . . . . .	178
29.6 LAB 5: Memory Manager . . . . .	179
29.7 LAB 6: Multiprocessor Management . . . . .	179
<b>X Laboratory Projects for IS or IT</b>	<b>181</b>
<b>30 IS/IT OS Project</b>	<b>183</b>

# Preface

Coordination with course text (Tanenbaum & Woodhull *Operating Systems Design and Implementation 3rd ed.*):

The Lecture notes focus on operating system concepts. Students are expected to read the Minix portions of the text in order to do the laboratory exercises.



# Assignments

## Exercises from the course textbook

The exercises are selected from the course textbook: Tanenbaum & Woodhull *Operating Systems Design and Implementation 3rd ed.*. Each student is expected to turn in their own solutions to the assignment. However, students may collaborate in the production of solutions and are expected to list their collaborators including internet resources.

Chap.	Pages	Assignment	Due Date
1	52-54	Do any 10 problems	Wed, 2nd week of classes
2	215-220	Select 10 problems, at least one from each decade	Wed, 4th week of classes
3			Wed, 6th week of classes
4	476-480	Select 10 problems at least one from each decade	Wed, 8th week of classes
5			Wed, 10th week of classes

## Laboratory projects

These lecture notes include several sets of laboratory exercises. Students are expected to select laboratory exercises appropriate to their major – CE, CS, IS, IT, or SE. Consult the appropriate chapters in these lecture notes for more details.

The Lecture notes focus on operating system concepts. Students are expected to read the Minix portions of the text and consult information available at <http://www.minix3.org> in order to do the laboratory exercises.

## Tests

Three to five tests are planned.

1. Process Management
2. Memory Management
3. File System Management

4. Protection and Security

5. Final (comprehensive)

Details to follow.

## Part I

# Introduction





# Chapter 1

## OS1: Overview of Operating Systems

*Suggested time:* 2 hours

*Topics:*

- Role and purpose of the operating system
- History of operating system development
- Functionality of a typical operating system
- Mechanisms to support client-server models, hand-held devices
- Design issues (efficiency, robustness, flexibility, portability, security, compatibility)
- Influences of security, networking, multimedia, windows

*Learning objectives:*

1. Explain the objectives and functions of modern operating systems.
2. Describe how operating systems have evolved over time from primitive batch systems to sophisticated multiuser systems.
3. Analyze the tradeoffs inherent in operating system design.
4. Describe the functions of a contemporary operating system with respect to convenience, efficiency, and the ability to evolve.
5. Discuss networked, client-server, distributed operating systems and how they differ from single user operating systems.
6. Identify potential threats to operating systems and the security features design to guard against them.
7. Describe how issues such as open source software and the increased use of the Internet are influencing operating system design.

## 1.1 A Computer System

A modern computer system consists of one or more processors, memory, timers, disks, printers, keyboard, pointing device (mouse), display, network interface, and other I/O devices.

### A Computer System

Banking system	Airline reservation	Web browser	<i>Application programs</i>
Compilers	Editors	Command Interpreter	<i>User mode system programs</i>
<b>Operating System</b>			<i>Kernal mode system programs</i>
Machine Language			<i>Hardware</i>
Microarchitecture			<i>Hardware</i>
Physical devices			<i>Hardware</i>

Each level provides an interface or virtual machine that is easier to understand and use.

## 1.2 Role and Purpose of the OS

An operating system is the most fundamental system program. It controls all of the system's resources and provides a base upon which application programs can be written. From the use perspective, an OS provides

1. a platform for user applications (process management)
2. communication management (device management)
3. data storage (file system management)

The most conservative definition of an OS is to limit it to the software that must run in kernal (or supervisor) mode<sup>1</sup>. Large monolithic operating systems place most of their services in the kernel while modular systems place most of their services in the user mode.

In these lecture notes the user interface is not part of the operating system, whethethat r it is a GUI such as MSWindows or a text mode CLI (command line interface).

Questions:

- What are the design differences between operating systems designed for personal applications on palm tops or PCs and operating systems designed for enterprise computing?
- How do OSes change over time?

---

<sup>1</sup>At one point, MicroSoft argued that their browser was part of the operating system.

Themes -

- Virtual machines, layering, & levels of abstraction
- Resource management
- Liveness - something good happens (fairness)
- Safety - nothing bad happens (security and protection)

### 1.2.1 Provide a service for clients - a virtual machine (top-down view)

An *operating system* is a layer of the computer system (a virtual machine) between the hardware and user programs.

- Multiple processes
- Multiple address spaces
- File system

### 1.2.2 Resource manager (bottom-up view)

An *operating system* is a *resource manager*. The operating system provides an orderly controlled allocation of processors, memory, and I/O devices among the various process competing for access. This is a consequence of the fact that for the integrity of the task, a process must have exclusive access to the resource.

Client Process <sub>0</sub>	...		Client Process <sub>n</sub>
Operating System			
CPU	Memory	Files	I/O De- vices

Resource management includes

- Scheduling resources - when and who gets a resource (cpu, devices, memory block)
- Transforming resources - to provide an easier to use version of a resource (disk blocks vs file system, device drivers)
- Multiplexing resources - create the illusion of multiple resources (cpu, spooled printing, swap space)

The hardware resources it manages include

- Processors - process management system
- Memory - memory management system

- I/O devices - I/O system
- Disk space - file system

Management values:

- Community values:
  - Stable, reliable, predictable, efficient
- Process values:
  -

The hardware resources are transformed into virtual resources so that an operating system may be viewed as providing a **virtual computers**, one to each user. A **virtual machine** (top down view) consists of

- Processes - virtualization of the computer including a virtual processor that abstracts the cpu - user mode instruction set + system calls
- Virtual memory - virtualization of physical memory
- Logical devices - virtualization of physical devices
- Files - virtualization of disk space

### 1.2.3 The general functions of an operating system

- Allocation - assigns resources to processes needing the resource
- Accounting - keeps track of resources - knows which are free and which process the others are allocated to.
- Scheduling - decides which process should get the resource next.
- Protection - makes sure that a process can only access a resource when it is allowed

Basic Functions

- Process management
- Resource management
  - Device management
  - Memory management
  - File management

## 1.3 History of OS Development

The digital computer

- Analytical engine - Charles Babbage (1792-1871)
- Vacuum tubes and plugboards (1945-55) - Aiken, von Neumann, Ecker, Mauchley, Zuse
- Transistors and batch systems (1955-65)
- ICs and multiprogramming (1965-1980) - OS/360, timesharing, MULTICS, UNIX
- Personal computers (1980- ) - CP/M, DOS, GUI, X, MS-NT.

Operating systems

- Mainframe
  - large I/O capacity
  - provide batch, transaction, and timesharing services
  - high reliability
- Server - file, print, web
- Personal computer
- Real-time - temporality is a key design factor
  - soft real-time - multimedia systems
  - hard real-time - assembly line, nuclear power station
- Embedded systems - PDA, control devices
- Smart-card OS

## 1.4 Functionality of a Typical OS

Functionality - system requirements

- Support for separate activities
- Manage multiple clients and hardware devices
- Manage cooperative access to resources
- Manage competition for resources
- Support for concurrent access to data structure while maintaining data invariants

- Support for composite tasks with potentially interfering subtasks and possible failures

Typical services

- Job sequencing
- Job control language
- Error handling
- I/O
- Interrupt handler
- Scheduling
- Resource control
- Protection
- Multi-access
- Security
  - Memory protection
  - File access control (authorization)
  - Authentication (secure establishment of identity)
  - Secure communication (encryption)

## 1.5 Mechanisms to Support Client-Server Models, Hand-held Devices

- Processes (sockets)
- Communication
  - Network
  - Infrared
  - Bluetooth
  - etc

Operational Qualities	Maintenance Qualities
<ul style="list-style-type: none"> <li>• Functionality: suitability, accuracy, interoperability, security, compliance</li> <li>• Reliability: maturity, fault tolerance, recoverability compliance</li> <li>• Usability: understandability, learnability, operability, attractiveness, compliance</li> <li>• Efficiency: time behavior, resource utilization, compliance</li> </ul>	<ul style="list-style-type: none"> <li>• Maintainability: analyzability, changeability, stability, compliance</li> <li>• Portability: adaptability, installability, co-existence, replaceability, compliance</li> </ul>

Figure 1.1: ISO 9126 Software Quality Description and other Software Quality Factors

## 1.6 Design Issues

An Approach to Design<sup>2</sup>

<i>Philosophy</i>	Ontology, epistemology, axiology (values)
<i>Policy</i>	An implementation plan
<i>Procedures</i>	Key algorithms
<i>Mechanisms</i>	Used to realize (implement) the algorithms

The values in a design philosophy include those identified in the ISO 9126 Software Quality Characteristics (See Figure 1.1). Specific to operating systems are the following:

- Efficiency - resources should be used as much as possible
- Robustness -
- Flexibility -
- Portability -
- Compatibility -
- Fairness - processes should get the resources they need
- Absence of deadlock or starvation - no process should wait forever for a resource.

<sup>2</sup>IS and IT Majors should notice the resemblance to the strategic management process.

- Protection/security - no process should be able to access a resource without permission

*Question:* which of these goals/issues are safety properties and which are liveness properties?

Functional factors

- Performance
- Integrity
  - Protection & security
    - \* *Security*: Protection against unauthorized disclosure, alteration, or destruction - protection against unauthorized users.
    - \* An organization's *security policy* defines the rules for authorizing access to computer and information resources.
  - *Security objectives*:
    - \* *Secrecy*: Information should not be disclosed to unauthorized users - protection against authorized users.
    - \* *Integrity*: Maintain the accuracy or validity of data - protection against authorized users
    - \* *Availability*: Authorized users should not be denied access.
  - The computer's *protection mechanisms* are tools for implementing the organization's security policy.
- Correctness - based on requirements
- Maintainability - designed for evolution

## 1.7 Influences of Security, Networking, Multimedia, Windows

- Economics - hardware cost
- Open source community -
- Commercial influence -
- Standards
  - Application interface
  - portability
  - interoperability
- Environmental factors



## Chapter 2

# OS2: Operating System Principles

*Minimum core coverage time:* 2 hours

*Topics:*

- Structuring methods (monolithic, layered, modular, micro-kernel models)
- Abstractions, processes, and resources
- Concepts of application program interfaces (APIs)
- Application needs and the evolution of hardware/software techniques
- Device organization
- Interrupts: methods and implementations
- Concept of user/system state and protection, transition to kernel mode

*Learning objectives:*

1. Explain the concept of a logical layer.
2. Explain the benefits of building abstract layers in hierarchical fashion.
3. Defend the need for APIs and middleware.
4. Describe how computing resources are used by application software and managed by system software.
5. Contrast kernel and user mode in an operating system.
6. Discuss the advantages and disadvantages of using interrupt processing.
7. Compare and contrast the various ways of structuring an operating system such as object-oriented, modular, micro-kernel, and layered.
8. Explain the use of a device list and driver I/O queue.

## 2.1 Elements

- Processes
  - Process table
  - Process: address space, process table entry
  - Process management system calls
    - \* Process creation
    - \* Interprocess communication
    - \* Alarm Signal SIGALRM (process interrupts)
    - \* Process termination
- Files
- Shell

## 2.2 Structuring Methods

An *operating system* usually consists of the following software components

- User interface (command interpreter)
- Programming interface (system calls)
- Process manager
- Memory manager
- File manager
- Interrupt handler
- Device drivers

which supply the following

- Processor modes
- Kernels
- Requesting services from the OS
  - System call - call, trap, return
  - Message passing - send, receive

General purpose operating systems typically have four major components:

1. process management
2. I/O device management
3. memory management
4. file management

### 2.2.1 Monolithic systems

The structure that has no structure ... basic structure:

- Main program that invokes the requested services procedure
- A set of service procedures that carry out the system calls
- A set of utility procedures

#### Monolithic System

Main Procedure
Service Procedures
Utility Procedures

### 2.2.2 Layered systems

Examples: THE, MULTICS

Layer	Function
5	The operator
4	User programs
3	I/O management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

### 2.2.3 Virtual machines

Examples: CP/CMS-VM/370, DOS under Windows, Java Virtual Machine

#### VM/370 with CMS

	Virtual 370	Virtual 370	Virtual 370	<i>System calls here</i>
<i>I/O instructions here</i>	CMS	CMS	CMS	<i>trap here</i>
<i>trap here</i>	VM/370			
	370 Bare hardware			

Kernel mode

### 2.2.4 Exokernels

Exokernel - functionality limited to protection and multiplexing of resources - allocation of resources to virtual machines.

### 2.2.5 Client-server

Move as much code out of the kernel as possible

<i>User Mode</i>	Client Process	Client Process	Process Server	Terminal Server	...	File Server	Memory Server
<i>Kernel mode</i>	Microkernel						

Servers may be local or distributed.

Machine 1		Machine 2		Machine $n$
Client		Server		Server
Kernel		Kernel		Kernel
The Network				

### 2.2.6 OS Research

- Microkernels
- Extensible operating systems
- Exokernels

## 2.3 Abstractions, processes, and resources

Abstract model of computing

The OS provides an abstraction of the hardware that is easier to use.

### 2.3.1 Resources

Resources

- are requested by processes from the OS
- the process must suspend its operation until it receives the resource

Common resources

- Files
- Other resources
  - CPU
  - Memory
  - I/O devices

### 2.3.2 Processes

A *process* is a sequential program in execution and consists of

- the *object program* (or *code*) to be executed
- the *data* on which the program will execute
- *resources* required by the program
- the *status* of the process's execution

Abstract machine

Process creation

- fork
- quit
- join

Process scheduler - ready, running, blocked

### 2.3.3 Threads

A *thread* (*lightweight process*) is an entity that executes using the program and other resources of its associated process - there can be several threads associated with a single process.

Single program multiple data programming model.

Thread state consists of the process state plus the thread stack, some process status information, and OS table entries.

Thread scheduler

Minimal context switching time.

### 2.3.4 Objects

Simulation - OOP

## 2.4 Concepts of APIs

## 2.5 Application needs and evolution of techniques

## 2.6 Device Organization

## 2.7 Interrupts: methods and implementations

## 2.8 User and System State



# Part II

## Processes





## Chapter 3

# OS3: Concurrency

*Minimum core coverage time:* 6 hours

*Topics:*

- States and state diagrams
- Structures (ready list, process control blocks, and so forth)
- Dispatching and context switching
- The role of interrupts
- Concurrent execution: advantages and disadvantages
- The "mutual exclusion" problem and some solutions
- Deadlock: causes, conditions, prevention
- Models and mechanisms (semaphores, monitors, condition variables, rendezvous)
- Producer-consumer problems and synchronization
- Multiprocessor issues (spin-locks, reentrancy)

*Learning objectives:*

1. Describe the need for concurrency within the framework of an operating system.
2. Demonstrate the potential run-time problems arising from the concurrent operation of many separate tasks.
3. Summarize the range of mechanisms that can be employed at the operating system level to realize concurrent systems and describe the benefits of each.
4. Explain the different states that a task may pass through and the data structures needed to support the management of many tasks.

5. Summarize the various approaches to solving the problem of mutual exclusion in an operating system.
6. Describe reasons for using interrupts, dispatching, and context switching to support concurrency in an operating system.
7. Create state and transition diagrams for simple problem domains.
8. Discuss the utility of data structures, such as stacks and queues, in managing concurrency.
9. Explain conditions that lead to deadlock.

*Concepts*

States and state diagrams

Structures (ready list, process control blocks, and so forth)

Dispatching and context switching

The role of interrupts

Concurrent execution

The “mutual exclusion” problem

Deadlock: causes, conditions, prevention

Models and mechanisms (semaphores, monitors, rendezvous)

Producer-consumer problems

# Chapter 4

## Processes

### 4.1 Process Concept

#### 4.1.1 Definition

A *program* is

- a sequence of instructions
- a static object that can exist in a file

A *process* is

- a sequence of instruction executions
- a dynamic object – a program in execution - code and data, program counter, register contents, and the process stack (return address, parameters, local variables).

*Multiprogramming* is the sharing of the CPU through switching back and forth between processes.

*Parallel execution* requires either multiple pipelines in the CPU or multiple CPUs.

Types of processes (by resource use)

- Independent: process cannot affect or be affected by the other processes
  - Dependent: processes can affect or be affected by the other processes
- There are several subcategories

- cooperating – share resources to accomplish a task
- competing – may starve opponent
- hostile – attempt to destroy another's resources

### 4.1.2 Goal/Rationale

Why do we need multiple processes?

- simplify programming when a task naturally decomposed into multiple processes
- permit the full use of CPU cycles
- support multiple tasks/users

### 4.1.3 Design

- Types: code and data, program counter, registers, process stack.
- Functions
  - *Process Creation*: parent - child - tree of processes  
child needs resources (cpu time, memory, files, i/o devices) either from parent or OS; initialization data from parent.
    - \* System initialization.
      - foreground processes - interact with the user
      - background processes - *daemons*.
    - \* Execution of a process creation system call by a running process.
    - \* User request to create a new process.
    - \* Initiation of a batch job.

Examples:

- \* Unix: a hierarchy of processes
- \* Windows: all processes are equal except when a parent creates a child but ownership of the child process may be passed to other processes.

– *Execution*:

- \* Parent executed concurrently with child
- \* Parent waits until some or all of its children have terminated

Address space:

- \* Child process has a duplicate of the parent process (Unix)
- \* Child process has a program loaded into it

Examples:

- \* MS-DOS: sys call to load a binary file and execute it as a child process. Parent suspends until child exits.
- \* UNIX: processes created with the `fork()` system call which creates an identical copy of the calling process. Parent and child execute in parallel. Note that FORK – returns 0 to the child and the child's pid to the parent; child can call `execve` to replace memory space with a new program; Parent can call `wait` to remove itself from the queue until child has terminated.

- \* DEC VMS: create, load program, and start it running
- \* MS WindowsNT: supports both the UNIX and DEC VMS models.
- *Process Termination* A process terminates by using the EXIT system call returning data to the parent process via the FORK system call.
  - \* Normal exit
  - \* Error exit (voluntary)
  - \* Fatal error (involuntary)
  - \* Killed by another process (involuntary)

A process may be terminated by an ABORT system call – usually only by parent process. Reasons:

- \* Child has exceeded its usage of some of the resources
- \* Task assigned to child is no longer needed
- \* Parent is exiting and OS does not allow children to play unsupervised

Idealized OS structure: Process structured OS–

Processes			
0	1	...	n-1
Scheduler			

#### 4.1.4 Implementation

Processes generate system calls and hardware generates interrupts.

A process is in one of 5 states:

- New: the process is being created
- Running: instructions are being executed
- Blocked (Waiting): process is waiting for an event to occur
- Ready: waiting for a processor
- Terminated: process has finished execution

Data structures

- Process table (of process control blocks) each which contains the following information
  - Process Management
    - \* Process State
    - \* Program Counter
    - \* CPU Registers
    - \* CPU Scheduling Information: priority, pointers to queues, ...

- \* Accounting Information: cpu time used, account numbers, job or process numbers, ...
- Memory Management
  - \* Base and limit registers
  - \* Page table
  - \* Segment tables (data segment, code segment)
- File Management
  - \* I/O devices allocated to the process
  - \* Root and working directory
  - \* File descriptors of open files
- Interrupt vector - contains the addresses of the interrupt service procedures. *Context Switch* – saving the state of the old process and loading the saved state of the new process. Interrupt/SysCall(Trap) processing (actual details vary)
  1. Hardware
    - Interrupt hardware saves some user process information on the stack and
    - loads the `pc` with handler address from interrupt vector
  2. Assembly routine
    - saves user data to PCB
    - sets up new stack, and
    - calls the interrupt handler
  3. Interrupt handler runs (reads and buffers input) and possibly marks some blocked process as ready then calls the scheduler
  4. Scheduler selects next process returns
  5. Handler returns to assembly code
  6. Assembly routine loads `pc` and registers from PCB of next process and starts the process The *dispatcher* is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves:
    - Switching context
    - Switching to user mode
    - Jumping to proper location in user program to restart the program.

Must be FAST – *dispatcher latency* – time to complete action.

```

dispatcher() { ... }

scheduler() { ... }

interruptHandler() { ... }

sysCall () { ... }

```

## 4.2 Threads

A process has an address space and a single thread of control.

*Threads* (light-weight processes) are processes that share an address space (code and data). That is, there are multiple threads of control and a single address space.

Each thread has a program counter, registers, and run-time stack.

### 4.2.1 Goal/Rationale

Cooperating (dependent or competing) processes can be affect or be affected by the other processes. There are several advantages

- Modularity - task oriented program decomposition
- Information sharing - shared address space
- Computation speedup - on a multithreaded CPU or multiCPU environment.
- Convenience - program need not be designed as a sequential process
- Context switching with threads may be upto 100 times faster than with processes.

They can be organized in several ways, Dispatcher-Worker, Team, or Pipeline Example

Producer-consumer p(110) (busy waiting) bounded; unbounded buffer  
 With cooperating processes (non-competing or hostile processes),  
 the implementation may be simplified to shared address space i.e. a  
 single address space with multiple threads of control.

Examples: Mach, OS/2, Solaris 2 - The program counter, register contents  
 and the process stack are a thread.

### 4.2.2 Thread Structure

- *task* a collection of threads

- *thread* or *lightweight process (LWP)* pc, register set, stack space; shares code, data and OS resources
- *heavy weight process* task with one thread
- *user-level threads* thread switching does not call OS – fast!

### 4.2.3 Design

- Mutex
- Scheduling
- Critical regions (mutex + condition variables)
- Global variables

### 4.2.4 Implementation

- User level
- Kernel level

### 4.2.5 Solaris 2

## 4.3 Exercises

Analysis, Design, Implementation

1. Find out what processes are running on your favorite system. Which processes would you combine or eliminate?
2. Write a program that forks a child process and then both the parent and child print 20 copies of their pids.
3. Implement a multiple process executive



# Chapter 5

## Deadlock

*One hour of lecture is allocated to this chapter.*

“When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.”

*–Kansas law early 20th century*

### 5.1 Basic Concepts

Resource utilization sequence

1. Request and wait to acquire the resource
2. Use the resource
3. Release the resource

A *preemptable resource* is a resource that can be taken away from the process owning it with no ill effects e.g. CPU, memory.

*Non-preemptable resources* are resources that can only be used by one process at a time e.g. printer, tape drive, slot in I-node table, ethernet, database record.

A set of processes is *deadlocked* if each process in the set is waiting for an event that only another process in the set can cause. Events that may require a process to wait include semaphore, message, data item.

Necessary and sufficient conditions for deadlock

1. Mutual Exclusion: the resources are not shareable
2. Hold and Wait: at least one process that is holding a resource and waiting to acquire more.
3. No preemptions: the resources are non-preemptive
4. Circular Wait: P<sub>0</sub> waiting for P<sub>1</sub> waiting for ... waiting for P<sub>n</sub> waiting for P<sub>0</sub>

### 5.1.1 Resource-Allocation Graph

- Represent a process as a circle.
- Represent a resource type as a rectangle.
- Directed edge from process to resource indicates a request.
- Directed edge from resource to process indicates allocation. If the graph is cycle free then there is no deadlock.
- If the graph contains a cycle then it may be deadlocked.

## 5.2 Methods for handling deadlocks

- Ignore/Ostrich Algorithm
- Prevent
- Avoid
- Recover (detect and recover)

## 5.3 Ignore/Ostrich Algorithm

Unix

## 5.4 Deadlock Prevention

Prevent one of the 4 conditions for deadlock from holding:

- *Mutual Exclusion* spool non-shareable resources
- *Hold and Wait* request all resources initially; may request when none
- *No Preemption* preempt when waiting; allocate if available (may preempt waiting processes)
- *Circular Wait* impose total ordering on resources and require requests in increasing order (PROOF)

## 5.5 Deadlock Avoidance

One algorithm: each process declares max number of resources of each type

### 5.5.1 Safe State

safe – unsafe – deadlock

### 5.5.2 Resource-Allocation Graph Algorithm

### 5.5.3 Banker's Algorithm

## 5.6 Deadlock Detection and Recovery

### Detection

- Single instance of each resource; collapse resource allocation graph into wait-for graph and do a depth first search with checks for cycles
- Multiple instances of a resource type; Key idea: sequentially for each process  $P_i$ : if its requested resources is less than available resources, then delete the process from the list and add its allocated resources to available resources. Any processes remaining are deadlocked.
- Recover
- Detection-Algorithm Usage
  - for each request (determines *cause* of deadlock but is expensive)
  - at random times (cannot determine cause of deadlock)

### Recovery

- Process termination
  - Abort all deadlocked processes
  - Abort one process at a time until the deadlock cycle is eliminated
- Resource preemption Issues in preemption
  - Selecting a victim (minimize cost)
  - Rollback to safe state and restart
  - Starvation, will it always be the same process?

## 5.7 Combined Approach to Deadlock Handling

## 5.8 Exercises

### Analysis, Design, Implementation

1. Write a program to find cycles in a graph.
2. Implement the Resource-Allocation graph algorithm
3. Implement the Banker's algorithm.



## Chapter 6

# Synchronization and Communication

*Four hours of lecture are allocated to this chapter.*

### 6.1 Basic Concepts

**Race condition** A situation where the outcome of the execution depends on the particular order of execution is called a *race condition* (usually occurs when two processes (or database transactions) modify the same variable). Examples: Producer-consumer (bounded buffer) with a counter. Spooler with table of print jobs. Database update.

**Critical Section** The segment of code in which a process changes a shared data structure is called a *critical section*. The critical section of code is bounded by an *entry* section and followed by an *exit* section.

```
...  
code to enter critical section  
    critical section code  
code to exit critical section  
...
```

**Safety Property** nothing bad happens

**Liveness Property** something good happens

**Critical Section Problem** The *critical section problem* is to design a protocol that the processes can use to cooperate so that race conditions do not result. That protocol must satisfy the following:

1. Mutual Exclusion: no two processes may be simultaneously inside their critical sections. (safety property)

2. Progress: if a process attempts to enter its critical section and there are no processes executing in their critical section, then eventually the process will enter its critical section. (liveness property)
3. Bounded waiting: there must be a bound on the number of processes that enter their critical section before a waiting process gains access to its critical section. (liveness property)

The last two properties may be combined to state that if a process is waiting to enter its critical section, eventually it will enter its critical section. Assumptions:

- processes execute at a non-zero speed
- no assumption regarding relative speed
- basic machine-language instructions are executed atomically.

## 6.2 Software Solutions

### 6.2.1 Two-Process "Solutions"

1. Gain access to the shared variable and lock out the other process.

```
boolean locked := false;

P0, P1:
    while locked do;
    locked := true;
    critical section;
    lock := false
    non-critical section;
```

violates mutual exclusion (both test, set to true and enter critical section)

2. Take turns using the shared variable.

```
turn := 0;

P0:                                     P1:
    while turn = 1 do;                   while turn = 0 do;
    critical section;                     critical section;
    turn := 1;                             turn := 0;
    non-critical section                  non-critical section;
```

violates progress condition (due to strict alternation)

3. Announce intentions and check to see if the other process is using the variable.

Declarations

```
boolean ready[1];
```

Initialization

```
ready[0] := ready[1] := false;
```

P0:

```
ready[0] := true;
while ready[1] do;
critical section;
ready[0] := false;
```

P1:

```
ready[1] := true;
while ready[0] do;
critical section;
ready[1] := false;
```

violates progress condition (when both set flags to true)

### G. L. Peterson's Solution 1981

(Dekker, a Dutch mathematician, provided the first solution but, Peterson's is simpler)

Declarations

```
boolean ready[1];
integer turn;
```

Initialization

```
ready[0] := false;
ready[1] := false;
```

```
procedure enter( integer process )
```

```
integer other = 1 - process;
```

```
ready[process] := true;
```

```
turn := other;
```

```
// flag to let other in when finished
```

```
while turn = other and ready[other] do; // other is in critical section
```

```
end;
```

```
procedure exit ( integer process )
```

```
ready[process] := false;
```

```
end;
```

### CORRECTNESS PROOF

**Mutual exclusion** Assume that both P0 and P1 are in their critical sections

P0 in critical section implies: ready[0] and (turn=0 or not ready[1])

P1 in critical section implies: ready[1] and (turn=1 or not ready[0])

but turn cannot be both 0 and 1 therefore, both P0 and P1 cannot be in their critical sections.

**Progress and bounded waiting** Assume that a process  $P_i$  is stuck waiting i.e.  $\text{turn} = 1-i$  and  $\text{ready}[1-i]$  (do case analysis on  $P_{1-i}$ )

## 6.2.2 Multiple-Process Solutions

the *bakery algorithm* overhead OSC fig 6.5

## 6.3 Hardware Solutions

- Disable interrupts
- Test-and-Set

```
function Test-and-Set (var target: boolean): boolean;
    Test-and-Set := target;
    target := true;
end;
```

Example:

```
lock := false; // initialization
...
while Test-and-Set( lock ) do;
    critical section;
    lock := false;
...

```

violates bounded waiting since faster processes may prevent slower processes from gaining access.

- Swap

```
procedure Swap(var a, b: boolean)
    var temp : boolean;
    temp := a;
    a := b;
    b := temp;
end;
```

Example:

```
lock := false; // initialization
...
key := true;
```



```

repeat
    Swap( lock, key );
until key = false;
critical section;
lock := false;
...

```

violates bounded waiting since faster processes may prevent slower processes from gaining access.

- Complete solution using test-and-set

```

Declarations
    var waiting : array [0..n-1] of boolean; //one slot for each process
                                              //initialized to false

    lock : boolean := false;

procedure Enter( process : integer );
    var key : boolean := true;
    waiting[process] := true;
    while (waiting[process] and key) do key := test-and-Set(lock);
    waiting[process] := false;
end;

procedure Exit( process : integer );
    var j : integer := i + 1 mod n;
    while (j <> i) and (not waiting[j]) do j := j+1 mod n; if j=i then lock :=false else w

```

#### CORRECTNESS PROOF

These solutions require busy waiting and are not easy to generalize to more complex problems. In the following sections, we examine primitives which block instead of wasting CPU time when they cannot enter their critical sections. A process can change its state to Blocked (waiting for some condition to change) and can signal Blocked processes so that they can continue.

In this case, the OS must provide the system calls BLOCK and WAKEUP.

## 6.4 Semaphores

### 6.4.1 Definition

A semaphore is an integer variable that is accessed through two atomic operations: DOWN and UP (WAIT and SIGNAL).

### 6.4.2 Goal/Rationale

### 6.4.3 Design

Spinlock version of a semaphore

```
S := 0;

down(S): while S <= 0 do; // wait S :=S-1; up(S): S :=S+1 // signal
```

Blocking version of a semaphore

```
type semaphore = record
    value : integer;
    L : list of processes; // or queue blocked waiting for
end;                      // the signal

down(S): S.value := S.value - 1; // wait
    if S.value < 0 then
        add this process to S.L;
        block;
    end;

up(S): S.value := S.value + 1;    // signal
    if S.value <= 0 then
        remove a process P from S.L;
        wakeup(P);
    end;
```

### 6.4.4 Implementation

*Single processor:* The normal way is to implement the semaphore operations (up and down) as system calls with the OS disabling the interrupts while executing the code.

*Multiprocessor:* Each semaphore should be protected by a lock variable, with the TSL instruction used to be sure that only one CPU at a time examines the semaphore. Using the TSL instruction to prevent several CPUs from accessing the semaphore at the same time is different from busy waiting.

### 6.4.5 Usage

### 6.4.6 Deadlocks and starvation

### 6.4.7 Binary Semaphores

See OSC p. 180

## 6.5 Classical Problems of Synchronization

### 6.5.1 Bounded Buffer

(Models race conditions) There is a pool of  $n$  buffers that are filled by a producer process and emptied by a consumer process. The problem is to keep the producer from overwriting full buffers and the consumer from rereading empty buffers.

### 6.5.2 Dining Philosophers

(Models exclusive access to limited resources) Five philosophers spend their lives seated around a circular table thinking and eating. Each philosopher has a plate of spaghetti and, on each side, shares a fork his/her neighbor. To eat, the philosopher must acquire two forks. The problem is to write a program that lets each philosopher eat and think.

### 6.5.3 Readers and Writers

(Models access to a database) A data object is shared among several concurrent processes. Some of which only want to read the content of the shared object, whereas others want to update (read and write) the shared object. The problem is insure that only one writer at a time has access to the object.

### 6.5.4 Sleeping Barber

The barber shop has one barber, a barber chair, and  $n$  chairs for waiting customers. The problem is to construct an appropriate simulation.

## 6.6 Critical Regions

### Definition

The *critical-region* high-level synchronization construct requires that a variable  $v$  of type  $T$ , which is shared among many processes, be declared as:

```
var v : shared T;
```

The variable can be accessed only inside a *region* statement of the form:

```
region v when B do S;
```

### Goal/Rationale

### Design

### Implementation

See OSC fig 6.16 for an implementation using semaphores

## 6.7 Monitors

### Definition

The *monitor* high-level synchronization construct provides access to shared variables through entry procedures and ensures that only one process at a time can be active within the monitor.

### Goal/Rationale

### Design

```

type monitor-name = monitor
  variable declarations including condition variables
    e.g. var x, y : condition; //wait and signal on these variables

  procedure entry P1(...);
    ...
  procedure entry PN(...);

  other procedures
begin
  initialization code
end.
```

### Implementation

See OCS p. 195 for an implementation using semaphores

### 6.7.1 Example: Solaris 2

## 6.8 Message Passing

The previous primitives are unsuitable for use in a multiprocessor environment with local memory and in a networked environment. In such an environment, message passing is used.

`send( message ), receive( message )`

*communication link*: Basic implementation questions

- How are the links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of processes?
- What is the capacity of a link? buffer space etc
- What is the size of messages? variable or fixed size

- Unidirectional or bidirectional?

Logical implementation

- Direct or indirect communication
- symmetric or asymmetric communication
- Automatic or explicit buffering
- Send by copy or by reference
- Fixed size or variable size

### Basic Structure

#### Naming

1. Direct Communication: Messages are exchanged between named processes

```
send(P, message) send message to process P
receive( Q, message ) receive message from process Q
```

Communication links have the following properties

- A link is established automatically; processes need to know each other's name
- A link is associated with exactly two processes
- Between each pair of processes, there exists exactly one link.
- The link may be unidirectional, but is usually bidirectional.
- Modularity is limited (changing a process name requires global search)

2. Indirect Communication: Messages are exchanged via *mailboxes* or *ports*.

Communication links have the following properties

- A link is established between processes if they share a mailbox.
- A link may be associated with more than two processes
- Between each pair of processes, there may exist more than one link.
- The link may be either unidirectional or bidirectional.

In the case of multiple receivers,

- all link to be associated with

#### Buffering

- Zero capacity – sender and receiver must synchronize *rendezvous*
- Bounded capacity – may delay sender
- Unbounded capacity – no delay

Other: *remote procedure call (RPC)*

**Exception Conditions**

1. Process termination
2. Lost messages
3. Scrambled messages

**Implementation**

See MOS p. 53, 54

**An Example: Mach**

## 6.9 Atomic Transactions

### 6.10 Exercises

Analysis, Design, Implementation

1. Pick a popular processor, what hardware instruction(s) are available to solve the critical section problem?
2. Implement shared memory communication using either
  - Peterson's algorithm
  - Test and Set
  - Semaphores
  - message passing
3. Show why the attempted solutions to the critical section problem are incorrect.
4. Construct solutions to the classical synchronization problems using
  - Semaphores
  - Critical regions
  - Monitors
  - Message passing

## Chapter 7

# OS4 - Scheduling and dispatch

*Minimum core coverage time:* 3 hours

*Topics:*

- Preemptive and nonpreemptive scheduling
- Schedulers and policies
- Processes and threads
- Deadlines and real-time issues

*Learning objectives:*

1. Compare and contrast the common algorithms used for both preemptive and non-preemptive scheduling of tasks in operating systems, such as priority, performance comparison, and fair-share schemes.
2. Describe relationships between scheduling algorithms and application domains.
3. Discuss the types of processor scheduling such as short-term, medium-term, long-term, and I/O.
4. Describe the difference between processes and threads.
5. Compare and contrast static and dynamic approaches to real-time scheduling.
6. Discuss the need for preemption and deadline scheduling.
7. Identify ways that the logic embodied in scheduling algorithms are applicable to other domains, such as disk I/O, network scheduling, project scheduling, and other problems unrelated to computing.

*Concepts:*

Preemptive and nonpreemptive scheduling

Schedulers and policies

Processes and threads

Deadlines and real-time issues



## Chapter 8

# CPU Scheduling

*Three hours of lecture are allocated to this chapter.*

### 8.1 Ontology: Basic Concepts

Process Behavior

CPU-I/O Burst Cycle: Typical process – large number of short CPU bursts and a small number of long CPU bursts – See OSC histogram Fig 5.2 p. 133

CPU Bound: mostly long CPU bursts

I/O Bound: mostly short CPU bursts

Gantt chart: see OSC p. 137

State diagram

	Ready	Ready, Suspended	Running	Blocked	Blocked, suspended	Exit
<b>New</b>	LTS					Sys
<b>Ready</b>		MTS	STS			Sys
Ready, sus- pended	MTS					Sys
Running	Quantum Consumed			Sys Call (I/O Re- quest)		Process, Sys
Blocked (Waiting)	Intrupt (I/O done)					Sys
Blocked, suspended				MTS		Sys
<b>Exit</b>						

Scheduling decisions may take place when a process switches from:

1. running to waiting
2. running to ready
3. waiting to ready
4. running to terminated

*Non-preemptive* scheduling: The current process keeps the CPU until it releases the CPU by either terminating or by switching to a waiting state.

Non-preemptive scheduling occurs only under 1 and 4 (MS-Windows); it does not require special hardware (timer).

*Preemptive* scheduling: The currently running process may be interrupted and moved to the ready state by the operating system. It requires

- special hardware (timer)
- mechanisms to coordinate access to shared data
- a kernel designed to protect the integrity of its own data structures. Some Unix systems wait for system calls to complete or for an I/O block to take place before a context switch (this does not support real-time processing). Further, interrupts may be disabled.

Policy versus Mechanism

- Policy - sets priority on individual processes
- Mechanism - implements a scheduling policy

Scheduling Queues

- Job Queue - incoming jobs
- Ready Queue -
- Device Queues - blocked processes

See OSC fig 4.5 - Queueing-diagram

Selection of a process from a queue is performed by a *scheduler*.

- Long-term (Job) Scheduler – active when a new job enters the system; most often found in batch systems; determines the degree of multiprogramming
  - Compute bound
  - I/O bound
- Medium-term Scheduler – active swaps jobs out to improve job mix
  - Time since swapped in or out
  - CPU time used
  - Size
  - Priority
- Short-term (CPU) Scheduler: Ready queue may be implemented as a FIFO queue, priority queue, a tree, or a linked list. (records in the queue are the PCBs)

## 8.2 Values - Scheduling Criteria (Goals)

- Fairness: each process gets its fair share
- Efficiency: CPU utilization

Variables

- Throughput: number of processes/time unit
- Turnaround: time it takes to execute a process from start to finish
- Waiting time: total time spent in the ready queue
- Response time: amount of time it takes to start responding (average, variance)

**Primary Value (General Purpose OS):** It is desirable to ensure that all processes get the cpu time they need and to *maximize* CPU utilization and throughput, and *minimize* turnaround time, waiting time, and response time. And may want to

- optimize the minimum or maximum (minimize maximum response time)
- minimize variance in response time (i.e. predictable response time)

## 8.3 Methods - Scheduling Algorithms

### 8.3.1 First-Come, First-Served (FCFS)

non-preemptive; average waiting time is dependent on order of arrival (consider cpu burst times and waiting time for each process). 24,3,3 vs 3,3,24

### 8.3.2 Shortest-Job-First (SJF)- shortest process next (SPN)

- Nonpreemptive
- SJF is really shortest next CPU burst
- Is provably optimal
- Use user estimated process time
- Shortest remaining time is a preemptive version
- Use approximation of next CPU burst (exponential average)

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

where

- $t_n$  is length of  $n^{th}$  CPU burst
- $\tau_i$  be the predicted value of the  $i^{th}$  CPU burst
- $\alpha$  is often set at 1/2 (equal weight to past and most recent activity)

The expansion of the formula explains why it is called the exponential average

### 8.3.3 Priority Scheduling

- External Priorities: political and economic factors.
- Internal Priorities: measureable quantity – time limits, memory requirements, open files, ave I/O burst/ave CPU burst,  $1/f$  where  $f$  is fraction of last quantum used (favors I/O bound processes)
- May be preemptive or non-preemptive
- Problem: *indefinite blocking* or *starvation*
- Solution: *aging*—gradually increase priority of waiting processes

### 8.3.4 Round-Robin (RR)

- FCFS with preemption – each process is given a *time quantum* or time slice.
- Ready queue is a FIFO queue implemented as a circular queue
- Time quantum too large = FCFS
- Time quantum too short = processor sharing and each job ( $n$  jobs) runs  $1/n$  the speed
- Time quantum should be large wrt context-switch time
- Turnaround time improves if most processes finish CPU burst within the time quantum
- 80
- Particularly effective for general-purpose time-sharing systems or transaction processing systems.
- Amount of processor time depends on length of CPU burst – a source of unfairness
- Virtual round robin: I/O bound processes given priority to finish our their quantum.

### 8.3.5 Multilevel Queue Scheduling

- Processes are easily classified into different groups
- Example: *foreground* (interactive), *background* (batch)
- *Multilevel queue-scheduling algorithm*: partition the ready queue into several queues – each with own scheduling algorithm and scheduling between queues – usually fixed-priority preemptive scheduling
- Example:
  1. system processes
  2. interactive processes
  3. interactive editing processes
  4. batch processes
  5. student processes
- queues have absolute priority
- queues have a fixed proportion of cpu time i.e. time slice between queues (foreground-background 80-20)

### 8.3.6 Multilevel Feedback Queue Scheduling

- *Multilevel Feedback Queue Scheduling* allows processes to move between queues
- Defined by the following parameters:
  - The number of queues
  - The scheduling algorithm for each queue
  - The method used to determine when to upgrade a process to a higher-priority queue
  - The method used to demote a process to a lower priority queue
  - The method used to determine which queue a process will enter when that process needs service.
- Most general—most complex

	FCFS	Round Robin	SJF	SRT	HRRN	Feedback
Selection Function	$\max[w]$	constant	$\min[s]$	$\min[s-e]$	$\max[(w+s)/s]$	see text
Decision mode	non-preemptive	preemptive	non-preemptive	preemptive	non-preemptive	preemptive at time quantum
Throughput	N/A	low for small quantum	high	high	high	N/A
Response time	May be high	Good for short processes	Good for short processes	good	good	N/A
Overhead	minimal	low	can be high	can be high	can be high	can be high
Effect on processes						
Starvation	No	No	Possible	Possible	No	Possible

$w$  = time spent in the system so far, waiting and executing

$e$  = time spent in execution so far.

$s$  = total service time required by the process, including  $e$ .

## 8.4 Multiple-Processor Scheduling

- *heterogeneous* set of processors – processes must be instruction set specific – as may be the case in distributed computing. (PCN – common intermediate code)
- *homogeneous* set of processors – *load sharing*
  - separate queue for each processor
  - common queue
  - *asymmetric multiprocessing* single processor for scheduling, I/O processing, & other system activities
  - *symmetric multiprocessing* either each processor is self-scheduling or a master-slave structure

## 8.5 Real-time Scheduling

- *Hard real-time* – complete a critical task within a guaranteed time – cannot have secondary storage or virtual memory
- *Soft real-time* – critical processes receive priority over less fortunate ones – can cause unfair allocation of resources and starvation, but can support
  - multimedia
  - high-speed interactive graphics
  - other speed critical tasks
- priority scheduling and real-time processes have highest priority and no aging
- dispatch latency must be small – preemption points in system calls or entire kernel is preemptible (Solaris 2)

## 8.6 Algorithm Evaluation

- define the criteria to be used – cpu utilization, response time, throughput
- evaluate the various algorithms.

### 8.6.1 Deterministic modeling

- *Analytic evaluation*: evaluate algorithm against workload
- *deterministic modeling*: predetermined workload
- too specific, too much exact knowledge



**8.6.2 Queueing models**

**8.6.3 Simulations**

**8.6.4 Implementation**

Clock, clock driver, interrupts

**8.7 Exercises**

Analysis, Design, Implementation



# Part III

## I/O



## Chapter 9

# OS6 - Device management

### *Topics:*

- Characteristics of serial and parallel devices
- Abstracting device differences
- Buffering strategies
- Direct memory access
- Recovery from failures

### *Learning objectives:*

1. Explain the key difference between serial and parallel devices and identify the conditions in which each is appropriate.
2. Identify the relationship between the physical hardware and the virtual devices maintained by the operating system.
3. Explain buffering and describe strategies for implementing it.
4. Differentiate the mechanisms used in interfacing a range of devices (including hand-held devices, networks, multimedia) to a computer and explain the implications of these for the design of an operating system.
5. Describe the advantages and disadvantages of direct memory access and discuss the circumstances in which its use is warranted.
6. Identify the requirements for failure recovery.
7. Implement a simple device driver for a range of possible devices.

### *Concepts*

Characteristics of a serial or parallel device

Buffering strategies

Free lists and device layout

Servers and interrupts

Recovery from failures

`;bLectures;/bL`

- `• |a href="ioSystems.html"|I/O Devices & Systems;/aL`

# Chapter 10

## I/O Devices and Systems

*Note: Two hours of lecture time is allocated to this chapter.*

### 10.1 Devices as files

- Block structured devices
- Character structured devices

### 10.2 I/O Devices

- Devices -
- Controllers -

#### 10.2.1 Disk Devices

Structure

- Spindle
- Platter/Surface
- Track (cylinder – tracks at same head position)
- Sector - A sector contains the smallest amount of information that can be read or written to the disk (32-4096 bytes)

Address

- Cylinder
- Track

- Sector
- $s$  = number of sectors per track
- $t$  = number of tracks per cylinder
- $i$  = cylinder number ( $0 - n_{\text{sub}_i i_i} / \text{sub}_i$ )
- $j$  = surface number ( $0 - n_{\text{sub}_i j_i} / \text{sub}_i$ )
- $k$  = sector number ( $0 - n_{\text{sub}_i k_i} / \text{sub}_i$ )
- $b$  = block number
- $b = k + s \cdot (j + i \cdot t)$

#### Timing

- Seek - move r/w head to cylinder
- Latency - wait for sector to rotate into position
- Transfer - data transfer

## 10.3 I/O Systems

### 10.3.1 Software

#### Context

- User process
- Device independent software (eg. filesystem)
- Device driver
- Interrupt handler
- Hardware (eg. disk drive & controller)

### 10.3.2 Disk Scheduling

#### FCFS Scheduling

First-come, first-served (fair but not efficient)

#### SSTF Scheduling

#### Shortest-seek-time-first

- efficient disk arm movement (half of FCFS)
- susceptible to starvation



**SCAN Scheduling**

Elevator Algorithm (reconciles conflicting goals of efficiency and fairness)

**C-SCAN Scheduling**

Circular Scan

**LOOK Scheduling**

Look for a request in that direction before moving

**Algorithm Selection**

- Rotational time is beginning to dominate (i.e. elevator algorithm is not as important)
- Algorithms are moving into hardware
- Raid technology & disk striping - parallel access

**10.3.3 Disk Management****Disk Formatting**

- Physical Formatting: track and sector marks; space for ECC(error correcting code)
- Logical Formatting: partition and an initial empty file system.

**Boot Block**

bootstrap program: initializes system and loads OS memory – stored in ROM and/or boot block of disk

**Error Handling**

- Programming error (e.g., request for non-existent sector) – should not occur but if it does, terminate disk request
- Transient checksum error (e.g., caused by dust on the head) – retry
- Permanent checksum error (e.g., disk physically damaged) – mark as bad and substitute a new block
- Seek error (e.g. the arm sent to sector 6 but went to sector 7) – recalibrate
- Controller error (e.g., controller refuses to accept commands) –

**Bad Blocks**

- IDE
- SCSI bad blocks are remapped to block from a special pool – usually on the same cylinder

**10.3.4 Swap Space Management****Swap-Space Use**

swapping – process image: code and data paging – pages that are pushed out

**Swap-Space Location**

part of normal file system: simple to implement but, inefficient  
separate disk partition: no file structure; special optimized management algorithms

**Swap-Space Management****10.3.5 Disk Reliability**

- Disks used to be the least reliable component of the system
- Disk striping (interleaving): break block in to subblocks and store subblocks on different drives to improve speed of block access
- RAID: several levels
  - mirroring or shadowing: keep a duplicate copy
  - ...
  - block interleaved parity: one disk contains a parity block for all corresponding blocks so data can be reconstructed when one disk crashes
  - With 100 disks and 10 parity disks, mean time to data loss (MTDL) is 90 years compared to standard 2 or 3 years

**10.3.6 Stable-Storage Implementation**

write more than once – write is not complete until both writes have occurred.  
if error occurs in a write, restore to uncorrupted copy, if no error occurs but copies differ, restore to contents of second.

**Part IV**

**Memory Management**



## Chapter 11

# OS5 - Memory Management

*Minimum core coverage time:* 5 hours

Topics:

- Review of physical memory and memory management hardware
- Overlays, swapping, and partitions
- Paging and segmentation
- Placement and replacement policies
- Working sets and thrashing
- Caching

Learning objectives:

1. Explain memory hierarchy and cost-performance tradeoffs.
2. Explain the concept of virtual memory and how it is realized in hardware and software.
3. Summarize the principles of virtual memory as applied to caching, paging, and segmentation.
4. Evaluate the tradeoffs in terms of memory size (main memory, cache memory, auxiliary memory) and processor speed.
5. Defend the different ways of allocating memory to tasks, citing the relative merits of each.
6. Describe the reason for and use of cache memory.
7. Compare and contrast paging and segmentation techniques.

8. Discuss the concept of thrashing, both in terms of the reasons it occurs and the techniques used to recognize and manage the problem.
9. Analyze the various memory portioning techniques including overlays, swapping, and placement and replacement policies.

**Concepts**

Review of physical memory and memory management hardware

Overlays, swapping, and partitions

Paging and segmentation

Memory mapped files

Placement and replacement policies

Working sets and thrashing

Real-time issues

## Chapter 12

# Memory Management

*Two hours of lecture are allocated to this chapter.*

To keep several processes in memory; we must *share* memory.

## 12.1 Background

### 12.1.1 Modeling CPU Utilization

- Naive model: If a process computes only  $P\%$  of the time, a system with  $100/P$  processes will compute 100% of the time.
- Probabilistic model: If a process spends a fraction  $p$  of its time in an I/O wait state, then in a system with  $n$  processes, the CPU utilization  $= 1 - p^n$ .
- Queueing theory: an even more accurate model.

Multiple processes can lead to improved CPU utilization, but there are diminishing returns with more processes.

### 12.1.2 Basic memory layout

Interrupt vector, OS kernel, User Programs

### 12.1.3 Address Binding

- Compile time – compiler generates absolute code – not relocatable
- Load time – relocatable code – table of addresses ie code modified at load time
- Execution time – requires special hardware (base register, page table)

### 12.1.4 Dynamic Loading

Load a routine when it is called – unused routines are not loaded; the programmer must partition program and sometimes there are library routines to assist with dynamic loading

### 12.1.5 Static Linking

Routines are linked together into a single address space. Each program has its own copy of shared libraries.

### 12.1.6 Dynamic Linking

Routines are linked at run-time – programs can share library code. There must be OS support for access to shared library since the shared library must be in a shared address space.

### 12.1.7 Overlays

A program is segmented and at execution time, only the active segment need be in memory. When another segment is needed, it is loaded into the same physical address space that was occupied previously.

The programmer must design and program the overlay structure properly since it requires no OS support

### 12.1.8 Logical versus Physical Address Space

- logical address – address generated by the CPU
- physical address – address seen by the memory address register
- physical address=logical address for compile and load time binding
- physical address!=logical address(virtual address) for execution time binding
- MMU (memory mapping unit) maps virtual address to physical address
- Relocation Example: physical address = logical address + relocation register
- Relocation and Protection: physical address = logical address + relocation register < limit register

### 12.1.9 Implementation of Protection & Relocation

- Relocation: load with relocation table or base register
- Protection: protection code for each block
- Relocation and Protection: base and limit registers



### 12.1.10 Supervisor mode/User mode

In user mode, the program has access only to its address space. In supervisor mode, the OS has full access to all memory. Hardware provides for user and supervisor (OS) modes (special instructions). When a user executes a special instruction, control is transferred to the OS. See MOS p. 19

### 12.1.11 Swapping

A process is swapped out to backing store to make space for higher priority processes. Note: under MS Windows, swapping is under user not OS control.

## 12.2 Contiguous Allocation

For most processes, memory use is fixed at compile time, the exception being, recursion and dynamic data structures.

### 12.2.1 Single-Partition Allocation

In single partition allocation there is a single user space memory partition protected with relocation and limit registers. Processes are swapped in and out to provide for multiprogramming.

### 12.2.2 Multiple-Partition Allocation

Multiple partitions are available to permit multiple processes to be simultaneously resident in memory. This reduces lost time due to swapping.

- fixed size partitions - poor memory utilization due to fragmentation
- variable size partitions - allocation algorithms
  - First-fit: allocate the  $j_i$  first  $i_i$  hole that is big enough. (generates large holes); A variant is Next-fit: begin searching where the last one left off (slightly worse performance than first fit)
  - Best-fit: allocate the  $j_i$  smallest  $i_i$  hole that is big enough. slow and results in more wasted memory (tiny useless holes)
  - Worst-fit: allocate the  $j_i$  largest  $i_i$  hole that is big enough. (maximize holes)

Simulations show that first-fit and best-fit are better than worst-fit in terms of decreasing both time and storage utilization, but first-fit is faster.

External and internal fragmentation and compaction  
Memory management

- Bit map - each bit indicates whether or not a block of memory is in use - search for free memory is slow so it is not often used

- Linked Lists - sorted by address to allow easy updating.
- Buddy System - see MOS p. 86

## 12.3 Swap Space Management

A process is swapped out to backing store to make space for higher priority processes Note: under MS Windows, swapping is under user not OS control.

The swap space management problems and solutions are essentially the same as for main memory.

## 12.4 Paging (non-contiguous allocation)

Used to minimize external fragmentation and to simplify allocation

### 12.4.1 Basic Method

- Physical memory is partitioned into fixed sized  $i_L$  frames. The size will determine the degree of internal fragmentation.
- Logical memory is partitioned into blocks of the same size called  $i_L$  pages.
- Backing store is divided into fixed sized blocks the same size as the memory frames
- Hardware support: page table (see page 268)
- High-order bits: page; low-order bits: page offset
- i.e. base(relocation) register for each page
- each process has a page table
- OS maintains a frame table (for each page – free or allocated, and which process)

### 12.4.2 Structure of the Page Table

Hardware Support

- Registers
- Memory – accessed via page-table base register (PTBR)
- Hardware Cache (associative registers) transition look-aside buffer TLB)
- Context switch is expensive

see page 274

Protection: associate protection bits with each page: read-only, read-write; valid (in user space)

PTLR page table length register

### 12.4.3 Multilevel Paging

in large address space the page table are LARGE, the page tables themselves may be paged

### 12.4.4 Inverted Page Table

pid + page + displacement; if page is not in page table, an external page table (one/process) is consulted

### 12.4.5 Shared Pages

two virtual addresses mapped to the same physical address; requires reentrant code; does not work well with inverted tables

## 12.5 Segmentation

Extreme: each subroutine has its own segment

### 12.5.1 Basic Method

Logical address = (segment number, offset)

### 12.5.2 Hardware

see figure p. 285

### 12.5.3 Implementation of Segment Tables

### 12.5.4 Protection and Sharing

### 12.5.5 Fragmentation

segments are of variable length!

## 12.6 Segmentation with Paging

### 12.6.1 Multics

### 12.6.2 OS/2 32-Bit Version

## 12.7 Exercises

Analysis, Design, Implementation

1. If a computer has 1M of memory, with the OS requiring 200K, and each user program 200K and an average of 80% average I/O wait, what is the CPU utilization (ignore the OS overhead). Should the owner add 1 or 2M of additional memory?

# Chapter 13

## Virtual Memory

Two hours of lecture are allocated to this chapter. The first lecture covers sections 1-5, the second lecture sections 5-9.

### 13.1 Background

Instructions must be in memory.

- Entire logical address space must be in memory – overlays and dynamic loading
- BUT entire program is not needed, error conditions, actual array size vs declared array size, options and features not used

The benefits of partial residence

- Could have larger *virtual* address space
- More programs could be executing
- Less I/O for swapping

*Virtual Memory* is the separation of Logical memory from Physical memory

- Demand paging
- Demand segmentation

### 13.2 Demand Paging

- Lazy swapper – pager rather than swapper
- Uses valid-invalid page bit – address is memory or disk
- Memory resident

- Page fault – trap to OS
- Procedure for handling Page Fault
  1. Check internal table (PCB) to determine valid address range
  2. If invalid, terminate process
  3. Find free frame
  4. Schedule disk access to read desired page
  5. After read, modify internal table and page table
  6. restart the instruction which caused the page fault
- Pure demand paging (start with no pages)
- Programs have *locality of reference*
- Hardware support is same as for paging and swapping
  - Page table
  - Secondary memory (swap space)
- Architectural support: restart instruction – consider add, string copy, auto increment (decrement)

### 13.3 Performance of Demand Paging

- Page from file system
- Page from swap space (load job into swap space and page from there).
- Load from file system, swap to swap space. (BSD UNIX)

### 13.4 Page Replacement

Over allocate to increase multiprogramming; what happens if physical memory is fully utilized when a page fault occurs?

- Terminate a user process (reduce level of multiprogramming)
- Swap out a process
- Page replacement (two pages – one out, one in; one page if dirty bit is not set!)

Reference Strings

## 13.5 Page Replacement Algorithms

Want an algorithm with lowest *page-fault rate*. The theoretical approach is to run the algorithm on a reference string (string of page numbers representing changes in page references over the life of a process – either actual or randomly generated)

### 13.5.1 FIFO

Associate the time the page was loaded with the page and replace oldest page. The algorithm is good for initialization code, bad for frequently used early data pages

*Belady's anomaly*: page faults may increase with additional memory.  
(use reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.)

### 13.5.2 Optimal Algorithm

*Replace the page that will not be used for the longest period of time*  
unimplementable as is SJF

### 13.5.3 LRU Algorithm

page that has not been used for the longest period of time.

Implementation:

- Counters – time of use field
- Stack of page numbers

### 13.5.4 LRU Approximation Algorithm

- Additional-Reference-Bits Algorithm - count bits
- Second-Chance Algorithm - circular queue
- Enhanced Second-Chance Algorithm - MacOS
- Counting Algorithms - LFU, MFU
- Page Buffering Algorithm -
- choose victim, read page into a free frame from a *pool* of free frames, restart, write out victim page.
- maintain list of modified pages. write out modified pages when disk is free.
- reuse old page if still available

## 13.6 Allocation of Frames

MUST maintain 3 free frames

### 13.6.1 Minimum number of frames

defined by the computer architecture, instruction set and levels of indirection

### 13.6.2 Allocation Algorithms

- Equal allocation
- Proportional allocation

### 13.6.3 Global vs Local Allocation

## 13.7 Thrashing

Spending more time paging than executing

### 13.7.1 Cause of thrashing

use example from text

### 13.7.2 Working-Set Model

## 13.8 Other Considerations

### 13.8.1 Prepaging

### 13.8.2 Page Size

### 13.8.3 Program Structure

## 13.9 Demand Segmentation



**Part V**

**File Management**



# Chapter 14

## OS8 - File systems

### *Topics:*

- Files: data, metadata, operations, organization, buffering, sequential, non-sequential
- Directories: contents and structure
- File systems: partitioning, mount/unmount, virtual file systems
- Standard implementation techniques
- Memory-mapped files
- Special-purpose file systems
- Naming, searching, access, backups

### Learning objectives:

1. Summarize the full range of considerations that support file systems.
2. Compare and contrast different approaches to file organization, recognizing the strengths and weaknesses of each.
3. Summarize how hardware developments have lead to changes in our priorities for the design and the management of file systems.

### Concepts

File layout

Directories: contents and structure

Naming, searching, access, backups

Fundamental file concepts (organization, blocking, buffering)

Sequential files

Nonsequential files



## Chapter 15

# File System Interface

Two hours of lecture are allocated to this chapter. The first lecture covers sections 1-3, the second lecture sections 4-5.

- files
- directory structure
- partitions
- file protection

### 15.1 The File Concept

A *file* is a sequence of bytes of arbitrary length. Files are implemented by the operating system to provide persistent storage.

- File Attributes/meta data
  - Name
  - Type
  - Size
  - Location - disk, sectors
  - Protection - owner, group, access rights (read, write, execute)
  - Time & date (creation, last use, last modification)
- File ADT: The basic operations<sup>1</sup>

---

<sup>1</sup>An abstract data type (ADT) is characterized by the following properties: 1. It exports a type. 2. It exports a set of operations. This set is called interface. 3. Operations of the interface are the one and only access mechanism to the type's data structure. 4. Axioms and preconditions define the application domain of the type.

- Creating a file: allocate space, make directory entry
- Writing a file: given name and data, find file and write at write pointer, then update write pointer.
- Reading a file: given name and a block in memory, find file and copy data at read pointer into block, then update read pointer.
- Repositioning within a file: reposition the file read/write pointer (file seek)
- Deleting a file: find file, release space and directory entry
- Truncating a file: reset file length (release space)

Open-file table, per-process table

- File pointer
- File open count
- Disk location of file

sharing of files (multiple edit sessions)

- File Types: Should the OS recognize and support file types? File extensions (Table OSC p. 355) are required in DOS, optional in Unix.
  - Extensions may be required by the OS or software
  - TOPS-20: auto recompile if source is modified.
  - Apple Macintosh: a file is associated with the program that created it – enables editing
  - While not possible in Unix, some X windowing environments provide such capabilities.
  - File extensions facilitate viral infections
- File Structure: Should the OS support alternative internal file structures?
  - Internal structure: logical record size; physical record size; packing technique; file is a sequence of blocks (internal fragmentation)
  - VMS and IBM mainframe OSs provide multiple file types; OS is more complex
  - Unix – One: byte sequence
  - Mac – resource fork-user modifiable button labels, data fork-code and data
- Access Methods
  - Sequential Access: The tape model
  - Direct Access: fixed-length logical records; the disk model (array); requires block number to be a file operation parameter
  - Indexed files: index with pointers to blocks – the index often contains a key (IBM: ISAM)

## 15.2 The Directory Concept

A file system is broken into partions (IBM: minidisks; DOS: volumes) which contain files and directories. Directories contain information (device directory or volume table) about the files within it.

- Directory ADT:
  - Search for a file – name/pattern
  - Create a file
  - Delete a file
  - List a directory
  - Rename a file
  - Traverse the file system
- Directory structure
  - Single-Level Directory
  - Two-Level Directory
  - Tree-Structured Directories
  - Acyclic-Graph Directories
    - \* Shared files/directories
    - \* Symbolic link(pointer)
    - \* Problems:
      - multiple names, file system traversal
      - deletion: dangling pointers, reference counts
  - General Graph Directory; cycles, garbage collection
- Protection
  - Access Lists and Groups
  - Other Protection Approaches; passwords

### 15.2.1 Unix

## 15.3 Consistency Semantics

### 15.3.1 Unix Semantics

the Unix file system

### 15.3.2 Session Semantics

the Andrew file system

### 15.3.3 Immutable-Shared-Files Semantics





## Chapter 16

# File System Implementation

Two hours of lecture are allocated to this chapter.

Issues

- Disk file systems
- Allocation
- Free space management
- Performance

### 16.1 File-System Structure

Byte sequence, Record sequence, Tree of records

Efficient I/O requires block transfers; disk sectors usually vary from 32 to 4096 bytes with 512 a common size.

Disk file systems: supports either sequential or direct access; can be rewritten in place.

#### 16.1.1 File System Organization

Goal is efficient and convenient access with the following layered architecture

- I/O control – device drivers, interrupt handlers
- Basic file system – issues generic commands to device drivers: drive, cylinder, surface, sector
- File-organization module – files and logical blocks, physical blocks – translates logical structure to physical structure; free space manager
- Logical file system – responsible for protection and security

### 16.1.2 File-System Mounting

Unix and Macintosh

## 16.2 Allocation Methods

contiguous, linked, indexed (DG nova supports all three)

- Contiguous Allocation (IBM VM/CMS) - good performance but fragmentation occurs (compare with contiguous memory allocation)
- Linked Allocation - efficient management but requires space for pointers, has poor random access and may have reliability problems; in contrast *file allocation table* or FAT (MS-DOS & OS/2) solves some of the problems.
- Indexed - provides efficient memory management and access through an index block but suffers from wasted space.
  - Linked scheme: link index blocks
  - Multilevel index: index block points to index blocks
  - Combined scheme: Unix inode (direct blocks, single indirect, double indirect, triple indirect).
- Performance: Combine contiguous and indexed allocation, CPU speed vs Disk access time

## 16.3 Free-Space Management

- Bit Vector: fast for contiguous allocation and small disks especially with hardware support.
- Linked List:
- Grouping: free block contains addresses of free blocks and last free block contains address of next block containing addresses of free blocks.
- Counting: disk address and free count of contiguous blocks.

## 16.4 Directory Implementation

- Linear List: (implemented in an array, linked list, linked binary tree)
- Hash Table:

## 16.5 Efficiency and Performance

The disk subsystem is the major bottleneck in system performance.

- Efficiency
- Performance
- Recovery
- Backup and Restore



**Part VI**

**Other**



## Chapter 17

# OS7 - Security and protection

### *Topics:*

- Overview of system security
- Policy/mechanism separation
- Security methods and devices
- Protection, access, and authentication
- Models of protection
- Memory protection
- Encryption
- Recovery management

### *Learning objectives:*

1. Defend the need for protection and security, and the role of ethical considerations in computer use.
2. Summarize the features and limitations of an operating system used to provide protection and security.
3. Compare and contrast current methods for implementing security.
4. Compare and contrast the strengths and weaknesses of two or more currently popular operating systems with respect to security.
5. Compare and contrast the security strengths and weaknesses of two or more currently popular operating systems with respect to recovery management.

## 17.1 Overview of system security

*Security*: Protection against unauthorized disclosure, alteration, or destruction of data and protection against unauthorized users. There are three goals—secrecy, integrity, and availability.

Goal	Threat
<i>Secrecy</i> (Data Confidentiality)	Exposure of data (Information should not be disclosed to unauthorized users.)
Data <i>Integrity</i>	Tampering with data (Only authorized users should be allowed to modify data. Accuracy or validity is absolutely fundamental, security is more of a secondary issue.)
System <i>Availability</i>	Denial of service (Authorized users should not be denied access.)

## 17.2 Policy/mechanism separation

- An organization's *security policy* defines the rules for authorizing access to computer and information resources.
- A *protection mechanism* is a set of hardware and software components used to implement any one of different sets of strategies.
- A *policy* is a particular strategy that dictates the way a mechanism is used to achieve specific goals.

## 17.3 Security methods and devices

- Physical –
- Administrative – employee education & monitoring
- Automated –

## 17.4 Protection, access, and authentication

- Protection – physical, encryption, user background checks and monitoring.
- Access –
- Authentication – the process of determining who the user is. The methods are based on one of three general principles
  1. Something the user knows. Name-Password pairs.
    - Login: Password: issues
    - Password security



- (a) Length  $> 7$
  - (b) Mixed case
  - (c) Digits and special characters
  - (d) Avoid words and names
- One-time passwords
- Challenge-Response
- 2. Something the user has.
  - Magnetic stripe card (140 bytes)
  - Chip card
    - (a) Stored value card ( $< 1K$ )
    - (b) Smart card (4MHz 8-bit CPU 16 KB of ROM, 4 KB of EEPROM, 512 Bytes of RAM 9600-bps com channel)
- 3. Something the user is. Biometrics - enrollment and identification - fingerprint, voiceprint, retinal pattern, signature analysis, etc

## 17.5 Models of protection

User-oriented access control

- ID/password

Data-oriented access control

- file
- database

Reference monitor – each time an access to a protected resource is attempted, the system first consults the reference monitor to check its legality. The reference monitor consults its policy tables and makes a decision.

General model of access control *Access Matrix*

- Subject: process
- Object = unique name, finite set of operations; Examples: files, programs/software package, services (login, ftp, web, DB, etc), hardware (system, switch/router, server), ...
- Rights: a subset of operations on an object. Example: Unix access rights –read, write, execute/create
- Domain = (object, rights), ... . A domain may correspond to one or more users.

- Access matrix – networks and systems; processes, users, groups

Access Matrix	Object <sub>0</sub>	Object <sub>1</sub>	...
Process <sub>0</sub>	Access rights	Access rights	...
Process <sub>1</sub>	Access rights	Access rights	...
...	...	...	...

*Example:* Objects – Unix files; Processes – users and programs; Access rights – as defined by ugo permissions *Implementation* - the Access Matrix is a large sparse matrix.

- Decomposition by columns: *access control lists* - For each object, a list of domains/processes with their access rights.
- Decomposition by rows: *capability tickets* - For each domain/process, a list of objects and corresponding access rights. Implemented in
  - hardware using tagged architecture (each memory word has an extra bit - IBM AS/400)
  - the OS (e.g., in the process control block)
  - user space (but cryptographically protected)
- Trade offs
  - Efficiency - capabilities (no checking needed), ACL (potentially long search)
  - Encapsulation - easy with processes and capabilities
  - Selective revocation of rights - easy with ACLs
  - Removal of object or capabilities but not both - easily handled with ACLs but not capabilities.

## 17.6 Memory protection

virtual memory: segmentation or paging

- No sharing: no duplicate entries in page and/or segment tables
- Sharing: allow duplicate entries in page and or segment tables

## 17.7 Encryption

see security notes from the networking class

## 17.8 Recovery management

A plan for recovery from a security event.

## 17.9 Trusted systems

To build a secure system, have a security model at the core of the OS that is simple enough that the designers can actually understand it, and resist all pressure to deviate from it in order to add new features.

### 17.9.1 Covert channels

- Using timing information to send bits.
- Locking and unlocking files to send bits.
- Stenography.

Previous notes

- [Security.html](#)
- [Protection.html](#)
- Encryption - see networking
- Recovery management

## 17.10 References

- Infosyssec at <http://www.infosyssec.net/infosyssec/linux1.htm>



## Chapter 18

# OS9 - Real-time and embedded systems

### *Topics:*

- Process and task scheduling
- Memory/disk management requirements in a real-time environment
- Failures, risks, and recovery
- Special concerns in real-time systems

### Learning objectives:

1. Describe what makes a system a real-time system.
2. Explain the presence of and describe the characteristics of latency in real-time systems.
3. Summarize special concerns that real-time systems present and how these concerns are addressed.



## Chapter 19

# OS10 Fault tolerance

This chapter intentionally left blank.





## Chapter 20

# OS11 System performance evaluation

This chapter intentionally left blank.



## Chapter 21

# OS12 Scripting

This chapter intentionally left blank.



**Part VII**

**Computer Organization**



## Chapter 22

# System Organization

For more information see: <http://www.howstuffworks.com>

Central Processing Unit (CPU)

Control
ALU and the Registers

Memory Management Unit

MMU
-----

Memory - an array of storage units.

0	C&D
...	C&D
n	C&D

Other devices - hard drive, network, display, sound, ...

### 22.1 The von Neumann Architecture

- Processor (CPU)
- Memory and the Memory Management Unit (MMU)
- I/O modules and devices
- System interconnection

### 22.2 The Central Processing Unit (CPU)

#### 22.2.1 Basic Structure

- Arithmetical-logical unit (ALU)
- Control unit

Registers

- User visible - used to minimize memory references
  - Data registers
  - Address registers
    - \* index register
    - \* segment pointer
    - \* stack pointer
  - Condition codes (flags)
- Control and status registers - used by the processor to control the operation of the processor
  - Program counter
  - Instruction register

#### Instruction Execution

- Fetch execute cycle
- Instruction set
  - processor-memory
  - processor-i/o
  - data processing
  - control

#### Fetch-Execute Cycle:

```

PC = <machine start address>;
haltFlag = CLEAR;
while (haltFlag not SET during execution) {
    IR = Memory[PC]; // Fetch
    PC = PC + i; // Increment PC
    execute(IR); // Decode and execute
}

```

#### Various CPU architectures

- Stack machine e.g., the Java virtual machine
- Accumulator machine
- Register machine e.g., SPARC, MIPS, Alpha, PowePC

X86 architecture is a complex instruction set machine and includes aspects of all three architecture types.



### 22.2.2 Interrupts

Busy waiting (repeatedly checking to see if an I/O device has completed its task) wastes CPU cycles. The alternative is to have the device signal the OS when it is done. A timer is required to support multi-tasking.

- Program
- Timer
- I/O
- Hardware failure

I/O Communication Techniques

- Programmed I/O -
  - instruction set
    - \* control
    - \* test
    - \* read, write
- Interrupt-driven I/O
  - Direct memory access DM

The Fetch-Execute Cycle with an interrupt

```

PC = <machine start address>
haltFlag = CLEAR;
while (haltFlag not SET during execution) {
    IR = Memory[PC]; // Fetch
    PC = PC + i;
    execute(IR); // Execute
    if ( interruptRequest ){// Interrupt the current process
        Memory[0] = PC; // Save the current PC in address 0
        PC = Memory[1]; // Branch indirect through address 1
    }
}

```

The Interrupt Handler

```

InterruptHandler {
    saveProcessorState();
    for ( i=0; i < NumberOfDevices; i++ )
        if ( device[i].done == 1 ) goto deviceHandler( i );
}

```

Disabling Interrupts

```

If ( interruptRequest  interruptEnabled ) {
    disableInterrupts();
    Memory[0] = PC;
    PC = Memory[1];
}

```

### 22.2.3 Processor Modes

CPU modes:

- user mode - user program is restricted to user level instructions and address space. Exception: System call or trap instruction which causes an interrupt and changes the processor mode.
- system/supervisor/privileged mode - all instructions and address space

Privileged instructions

- Instructions that change processor mode
- Memory management instructions: set page table base or TLB
- Timer instructions
- Instructions that set other important hardware registers

They are used to protect

1. The processor mode
2. The memory
3. The I/O devices
4. The processor itself.

#### The user mode trap instruction

The trap instruction and a trap handler table provides a safe way for user-mode process to execute only predefined software when the mode bit is set to supervisor mode.

Trap instruction: `trap argument`

Trap handler - assume trap handler table is loaded a location 1000

```

executeTrap( argument ) {
    setMode( supervisor );
    switch( argument ){
        case 1: PC = Memory[1001];
        ...<
        case n: PC = Memory[1000+n];}
}

```

OS resets mode to user before user program returns to execution.

## 22.3 Memory

### 22.3.1 Memory Hierarchy

- Registers
- Cache
- Main memory
- Disk cache
- Disk
- Removable media

#### Cache Memory

- Principles
  - Main memory
    - \*  $2^n$  addressable words with an  $n$ -bit address
    - \*  $M = 2^n / K$  blocks of memory;  $K$  words per block
  - Cache:  $C$  slots of  $K$  words (with  $C \ll M$ )
- Design
  - cache size
  - block size
  - mapping function
  - replacement algorithm
  - write policy

### 22.3.2 Protected memory

Base and limit registers

Effective address:  $\text{base} + \text{offset}$

- Supervisor mode
  - $\text{Base} = 0$
  - $\text{Limit} = y$  where  $y$  is memory size
- User mode
  - $\text{Base} = x$  where  $x$  is assigned by the OS
  - $\text{Limit} = y$  where  $y$  is assigned by the OS

```
fetch: IR := Memory[PC+Base] if PC + Base <= Limit
Load a r: r := Memory[a + Base] if
```

### 22.3.3 Paged Memory

User mode program addresses are pairs (page number, page offset); usually just user offsets interpreted as *page number* + *page offset*.

Supervisor mode program addresses are absolute addresses.

Page table - an array of page numbers  
Effective address:  $\text{PageTable}[\text{page number}] + \text{page offset}$

### 22.3.4 Virtual Memory

Usually paged memory with additional information to indicate whether page is in memory or not.

## 22.4 Devices

Device Abstractions
Application Software
High-level I/O Machine
Device Controller
Device

- Device characteristics
  - by data transmission
    - \* block-oriented
    - \* character-oriented
  - by function
    - \* communication
    - \* storage

- Device controllers

- Device drivers

[Hard Drives/h3](#)

- [Disks.html](#) Disk basics/a
- [techno.html](#) SCSI and RAID: DPT Technology White Papers/a
- [raid.html](#) RAID/a

[Network devices/h3](#)

- [../425/DirectLink.html](#) Network devices/a

### 22.4.1 Disk Devices

#### Structure

- Spindle
- Platter/Surface
- Track (cylinder – tracks at same head position)
- Sector - A sector contains the smallest amount of information that can be read or written to the disk (32-4096 bytes)

### 22.4.2 Addressing

- Cylinder
- Track
- Sector
- $s$  = number of sectors per track
- $t$  = number of tracks per cylinder
- $i$  = cylinder number ( $0 - n_{\text{sub}_i}/\text{sub}_i$ )
- $j$  = surface number ( $0 - n_{\text{sub}_j}/\text{sub}_j$ )
- $k$  = sector number ( $0 - n_{\text{sub}_k}/\text{sub}_k$ )
- $b$  = block number
- $b = k + s \cdot (j + i \cdot t)$

#### Timing

- Seek - move r/w head to cylinder
- Latency - wait for sector to rotate into position
- Transfer - data transfer

h3iDisk Management/h3i

h4iDisk Formatting/h4i

- Physical Formatting: track and sector marks; space for ECC(error correcting code)
- Logical Formatting: partition and an initial empty file system.

h4iBoot Block/h4i bootstrap program: initializes system and loads OS memory – stored in ROM and/or boot block of disk

h4iError Handling/h4i

- Programming error (e.g., request for non-existent sector) – should not occur but if it does, terminate disk request
- Transient checksum error (e.g., caused by dust on the head) – retry
- Permanent checksum error (e.g., disk physically damaged) – mark as bad and substitute a new block
- Seek error (e.g. the arm sent to sector 6 but went to sector 7) – recalibrate
- Controller error (e.g., controller refuses to accept commands) –

h4Bad Blocks/h4

- IDE
- SCSI bad blocks are remapped to block from a special pool – usually on the same cylinder

h3Disk Reliability/h3

- Disks used to be the least reliable component of the system
- Disk striping (interleaving): break block in to subblocks and store subblocks on different drives to improve speed of block access
- RAID: several levels
  - mirroring or shadowing: keep a duplicate copy
  - ...
  - block interleaved parity: one disk contains a parity block for all corresponding blocks so data can be reconstructed when one disk crashes
  - With 100 disks and 10 parity disks, mean time to data loss (MTDL) is 90 years compared to standard 2 or 3 years

h3Stable-Storage Implementation/h3 write more than once – write is not complete until both writes have occurred. if error occurs in a write, restore to uncorrupted copy, if no error occurs but copies differ, restore to contents of second.i/body

## RAID

bOlder RAID levels/b

center

Level	Description	Comments/Advantages	Disadvantages
RAID 0	files striped across multiple drives	high read & write performance	no redundancy
RAID 1	files are		

mirrored; on second drive; data redundancy; faster read performance; double disk space; slower write performance; `RAID 2`; `RAID 1` with error-correction code (ECC); not generally used since SCSI drives have ECC built in; `RAID 3`; files are striped at the byte level across multiple drives; parity value stored on a dedicated drive; hardware based; data redundancy; faster read and write performance; extra disk required; I/O can be a bottleneck; expensive; `RAID 4`; `RAID 3` except files are striped at block level; less expensive than `RAID 3`; data redundancy; faster read and write performance; I/O can be a bottleneck; `RAID 5`; `RAID 4` except parity information is distributed across all drives; data redundancy; faster reads; writes can be slow;

Some vendors provide combinations of RAID levels.

New RAID levels

Term	Description
FRDS (failure-resistant disk system)	system protects against data loss due to failure of a single part of the system; FRDS plus; FRDS + hot swapping; the ability to recover from cache and power failures; FTDS (failure-tolerant disk system); FRDS + reasonable protection against other failures; FTDS plus; FTDS + protection against bus failures; DTDS (disaster-tolerant disk system); two or more zones with cooperation to prevent data loss in case of complete failure of one machine or array; DTDS plus; DTDS + recovery in case of all manner of disasters – flood, fire, ...





## Chapter 23

# Assembly Programming in GNU/Linux

### 23.1 x86 Architecture and Assembly Instructions

#### 23.1.1 Programming Model

##### Memory

$2^{32}$ - bytes

##### Registers

###### 8 32-bit General Purpose Registers

Register	Function	16-bit low end	8-bit
eax	Accumulator	ax	ah, al
ebx	(base index)	bx	bh, bl
ecx	(count)	cx	ch, cl
edx	(data)	dx	dh, dl
edi	(destination index)	di	
esi	(source index)	si	
ebp	Frame pointer	bp	
esp	Stack top pointer	sp	

##### Registers

###### 6 16-bit Section Registers

Register	Function
cs	Code section
ds	Data section
ss	Stack section
es	(extra section)
fs	(supplemental section)
gs	(supplemental section)

**EFLAGS Register**

S	Sign
Z	Zero
C	Carry
P	Parity
O	Overflow

32-bit EFLAGS Register

32-bit EIP (Instruction Pointer Register)

### 23.1.2 AT & T Style Syntax (GNU C/C++ compiler and GAS)

- Instruction: **opcode**[**b+w+l**] **src, dest**
- Register: **%reg**
- Memory operand size: [**b+w+l**] for byte, word, longword - 8, 16, 32 bits
- Memory references: **section:disp(base, index, scale)** where *base* and *index* are optional 32-bit base and index registers, *disp* is the optional displacement, and *scale*, taking the values 1, 2, 4, and 8, multiplies *index* to calculate the address of the operand. – address is relative to section and is calculated by the expression:  $\text{base} + \text{index} * \text{scale} + \text{disp}$
- Constants (immediate operands)
  - 74 - decimal
  - 0112 - binary
  - 0x4A - hexadecimal
  - 0f-395.667e-36 - floating point
  - 'J - character
  - "string" - string

#### Operand Addressing

- Code: CS + IP (Code segment + Offset)
- Stack: SS + SP (Stack segment + Offset (stack top))
- Immediate Operand:  $\$constant\_expression$

- Register Operand: *%registerName*
- Memory Operand: **section:displacement(base, index, scale)** The section register is often selected by default. cs for code, ss for stack instructions, ds for data references, es for strings.

Base	+(	Index	*	Scale	)+	Displacement
eax		eax		1		Name
ebx		ebx		2		Number
ecx		ecx		3		
edx		edx		4		
esp		ebp				
ebp		esi				
esi		edi				
edi						

- **Direct**Operand: displacement (often just the symbolic name for a memory location)
- **Indirect** Operand: (base)
- **Base+displacement**: displacement(base)
  - \* index into an array
  - \* access a field of a record
- **(index\*scale)+displacement**: displacement(,index,scale)
  - \* index into an array
- **Base + index + displacement**: displacement(base,index)
  - \* two dimensional array
  - \* one dimensional array of records
- **Base+(index\*scale)+ displacement**: displacement(base, index,scale)
  - \* two dimensional array

### 23.1.3 Subroutines

- Function – returns an explicit value
- Procedure – does not return and explicit value

The flow of control and the interface between a subroutine and its caller is described by the following:

Caller ...	
<b>call</b> <i>target</i>	Transfer of control from caller to the subroutine by
Subroutine	
<b>pushl</b> %ebp <b>movl</b> %esp, %ebp	Save base pointer of the caller New base pointer (activation record/frame)
Callee ...	Body of Subroutine
<b>movl</b> %ebp,%esp <b>popl</b> %ebp	Restore the callers stack top pointer Restore the callers base pointer
<b>ret</b>	Return of control from the subroutine to the caller by alter the program counter (CS:IP) register to the saved address of the caller.
Caller ...	

An alternative is to have the caller save and restore the values in the registers. (Prior to the call, the caller saves the registers it needs and after the return, restores the values of the registers)

### 23.1.4 Data

#### Data Representation

- Bits, Bytes, Wyde, word, double word – modulo  $2^n$
- Sign magnitude – sign bit 0=+, 1=-; magnitude
- One's complement – negative numbers are complement of positive numbers  
- problem: two representations for zero
- Twos complement (used by Intel) – to negate:
  - Invert (complement)
  - add 1
- Excess  $2^{(n-1)}$  (often used for exponent)
- ASCII - character data
- EBCDIC
- BCD

#### Data Definition Directives

Description provided to the assembler of how static data is to be organized.

- Symbolic name (variables and constants)
- Size (number of bytes)
- Initial value

- **.data**
- Define Byte (DB): (8-bit values) *[name] DB initial value [, initial value]*  
see key examples in text; multiple values, undefined, expression, C and Pascal strings, one or more lines of text, \$ for length of string
- Define Word (DW): (16-bit words) *[name] DW initial value [, initial value]*  
see key examples in text; reversed storage format, pointers
- Define Double Word (DD): (32-bit double words) *[name] DW initial value [, initial value]*
- Example: p. 80
- DUP Operator: *n dup( value )* see key examples in text; type checking

#### Constant Definitions

- **.CONST**
- **EQU**: *name EQU constant expression*

#### 23.1.5 Data Transfer Instructions

- **mov** *src, dest*
  - src: immediate value, register, memory
  - dest: register, memory
  - except memory, memory
- **xchg** *sd1, sd2*
  - Memory, Register
  - Register, Memory
  - Register, Register
- **push** *src*
  - src: immediate, register, or memory
- **pop** *dest*
  - dest: register or memory
- **pusha** - save all registers on the stack
- **popa** - restore all registers from the stack

### 23.1.6 Arithmetic Instructions

- **add***src, dest*; **sub***src, dest* - *src* +/- *dest*, result in *dest*
  - Memory, Register
  - Register, Memory
  - Register, Register
- Flags Affected by add and sub: OF (overflow), SF (sign), ZF (zero), PF (parity), CF (carry), AF (borrow)
- **inc***dest*; **dec** *dest* faster than add/subtract
  - Memory
  - Register
- Flags Affected by inc and dec: OF (overflow), SF (sign), ZF (zero), PF (parity), AF (borrow)
- **adc** & **sbb** add with carry/subtract with borrow - used for adding numbers with more than 32-bits
- **cmp** *src, dest* computes *src* - *dest* (neither *src* or *dest* changes) but may change flags.
  - Memory, Register
  - Register, Memory
  - Register, Register
- **cmpxchg** *src, dest* - compares *dest* with accumulator and if equal, *src* is copied into destination. If not equal, destination is copied to the accumulator.
- **neg** *dest* - change sign or two's complement
  - Memory
  - Register
- Flags Affected by NEG: SF (sign), ZF (zero), PF (parity), CF (carry), AF (borrow)
- **mul** *src* - unsigned multiplication EDX:EAX = *src* \* *eax*
- **imul** *src* - signed multiplication EDX:EAX = *src* \* *eax*
- Flags Affected by MUL, IMUL:
  - undefined: SF, ZF, AF, PF
  - OF, CF set if upper half is nonzero, set otherwise

- **div src** (unsigned) src is general register or memory quotient  $\text{eax} = \text{edx:eax}/\text{src}$ ; remainder  $\text{edx} = \text{edx:eax} \bmod \text{src}$
- **idiv src** (signed) src is general register or memory quotient  $\text{eax} = \text{edx:eax}/\text{src}$ ; remainder  $\text{edx} = \text{edx:eax} \bmod \text{src}$ 
  - Flags Affected by DIV, IDIV:
    - \* undefined: OF, SF, ZF, AF, PF
    - \* Type 0 interrupt if quotient is too large for destination register.
- CBW (change byte to word) expands AL to AX - signed arithmetic
- CWD (change word to double word) expands AX to DX:AX - signed arithmetic
- BCD Arithmetic - often used in point of sale terminals
- ASCII Arithmetic - rarely used

### 23.1.7 Logic Instructions

- **andl src, dest** -  $\text{dest} = \text{src} \text{ and } \text{dest}$
- **orl src, dest**
- **xorl src, dest**
- **notl dest** - logical inversion or one's complement
- **neg dest** - change sign or two's complement
  - Memory
  - Register
- **testl src, dest** (an AND that does not change dest, only flags)

### 23.1.8 Shift and Rotate Instructions

- Logical Shift
  - **shr count, dest** - shift dest count bits to the right
  - **shl count, dest** - shift dest count bits to the left
- Arithmetic Shift (preserves sign)
  - **sar count, dest** - shift dest count bits to the right
  - **sal count, dest** - shift dest count bits to the left
- Rotate without/With carry flag
  - **ror count, dest** - rotate dest count bits to the right

- **rol** *count, dest* - rotate dest count bits to the left
- **ror** *count, dest* - rotate dest count bits to the right
- **rcl** *count, dest* - rotate dest count bits to the left
- **test** *arg, arg*(an AND that does not change dest, only flags)
- **cmp** *src, dest* subtract src from dest (neither src or dest changes) but may change flags.
  - Memory, Register
  - Register, Memory
  - Register, Register
  - CMP
- Flag Bit Operations
  - Complement CF: CMC
  - Clear CF, DF, and IF: CLC,CLD,CLI,
- Set CF, DF, and IF: STC, STD, STI

### 23.1.9 Control Transfer Instructions

- **cmp** *src, dest* - compute dest - src and set flags accordingly
- Jump instructions: the transfer is one-way; that is, a return address is not saved.

```

NEXT: . . .
      . . .
      jmp NEXT ;GOTO NEXT

```

### Jump Instructions



<b>jmp</b> <i>dest</i>		unconditional	NEXT:... ... jmp NEXT ;GOTO NEXT
<i>Unsigned conditional jumps</i>			
<b>jcc</b> <i>dest</i>			
ja/jnbe	C=0 and Z=0	Jump if above	
jae/jnb	C=0	Jump if above or equal to	
jb/jnae	C=1	Jump if below	
jbe/jna	C=1 or Z=1	Jump if below or equal to	
jc	C=1	Jump if carry set	
je/jz	Z=1	Jump if equal to	
jnc	C=0	jump if carry cleared	
jne/jnz	Z=0	jump if not equal	
jnp/jpo	P=0	jump if no parity	
jp/jpe	P=1	jump on parity	
jcxz	cx=0	jump if cx=0	gcc does not use
jecxz	ecx=0	jump if ecx=0	gcc does not use
<i>Signed conditional jumps</i>			
<b>jcc</b> <i>dest</i>			
jg/jnle	Z=0 and S=0	jump if greater than	
jge/jnl	S=0	jump if greater than or equal	
jl/jnge	S=1	jump if less than	
jle/jng	Z=1 or S=1	jump if less than or equal	
jno	O=0	jump if no overflow	
jns	S=0	jump on no sign	
jo	O=1	jump on overflow	
js	S=1	jump on sign	

- Loop instructions: The loop instruction decrements the ecx register then jumps to the label if the termination condition is not satisfied.

```

        movl count, %%ecx
LABEL:

...

loop LABEL

```

	Termination condition	
<b>loop</b> <i>label</i>	ecx = 0	gcc does not use
<b>loopz/loope</b> <i>label</i>	ecx = 0 or ZF = 0	gcc does not use
<b>loopnz/loopne</b> <i>label</i>	ecx = 0 or ZF = 1	gcc does not use

- **call***name* - call subroutine *name*
- **ret** - return from subroutine

- **enter**
- **leave**
- **int***n* - interrupt
- **into** - interrupt on overflow
- **iret** - interrupt return
- **bound** - value out of range
- IF C THEN S;
- IF C THEN S1 ELSE S2;
- CASE E DO c1 : S1; c2 : S2; ... cn : Sn end;
- WHILE C DO S;
- REPEAT S UNTIL C;
- FOR I from J to K by L DO S;

### 23.1.10 String Instructions

The string instructions assume that by default, the address of the source string is in `ds:esi` (section register may be any of `cs`, `ss`, `es`, `fs`, or `gs`) and the address of the destination string is in `es:edi` (no override on the destination section). Typical code follow the scheme

```

    initialize esi and edi with addresses for source and destination strings
    initialize ecx with count
    Set the direction flag with cld to count up, with std to count down
    prefix string-operation

```

prefix **movs** - move string

prefix **cmps** - compare string WARNING: subtraction is dest - source, the reverse of the `cmp` instruction

prefix **scas** - scan string

prefix **lods** - load string

prefix **stos** - store string

- String instruction prefixes: The `ecx` register must be initialized and the DF flag in initialized to control the increment or decrement of the `ecx` register. Unlike the loop instruction, the test is performed before the instruction is executed.

- **rep** - repeat while ecx not zero
- **repe** - repeat while equal or zero (used only with cmps and scas)
- **repne** - repeat while not equal or not zero (used only with cmps and scas)

### 23.1.11 Miscellaneous Instructions

- **leal** *src, dest*(load effective address – the address of src into dest)
  - Memory, Register
- **nop**
- **xlat/xlatb**
- **cuid**

### 23.1.12 Floating Point Instructions

Floating Point: 8 32-bit registers

Register	Function
st	
st(0)	
st(1)	
...	
st(7)	

### 23.1.13 MMX Instructions

### 23.1.14 System Instructions

- **hlt**
- **lock**
- **esc**
- **bound**
- **enter leave**

Interrupts

- **int**
- **into**

Memory Management Unit

- **invlpg**

## Cache

### 23.1.15 References

- <http://www.x86.org>

## 23.2 x86 Assembly Programming

### 23.2.1 Assumptions

1. The program is in a single file.
2. All variables are 32-bit.

### 23.2.2 The G++ options

Source program: **program.cpp**

- Compile to a.out **g++ program.cpp**
- Compile to named file **g++ program.cpp -o program**
- Generate assembly program **g++ -S program.cpp**
- Optimize a program
- **g++ -O program.cpp**
- Generate and optimize an assembly program
- **g++ -O -S program.cpp**

### 23.2.3 Using GAS the GNU assembler

Source program: **program.s**

- Assemble **as program.s -o program.o**
- Compile **gcc program.o -o program**

### 23.2.4 Inline Assembly

Inline assembly code may be included as a string parameter, one instruction per line, to the `asm` function in a C/C++ source program.

```
...
asm("incl x;
movl 8(%ebp), %eax
");
```

Where the basic syntax is: **asm** [ **volatile** ] (*/\*asm statements\*/*  
 [: */\* outputs - comma separated list of constraint name pairs \*/*  
 [: */\* inputs - comma separated list of constraint name pairs \*/*  
 [: */\* clobbered space - registers, memory \*/*  
 ]]);

- asm statements - enclosed with quotes, at&t syntax, separated by new lines
- outputs & inputs - constraint-name pairs "constraint" (name), separated by commas
- registers-modified - names separated by commas

Constraints are

- **g** - let the compiler decide which register to use for the variable
- **r** - load into any available register
- **a** - load into the eax register
- **b** - load into the ebx register
- **c** - load into the ecx register
- **d** - load into the edx register
- **f** - load into the floating point register
- **D** - load into the edi register
- **S** - load into the esi register

The outputs and inputs are referenced by numbers beginning with %0 inside asm statements.

Example:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

int f( int );

int main (void)
{
    int x;
    asm volatile("movl $3,%0" : "=g"(x) : : "memory"); // x = 3;
    printf("%d -&gt; %d\n",x,f(x));
```

```

}

/*END   Main   */

int f( int x )
{
    asm volatile("movl %0,%%eax
                  imull $3, %%eax
                  addl $4,%%eax"
                  :
                  : "a" (x)
                  : "eax", "memory"
                  ); //return (3*x + 4);
}

```

### Global Variables

Assuming that `x` and `y` are global variables, the following code implements  $x = y*(x+1)$

```

asm("incl x
movl x, %eax
imull y
movl %eax,x
");

```

### Local variables

Space for local variables is reserved on the stack in the order that they are declared. So given the declaration: `int x, y;`

```

x is at -4(%ebp)
y is at -8(%ebp)

```

### Value Parameters

Parameters are pushed onto the stack from right to left and are referenced relative to the base pointer (`ebp`) at four byte intervals beginning with a displacement of 8. So in the body of `p(int x, int y, int z)`

```

x is at 8(%ebp)
y is at 12(%ebp)
z is at 16(%ebp)

```

### Reference parameters

Reference parameters are pushed onto the stack in the same order that value parameters are pushed onto the stack. The difference is that access to the value to which the parameter points is as follows

```
p(int& x,...  
  
    movl 8(%ebp), %eax # reference to x copied to eax  
    movl $5,  
    (%eax)           # x = 5
```

### 23.2.5 References

- <http://www.x86.org>
- <http://www.delorie.com/djgpp>
- <http://www.castle.net/~avly/djasm.html> the DJGPP QuickAsm Programming Guide





## Part VIII

# Minix Laboratory Projects for CE, CS, or SE



## Chapter 24

# Minix Project Information

The laboratory projects are open-ended design problems giving students opportunities to consider important design decisions in a modern operating system. Students are graded on the quality of the design and how it is validated through sample test programs. Examinations also include an important component of design questions.

Here, you'll find information that applies across all of the projects. Project-specific information can be found the following chapters. There will be four projects assigned this quarter, each one due about 2 weeks after it is assigned. Individual project descriptions are found in the following pages.

Project 1 : Writing a simple shell	Due Wed, 3rd week
Project 2 : Scheduling	Due Wed, 5th week
Project 3 : Memory System	Due Wed, 8th week
Project 4 : File System	Due Wed, 10th week

### 24.1 Design & documentation

The assignments in this class do not typically require you to write large quantities of code. For most assignments, you need only write several hundred lines of code, if not fewer. However, the concepts covered in class and in the operating system are quite difficult for most people. As a result, deciding which lines of code to write is very difficult. This means that a good design is crucial to getting your code to work, and well-written documentation is necessary to help you and your group to understand what your code is doing. The design document should contain the following sections:

- Purpose
- Available Resources
- Design
- Testing

The most important thing to do is write your design first and do it early! Each assignment is about two weeks long; your design should be complete by the end of the first week. Doing the design first has many advantages:

- You understand the problem and the solution before writing code.
- You can discover issues with your design before wasting time writing code that you'll never use.
- You can get help with the concepts without getting bogged down in complex code.
- You'll save debugging time by knowing exactly what you need to do.

Doing your design early is the single most important factor for success in completing your programming projects!

## 24.2 Packaging Your Software - tar

The file compression utilities are used to decrease storage requirements and to reduce the time it takes to transmit files.

**compress** *file* *compressedFile.Z* The compress utility. Compressed files should have a .Z suffix.

**uncompress** *file.Z* The uncompress utility.

**gzip** *file* *zippedFile.gz* The GNU zip utility. Zipped files should have a .gz suffix

**gunzip** *file.gz* The GNU unzip utility.

The tar (tape **a**rchive) utility is used to construct a linear representation of the UNIX hierarchical file structure and restore the hierarchical structure from the linear representation. In addition, it may be used to compress and decompress the files. It was originally designed to save file systems on tapes as a backup.

**tar cf** *theArchive.tar theDirectory* - creates a tar file of *theDirectory* and its contents and places it in the working directory.

**tar cfz** *theArchive.tgz theDirectory* - creates a gzipped tar file of *theDirectory* and its contents and places it in the working directory.

**tar xf** *theTarFile.tar* - extracts the tar file into the working directory, duplicating the tarred directory structure.

**tar xfz** *theTarFile.tgz* - gunzips and extracts the tar file into the working directory

**tar tzf** *theTarFile.tgz* - List of the contexts of the tar file

**tar xf** *theTarFile FileOfInterest* - extract the file of interest from the tar file.

## 24.3 Controlling Recompilation – make

**make** is a utility for automatically building applications. Files specifying instructions for make are called Makefiles (usually named **Makefile** or **makefile**). **make** can be used with almost any compiled language.

The basic tool for building an application from source code is the compiler. **make** is a separate, higher-level utility which tells the compiler which source code files to process. It tracks which ones have changed since the last time the project was built and invokes the compiler on only the components that depend on those files. A makefile can be seen as a kind of advanced shell script which tracks dependencies instead of following a fixed sequence of steps.

A makefile consists of lines of text which define a file (or set of files) or a rule name as depending on a set of files. Output files are marked as depending on their source files, for example, and source files are marked as depending on files which they include internally. After each dependency is listed, a series of lines of tab-indented text may follow which define how to transform the input into the output, if the former has been modified more recently than the latter. In the case where such definitions are present, they are referred to as “build scripts” and are passed to the shell to generate the target file. The basic structure is:

```
# Comments use the pound sign (aka hash)
target: dependencies
    command 1
    command 2
    .
    .
    .
    command n
```

Below is a very simple makefile that would compile a source called **helloworld.c** using **cc**, a C compiler. It is executed by the command:

```
make
```

The PHONY tag is a technicality that tells make that a particular target name does not produce an actual file. It is executed by the command:

```
make clean
```

The **\$@** and **\$i** are two of the so-called automatic variables and stand for the target name and so-called “implicit” source, respectively. There are a number of other automatic variables.

Note that in the “clean” target, a minus prefixes the command, which tells make to ignore errors in running the command; make will normally exit if execution of a command fails at any point. In the case of a target to cleanup, typically called “clean”, one wants to remove any files generated by the build process, without exiting if they don’t exist. By tagging the clean target PHONY, we prevent make expecting a file to be produced by that target. Note that in this

particular case, the minus prefixing the command is redundant in the common case, the `-f` or “force” flag to `rm` will prevent `rm` exiting due to files not existing. It may exit with an error on other, unintended errors which may be worth stopping the build for.

```
helloworld: helloworld.o
    cc -o $@ $<

helloworld.o: helloworld.c
    cc -c -o $@ $<

.PHONY: clean
clean:
    -rm -f helloworld helloworld.o
```

### An example makefile for Lab #1

```
# Macro definitions
SHELL = /bin/sh # Limit commands to Bourne shell

myshell: lex.yy.o myshell.o
    gcc -o myshell -L/usr/lib myshell.o lex.yy.o

lex.yy.c: shell.l
    flex shell.l
lex.yy.o: lex.yy.c
    gcc -c lex.yy.c

# assuming files in directory temp, creates a tape archive
tarfile:
    cd ../; tar cfz tarfile.tgz temp; mv tarfile.tgz temp/.; cd temp

# Remove all non-source files.
clean :
    /bin/rm -f myshell lex.yy.* *.o *.tgz
```

## 24.4 Version Control

## 24.5 Alternate Projects

For some alternative project possibilities visit

- <http://www.gumstix.com/>
- <http://www.minix3.org/>

## Chapter 25

# Programming Project #1

### 25.1 Purpose

The main goals for this project are to familiarize you with the MINIX 3 operating system – how it works, how to use it, and how to compile code for it and to give you an opportunity to learn how to use system calls. To do this, you’re going to implement a Unix shell program. A shell is simply a program that conveniently allows you to run other programs; your shell will resemble the shell that you’re familiar with from logging into Unix computers.

### 25.2 Basics

Before going on to the rest of the assignment, get MINIX running.

You are provided with the files `shell.1` and `myshell.c` that contain some code that calls `getline()`, a function provided by `shell.1` to read and parse a line of input. The `getline()` function returns an array of pointers to character strings. Each string is either a word containing the letters, numbers, period (`.`), and forward slash (`/`), or a character string containing one of the special characters: `( ) < > | & ;` (these all have syntactical meaning to the shell).

The files are found in the directory with this document and in the last two sections of this chapter.

To compile `shell.1`, you have to use the `lex` command in MINIX.

```
lex shell.1
```

This will produce a file called `lex.yy.c`. You must then compile and link `lex.yy.c` and `myshell.c` in order to get a running program. In the link step, you also have to use `-L/usr/lib` to get everything to work properly. Use `cc` for compiling and linking.

```
cc -L/usr/lib myshell.c lex.yy.c
```

The resulting executable is `a.out` which is executed by `./a.out`. If you prefer the executable to be named `shell`, the compile using

```
cc -o shell -L/usr/lib myshell.c lex.yy.c
```

## 25.3 Details

Your shell must support the following:

1. The internal shell command `exit` which terminates the shell.  
**Concepts:** shell commands, exiting the shell  
**System calls:** `exit()`
2. A command with no arguments.  
**Example:** `ls`  
**Details:** Your shell must block until the command completes and, if the return code is abnormal, print out a message to that effect. This holds for *all* command strings in this assignment.  
**Concepts:** Forking a child process, waiting for it to complete, synchronous execution.  
**System calls:** `fork()`, `execvp()`, `exit()`, `wait()`
3. A command with arguments.  
**Example:** `ls -l`  
**Details:** Argument zero is the name of the command other arguments follow in sequence.  
**Concepts:** Command-line parameters.
4. A command, with or without arguments, whose output is redirected to a file.  
**Example:** `ls -l > file`  
**Details:** This takes the output of the command and puts it in the named file.  
**Concepts:** File operations, output redirection.  
**System calls:** `close()`, `dup()`
5. A command, with or without arguments, whose input is redirected from a file.  
**Example:** `sort < scores`  
**Details:** This uses the named file as input to the command.  
**Concepts:** Input redirection, file operations.  
**System calls:** `close()`, `dup()`
6. A command, with or without arguments, whose output is piped to the input of another command.  
**Example:** `ls -l | more`  
**Details:** This takes the output of the first command and makes it the input to the second command.



**Concepts:** Pipes, synchronous operation

**System calls:** `pipe()`, `close()`, `dup()`

Your shell *must* check and correctly handle *all* return values. This means that you need to read the manual pages for each function and system call to figure out what the possible return values are, what errors they indicate, and what you must do when you get that error.

## 25.4 Deliverables

You must hand in a compressed `tar` file of your project directory, including your design document. You must do a “`make clean`” before creating the tar file. In addition, you should include a `README` file to explain anything unusual to the instructor. Your code and other associated files must be in a single directory; the instructor will copy them to his MINIX installation and compile and run them there.

Do not submit object files, assembler files, or executables. Every file in the `tar` file that could be generated automatically by the compiler or assembler will result in a 5 point deduction from your programming assignment grade. Your design document should be called `design.txt` (if plain ASCII text, with a maximum line length of 75 characters) or `design.pdf` (if in Adobe PDF), and should reside in the project directory with the rest of your code. Formats other than plain text or PDF are not acceptable; please convert other formats (MS Word, LaTeX, HTML, etc.) to PDF. Your design document should describe the design of your assignment in enough detail that a knowledgeable programmer could duplicate your work. This includes descriptions of the data structures you use, all non-trivial algorithms and formulas, and a description of each function including its purpose, inputs, outputs, and assumptions it makes about the inputs or outputs.

## 25.5 Hints

- START EARLY! You should start with your design.
- Build your program a piece at a time. Get one type of command working before tackling another.
- Experiment! You’re running in an emulated system—you *can’t* crash the whole computer (and if you can, let us know...).
- You may want to edit your code outside of MINIX (using your favorite text editor) and copy it into MINIX to compile and run it. This has several advantages:
  - Crashes in MINIX don’t harm your source code (by not writing changes to disk, perhaps).

- Most OSes have better editors than what's available in MINIX.

### *START EARLY!*

- Test your shell. You might want to write up a set of test lines that you can cut and paste (or at least type) into your shell to see if it works. This approach has two advantages: it saves you time (no need to make up new commands) and it gives you a set of tests you can use every time you add features. Your tests might include:
  - Different sample commands with the features listed above
  - Commands with errors: command not found, non-existent input file, etc.
  - Malformed command lines (e.g., `ls -l > | foo`)
- Use RCS to keep multiple revisions of your files. RCS is very space-efficient, and allows you to keep multiple “coherent” versions of your source code and other files (such as Makefiles).
- Did we mention that you should START EARLY!

We assume that you are already familiar with `makefiles` and debugging techniques from earlier classes. If not, this will be a considerably more difficult project because you will have to learn to use these tools as well.

This project doesn't require a lot of coding (typically fewer than 200 lines of code), but does require that you understand how to use MINIX and how to use basic system calls.

You should do your design first, before writing your code. To do this, experiment with the existing shell template (if you like), inserting debugging print statements if it'll help. It may be more “fun” to just start coding without a design, but it'll also result in spending more time than you need to on the project.

IMPORTANT: As with all of the projects this quarter, the key to success is starting early. You can always take a break if you finish early, but it's impossible to complete a 20 hour project in the remaining 12 hours before it's due....

## 25.6 Project groups

The first project must be done individually however, later projects may be done in pairs. It's vital that every student in the class get familiar with how to use the MINIX system; the best way to do that is to do the first project yourself. For the second, third, and fourth projects, you may pick a partner and work together on the project.

## 25.7 shell.l

The following is a `lex` source file.

```
%{
#include <stdio.h>
#include <strings.h>

int _numargs = 10;
char *_args[10];
int _argcount = 0;
%}

WORD [a-zA-Z0-9\\/\.\-]+
SPECIAL [()><|&;*]

%%
_argcount = 0; _args[0] = NULL;

{WORD}|{SPECIAL} {
    if(_argcount < _numargs-1) {
        _args[_argcount++] = (char *)strdup(yytext);
        _args[_argcount] = NULL;
    }
}

\n return (int)_args;

[ \t]+

.

%%
int yywrap(void){return 1;}
/*added 9/27/2006 by A. Aaby with this addition compile without -lfl */

/*added void parameter 9/27/2006 by A. Aaby*/
char **getline() { return (char **)yylex(); }
```

## 25.8 myshell.c

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

```
extern char **getline(void);

int main(void) {
    int i;
    char **args;

    while(1) {
        args = getline();
        for(i = 0; args[i] != NULL; i++) {
            printf("Argument %d: %s\n", i, args[i]);
        }
    }
}
```

## Chapter 26

# Programming Project #2

### 26.1 Purpose

The main goal for this project is to modify the scheduler to be more flexible. You must implement:

- A dual queue scheduler.
- A lottery scheduler.

This project will also teach you how to experiment with operating system kernels, and to do work in such a way that might crash a computer. You'll get experience with modifying a kernel, and may end up with an OS that doesn't work, so you'll learn how to manage multiple kernels, at least one of which works.

### 26.2 Basics

The goal of this assignment is to get everyone up to speed on modifying MINIX 3 and to gain some familiarity with scheduling. In this assignment you are to implement a dual queue scheduler and a lottery scheduler. A lottery scheduler assigns each process some number of tickets, then randomly draws a ticket among those allocated to ready processes to decide which process to run. That process is allowed to run for a set time quantum, after which it is interrupted by a timer interrupt and the process is repeated. The number of tickets assigned to each process determines both the likelihood that it will run at each scheduling decision as well as the relative amount of time that it will get to execute. Processes that are more likely to get chosen each time will get chosen more often, and thus will get more CPU time.

One goal of best-effort scheduling is to give I/O-bound processes both faster service and a larger percentage of the CPU when they are ready to run. Both of these things lead to better responsiveness, which is one subjective measure of

computer performance for interactive applications. CPU-bound processes, on the other hand, can get by with slower service and a relatively lower percentage of the CPU when there are I/O-bound processes that want to run. Of course, CPU-bound processes need lots of CPU, but they can get most of it when there are no ready I/O-bound processes. One fairly easy way to accomplish this in a lottery scheduler is to give I/O-bound processes more tickets – when they are ready they will get service relatively fast, and they will get relatively more CPU than other CPU-bound processes.

The key question is how to determine which processes are I/O-bound and which are CPU-bound. One way to do this is to look at whether or not processes block before using up their time quantum. Processes that block before using up their time quantum are doing I/O and are therefore more I/O-bound than those that do not. On the other hand, processes that do not block before using up a time quantum are more CPU-bound than those that do. So, one way to do this is to start with every process with some specified number of tickets. If a process blocks before using up its time quantum, give it another ticket (up to some set maximum, say 10). If it does not block before using up its time quantum, take a ticket away (down to some set minimum, say 1). In this way, processes that tend to do relatively little processing after each I/O completes will have relatively high numbers of tickets and processes that tend to run for a long time will have relatively low numbers of tickets. Those that are in the middle will have medium numbers of tickets.

This system has several important parameters: time quantum, minimum and maximum numbers of tickets, and the speed at which tickets are given and taken away.

## 26.3 Details

In this project, you will modify the scheduler for MINIX. This should mostly involve modifying code in `kernel/proc.c` (All of the source code, except where specified explicitly, is in `/usr/src`), specifically the `sched()` and `pick_proc()` functions (and perhaps `enqueue()` and `dequeue()`). You may also need to modify `kernel/proc.h` to add elements to the `proc` structure and modify queue information (`NR_SCHED_QUEUES`, `TASK_Q`, `IDLE_Q`, etc.) and may need to modify `PRIO_MIN` and `PRIO_MAX` in `/usr/include/sys/resource.h`. Process priority is set in `do_getsetpriority()` in `servers/pm/misc.c` (don't worry—the code in here is *very* simple), which calls `do_nice()` in `kernel/system.c`. You might be better off just using the `nice()` system call, which calls `do_nice()` directly. You'll probably want to modify what `do_nice()` does—for lottery scheduling, `nice()` can be used to assign or take away tickets.

The current MINIX scheduler is relatively simple. It maintains 16 queues of “ready” processes, numbered 0-15. Queue 15 is the lowest priority (least likely to run), and contains only the `IDLE` task. Queue 0 is the highest priority, and contains several kernel tasks that never get a lower priority. Queues 1-14 contain all of the other processes. Processes have a maximum priority (remember, higher

priorities are closer to 0), and should never be given a higher priority than their maximum priority.

### 26.3.1 Lottery Scheduling

The first approach to scheduling is to use lottery scheduling.<sup>1</sup> There are a number of problems with standard priority-based Unix scheduling algorithms. First, there is no way to insulate users from each other. It is possible for a single user to monopolize the CPU simply by starting many processes. Second, there is no way to directly control relative execution rates. I.e. a user or administrator cannot specify that one task should get half as much CPU as another.

Lottery scheduling was proposed to address the problems mentioned above. In lottery scheduling each task is given some number of tickets. When it is time choose a new task, a lottery is held, and the task holding the winning ticket is allowed to run. This addresses the problem of specifying relative execution rates; a task is expected to run in proportion to the number of tickets it holds.

The problem of user insulation can be addressed by a slight extension of the basic lottery scheduling algorithm: Each user is assigned a number of base tickets. A currency can then be defined that allows the user to distribute as many tickets as he likes to his own tasks, backing those tickets with his base tickets. Lotteries are then performed after task's tickets are scaled by the number of base tickets that they represent. The idea of currencies can be extended indefinitely to implement hierarchical resource management.

As it stands, this lottery scheduling algorithm suffers from at least one major drawback. Tasks that consistently use less than their share of the quantum will not receive their full share of the processor. Waldsprger and Weihl suggest the use of compensation tickets to address this issue. This involves boosting a task's tickets by a factor. With this modification, tasks can expect to receive the appropriate share of the CPU even if they do not use their entire quantum.

Compensation tickets also have the effect of increasing interactivity in lottery scheduling. In general, interactive processes spend most of their time waiting for user input, and do not usually use all of their quantum. With compensation tickets, these tasks will tend to be scheduled more frequently.

In an implementation of lottery scheduling, one can make use of the existing linux scheduling infrastructure as much as possible.<sup>2</sup> That is, rather than creating data structures and support code specifically to be used for lottery scheduling, simply reinterpret existing code wherever possible. For example, rather than defining a new field: `number_of_tickets`, store a task's tickets in the existing `priority` field.

The primary advantage of reusing code in this manner is that development can be accomplished more quickly. There are a number of possible disadvantages:

---

<sup>1</sup>[http://www.usenix.org/publications/library/proceedings/osdi/full\\_papers/waldsprger.pdf](http://www.usenix.org/publications/library/proceedings/osdi/full_papers/waldsprger.pdf)

<sup>2</sup>see also [http://www.usenix.org/events/usenix99/full\\_papers/petrou/petrou.pdf](http://www.usenix.org/events/usenix99/full_papers/petrou/petrou.pdf)

- Lottery scheduling cannot coexist with the standard linux scheduler in the same kernel.
- Existing code may alter scheduling values in unanticipated ways.
- Names are not consistent with the concepts they represent.

Given the time constraints, these disadvantages were not serious enough to justify starting from scratch by writing data structures specifically for lottery scheduling.

System processes (queues 0-15) are run using their original algorithm, and queue 20 now contains the idle process. However, queue 16 contains *all* of the user processes, each of which has some number of tickets. The default number of tickets for a new process is 5. However, processes can add or subtract tickets by calling `setpriority(ntickets)`, which will increase the number of tickets by `ntickets` (note that a negative argument will take tickets away). Each time the scheduler is called, it should randomly select a ticket (by number) and run the process holding that ticket. Clearly, the random number must be between 0 and `nTickets-1`, where `nTickets` is the sum of all the outstanding tickets. You may use the `random()` call (you may need to use the random number code in `/usr/src/lib/other/random.c`) to generate random numbers and the `srandom()` call to initialize the random number generator. A good initialization function to use would be the current date.

For dynamic priority assignment, you should modify lottery scheduling to decrease the number of tickets a process has by 1 each time it receives a full quantum, and increase its number of tickets by 1 each time it blocks without exhausting its quantum. A process should never have fewer than 1 ticket, and should never exceed its original (desired) number of tickets.

You must implement lottery scheduling as follows:

1. Basic lottery scheduling. Start by implementing a lottery scheduler where every process starts with 5 tickets and the number of tickets each process has does not change.
2. Lottery scheduling with dynamic priorities. Modify your scheduler to have dynamic priorities, as discussed above.

New processes are created and initialized in `kernel/system/do_fork.c`. This is probably the best place to initialize any data structures.

### 26.3.2 Dual Round-Robin Queues

The second algorithm you need to implement uses two round robin queues. Processes are placed into the first queue when they are created, and move to the second queue after they have *completed* five quanta (that is, after they have been scheduled five times without waiting for I/O first). The scheduler runs all of the processes in the first queue once and then runs a single process from the second queue. This can be implemented in several ways; one possibility is



to include a “pseudo-process” in the first queue that, when at the front of the queue, causes a process from the second queue to be run.

Assume the following processes are in the two queues:

- Queue 1: P1 , P2 , P3
- Queue 2: P4 , P5 , P6 , P7

The scheduler would run processes in this order:

P1 , P2 , P3 , P4 , P1 , P2 , P3 , P5 , P1 , P2 , P3 , P6 ...

This allows long-running processes to make (slow) progress, but gives high priority to short-running processes.

To do this, you should add two *additional* queues to `kernel/proc.c` (perhaps using queues 17 and 18). System processes are scheduled by the same mechanism they use currently, but user processes are scheduled by being initially placed into queue 1, with a later move to queue 2. You can do this by modifying `sched()` and `pick_proc()` in `kernel/proc.c`. You might also need to modify `enqueue()` and `dequeue()`, and should feel free to modify any other files you like.

## 26.4 Deliverables

You must hand in a compressed `tar` file of your project directory, including your design document. You must do a “`make clean`” before creating the tar file. In addition, you should include a `README` file to explain anything unusual to the instructor. Your code and other associated files must be in a single directory; the instructor will copy them to his MINIX installation and compile and run them there.

Do not submit object files, assembler files, or executables. Every file in the `tar` file that could be generated automatically by the compiler or assembler will result in a 5 point deduction from your programming assignment grade.

Your design document should be called `design.txt` (if plain ASCII text, with a maximum line length of 75 characters) or `design.pdf` (if in Adobe PDF), and should reside in the project directory with the rest of your code. Formats other than plain text or PDF are not acceptable; please convert other formats (MS Word, LaTeX, HTML, etc.) to PDF. Your design document should describe the design of your assignment in enough detail that a knowledgeable programmer could duplicate your work. This includes descriptions of the data structures you use, all non-trivial algorithms and formulas, and a description of each function including its purpose, inputs, outputs, and assumptions it makes about the inputs or outputs.

## 26.5 Hints

- *START EARLY!* You should start with your design, and check it over with the course staff.

- Experiment! You're running in an emulated system—you *can't* crash the whole computer (and if you can, let us know...).
- You may want to edit your code outside of MINIX (using your favorite text editor) and copy it into MINIX to compile and run it. This has several advantages:
  - Crashes in MINIX don't harm your source code (by not writing changes to disk, perhaps).
  - Most OSes have better editors than what's available in MINIX.

### *START EARLY!*

- Test your scheduler. To do this, you might want to write several programs that consume CPU time and occasionally print out values, typically identifying both current process progress and process ID (example :P1-0032 for process 1, iteration 32). Keep in mind that a smart compiler will optimize away an empty loop, so you might want to use something like `longrun.c` for your long-running programs.
- Your scheduler should be statically selected at boot time. However, there's no reason you can't have the code for both lottery and dual-queue scheduling in the OS at one time. At the least, you should have a single file and use `#ifdef` to select which scheduling algorithm to include.
- For lottery scheduling, keep track of the total number of tickets in a global variable in `proc.c`. This makes it easier to pick the ticket. You can then walk through the list of processes to find the one to use next.
- Use RCS to keep multiple revisions of your files. RCS is very space-efficient, and allows you to keep multiple "coherent" versions of your source code and other files (such as Makefiles).
- Did we mention that you should START EARLY!

We assume that you are already familiar with `makefiles` and debugging techniques from earlier classes. If not, this will be a considerably more difficult project because you will have to learn to use these tools as well.

This project doesn't require a lot of coding (typically fewer than 200 lines of code), but does require that you understand how to use MINIX and how to use basic system calls.

You should do your design *first*, before writing your code. To do this, experiment with the existing shell template (if you like), inserting debugging print statements if it'll help. It may be more "fun" to just start coding without a design, but it'll also result in spending more time than you need to on the project.

**IMPORTANT:** As with all of the projects this quarter, the key to success is starting early. You can always take a break if you finish early, but it's impossible to complete a 20 hour project in the remaining 12 hours before it's due....

## 26.6 Project groups

You may do this project, as well as the third and fourth projects with a project partner of your choice. However, you can't switch partners after this assignment, so please choose wisely. If you choose to work with a partner (and we encourage it), you both receive the same grade for the project. One of you should turn in a single file called `partner.txt` with the name of your partner. The other partner should turn in files as above. Please make sure that *both* partners' names and accounts appear on all project files.

## 26.7 longrun.c

```
/*
 * longrun.c
 *
 * This program runs for a very long time, and occasionally prints
 * out messages that identify itself.
 *
 * Author: Ethan L. Miller (elm at cs.ucsc.edu)
 *
 * $Id: longrun.c,v 1.1 2006/05/02 17:23:29 elm Exp $
 */

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

#define LOOP_COUNT_MIN 100
#define LOOP_COUNT_MAX 100000000
int
main (int argc, char *argv[])
{
    char *idStr;
    unsigned int v;
    int i = 0;
    int iteration = 1;
    int loopCount;
    int maxloops;

    if (argc < 3 || argc > 4) {
        printf ("Usage: %s <id> <loop count> [max loops]\n", argv[0]);
        exit (-1);
    }
    /* Start with PID so result is unpredictable */
    v = getpid ();
```

```

/* ID string is first argument */
idStr = argv[1];
/* loop count is second argument */
loopCount = atoi (argv[2]);
if ((loopCount < LOOP_COUNT_MIN) || (loopCount > LOOP_COUNT_MAX)) {
    printf ("%s: loop count must be between %d and %d (passed %d)\n",
        argv[0], LOOP_COUNT_MIN, LOOP_COUNT_MAX, argv[2]);
    exit (-1);
}
/* max loops is third argument (if present) */
if (argc == 4) {
    maxloops = atoi (argv[3]);
} else {
    maxloops = 0;
}

/* Loop forever - use CTRL-C to exit the program */
while (1) {
    /* This calculation is done to keep the value of v unpredictable. Since
       the compiler can't calculate it in advance (even from the original
       value of v and the loop count), it has to do the loop. */
    v = (v << 4) - v;
    if (++i == loopCount) {
        /* Exit if we've reached the maximum number of loops. If maxloops is
           0 (or negative), this'll never happen... */
        if (iteration == maxloops) {
            break;
        }
        printf ("%s:%06d\n", idStr, iteration);
        fflush (stdout);
        iteration += 1;
        i = 0;
    }
}
/* Print a value for v that's unpredictable so the compiler can't
   optimize the loop away. Note that this works because the compiler
   can't tell in advance that it's not an infinite loop. */
printf ("The final value of v is 0x%08x\n", v);
}

```

## Chapter 27

# Programming Project #3

### 27.1 Purpose

The main goal for this project is to modify the MINIX 3 memory system to implement several different allocation strategies: *first fit* (done), *next fit* and *best fit*. You must implement:

- A system call to select the allocation algorithm.
- The various allocation algorithms.
- A user process to collect and process statistics.

You will run a synthetic workload to evaluate the performance of the allocation algorithms that you have developed.

Just like the previous project you will experiment with operating system kernels, and to do work in such a way that may very well crash the (simulated) computer. You'll get experience with modifying a kernel, and may end up with an OS that doesn't work, so you'll learn how to manage multiple kernels, at least one of which works.

You should also review the general project information page before you start this project.

### 27.2 Basics

The goal of this assignment is to give you additional experience in modifying MINIX 3 and to gain some familiarity with memory management. In this assignment you are to implement at least three allocation policies: *first fit* (which is already done, but will need to be made to live peacefully with the new algorithms), *next fit* and *best fit*. An ambitious student (one who wants extra credit, perhaps) would also implement *random fit*, *worst fit* and perhaps a policy of their own creation.

You can find discussions of these algorithms in either Tanenbaum text (indeed, in any operating systems text). Briefly, *first fit* chooses the first hole in which a segment will fit; *next fit*, like *first fit* chooses the first hole where a segment will fit, but beginning its search where it left off the last time (so you will need some persistent state), and *best fit* chooses the hole that is the tightest fit.

You need to implement a system call that will allow the selection of the allocation policy. Note that the policy has a *global* affect on the system, since it applies to all processes. Such a system call should only be executable by root, so you should check the *effective uid* of the process making the call. The default policy should be *first fit*. Each time *next fit* is selected by a system call, the next memory allocation will start at the front of the list (in other words, the *next* pointer is reset). Subsequent allocations using *next fit* pick up where the previous one left off until a new policy is selected by the system call.

## 27.3 Details

In this project, you will modify the memory allocation policy for MINIX. The current MINIX allocation policy is simple: it implements *first fit* only. Changing this policy should mostly involve modifying code in `servers/pm/alloc.c` (All of the source code, except where specified explicitly, is in `/usr/src`).

There needs to be a system call to select the allocation policy. You may create your own system call or modify an existing system call. Your design document should contain the details of how you're going to implement this.

You will implement a user process that will gather statistics regarding the number and the size of the holes. You can get this information via the system call `getsysinfo` (see `servers/pm/misc.c`). You should gather this information *once per second* and compute the number of holes as well as cumulative statistics on their average size and the standard deviation of their size. This information will be printed to a file in the following format:

```
%d\t%d\t%.2f\t%.2f\n", t, nholes, avg_size_in_mb, std_dev_size_in_mb
```

Your program should take one argument: the name of a file to print to. You should use `fopen` and `fprintf` to print the lines to the log file. The value for *t* should start at 0, and increment each time a line is printed.

This experiment wouldn't be much fun without a workload. Since memory allocation in MINIX is pretty much static (pre-allocated data segment sizes), a set of programs (`memuse.tgz.gz`) that will use differing amounts of memory is available in this directory. The main program, `memuse`, will fork off a bunch of other processes that use memory in differing amounts for varying amounts of time. Feel free to modify the code if you like. Further details on specific experiments to run will follow shortly; we will supply specific workloads for you to run against all three (or more) memory allocation algorithms.

### 27.3.1 Deliverables

You must hand in a compressed **tar** file of your project directory, including your design document. You must do a “**make clean**” before creating the tar file. In addition, you should include a **README** file to explain anything unusual to the teaching assistant. Your code and other associated files must be in a single directory; the TA will copy them to his MINIX installation and compile and run them there. You should have two subdirectories in your tar file below your main directory: one containing the kernel source files from the **servers/pm** directory, and the other containing your user program.

Do not submit object files, assembler files, or executables. Every file in the **tar** file that could be generated automatically by the compiler or assembler will result in a 5 point deduction from your programming assignment grade.

Your design document should be called **design.txt** (if plain ASCII text, with a maximum line length of 75 characters) or **design.pdf** (if in Adobe PDF), and should reside in the project directory with the rest of your code. Formats other than plain text or PDF are not acceptable; please convert other formats (MS Word, LaTeX, HTML, etc.) to PDF. Your design document should describe the design of your assignment in enough detail that a knowledgeable programmer could duplicate your work. This includes descriptions of the data structures you use, all non-trivial algorithms and formulas, and a description of each function including its purpose, inputs, outputs, and assumptions it makes about the inputs or outputs

## 27.4 Hints

- *START EARLY!* You should start with your design, and check it over with the course staff.
- Experiment! You’re running in an emulated system—you *can’t* crash the whole computer (and if you can, let us know...).
- You may want to edit your code outside of MINIX (using your favorite text editor) and copy it into MINIX to compile and run it. This has several advantages:
  - Crashes in MINIX don’t harm your source code (by not writing changes to disk, perhaps).
  - Most OSes have better editors than what’s available in MINIX.
- *START EARLY!*
- Look over the operating system code before writing your design document (not to mention your code!). Leverage existing code as much as possible, and modify as little as possible. For this assignment, you should write less than 100 lines of kernel code (unless you do extra allocation algorithms). You might also look at the kernel code to learn how to implement a system call by seeing how it’s done already.

- Test your implementation. To do this, you might want to write several programs that consume various amounts of memory. Keep in mind that a smart compiler will optimize away an empty loop.
- Your allocation must be dynamically selected using a system call. That means the code for every policy must be part of the operating system.
- Use RCS to keep multiple revisions of your files. RCS is very space-efficient, and allows you to keep multiple “coherent” versions of your source code and other files (such as Makefiles).
- Did we mention that you should START EARLY!

We assume that you are already familiar with `makefiles` and debugging techniques from earlier classes. If not, this will be a considerably more difficult project because you will have to learn to use these tools as well.

This project doesn’t require a lot of coding (typically fewer than 200 lines of code), but does require that you understand how to use MINIX and how to use basic system calls.

You should do your design *first*, before writing your code. To do this, experiment with the existing shell template (if you like), inserting debugging print statements if it’ll help. It may be more “fun” to just start coding without a design, but it’ll also result in spending more time than you need to on the project.

IMPORTANT: As with all of the projects this quarter, the key to success is starting early. You can always take a break if you finish early, but it’s impossible to complete a 20 hour project in the remaining 12 hours before it’s due....

## 27.5 Project groups

You may do this project, as well as the fourth project with a project partner of your choice. However, you can’t switch partners if you already had a partner for Project #2. If you choose to work with a partner (and we encourage it), you both receive the same grade for the project. One of you should turn in a single file called `partner.txt` with the name of your partner. The other partner should turn in files as above. Please make sure that *both* partners’ names and accounts appear on all project files.



## Chapter 28

# Programming Project #4

### 28.1 Purpose

The main goal for this project is to use a combination of system calls and user program to maintain Merkle hash trees in MINIX 3. You must implement:

- A system call that returns the path to a file that has been modified.
- A user program that recalculates the MD5 hash value for a file or directory, and recursively adjusts the hash values for parent directories (see below for details).

As with the previous project, you will experiment with operating system kernels, and to do work in such a way that may very well crash the (simulated) computer. You'll get experience with modifying a kernel, and may end up with an OS that doesn't work, so you'll learn how to manage multiple kernels, at least one of which works.

You should also read over the general project information page before you start this project.

### 28.2 Basics

The goal of this assignment is to give you additional experience in modifying MINIX 3 and to gain some familiarity with file systems, system calls, and Merkle hash trees (see Wikipedia on hash trees). You should make minimal changes to the kernel, with most of your code being written in a user-level program.

This assignment requires you to implement Merkle hash trees, which can be used to easily detect files that have been modified. The hash of the files and subdirectories in a directory are stored in a file named `.hashes` in the directory. Calculating the hash of a file is relatively straightforward—simply use the functions in `md5sum.c` to step through all of the bytes, generating a hash value. Calculating a hash value of a directory is similar; the hash of a directory

is the hash of the `.hashes` file in the directory. If a file's content changes, its hash value will change as well. This will cause a change to the `.hashes` file in its directory, which will result in the directory's hash value changing. This change will then "ripple" up the directory tree until it reaches the root. The user process you write will have to keep the hash tree up to date, recursively going up the directory tree (using the `..` link to a directory's parent) to keep the hash values correct. The program should take as an argument a "root" directory below which it initializes the hash tree (checks correctness on startup), and then go into a loop calling the system for the names of changed files or directories, ignoring names that don't start with the "root," and making updates to the tree below the root when they occur. Note that the `.hashes` file must be kept sorted numerically (alphabetically) and should include both hash values (in hexadecimal) and file names. A sample `.hashes` file (`hashes-sample.txt`) is available.

So far, all of the code can (and should!) be implemented at user level. However, the user process that manages these changes needs to know when it should recheck a file. To do this, you need to implement a system call that returns the name of each file that is closed (after being written), created, unlinked, truncated or renamed, or when a directory is created (`mkdir`) or deleted (`rmdir`). You only need a single system call to do this; all of the changes can be reported to the same call, with the user process figuring out the difference if necessary. The system call should buffer up file names, returning one name per call to the system call. It may be difficult to figure out the file name when the file is closed; you should consider putting the name into a buffer when the file is opened for writing, and then returning the name when the file is closed. Don't worry about files that are opened for writing but never actually written; they can be returned and will be ignored by the user program if needed. However, your system call should ignore `.hashes` files. You may use a set of fixed-size buffers if you like; 200 buffers of 150 characters each should be plenty of space (we won't test the program by sending lots of files at it). If you run out of buffer space, you may throw out a randomly-selected buffered name. This way, your code will work correctly even if nobody calls the system call.

## 28.3 Deliverables

You must hand in a compressed `tar` file of your project directory, including your design document. You must do a "`make clean`" before creating the `tar` file. In addition, you should include a `README` file to explain anything unusual to the teaching assistant. Your code and other associated files must be in a single directory; the TA will copy them to his MINIX installation and compile and run them there. You should have two subdirectories in your `tar` file below your main directory: one containing the kernel source files from the `servers/fs` directory, and the other containing your user program.

Do not submit object files, assembler files, or executables. Every file in the `tar` file that could be generated automatically by the compiler or assembler will

result in a 5 point deduction from your programming assignment grade.

Your design document should be called `design.txt` (if plain ASCII text, with a maximum line length of 75 characters) or `design.pdf` (if in Adobe PDF), and should reside in the project directory with the rest of your code. Formats other than plain text or PDF are not acceptable; please convert other formats (MS Word, LaTeX, HTML, etc.) to PDF. Your design document should describe the design of your assignment in enough detail that a knowledgeable programmer could duplicate your work. This includes descriptions of the data structures you use, all non-trivial algorithms and formulas, and a description of each function including its purpose, inputs, outputs, and assumptions it makes about the inputs or outputs.

## 28.4 Hints

- *START EARLY!* You should start with your design, and check it over with the course staff.
- Experiment! You're running in an emulated system—you *can't* crash the whole computer (and if you can, let us know...).
- You may want to edit your code outside of MINIX (using your favorite text editor) and copy it into MINIX to compile and run it. This has several advantages:
  - Crashes in MINIX don't harm your source code (by not writing changes to disk, perhaps).
  - Most OSes have better editors than what's available in MINIX.
- *START EARLY!*
- Look over the operating system code **before** writing your design document (not to mention your code!). Leverage existing code as much as possible, and modify as little as possible.
- A good place to modify the system calls is right before they exit. You'll want to modify `do_close`, `do_mkdir`, *etc.* right before they return to the caller. Don't worry about returning a file that wasn't really modified—your user program can simply ignore it if it discovers no changes were made.
- For this assignment, you should write fewer than 200 lines of kernel code. You might want to look at the kernel code to learn how to implement a system call by seeing how it's done already.
- Waiting for something to happen and then resuming the process afterwards is very similar to what the `select()` system call does (in `fs/select.c`). You can probably reuse much of that code for your system call. In particular, look at what `suspend()` and `revive()` do. Use the same mechanisms you used in Project #3 to return strings from the kernel.

- Write the hash tree generator first, without the system call. You can still test whether the generator works without any system calls.
- Use RCS to keep multiple revisions of your files. RCS is very space-efficient, and allows you to keep multiple “coherent” versions of your source code and other files (such as Makefiles).
- Did we mention that you should START EARLY!

We assume that you are already familiar with `makefiles` and debugging techniques from earlier classes. If not, this will be a considerably more difficult project because you will have to learn to use these tools as well.

This project doesn’t require a lot of coding (typically fewer than 200 lines of code), but does require that you understand how to use MINIX and how to use basic system calls.

You should do your design *first*, before writing your code. To do this, experiment with the existing shell template (if you like), inserting debugging print statements if it’ll help. It may be more “fun” to just start coding without a design, but it’ll also result in spending more time than you need to on the project.

**IMPORTANT:** As with all of the projects this quarter, the key to success is starting early. You can always take a break if you finish early, but it’s impossible to complete a 20 hour project in the remaining 12 hours before it’s due....

## 28.5 Project groups

You may do this project with a project partner of your choice. However, you can’t switch partners if you already had a partner for Project #2. If you choose to work with a partner (and we encourage it), you both receive the same grade for the project. One of you should turn in a single file called `partner.txt` with the name of your partner. The other partner should turn in files as above. Please make sure that *both* partners’ names and accounts appear on all project files.

## Part IX

# Simulated System Laboratory Projects for CE, CS, or SE



## Chapter 29

# Operating System Project & Labs

The purpose of the OS Project is to strengthen your software design and implementation skills, responsible team work, clear communication, and to facilitate your understanding of OS design concepts.

The class will be divided into teams of two or three. The class must agree on a common interface for each of the modules so that modules may be exchanged and compiled without change.

Test harnesses must be constructed to test and validate each module.

Evaluate your design and implementation from the perspective of

- functionality
- reliability
- usability
- efficiency
- maintainability
- portability

Your project should be representative of the highest quality work you can do and should conform to common software engineering standards for coding and documentation practice. You may be asked to demonstrate your project to the class.

In your labs you will design and implement each of the following:

1. Security and Protection
2. File System Manager
3. Memory Manager

4. Process Manager
5. CPU and Interrupts
6. User Interface

## 29.1 Lab 0: EWD and SM

The goals of this lab are to improve your knowledge of and skill in using the C language and to begin the construction of a simulated hardware platform and an OS for the platform.

1. Learn the programming language EWD and the architecture of SM
2. Critique the documentation and code for ewd and sm
3. Modify the EWD code to produce a collection of executable files from the given programs.
4. Begin the construction of the hardware platform and OS. Using the processor code (SM.h)
  - Construct a simple process manager to manage PCBs for processes
  - Construct a control program and user shell for interacting with your OS project.

The resulting system

- should provide single step capabilities and display of system state.
- should allow the user to load programs
- should permit interaction with the file system.
- should permit interaction with the process manager.

<http://www.cs.wvc.edu/~aabyan/Code/EWD>

## 29.2 Lab 1: Modify the SM to support the following:

The goal of this lab is to enhance the simulated hardware to support process and memory management.

Modify the simulated hardware to provide

1. An interrupt mechanism
2. Memory protection mechanisms include
  - (a) base and limit registers



### 29.3. LAB 2: CREATE A SIMPLE NUCLEUS OF AN OPERATING SYSTEM FOR SM177

- (b) paging
- (c) segments

provide support for paging.

3. Privileged instruction set
  - (a) enabling and disabling interrupts
  - (b) switching a processor between processes
  - (c) accessing registers used by the memory protection hardware
  - (d) halting the central processor
4. Real time clock: interrupts at fixed intervals

Best code will be selected to be used by all groups in succeeding phases.

## 29.3 LAB 2: Create a simple nucleus of an operating system for SM

1. Process representation, loader, and command interpreter
  - (a) Create a representation for processes (process control block and appropriate queues)
  - (b) Create a simple loader to load programs from files into the memory of SM
  - (c) Create a simple batch and interactive command line interpreter to allow programs to be loaded
2. First level interrupt handler
  - (a) determine the source of the interrupt
  - (b) service the interrupt
3. A *dispatcher* (low-level scheduler) which switches the processor between processes
  - If current process is still suitable to run, continue it else
  - (a) Save environment of current process
  - (b) Retrieve the environment of the most suitable process from its descriptor
  - (c) Transfer control to the restored process
4. Wait and signal primitives

Best code will be selected to be used by all groups in succeeding phases.

The process manager manages processes in response to interrupts and system calls. It interacts with the memory manager and the file manager.

1. Define a process control block (PCB) and a collection of queues..
2. Construct a short term scheduler.
3. Design and implement a scheme to describe process behavior that may be used to simulate process behavior for your project. It should provide for long and short cpu bursts, system calls, and require interaction with the memory manager and the file system manager.
4. System security has three goals secrecy, integrity, and availability. How may each of these goals be satisfied in the design of the process manager?

## 29.4 LAB 3: Hard Drive and I/O

Create a hard drive subsystem with a simple interface to transfer blocks between RAM and the hard drive, a mechanism to keep trace of free blocks, and a an interface used to allocate and free blocks. Specifically,

1. Construct a module to simulate a hard drive. The hard drive consists of  $N$  blocks. The device driver for the hard drive provides the following: given a block number, a memory frame number, and one of two commands, a block of data is written from the hard drive or memory to memory or the hard drive.

```
hd(Block#, Frame#, Read)
hd(Block#, Frame#, Write)
```

It must be a persistent data structure.

2. A mechanism to keep track of free blocks
3. An interface for allocation of free blocks and deallocation of blocks.

Best code will be selected to be used by all groups in succeeding phases.

## 29.5 LAB 4: File System, Memory Manager, and Swap System

The file system manager manages free space, files, directories, and swap space.

1. Construct a module to manage free space. Remember to consider what must happen when the system is booted up and when the system is shut down. Provide a description of an interface for the free space manager.
2. Design and implement a file system. Remember to include interaction with the free space manager. Provide a description of an interface for primitive file operations.

3. Design and implement a directory system. Provide a description of an interface for a directory system.
4. System security has three goals secrecy, integrity, and availability. How may each of these goals be satisfied in the design of the file system manager?
5. Provide support for
  - a paged memory management system and
  - a swap system.

Best code will be selected to be used by all groups in succeeding phases.

## 29.6 LAB 5: Memory Manager

The memory manager manages the primary store (RAM) allocating frames to processes and interacts with the file system manager. The primary store consists of  $N$  frames.

1. Construct a module to manage free frames.
2. Construct a module to allocate frames to and reclaim frames from processes.
3. Construct a module to swap pages between the file system manager and the the memory manager.
4. Construct a virtual memory module to provide demand paging
5. System security has three goals secrecy, integrity, and availability. How may each of these goals be satisfied in the design of the memory manager?

## 29.7 LAB 6: Multiprocessor Management

Extend the system with muliprocessor (multiple instances of the cpu). Provide for synchronized access to shared resources, appropriate scheduling algorithms, run queues, and load balancing..



**Part X**

**Laboratory Projects for IS  
or IT**



## Chapter 30

# IS/IT OS Project

Setup and administer a Linux, Solaris, and/or MS-Windows2000 Server  
References:

- Kaplenk, Joe Unix System Administrator's Interactive Workbook Prentice-Hall PTR 1999
- Helmick, Jason Preparing for MCSE Certification (Windows 2000 Server) DDC Publishing 2000

Install and evaluate the AFS