

Computer Systems: An Integrated Approach to Architecture and Operating Systems

Table of Contents

Preface.....	i
Why a New Book on Computer Systems?	i
The structure of the book	ii
Where Does This Textbook Fit into the Continuum of CS Curriculum?.....	iii
Supplementary Material for Teaching an Integrated Course in Systems	v
Example project ideas included in the supplementary material	vi
Chapter 1 Introduction.....	1-1
1.1 What is Inside a Box?	1-1
1.2 Levels of Abstraction in a Computer System	1-2
1.3 The Role of the Operating System	1-5
1.4 What is happening inside the box?.....	1-7
1.4.1 Launching an application on the computer	1-9
1.5 Evolution of Computer Hardware	1-9
1.6 Evolution of Operating Systems.....	1-11
1.7 Roadmap of the rest of the book.....	1-12
1.8 Review Questions.....	1-13
Chapter 2 Processor Architecture.....	2-1
2.1 What is involved in processor design?.....	2-2
2.2 How do we design an instruction set?.....	2-2
2.3 A Common High-Level Language Feature Set.....	2-3

2.4 Expressions and Assignment Statements	2-4
2.4.1 Where to keep the operands?	2-4
2.4.2 How do we specify a memory address in an instruction?	2-9
2.4.3 How wide should each operand be?.....	2-10
2.4.4 Endianness	2-12
2.4.5 Packing of operands and Alignment of word operands.....	2-15
2.5 High-level data abstractions	2-17
2.5.1 Structures.....	2-17
2.5.2 Arrays	2-18
2.6 Conditional statements and loops	2-20
2.6.1 If-then-else statement	2-20
2.6.2 Switch statement	2-22
2.6.3 Loop statement.....	2-24
2.7 Checkpoint	2-24
2.8 Compiling Function calls	2-24
2.8.1 State of the Caller	2-25
2.8.2 Remaining chores with procedure calling.....	2-28
2.8.3 Software Convention	2-30
2.8.4 Activation Record.....	2-36
2.8.5 Recursion	2-37
2.8.6 Frame Pointer	2-37
2.9 Instruction-Set Architecture Choices	2-40
2.9.1 Additional Instructions.....	2-40
2.9.2 Additional addressing modes.....	2-41
2.9.3 Architecture styles	2-41
2.9.4 Instruction Format	2-42
2.10 LC-2200 Instruction Set.....	2-45
2.10.1 Instruction Format	2-45
2.10.2 LC-2200 Register Set	2-48
2.11 Issues influencing processor design.....	2-48
2.11.1 Instruction-set.....	2-48
2.11.2 Influence of applications on instruction-set design.....	2-50

2.11.3 Other issues driving processor design	2-51
2.12 Summary	2-52
2.13 Review Questions.....	2-53

Chapter 3 Processor Implementation3-1

3.1 Architecture versus Implementation	3-1
3.2 What is involved in Processor Implementation?	3-2
3.3 Key hardware concepts.....	3-3
3.3.1 Circuits	3-3
3.3.2 Hardware resources of the datapath.....	3-3
3.3.3 Edge Triggered Logic	3-5
3.3.4 Connecting the datapath elements	3-7
3.3.5 Towards bus-based Design	3-10
3.3.6 Finite State Machine (FSM).....	3-13
3.4 Datapath Design.....	3-15
3.4.1 ISA and datapath width.....	3-17
3.4.2 Width of the Clock Pulse.....	3-18
3.4.3 Checkpoint	3-18
3.5 Control Unit Design	3-18
3.5.1 ROM plus state register	3-19
3.5.2 FETCH macro state	3-23
3.5.3 DECODE macro state.....	3-25
3.5.4 EXECUTE macro state: ADD instruction (part of R-Type).....	3-26
3.5.5 EXECUTE macro state: NAND instruction (part of R-Type)	3-28
3.5.6 EXECUTE macro state: JALR instruction (part of J-Type)	3-28
3.5.7 EXECUTE macro state: LW instruction (part of I-Type)	3-29
3.5.8 EXECUTE macro state: SW and ADDI instructions (part of I-Type)	3-30
3.5.9 EXECUTE macro state: BEQ instruction (part of I-Type)	3-31
3.5.10 Engineering a conditional branch in the microprogram.....	3-32
3.5.11 DECODE macro state revisited	3-34
3.6 Alternative Style of Control Unit Design	3-35
3.6.1 Microprogrammed Control.....	3-35

3.6.2 Hardwired control.....	3-36
3.6.3 Choosing between the two control design styles.....	3-37
3.7 Summary	3-38
3.8 Historical Perspective.....	3-38
3.9 Review Questions.....	3-41

Chapter 4 Interrupts, Traps and Exceptions 4-1

4.1 Discontinuities in program execution	4-2
4.2 Dealing with program discontinuities	4-4
4.3 Architectural enhancements to handle program discontinuities	4-7
4.3.1 Modifications to FSM	4-8
4.3.2 A simple interrupt handler.....	4-9
4.3.3 Handling cascaded interrupts	4-10
4.3.4 Returning from the handler	4-13
4.3.5 Checkpoint	4-14
4.4 Hardware details for handling program discontinuities	4-14
4.4.1 Datapath details for interrupts	4-14
4.4.2 Details of receiving the address of the handler	4-16
4.4.3 Stack for saving/restoring	4-18
4.5 Putting it all together	4-20
4.5.1 Summary of Architectural/hardware enhancements.....	4-20
4.5.2 Interrupt mechanism at work	4-20
4.6 Summary	4-23
4.7 Review Questions.....	4-25

Chapter 5 Processor Performance and Rudiments of Pipelined Processor Design 5-1

5.1 Space and Time Metrics.....	5-1
5.2 Instruction Frequency	5-4
5.3 Benchmarks.....	5-5
5.4 Increasing the Processor Performance	5-9
5.5 Speedup	5-10
5.6 Increasing the Throughput of the Processor	5-14

5.7 Introduction to Pipelining	5-14
5.8 Towards an instruction processing assembly line	5-15
5.9 Problems with a simple-minded instruction pipeline.....	5-17
5.10 Fixing the problems with the instruction pipeline.....	5-18
5.11 Datapath elements for the instruction pipeline	5-20
5.12 Pipeline-conscious architecture and implementation.....	5-22
5.12.1 Anatomy of an instruction passage through the pipeline	5-23
5.12.2 Design of the Pipeline Registers.....	5-26
5.12.3 Implementation of the stages.....	5-27
5.13 Hazards.....	5-27
5.13.1 Structural hazard.....	5-28
5.13.2 Data Hazard.....	5-30
5.13.3 Control Hazard	5-41
5.13.4 Summary of Hazards	5-51
5.14 Dealing with program discontinuities in a pipelined processor	5-52
5.15 Advanced topics in processor design	5-55
5.15.1 Instruction Level Parallelism	5-55
5.15.2 Deeper pipelines	5-56
5.15.3 Revisiting program discontinuities in the presence of out-of-order processing	5-59
5.15.4 Managing shared resources.....	5-60
5.15.5 Power Consumption	5-62
5.15.6 Multi-core Processor Design	5-63
5.15.7 Intel Core Microarchitecture: An example pipeline	5-64
5.16 Summary	5-67
5.17 Historical Perspective.....	5-67
5.18 Review Questions.....	5-68

Chapter 6 Processor Scheduling.....6-1

6.1 Introduction	6-1
6.2 Programs and Processes	6-2
6.3 Scheduling Environments.....	6-7
6.4 Scheduling Basics	6-9

6.5 Performance Metrics	6-12
6.6 Non-preemptive Scheduling Algorithms.....	6-15
6.6.1 First-Come First-Served (FCFS).....	6-15
6.6.2 Shortest Job First (SJF)	6-19
6.6.3 Priority.....	6-21
6.7 Preemptive Scheduling Algorithms.....	6-23
6.7.1 Round Robin Scheduler.....	6-26
6.8 Combining Priority and Preemption	6-31
6.9 Meta Schedulers	6-31
6.10 Evaluation	6-32
6.11 Impact of Scheduling on Processor Architecture.....	6-34
6.12 Summary and a Look ahead.....	6-36
6.13 Linux Scheduler – A case study	6-36
6.14 Historical Perspective.....	6-39
6.15 Review Questions.....	6-41

Chapter 7 Memory Management Techniques..... 7-1

7.1 Functionalities provided by a memory manager	7-1
7.2 Simple Schemes for Memory Management	7-4
7.3 Memory Allocation Schemes	7-8
7.3.1 Fixed Size Partitions	7-9
7.3.2 Variable Size Partitions	7-10
7.3.3 Compaction	7-13
7.4 Paged Virtual Memory	7-14
7.4.1 Page Table	7-17
7.4.2 Hardware for Paging	7-19
7.4.3 Page Table Set up.....	7-20
7.4.4 Relative sizes of virtual and physical memories	7-20
7.5 Segmented Virtual Memory.....	7-21
7.5.1 Hardware for Segmentation	7-26
7.6 Paging versus Segmentation	7-27
7.6.1 Interpreting the CPU generated address.....	7-29

7.7 Summary	7-30
7.8 Historical Perspective.....	7-32
7.8.1 MULTICS	7-33
7.8.2 Intel's Memory Architecture.....	7-35
7.9 Review Questions.....	7-36

Chapter 8 Details of Page-based Memory Management.... 8-1

8.1 Demand Paging	8-1
8.1.1 Hardware for demand paging.....	8-1
8.1.2 Page fault handler.....	8-2
8.1.3 Data structures for Demand-paged Memory Management.....	8-3
8.1.4 Anatomy of a Page Fault	8-5
8.2 Interaction between the Process Scheduler and Memory Manager.....	8-8
8.3 Page Replacement Policies	8-9
8.3.1 Belady's Min.....	8-10
8.3.2 Random Replacement.....	8-10
8.3.3 First In First Out (FIFO)	8-11
8.3.4 Least Recently Used (LRU)	8-13
8.3.5 Second chance page replacement algorithm.....	8-17
8.3.6 Review of page replacement algorithms	8-20
8.4 Optimizing Memory Management.....	8-22
8.4.1 Pool of free page frames.....	8-22
8.4.2 Thrashing.....	8-23
8.4.3 Working set	8-25
8.4.4 Controlling thrashing	8-26
8.5 Other considerations	8-28
8.6 Translation Lookaside Buffer (TLB)	8-28
8.6.1 Address Translation with TLB.....	8-29
8.7 Advanced topics in memory management	8-31
8.7.1 Multi-level page tables.....	8-31
8.7.2 Access rights as part of the page table entry.....	8-34
8.7.3 Inverted page tables	8-34

8.8 Summary	8-34
8.9 Review Questions.....	8-35

Chapter 9 Memory Hierarchy 9-1

9.1 The Concept of a Cache	9-2
9.2 Principle of Locality	9-3
9.3 Basic terminologies	9-4
9.4 Multilevel Memory Hierarchy	9-5
9.5 Cache organization.....	9-8
9.6 Direct-mapped cache organization.....	9-9
9.6.1 Cache Lookup	9-11
9.6.2 Fields of a Cache Entry	9-13
9.6.3 Hardware for direct mapped cache	9-14
9.7 Repercussion on pipelined processor design.....	9-16
9.8 Cache read/write algorithms	9-17
9.8.1 Read access to the cache from the CPU	9-18
9.8.2 Write access to the cache from the CPU	9-19
9.9 Dealing with cache misses in the processor pipeline	9-22
9.9.1 Effect of memory stalls due to cache misses on pipeline performance	9-23
9.10 Exploiting spatial locality to improve cache performance.....	9-25
9.10.1 Performance implications of increased blocksize.....	9-30
9.11 Flexible placement	9-31
9.11.1 Fully associative cache	9-32
9.11.2 Set associative cache	9-34
9.11.3 Extremes of set associativity.....	9-37
9.12 Instruction and Data caches.....	9-39
9.13 Reducing miss penalty	9-40
9.14 Cache replacement policy.....	9-41
9.15 Recapping Types of Misses	9-43
9.16 Integrating TLB and Caches.....	9-46
9.17 Cache controller.....	9-48
9.18 Virtually indexed physically tagged cache	9-49

9.19 Recap of Cache Design Considerations.....	9-52
9.20 Main memory design considerations.....	9-52
9.20.1 Simple main memory	9-53
9.20.2 Main memory and bus to match cache block size.....	9-54
9.20.3 Interleaved memory	9-55
9.21 Elements of a modern main memory systems	9-56
9.21.1 Page mode DRAM	9-61
9.22 Performance implications of memory hierarchy	9-62
9.23 Summary	9-63
9.24 Memory hierarchy of modern processors – An example	9-65
9.25 Review Questions.....	9-66

Chapter 10 Input/Output and Stable Storage..... 10-1

10.1 Communication between the CPU and the I/O devices	10-1
10.1.1 Device controller.....	10-2
10.1.2 Memory Mapped I/O	10-3
10.2 Programmed I/O	10-5
10.3 DMA	10-6
10.4 Buses	10-9
10.5 I/O Processor.....	10-10
10.6 Device Driver.....	10-11
10.6.1 An Example	10-12
10.7 Peripheral Devices	10-15
10.8 Disk Storage	10-17
10.8.1 Saga of Disk Technology.....	10-24
10.9 Disk Scheduling Algorithms.....	10-27
10.9.1 First-Come First Served	10-30
10.9.2 Shortest Seek Time First	10-30
10.9.3 Scan (elevator algorithm).....	10-31
10.9.4 C-Scan (Circular Scan)	10-32
10.9.5 Look and C-Look.....	10-33
10.9.6 Disk Scheduling Summary.....	10-33

10.9.7 Comparison of the Algorithms	10-34
10.10 Solid State Drive	10-36
10.11 Evolution of I/O Buses and Device Drivers	10-38
10.11.1 Dynamic Loading of Device Drivers.....	10-39
10.11.2 Putting it all Together	10-39
10.12 Summary	10-42
10.13 Review Questions.....	10-42

Chapter 11 File System..... 11-1

11.1 Attributes	11-2
11.2 Design Choices in implementing a File System on a Disk Subsystem	11-8
11.2.1 Contiguous Allocation	11-9
11.2.2 Contiguous Allocation with Overflow Area.....	11-12
11.2.3 Linked Allocation.....	11-12
11.2.4 File Allocation Table (FAT)	11-13
11.2.5 Indexed Allocation	11-15
11.2.6 Multilevel Indexed Allocation	11-17
11.2.7 Hybrid Indexed Allocation.....	11-18
11.2.8 Comparison of the allocation strategies	11-21
11.3 Putting it all together	11-22
11.3.1 i-node	11-28
11.4 Components of the File System	11-29
11.4.1 Anatomy of creating and writing files.....	11-30
11.5 Interaction among the various subsystems	11-31
11.6 Layout of the file system on the physical media.....	11-34
11.6.1 In memory data structures	11-37
11.7 Dealing with System Crashes	11-38
11.8 File systems for other physical media.....	11-39
11.9 A summary of modern file systems	11-39
11.9.1 Linux.....	11-39
11.9.2 Microsoft Windows.....	11-45
11.10 Summary	11-47

11.11 Review Questions.....	11-48
-----------------------------	-------

Chapter 12 Multithreaded Programming and Multiprocessors 12-1

12.1 Why Multithreading?	12-1
12.2 Programming support for threads	12-3
12.2.1 Thread creation and termination.....	12-3
12.2.2 Communication among threads	12-6
12.2.3 Read-write conflict, Race condition, and Non-determinism	12-7
12.2.4 Synchronization among threads	12-12
12.2.5 Internal representation of data types provided by the threads library.....	12-19
12.2.6 Simple programming examples	12-20
12.2.7 Deadlocks and livelocks	12-25
12.2.8 Condition variables	12-27
12.2.9 A complete solution for the video processing example	12-30
12.2.10 Rechecking the predicate.....	12-33
12.3 Summary of thread function calls and threaded programming concepts.....	12-36
12.4 Points to remember in programming with threads.....	12-38
12.5 Using threads as software structuring abstraction.....	12-39
12.6 POSIX pthreads library calls summary	12-40
12.7 OS support for threads.....	12-42
12.7.1 User level threads	12-45
12.7.2 Kernel level threads	12-47
12.7.3 Solaris threads: An example of kernel level threads	12-49
12.7.4 Threads and libraries.....	12-50
12.8 Hardware support for multithreading in a uniprocessor.....	12-51
12.8.1 Thread creation, termination, and communication among threads	12-51
12.8.2 Inter-thread synchronization	12-51
12.8.3 An atomic test-and-set instruction	12-52
12.8.4 Lock algorithm with test-and-set instruction.....	12-54
12.9 Multiprocessors	12-55
12.9.1 Page tables	12-56

12.9.2 Memory hierarchy	12-56
12.9.3 Ensuring atomicity	12-59
12.10 Advanced Topics	12-59
12.10.1 OS topics	12-60
12.10.2 Architecture topics.....	12-76
12.10.3 The Road Ahead: Multi- and Many-core Architectures	12-87
12.11 Summary	12-89
12.12 Historical Perspective.....	12-90
12.13 Review Questions.....	12-92

Chapter 13 Fundamentals of Networking and Network Protocols..... 13-1

13.1 Preliminaries	13-1
13.2 Basic Terminologies	13-2
13.3 Networking Software	13-6
13.4 Protocol Stack	13-8
13.4.1 Internet Protocol Stack	13-9
13.4.2 OSI Model.....	13-12
13.4.3 Practical issues with layering	13-13
13.5 Application Layer	13-14
13.6 Transport Layer.....	13-15
13.6.1 Stop and wait protocols	13-17
13.6.2 Pipelined protocols	13-20
13.6.3 Reliable Pipelined Protocol	13-22
13.6.4 Dealing with transmission errors	13-28
13.6.5 Transport protocols on the Internet.....	13-28
13.6.6 Transport Layer Summary.....	13-32
13.7 Network Layer.....	13-32
13.7.1 Routing Algorithms	13-33
13.7.2 Internet Addressing	13-40
13.7.3 Network Service Model.....	13-43
13.7.4 Network Routing Vs. Forwarding	13-47

13.7.5 Network Layer Summary	13-48
13.8 Link Layer and Local Area Networks	13-50
13.8.1 Ethernet	13-50
13.8.2 CSMA/CD.....	13-51
13.8.3 IEEE 802.3.....	13-53
13.8.4 Wireless LAN and IEEE 802.11	13-54
13.8.5 Token Ring.....	13-55
13.8.6 Other link layer protocols	13-57
13.9 Networking Hardware.....	13-58
13.10 Relationship between the Layers of the Protocol Stack	13-63
13.11 Data structures for packet transmission.....	13-63
13.11.1 TCP/IP Header.....	13-65
13.12 Message transmission time	13-66
13.13 Summary of Protocol Layer Functionalities	13-72
13.14 Networking Software and the Operating System	13-73
13.14.1 Socket Library.....	13-73
13.14.2 Implementation of the Protocol Stack in the Operating System.....	13-75
13.14.3 Network Device Driver.....	13-76
13.15 Network Programming using Unix Sockets.....	13-77
13.16 Network Services and Higher Level Protocols	13-85
13.17 Summary	13-86
13.18 Historical Perspective.....	13-87
13.18.1 From Telephony to Computer Networking.....	13-87
13.18.2 Evolution of the Internet	13-90
13.18.3 PC and the arrival of LAN	13-91
13.18.4 Evolution of LAN	13-91
13.19 Review Questions.....	13-94

Chapter 14 Epilogue: A Look Back at the Journey..... 14-1

14.1 Processor Design.....	14-1
14.2 Process	14-1
14.3 Virtual Memory System and Memory Management.....	14-2

14.4 Memory Hierarchy	14-2
14.5 Parallel System.....	14-3
14.6 Input/Output Systems.....	14-3
14.7 Persistent Storage	14-3
14.8 Network	14-4
14.9 Concluding Remarks.....	14-4

Appendix A Network Programming with Unix Sockets ... A-1

A.1 The problem.....	A-1
A.2 Source files provided	A-1
A.3 Makefile	A-1
A.4 Common header file	A-3
A.5 Client source code	A-3
A.6 Server source code	A-7
A.7 Instantiating the client/server programs.....	A-12

Chapter 2 Processor Architecture

(Revision number 19)

Two architectural issues surround processor design: the instruction set and the organization of the machine. There was a time, in the early days of computing (circa 60's and 70's) when processor design was viewed as entirely an exercise in hardware left to electrical engineers. Computers were programmed largely in assembly language and so the fancier the instruction set the simpler the application programs. That was the prevailing conventional wisdom. With the advent of modern programming languages such as Algol in the 60's, and rapid advances in compiler technology, it became very clear that processor design is not simply a hardware exercise. In particular, the instruction-set design is intimately related to how effectively the compiler can generate code for the processor. In this sense, programming languages have exerted a considerable influence on instruction-set design.

Let us understand how the programming language exerts influence on instruction set design. The constructs in high-level language such as assignment statements and expressions map to arithmetic and logic instructions and load/store instructions. High-level languages support data abstractions that may require different precision of operands and addressing modes in the instruction-set. Conditional statements and looping constructs would require conditional and unconditional branching instructions. Further, supporting modularization constructs such as procedures in high-level language may require additional supporting abstractions from the processor architecture.

Applications have a significant influence on the instruction-set design as well. For example, the early days of computing was dominated by scientific and engineering applications. Correspondingly, high-end systems of the 70's and 80's supported floating-point arithmetic in the instruction-set. Circa 2008, the dominant use of computing is in cell phones and other embedded systems, and this trend will no doubt continue as computing weaves its way into the very fabric of society. Streaming applications such as audio and video are becoming commonplace in such gadgets. Correspondingly, requirements of such applications (e.g., a single instruction operating on a set of data items) are influencing the instruction-set design.

It may not always be feasible or cost-effective to support the need of a particular system software or application directly in hardware. For example, in the early days of computing, low-end computers supported floating-point arithmetic via software libraries that are implemented using integer arithmetic available in the instruction-set. Even to this day, complex operations such as finding the cosine of an angle may not be supported directly by the instruction-set in a general-purpose processor. Instead, special system software called *math libraries* implements such complex operations by mapping them to simpler instructions that are part of the instruction-set.

The operating system has influence on instruction set design as well. A processor may appear to be running several programs at the same time. Think about your PC or PDA.

There are several programs running but there are not several processors. Therefore, there needs to be a way of remembering what a particular program is doing before we go on to another program. You may have seen an efficient cook working on four woks simultaneously making four different dishes. She remembers what state each dish is in and at appropriate times adds the right ingredients to the dishes. The operating system is the software entity (i.e., a program in itself) that orchestrates different program executions on the processor. It has its own influence on processor design as will become apparent in later chapters that deal with program discontinuities and memory management.

2.1 What is involved in processor design?

There are hardware resources that we know about from a course on logic design such as registers, arithmetic and logic unit, and the datapath that connects all these resources together. Of course, there are other resources such as main memory for holding programs and data, multiplexers for selecting from a set of input sources, buses for interconnecting the processor resources to the memory, and drivers for putting information from the resources in the datapath onto the buses. We will visit datapath design shortly.

As an analogy, we can think of these hardware resources as the alphabet of a language like English. Words use the alphabet to make up the English language. In a similar manner, the instruction set of the processor uses these hardware resources to give shape and character to a processor. Just as the repertoire of words in a natural language allows us to express different thoughts and emotions, the instruction set allows us to orchestrate the hardware resources to do different things in a processor. Thus, the instruction set is the key element that distinguishes an Intel x86 processor from a Power PC and so on.

As computer users we know we can program a computer at different levels: in languages such as C, Python, and Java; in assembly language; or directly in machine language.

The instruction set is the prescription given by the computer architect as to the capabilities needed in the machine and that should be made visible to the machine language programmer. Therefore, the instruction set serves as a contract between the software (i.e., the programs that run on the computer at whatever level) and the actual hardware implementation. There is a range of choices in terms of implementing the instruction set and we will discuss such choices in later chapters. First, we will explore the issues in designing an instruction set.

2.2 How do we design an instruction set?

As we know, computers evolved from calculating machines, and during the early stages of computer design the choice of instructions to have in a machine was largely dictated by whether it was feasible to implement the instruction in hardware. This was because the hardware was very expensive and the programming was done directly in assembly language. Therefore, the design of the instruction set was largely in the purview of the electrical engineers who had a good idea of the implementation feasibilities. However,

hardware costs came down and as programming matured, high-level languages were developed such that the attention shifted from implementation feasibility to whether the instructions were actually useful; we mean from the point of producing highly efficient and/or compact code for programs written in such high-level languages.

It turns out that while the instruction set orchestrates what the processor does internally, users of computers seldom have to deal with it directly. Certainly, when you are playing a video game, you are not worried about what instructions the processor is executing when you hit at a target. Anyone who has had an exposure to machine language programming knows how error prone that is.

The shift from people writing assembly language programs to compilers translating high-level language programs to machine code is a primary influence on the evolution of the instruction set architecture. This shift implies that we look to a simple set of instructions that will result in efficient code for high-level language constructs.

It is important to state a note of caution. Elegance of instruction-set is important and the architecture community has invested a substantial intellectual effort in that direction. However, an equally important and perhaps a more over-arching concern is the efficacy of the implementation of the instruction-set. We will revisit this matter in the last section of this chapter.

Each high-level language has its own unique syntactic and semantic flavor. Nevertheless, one can identify a baseline of features that are common across most high-level languages. We will identify such a feature set first. We will use compiling such a feature set as a motivating principle in our discussion and development of an instruction-set for a processor.

2.3 A Common High-Level Language Feature Set

Let us consider the following feature set:

1. Expressions and assignment statements: Compiling such constructs will reveal many of the nuances in an instruction-set architecture (ISA for short) from the kinds of arithmetic and logic operations to the size and location of the operands needed in an instruction.
2. High-level data abstractions: Compiling aggregation of simple variables (usually called structures or records in a high-level language) will reveal additional nuances that may be needed in an ISA.
3. Conditional statements and loops: Compiling such constructs result in changing the sequential flow of execution of the program and would need additional machinery in the ISA.
4. Procedure calls: Procedures allow the development of modular and maintainable code. Compiling a procedure call/return brings additional challenges in the design of an ISA since it requires remembering the state of the program before and after executing the procedure, as well as in passing parameters to the procedure and receiving results from the called procedure.

In the next several subsections (Sections 2.4 – 2.8), we will consider each of these features and develop the machinery needed in an ISA from the point of view of efficiently compiling them.

2.4 Expressions and Assignment Statements

We know that any high level language (such as Java, C, and Perl) has arithmetic and logical expressions, and assignment statements:

```
a = b + c; /* add b and c and place in a */ (1)
```

```
d = e - f; /* subtract f from e and place in d */ (2)
```

```
x = y & z; /* AND y and z and place in x */ (3)
```

Each of the above statements takes *two operands* as inputs, performs an operation on them, and then stores the result in a *third operand*.

Consider the following three instructions in a processor instruction-set:

```
add a, b, c; a ← b + c (4)
```

```
sub d, e, f; d ← e - f (5)
```

```
and x, y, z; x ← y & z (6)
```

The high level constructs in (1), (2), and (3) directly map to the instructions (4), (5), and (6) respectively.

Such instructions are called *binary* instructions since they work on two operands to produce a result. They are also called *three operand* instructions since there are 3 operands (two source operands and one destination operand). Do we always need 3 operands in such binary instructions? The short answer is no, but we elaborate the answer to this question in the subsequent sections.

2.4.1 Where to keep the operands?

Let us discuss the location of the program variables in the above set of equations: a, b, c, d, e, f, x, y , and z . A simple model of a processor is as shown in Figure 2.1.

We know that inside the processor there is an arithmetic/logic unit or ALU that performs the operations such as ADD, SUB, AND, OR, and so on. We will now discuss where to keep the operands for these instructions. Let us start the discussion with a simple analogy.

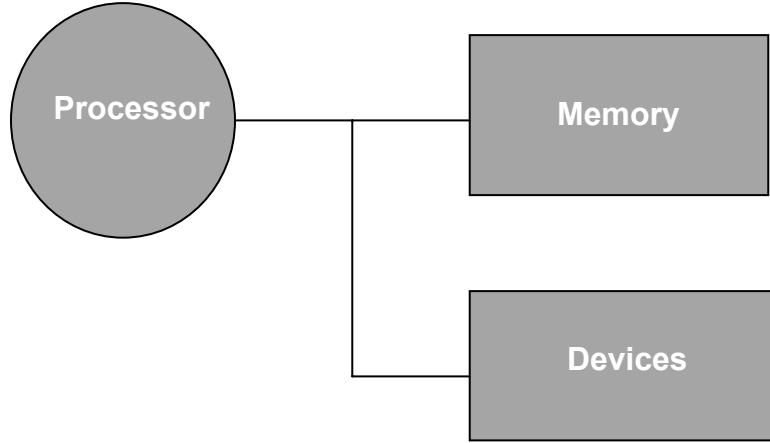


Figure 2.1: A basic computer organization

Suppose you have a toolbox that contains a variety of tools. Most toolboxes come with a tool tray. If you are working on a specific project (say fixing a leak in your kitchen faucet), you transfer a set of screwdrivers and pipe wrench from the toolbox into the tool tray. Take the tool tray to the kitchen sink. Work on the leak and then return the tools in the tool tray back into the toolbox when you are all done. Of course, you do not want to run back to the toolbox for each tool you need but instead hope that it is already there in the tool tray. In other words, you are optimizing the number of times you have to run to the toolbox by bringing the minimal set you need in the tool tray.

We want to do exactly that in the design of the instructions as well. We have heard the term *registers* being used in describing the resources available within a processor. These are like memory but they are inside the processor and so they are physically (and therefore electrically) close to the ALU and are made of faster parts than memory. Therefore, if the operands of an instruction are in registers then they are much quicker to access than if they are in memory. But that is not the whole story.

There is another compelling reason, especially with modern processor with very large memory. We refer to this problem as the *addressability* of operands. Let us return to the toolbox/tool-tray analogy. Suppose you are running an automobile repair shop. Now your toolbox is really big. Your project requires pretty much all the tools in your toolbox but at different times as you work on the project. So, as you work on different phases of your project, you will return the current set of tools in your tool tray and bring the appropriate ones for the new phase. Every tool has its unique place of course in the toolbox. On the other hand, you don't have a unique location in the tool tray for each tool. Instead, you are re-using the space in the tool tray for different tools from your toolbox.

An architect faces this same dilemma with respect to uniquely addressing operands. Modern processors have a very large memory system. As the size of memory grows, the size of a memory address (the number of bits needed to name uniquely a memory location) also increases. Thus if an instruction has to name three memory operands, that increases the size of each individual instruction. This is the addressability of operands is

a big problem, and will make each instruction occupy several memory locations in order to name uniquely all the memory operands it needs.

On the other hand, by having a small set of registers that cater to the immediate program need (*a la* the tool tray), we can solve this problem since the number of bits to name uniquely a register in this set is small. As a corollary to the addressability problem, the size of the register set has to be necessarily small to limit the number of addressing bits for naming the operands in an instruction (even if the level of integration would permit more registers to be included in the architecture).

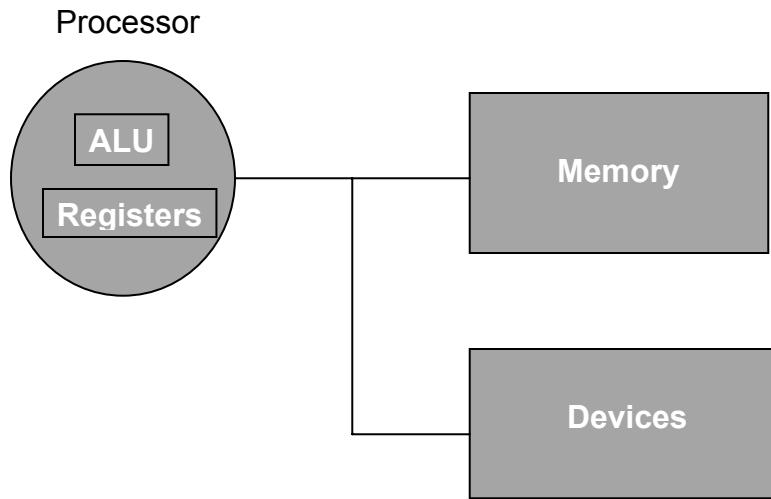


Figure 2.2: Adding registers inside the processor

Therefore, we may have instructions that look like:

```
add r1, r2, r3; r1 ← r2 + r3  
sub r4, r5, r6; r4 ← r5 - r6  
and r7, r8, r9; r7 ← r8 & r9
```

In addition, there is often a need to specify constant values for use in the program. For example, to initialize a register to some starting value or clearing a register that is used as an accumulator. The easiest way to meet this need is to have such constant values be part of the instruction itself. Such values are referred to as *immediate* values.

For example, we may have an instruction that looks like:

```
addi r1, r2, imm; r1 ← r2 + imm
```

In this instruction, an immediate value that is part of the instruction becomes the third operand. Immediate values are extremely handy in compiling high-level languages.

Example 1:

Given the following instructions

ADD Rx, Ry, Rz ;	Rx <- Ry + Rz
ADDI Rx, Ry, Imm ;	Rx <- Ry + Immediate value
NAND Rx, Ry, Rz ;	Rx <- NOT (Ry AND Rz)

Show how you can use the above instructions to achieve the effect of the following instruction

SUB Rx, Ry, Rz ;	Rx <- Ry - Rz
------------------	---------------

Answer:

NAND Rz, Rz, Rz ;	1's complement of Rz in Rz
ADDI Rz, Rz, 1 ;	2's complement of Rz in Rz
; Rz now contains -Rz	
ADD Rx, Ry, Rz ;	Rx <- Ry + (-Rz)
; Next two instructions restore	
; original value of Rz	
NAND Rz, Rz, Rz ;	1's complement of Rz in Rz
ADDI Rz, Rz, 1 ;	2's complement of Rz in Rz

All the operands are in registers for these arithmetic and logic operations. We will introduce the concept of **addressing mode**, which refers to how the operands are specified in an instruction. The addressing mode used in this case called **register addressing** since the operands are in registers.

Now with respect to the high-level constructs (1), (2), and (3), we will explore the relationship between the program variables **a**, **b**, **c**, **d**, **e**, **f** and **x**, **y**, **z** and these processor registers. As a first order, let us assume that all these program variables reside in memory, placed at well-known locations by the compiler. Since the variables are in memory and the arithmetic/logic instructions work only with registers, they have to be brought into the registers somehow. Therefore, we need additional instructions to move data back and forth between memory and the processor registers. These are called *load* (into registers from memory) and *store* (from registers into memory) instructions.

For example,

```
ld  r2, b; r2 ← b
st  r1, a; a ← r1
```

With these load/store instructions and the arithmetic/logic instructions we can now “compile” a construct such as

```
a = b + c
```

into

ld r2, b	(7)
ld r3, c	(8)
add r1, r2, r3	(9)
st r1, a	(10)

One may wonder why not simply use memory operands and avoid the use of registers. After all the single instruction

```
add a, b, c
```

seems so elegant and efficient compared to the 4-instruction sequence shown from (7)-(10).

The reason can be best understood with our toolbox/tool tray analogy. You knew you will need to use the screw-driver several times in the course of the project so rather than going to the toolbox every time, you paid the cost of bringing it once in the tool tray and *reuse* it several times before returning it back to the toolbox.

The memory is like the toolbox and the register file is like the tool tray. You expect that the variables in your program may be used in several expressions. Consider the following high-level language statement:

```
d = a * b + a * c + a + b + c;
```

One can see that once **a**, **b**, and **c** are brought from the memory into registers they will be *reused* several times in just this one expression evaluation. Try compiling the above expression evaluation into a set of instructions (assuming a multiply instruction similar in structure to the add instruction).

In a load instruction, one of the operands is a memory location, and the other is a register that is the destination of the load instruction. Similarly, in a store instruction the destination is a memory location.

Example 2:

An architecture has ONE register called an Accumulator (ACC), and instructions that manipulate memory locations and the ACC as follows:

LD	ACC, a ;	ACC <- contents of memory location a
ST	a, ACC ;	memory location a <- ACC
ADD	ACC, a ;	ACC <- ACC + contents of memory location

Using the above instructions, show how to realize the semantics of the following instruction:

ADD a, b, c; memory location a <-
contents of memory location b + contents of memory location c

Answer:

```
LD  ACC, b  
ADD ACC, c  
ST  a, ACC
```

2.4.2 How do we specify a memory address in an instruction?

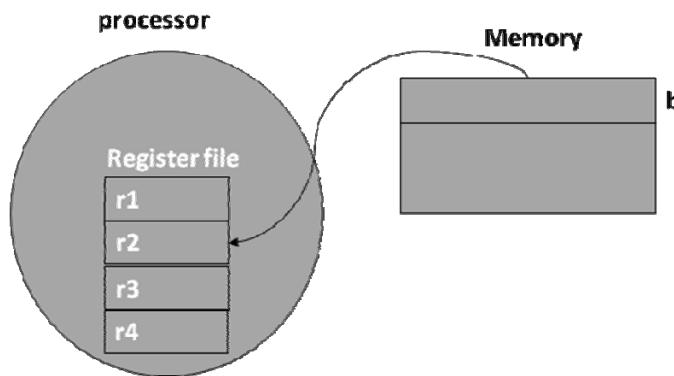


Figure 2.3: Loading a register from memory

Let us consider how to specify the memory address as part of the instruction. Of course, we can embed the address in the instruction itself. However, there is a problem with this approach. As we already mentioned earlier in Section 2.4.1, the number of bits needed to represent a memory address in an instruction is already large and will only get worse as the memory size keeps growing. For example, if we have a petabyte (roughly 2^{50} bytes) of memory we will need 50 bits for representing each memory operand in an instruction. Further, as we will see later on in Section 2.5, when compiling a program written in a high-level language (especially an object oriented language), the compiler may only know the offset (relative to the base address of the structure) of each member of a complex data structure such as an array or an object. Therefore, we introduce an

addressing mode that alleviates the need to have the entire memory address of an operand in each instruction.

Such an addressing mode is **base+offset** mode. In this addressing mode, a memory address is computed in the instruction as the sum of the contents of a register in the processor (called a base register) and an offset (contained in the instruction as an immediate value) from that register. This is usually represented mnemonically as

ld r2, offset(rb); r2 ← MEMORY[rb + offset]

If rb contains the memory address of variable b, and the offset is 0 then the above instruction equivalent to loading the program variable b into the processor register r2.

Note that rb can simply be one of the registers in the processor.

The power of the base+offset addressing mode is it can be used to load/store simple variables as above, and also elements of compound variables (such as arrays and structs) as we will see shortly.

Example 3:

Given the following instructions

LW Rx, Ry, OFFSET	;	Rx <- MEM[Ry + OFFSET]
ADD Rx, Ry, Rz	;	Rx <- Ry + Rz
ADDI Rx, Ry, Imm	;	Rx <- Ry + Immediate value

Show how you can realize a new addressing mode called autoincrement for use with the load instruction that has the following semantics:

LW Rx, (Ry)+	;	Rx <- MEM[Ry];
	;	Ry <- Ry + 1;

Your answer below should show how the LW instruction using autoincrement will be realized with the given instructions.

Answer:

LW Rx, Ry, 0	;	Rx <- MEM[Ry + 0]
ADDI Ry, Ry, 1	;	Ry <- Ry + 1

2.4.3 How wide should each operand be?

This is often referred to as the *granularity* or the *precision* of the operands. To answer this question, we should once again go back to the high-level language and the data types supported in the language. Let's use C as a typical high-level language. The base data types in C are **short**, **int**, **long**, **char**. While the width of these data types are implementation dependent, it is common to have **short** to be 16 bits; **int** to be 32 bits; **char** to be 8 bits. We know that the **char** data type is used to represent alphanumeric characters in C programs. There is a bit of a historic reason why **char** data type is 8 bits. ASCII was introduced as the digital encoding standard for alphanumeric characters (found in typewriters) for information exchange between computers and communication

devices. ASCII code uses 7-bits to represent each character. So, one would have expected **char** data type to be 7 bits. However, the popular instruction-set architectures at the time C language was born used 8-bit operands in instructions. Therefore, it was convenient for C to use 8 bits for the **char** data type.

The next issue is the choice of granularity for each operand. This depends on the desired precision for the data type. To optimize on space and time, it is best if the operand size in the instruction matches the precision needed by the data type. This is why processors support multiple precisions in the instruction set: word, half-word, and byte. **Word** precision usually refers to the *maximum precision* the architecture can support in hardware for arithmetic/logic operations. The other precision categories result in less space being occupied in memory for the corresponding data types and hence lead to space efficiency. Since there is less data movement between the memory and processor for such operands there is time efficiency accrued as well.

For the purposes of our discussion, we will assume that a word is 32-bits; half-word is 16-bits; and a byte is 8-bits. They respectively map to *int*, *short*, *char* in most C implementations. We already introduced the concept of addressability of operands in Section 2.4.1. When an architecture supports multiple precision operands, there is a question of *addressability* of memory operands. Addressability refers to the smallest precision operand that can be individually addressed in memory. For example, if a machine is byte-addressable then the smallest precision that can be addressed individually is a byte; if it is word addressed then the smallest precision that can be addressed individually is a word. We will assume byte addressability for our discussion. So, a word in memory will look as follows:



There are 4 bytes in each word (MSB refers to most significant byte; LSB refers to least significant byte). So if we have an integer variable in the program with the value
0x11223344

then its representation in memory it will look as follows:



Each byte can be individually addressed. Presumably, there are instructions for manipulating at this level of precision in the architecture. So the instruction-set may include instructions at multiple precision levels of operands such as:

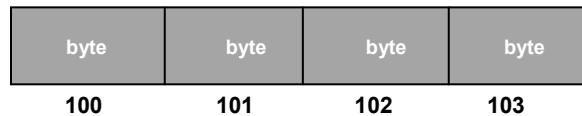
ld r1, offset(rb);	load a word at address rb+offset into r1
ldb r1, offset(rb);	load a byte at address rb+offset into r1
add r1, r2, r3;	add word operands in registers r2 and r3 and place the result in r1
addb r1, r2, r3;	add byte operands in registers r2 and r3 and place the result in r1

Necessarily there is a correlation between the architectural decision of supporting multiple precision for the operands and the hardware realization of the processor. The hardware realization includes specifying the width of the datapath and the width of the resources in the datapath such as registers. We discuss processor implementation in much more detail in later chapters (specifically Chapters 3 and 5). Since for our discussion we said word (32 bits) is the maximum precision supported in hardware, it is convenient to assume that the datapath is 32-bits wide. That is, all arithmetic and logic operations take 32-bit operands and manipulate them. Correspondingly, it is convenient to assume that the registers are 32-bits wide to match the datapath width. It should be noted that it is not necessary, but just convenient and more efficient, if the datapath and the register widths match the chosen word width of the architecture. Additionally, the architecture and implementation have to follow some convention for supporting instructions that manipulate precisions smaller than the chosen word width. For example, an instruction such as *addb* that works on 8-bit precision may take the lower 8-bits of the source registers, perform the addition, and place the result in the lower 8-bits of the destination register.

It should be noted that modern architectures have advanced to 64-bit precision for integer arithmetic. Even C programming language has introduced data types with 64-bit precision, although the name of the data type may not be consistent across different implementations of the compiler.

2.4.4 Endianness

An interesting question in a byte-addressable machine is the ordering of the bytes within the word. With 4 bytes in each word, if the machine is byte-addressable then four consecutive bytes in memory starting at address 100 will have addresses 100, 101, 102, and 103.

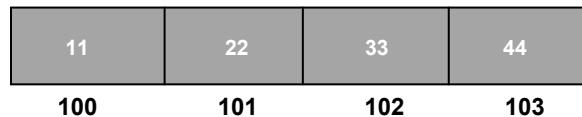


The composite of these 4 bytes make up a word whose address is 100.



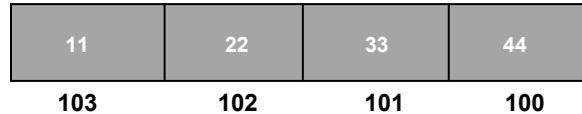
Let us assume the word at location 100 contains the values 0x11223344. The individual bytes within the word may be organized two possible ways:

Organization 1:



In this organization, the MSB of the word (containing the value 11_{hex}) is at the address of the word operand, namely, 100. This organization is called *big endian*.

Organization 2:



In this organization, the LSB of the word (containing the value 44_{hex}) is at the address of the word operand, namely, 100. This organization is called *little endian*.

So the endianness of a machine is determined by which byte of the word is at the word address. If it is MSB then it is big endian; if it is LSB then it is little endian.

In principle, from the point of view of programming in high-level languages, this should not matter so long as the program uses the data types in expressions in exactly the same way as they are declared.

However, in a language like C it is possible to use a data type differently from the way it was originally declared.

Consider the following code fragment:

```
int i = 0x11223344;
char *c;

c = (char *) &i;
printf("endian: i = %x; c = %x\n", i, *c);
```

Let us investigate what will be printed as the value of *c*. This depends on the endianness of the machine. In a big-endian machine, the value printed will be 11_{hex} ; while in a little-endian machine the value printed will be 44_{hex} . The moral of the story is if you declare a datatype of a particular precision and access it as another precision then it could be a recipe for disaster depending on the endianness of the machine. Architectures such as IBM PowerPC and Sun SPARC are examples of big endian; Intel x86, MIPS, and DEC Alpha are examples of little endian. In general, the endianness of the machine should not have any bearing on program performance, although one could come up with pathological examples where a particular endianness may yield a better performance for a particular program. This particularly occurs during string manipulation.

For example, consider the memory layout of the strings “RAMACHANDRAN” and “WAMACHANDRAN” in a big-endian architecture (see Figure 2.4). Assume that the first string starts at memory location 100.

```
char a[13] = "RAMACHANDRAN";
char b[13] = "WAMACHANDRAN";
```

100	R	A	M	A
104	C	H	A	N
108	D	R	A	N
 				
112	W	A	M	A
116	C	H	A	N
120	D	R	A	N

+0 +1 +2 +3

Figure 2.4: Big endian layout

Consider the memory layout of the same strings in a little-endian architecture as shown in Figure 2.5.

A	M	A	R	100
N	A	H	C	104
N	A	R	D	108
 				
A	M	A	W	112
N	A	H	C	116
N	A	R	D	120

+3 +2 +1 +0

Figure 2.5: Little endian layout

Inspecting Figures 2.4 and 2.5, one can see that the memory layout of the strings is left to right in big-endian while it is right to left in little-endian. If you wanted to compare the strings, one could use the layout of the strings in memory to achieve some coding efficiency in the two architecture styles.

As we mentioned earlier, so long as you manipulate a data type commensurate with its declaration, the endianness of the architecture should not matter to your program.

However, there are situations where even if the program does not violate the above rule, endianness can come to bite the program behavior. This is particularly true for network code that necessarily cross machine boundaries. If the sending machine is Little-endian and the receiving machine is Big-endian there could even be correctness issues in the resulting network code. It is for this reason, network codes use format conversion routines between host to network format, and vice versa, to avoid such pitfalls¹.

The reader is perhaps wondering why all the box makers couldn't just pick one endianness and go with it? The problem is such things become quasi-religious argument so far as a box maker is concerned, and since there is no standardization the programmers have to just live with the fact that there could be endianness differences in the processors they are dealing with.

To keep the discussion simple, henceforth we will assume a little-endian architecture for the rest of the chapter.

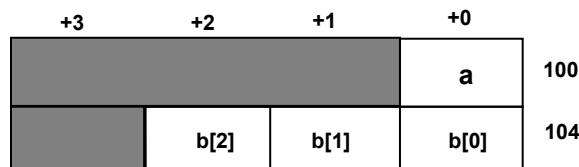
2.4.5 Packing of operands and Alignment of word operands

Modern computer systems have large amount of memory. Therefore, it would appear that there is plenty of memory to go around so there should be no reason to be stingy about the use of memory. However, this is not quite true. As memory size is increasing, so is the appetite for memory as far as the applications are concerned. The amount of space occupied a program in memory is often referred to as its *memory footprint*. A compiler, if so directed during compilation, may try **pack** operands of a program in memory to conserve space. This is particularly meaningful if the data structure consists of variables of different granularities (e.g., **int**, **char**, etc.), and if an architecture supports multiple levels of precision of operands. As the name suggests, packing refers to laying out the operands in memory ensuring no wasted space. However, we will also explain in this section why packing may not always be the right approach.

First, let us discuss how the compiler may lay out operands in memory given to conserve space. Consider the data structure

```
struct {
    char a;
    char b[3];
}
```

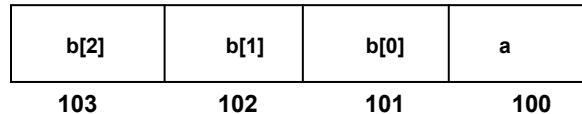
One possible lay out of this structure in memory starting at location 100 is shown below.



¹ If you have access to Unix source code, look up routines called htonl and ntoh, which stand for format conversion between host to network and network to host, respectively.

Let us determine the amount of memory actually needed to layout this data structure. Since each char is 1 byte, the size of the actual data structure is only 4 bytes but the above lay out wastes 50% of the space required to hold the data structure. The shaded region is the wasted space. This is the unpacked layout.

An efficient compiler may eliminate this wasted space and pack the above data structure in memory starting at location 100 as follows:



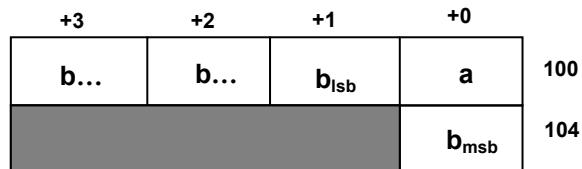
The packing done by the compiler is commensurate with the required precision for the data types and the addressability supported in the architecture. In addition to being frugal with respect to space, one can see that this layout would result in less memory accesses to move the whole structure (consisting of the two variables **a** and **b**) back and forth between the processor registers and memory. Thus, packing operands could result in time efficiency as well.

As we said before, packing may not always be the right strategy for a compiler to adopt.

Consider the following data structure:

```
struct {
    char a;
    int b;
}
```

We will once again determine how much memory is needed to layout this data structure in memory. A char is 1 byte and an int is 1 word (4 bytes); so totally 5 bytes are needed to store the structure in memory. Let's look at one possible lay out of the structure in memory starting at location 100.



The problem with this layout is that **b** is an int and it starts at address 101 and ends at address 104. To load **b** two words have to be brought from memory (from addresses 100 and 104). This is inefficient whether it is done in hardware or software. Architectures will usually require word operands to start at word addresses. This is usually referred to *alignment restriction* of word operands to word addresses.

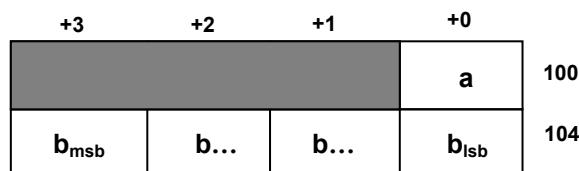
An instruction

ld r2, address

will be an illegal instruction if the address is not on a word boundary (100, 104, etc.).

While the compiler can generate code to load two words (at addresses 100 and 104) and do the necessary manipulation to get the int data type reconstructed inside the processor, this is inefficient from the point of view of time. So typically, the compiler will layout the data structures such that data types that need word precision are at word address boundaries.

Therefore, a compiler will most likely layout the above structure in memory starting at address 100 as follows:



Note that this layout wastes space (37.5% wasted space) but is more efficient from the point of view of time to access the operands.

You will see this classic space-time tradeoff will surface in computer science at all levels of the abstraction hierarchy presented in Chapter 1 from applications down to the architecture.

2.5 High-level data abstractions

Thus far we have discussed simple variables in a high-level language such as *char*, *int*, and *float*. We refer to such variables as *scalars*. The space needed to store such a variable is known *a priori*. A compiler has the option of placing a scalar variable in a register or a memory location. However, when it comes to data abstractions such as arrays and structures that are usually supported in high-level languages, the compiler may have no option except to allocate them in memory. Recall that due to the addressability problem, the register set in a processor is typically only a few tens of registers. Therefore, the sheer size of such data structures precludes allocating them in registers.

2.5.1 Structures

Structured data types in a high-level language can be supported with **base+offset** addressing mode.

Consider, the C construct:

```
struct {
    int a;
    char c;
    int d;
    long e;
}
```

If the base address of the structure is in some register **rb**, then accessing any field within the structure can be accomplished by providing the appropriate offset relative to the base register (the compiler knows how much storage is used for each data type and the alignment of the variables in the memory).

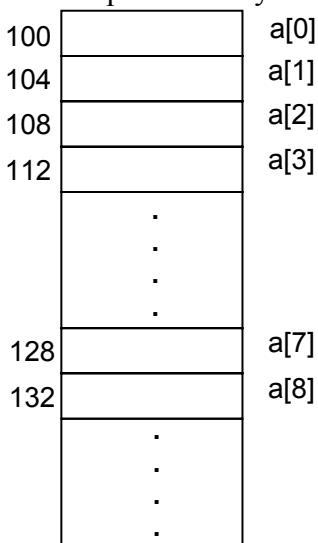
2.5.2 Arrays

Consider, the declaration

```
int a[1000];
```

The name **a** refers not to a single variable but to an array of variables $a[0]$, $a[1]$, etc. For this reason arrays are often referred to as *vectors*. The storage space required for such variables may or may not be known at compile time depending on the semantics of the high-level programming language. Many programming languages allow arrays to be dynamically sized at run time as opposed to compile time. What this means is that at compile time, the compiler may not know the storage requirement of the array. Contrast this with scalars whose storage size requirement is always known to the compiler at compile time. Thus, a complier will typically use memory to allocate space for such vector variables.

The compiler will lay this variable **a** out in memory as follows:



Consider the following statement that manipulates the array:

```
a[7] = a[7] + 1;
```

To compile the above statement, first **a[7]** has to be loaded from memory. It is easy to see that this doable given the **base+offset** addressing mode that we already introduced.

```
ld r1, 28(rb)
```

with **rb** initialized to **100**, the above instruction accomplishes loading **a[7]** into **r1**.

Typically, arrays are used in loops. In this case, there may be a loop counter (say **j**) that may be used to index the array. Consider the following statement

```
a[j] = a[j] + 1;
```

In the above statement, the offset to the base register is not fixed. It is derived from the current value of the loop index. While it is possible to still generate code for loading **a[j]**² this requires additional instructions before the load can be performed to compute the effective address of **a[7]**. Therefore, some computer architectures provide an additional addressing mode to let the effective address be computed as the sum of the contents of two registers. This is called the **base+index** addressing mode.

Every new instruction and every new addressing mode adds to the complexity of the implementation, thus the pros and cons have to be weighed very carefully. This is usually done by doing a cost/performance analysis. For example, to add base+index addressing mode, we have to ask the following questions:

1. How often will this addressing mode be used during the execution of a program?
2. What is the advantage (in terms of number of instructions saved) by using **base+index** versus **base+offset**?
3. What, if any, is the penalty paid (in terms of additional time for execution) for a load instruction that uses base+index addressing compared to base+offset addressing?
4. What additional hardware is needed to support base+index addressing?

Answers to the above questions, will quantitatively give us a handle on whether or not including the base+index addressing mode is a good idea or not.

We will come back to the issue of how to evaluate adding new instructions and addressing modes to the processor in later chapters when we discuss processor implementation and performance implications.

² Basically, the loop index has to be multiplied by 4 and added to rb to get the effective address. Generating the code for loading **a[j]** using only base+offset addressing mode is left as an exercise to the reader.

2.6 Conditional statements and loops

Before we talk about conditional statements, it is important to understand the concept of flow of control during program execution. As we know program execution proceeds sequentially in the normal flow of control:

100	I ₁
104	I ₂
108	I ₃
112	I ₄
116	I ₅
120	I ₆
124	I ₇
128	I ₈
132	.

Execution of instruction I₁ is normally followed by I₂, then I₃, I₄, and so on. We utilize a special register called *Program Counter (PC)*. Conceptually, we can think of the PC pointing to the currently executing instruction³. We know that program execution does not follow this sequential path of flow of control all the time.

2.6.1 If-then-else statement

Consider,

```
if (j == k) go to L1;  
a = b + c;  
L1: a = a + 1;
```

Let us investigate what is needed for compiling the above set of statements. The “if” statement has two parts:

1. Evaluation of the predicate “j == k”; this can be accomplished by using the instructions that we identified already for expression evaluation.
2. If the predicate evaluates to TRUE then it changes the flow of control from going to the next sequential instruction to the target L1. The instructions identified so far do not do accomplish this change of flow of control.

Thus, there is a need for a new instruction that changes the flow of control in addition to evaluating an arithmetic or logic expression. We will introduce a new instruction:

```
beq r1, r2, L1;
```

³ Note: We will see in Chapter 3 that for efficacy of implementation, the PC contains the memory address of the instruction immediately following the currently executing instruction.

The semantics of this instruction:

1. Compare r1 and r2
2. If they are equal then the next instruction to be executed is at address L1
3. If they are unequal then the next instruction to be executed is the next one textually following the *beq* instruction.

BEQ is an example of a conditional branch instruction to change the flow of control. We need to specify the address of the target of the branch in the instruction. While describing arithmetic and logic instructions, we discussed addressability (see Section 2.4.1) and suggested that it is a good idea to keep the operands in registers rather than memory to reduce the number of bits needed in the instruction for operand specifiers. This naturally raises the question whether the target of the branch instruction should also be in a register instead of being part of the instruction. That is, should the address L1 be kept in a register? A branch instruction takes the flow of control away from the current instruction (i.e., the branch) to another instruction that is usually not too far from it. We know that the PC points to the currently executing instruction. So, the target of a branch can be expressed relative to the current location of the branch instruction by providing an *address offset* as part of the instruction. Since the distance to the target of the branch from the current instruction is not too large, the address offset needs only a few bits in the branch instruction. In other words, it is fine to use a memory address (actually an address offset) as part of the instruction for branch instructions to specify the target of the branch.

Thus, the branch instruction has the format:

beq r1, r2, offset

The effect of this instruction is as follows:

1. Compare r1 and r2
2. If they are equal then the next instruction to be executed is at address $\text{PC} + \text{offset}_{\text{adjusted}}^4$
3. If they are unequal then the next instruction to be executed is the next one textually following the *beq* instruction.

We have essentially added a new addressing mode to computing an effective address. This is called **PC-relative** addressing.

The instruction-set architecture may have different flavors of conditional branch instructions such as BNE (branch on not equal), BZ (branch on zero), and BN (branch on negative).

Quite often, there may be an “else” clause in a conditional statement.

```
if (j == k) {
    a = b + c;
}
```

⁴ PC is the address of the *beq* instruction; $\text{offset}_{\text{adjusted}}$ is the offset specified in the instruction adjusted for any implementation specifics which we will see in the next Chapter.

```

    else {
        a = b - c;
    }
L1: ....
....
```

It turns out that we do not need anything new in the instruction-set architecture to compile the above conditional statement. The conditional branch instruction is sufficient to handle the predicate calculation and branch associated with the “if” statement. However after executing the set of statements in the body of the “if” clause, the flow of control has to be unconditionally transferred to **L1** (the start of statements following the body of the “else” clause).

To enable this, we will introduce an “unconditional jump” instruction:

j rtarget

where **rtarget** contains the target address of the unconditional jump.

The need for such an unconditional jump instruction needs some elaboration, since we have a branch instruction already. After all, we could realize the effect of an unconditional branch using the conditional branch instruction **beq**. By using the two operands of the **beq** instruction to name the same register (e.g., **beq r1, r1, offset**), the effect is an unconditional jump. However, there is a catch. The range of the conditional branch instruction is limited to the size of the offset. For example, with an offset size of 8 bits (and assuming the offset is a 2’s complement number so that both positive and negative offsets are possible), the range of the branch is limited to PC-128 and PC+127. This is the reason for introducing a new unconditional jump instruction, wherein a register specifies the target address of the jump.

The reader should feel convinced that using the conditional and unconditional branches that it is possible to compile any level of nested if-then-else statements.

2.6.2 Switch statement

Many high level languages provide a special case of conditional statement exemplified by the “switch” statement of C.

```

switch (k) {
    case 0:
    case 1:
    case 2:
    case 3:
    default:
}
```

If the number of cases is limited and/or sparse then it may be best to compile this statement similar to a nested if-then-else construct. On the other hand, if there are a number of contiguous non-sparse cases then compiling the construct as a nested if-then-else will result in inefficient code. Alternatively, using a jump table that holds the starting addresses for the code segments of each of the cases, it is possible to get an efficient implementation. See Figure 2.6.

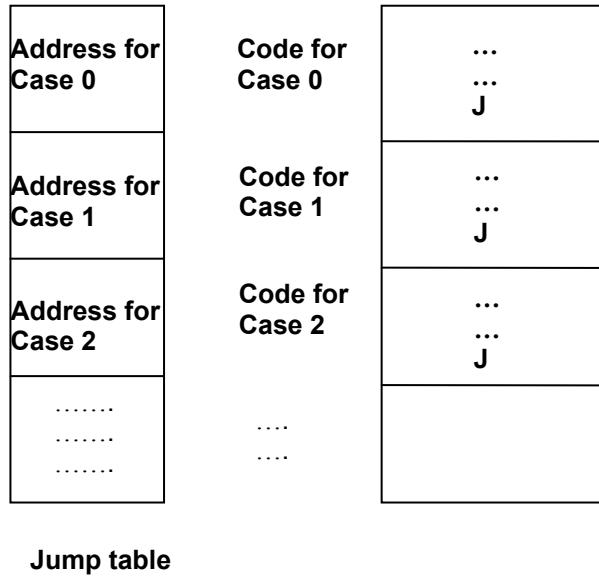


Figure 2.6: Implementing switch statement with a jump table

In principle, nothing new is needed architecturally to accomplish this implementation. While nothing new may be needed in the instruction-set architecture to support the switch statement, it may be advantageous to have unconditional jump instruction that works with one level of indirection.

That is,

J @ (rtarget)

where **rtarget** contains the address of the target address to jump to (unconditionally).

Additionally, some architectures provide special instructions for bounds checking. For example, MIPS architecture provides a set-on-less-than instruction

SLT s1, s2, s3

which works as follows:

```
if s2 < s3 then set s1 to 1
else set s1 to 0
```

This instruction is useful to do the bounds checking in implementing a switch statement.

2.6.3 Loop statement

High-level languages provide different flavors of looping constructs. Consider the code fragment,

```
j = 0;  
loop:    b = b + a[j];  
         j = j + 1;  
         if (j != 100) go to loop;
```

Assuming a register **r1** is used for the variable **j**, and the value 100 is contained in another register **r2**, the above looping construct can be compiled as follows:

```
loop:      ...  
          ...  
          ...  
          bne r1, r2, loop
```

Thus, no new instruction or addressing mode is needed to support the above looping construct. The reader should feel convinced that any other looping construct (such as “for”, “while”, and “repeat”) can similarly be compiled using conditional and unconditional branch instructions that we have introduced so far.

2.7 Checkpoint

So far, we have seen the following high-level language constructs:

1. Expressions and assignment statements
2. High-level data abstractions
3. Conditional statements including loops

We have developed the following capabilities in the instruction-set architecture to support the efficient compilation of the above constructs:

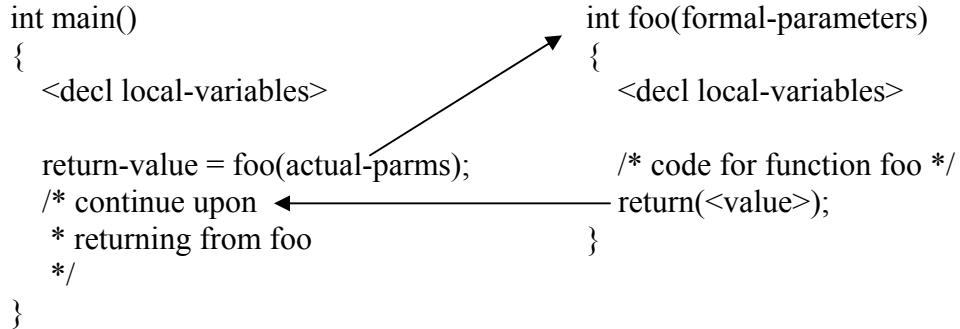
1. Arithmetic and logic instructions using registers
2. Data movement instructions (load/store) between memory and registers
3. Conditional and unconditional branch instructions
4. Addressing modes: register addressing, base+offset, base+index, PC-relative

2.8 Compiling Function calls

Compiling procedures or function calls in a high-level language requires some special attention.

First, let us review the programmer’s mental model of a procedure call. The program is in *main* and makes a call to function *foo*. The flow of control of the program is

transferred to the entry point of the function; upon exiting *foo* the flow of control returns to the statement following the call to *foo* in *main*.



First, let us define some terminologies: *caller* is the entity that makes the procedure call (*main* in our example); *callee* is the procedure that is being called (*foo* in our example).

Let us enumerate the steps in compiling a procedure call.

1. Ensure that the state of the caller (i.e. processor registers used by the caller) is preserved for resumption upon return from the procedure call
2. Pass the actual parameters to the callee
3. Remember the return address
4. Transfer control to callee
5. Allocate space for callee's local variables
6. Receive the return values from the callee and give them to the caller
7. Return to the point of call

Let us first ask the question whether the above set of requirements can be handled with the capabilities we have already identified in the instruction-set architecture of the processor. Let us look at each of the above items one by one. Let us first understand the ramifications in preserving the state of the caller in Section 2.8.1 and then the remaining chores in Section 2.8.2.

2.8.1 State of the Caller

Let us first define what we mean by the state of the caller. To execute the code of the callee, the resources needed are memory (for the code and data of the callee) and processor registers (since all arithmetic/logic instructions use them). The compiler will ensure that the memory used by the caller and callee are distinct (with the exception of any sharing that is mandated by the semantics of the high-level language). Therefore, the contents of the processor registers are the “state” we are worried about since partial results of the caller could be sitting in them at the time of calling the callee. Since we do not know what the callee may do to these registers, it is prudent to *save* them prior to the call and *restore* them upon return.

Now we need some place to save the registers. Let us try a hardware solution to the problem. We will introduce a *shadow register set* into which we will save the processor registers prior to the call; we will restore the processor registers from this shadow register set upon return from the procedure call. See figure 2.7

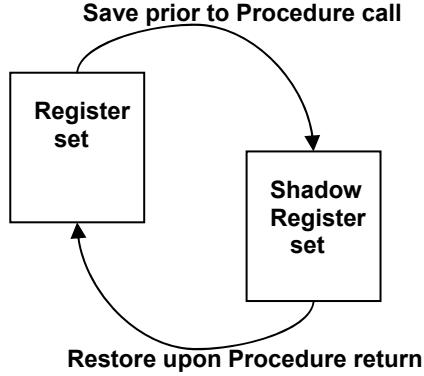


Figure 2.7: State save/restore for procedure call/return

We know that procedure calls are frequent in modular codes so rapid save/restore of state is important. Since the shadow register set is located inside the processor, and the save/restore is happening in hardware this seems like a good idea.

The main problem with this idea is that it assumes that the called procedure is not going to make any more procedure calls itself. We know from our experience with high-level languages that this is a bad assumption. In reality, we need as many shadow register sets as the level of nesting that is possible in procedure call sequences. See Figure 2.8.

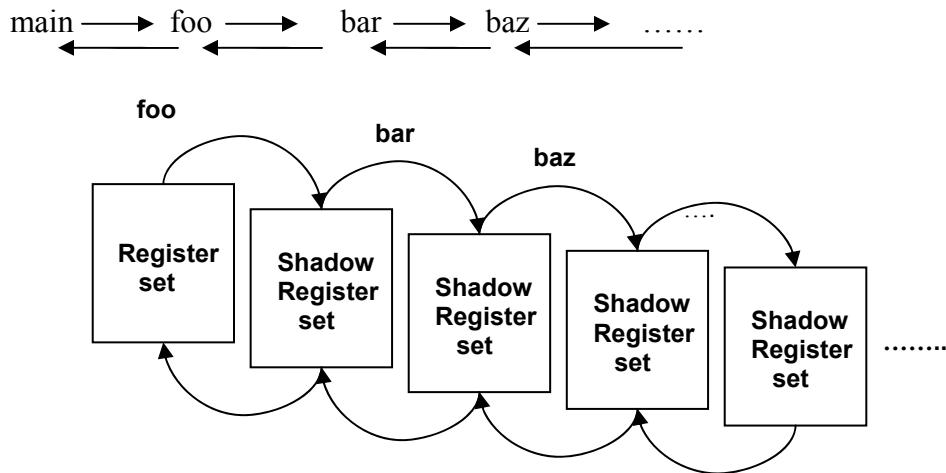


Figure 2.8: State save/restore for a nested procedure call

Let us discuss the implications of implementing these shadow register sets in hardware. The level of nesting of procedure calls (i.e., the chain of calls shown in Figure 2.8) is a dynamic property of the program. A hardware solution necessarily has to be finite in terms of the number of such shadow register sets. Besides, this number cannot be

arbitrarily large due to the cost and complexity of a hardware solution. Nevertheless, some architectures, such as Sun SPARC, implement a hardware mechanism called *register windowing*, precisely for this purpose. Sun SPARC provides 128 hardware registers, out of which only 32 registers are visible at any point of time. The invisible registers form the shadow register sets to support the procedure calling mechanism shown in Figure 2.8.

Figure 2.8 also suggests a solution that can be implemented in software: the *stack* abstraction that you may have learned in a course on data structures. We save the state in a stack at the point of call and restore it upon return. Since the stack has the *last-in-first-out (LIFO)* property, it fits nicely with the requirement of nested procedure calls.

The stack can be implemented as a software abstraction in memory and thus there is no limitation on the dynamic level of nesting.

The compiler has to maintain a pointer to the stack for saving and restoring state. This does not need anything new from the architecture. The compiler may dedicate one of the processor registers as the *stack pointer*. Note that this is not an architectural restriction but just a convenience for the compiler. Besides, each compiler is free to choose a different register to use as a stack pointer. What this means is that the compiler will not use this register for hosting program variables since the stack pointer serves a dedicated internal function for the compiler.

What we have just done with dedicating a register as the stack pointer is to introduce a software *convention* for register usage.

Well, one of the good points of the hardware solution was the fact that it was done in hardware and hence fast. Now if the stack is implemented in software then it is going to incur the penalty of moving the data back and forth between processor registers and memory on every procedure call/return. Let us see if we can have the flexibility of the software solution while having the performance advantage of the hardware solution. The software solution cannot match the speed advantage of the hardware solution but we can certainly try to reduce the amount of wasted effort.

Let us understand if we do really need to save all the registers at the point of call and restore them upon return. This implicitly assumes that the caller (i.e. the compiler on behalf of the caller) is responsible for save/restore. If the callee does not use ANY of the registers then the whole chore of saving and restoring the registers would have been wasted. Therefore, we could push the chore to the callee and let the callee save/restore registers *it is going to use* in its procedure. Once again, if the caller does not need any of the values in those registers upon return then the effort of the callee is wasted.

To come out of this dilemma, let us carry this idea of software convention a little further. Here again an analogy may help. This is the story of two lazy roommates. They would like to get away with as little house chores as possible. However, they have to eat

everyday so they come to an agreement. They have a frying pan for common use and each has a set of their own dishes. The convention they decide to adopt is the following:

- The dishes that are their own they never have to clean.
- If they use the other person's dishes then they have to leave it clean after every use.
- There is no guarantee that the frying pan will be clean; it is up to each person to clean and use it if they need it.

With this convention, each can get away with very little work...if they do not use the frying pan or the other person's dishes then practically no work at all!

The software convention for procedure calling takes an approach similar to the lazy roommates' analogy. Of course procedure call is not symmetric as the lazy roommates analogy (since there is an ordering - caller to callee). The caller gets a subset of the registers (the **s** registers) that are its own. The caller can use them any which way it wants and does not have to worry about the callee trashing them. The callee, if it needs to use the **s** registers, has to save and restore them. Similar to the frying pan, there is a subset of registers (the **t** registers) that are common to both the caller and the callee. Either of them can use the **t** registers, and do not have to worry about saving or restoring them. Now, as can be seen, similar to the lazy roommates' analogy, if the caller never needs any values in the **t** registers upon return from a procedure it has to do no work on procedure calls. Similarly, if the callee never uses any of the **s** registers, then it has to do no work for saving/restoring registers.

The saving/restoring of registers will be done on the stack. We will return to the complete software convention for procedure call/return after we deal with the other items in the list of things to be done for procedure call/return.

2.8.2 Remaining chores with procedure calling

1. **Parameter passing:** An expedient way of passing parameters is via processor registers. Once again, the compiler may establish a software convention and reserve some number of processor registers for parameter passing.

Of course, a procedure may have more parameters than allowed by the convention. In this case, the compiler will pass the additional parameters on the stack. The software convention will establish where exactly on the stack the callee can find the additional parameters with respect to the stack pointer.

2. **Remember the return address:** We introduced the processor resource, Program Counter (PC), early on in the context of branch instructions. None of the high-level constructs we have encountered thus far, have required remembering where we are in the program. So now, there is a need to introduce a new instruction for saving the PC in a well-known place so that it can be used for returning from the callee.

We introduce a new instruction:

```
JAL rtarget, rlink
```

The semantics of this instruction is as follows:

- Remember the return address in r_{link} (which can be any processor register)
- Set PC to the value in r_{target} (the start address of the callee)

We return to the issue of software convention. The compiler may designate one of the processor registers to be r_{target} , to hold the address of the target of the subroutine call; and another of the processor registers to be r_{link} to hold the return address. That is these registers are not available to house normal program variables.

So at the point of call, the procedure call is compiled as

```
JAL rtarget, rlink; /* rtarget containing the address  
of the callee */
```

Returning from the procedure is straightforward. Since we already have an unconditional jump instruction,

```
J rlink
```

accomplishes the return from the procedure call.

3. **Transfer control to callee:** Step 3 transfers the control to the callee via the JAL instruction.
4. **Space for callee's local variables:** The stack is a convenient area to allocate the space needed for any local variables in the callee. The software convention will establish where exactly on the stack the callee can find the local variables with respect to the stack pointer⁵.
5. **Return values:** An expedient way for this is for the compiler to reserve some processor registers for the return values. As in the case of parameters, if the number of returned values exceeds the registers reserved by the convention, then the additional return values will be placed on the stack. The software convention will establish where exactly on the stack the caller can find the additional return values with respect to the stack pointer.
6. **Return to the point of call:** As we mentioned earlier, a simple jump through r_{link} will get the control back to the instruction following the point of call.

⁵ See Exercise 18 for a variation of this description.

2.8.3 Software Convention

Just to make this discussion concrete, we introduce a set of processor registers and the software convention used:

- Registers **s0-s2** are the caller's s registers
- Registers **t0-t2** are the temporary registers
- Registers **a0-a2** are the parameter passing registers
- Register **v0** is used for return value
- Register **ra** is used for return address
- Register **at** is used for target address
- Register **sp** is used as a stack pointer

Before illustrating how we will use the above conventions to compile a procedure call we need to recall some details about stacks. A convention that is used by most compilers is that the stack grows down from high addresses to low addresses. The basic stack operations are

- Push: decrements the stack pointer and places the value at the memory location pointed to by the stack pointer
- Pop: takes the value at the memory location pointed to by the stack pointer and increments the stack pointer

The following illustrations (Figures 2.9-2.20) show the way the compiler will produce code to build the stack frame at run time:

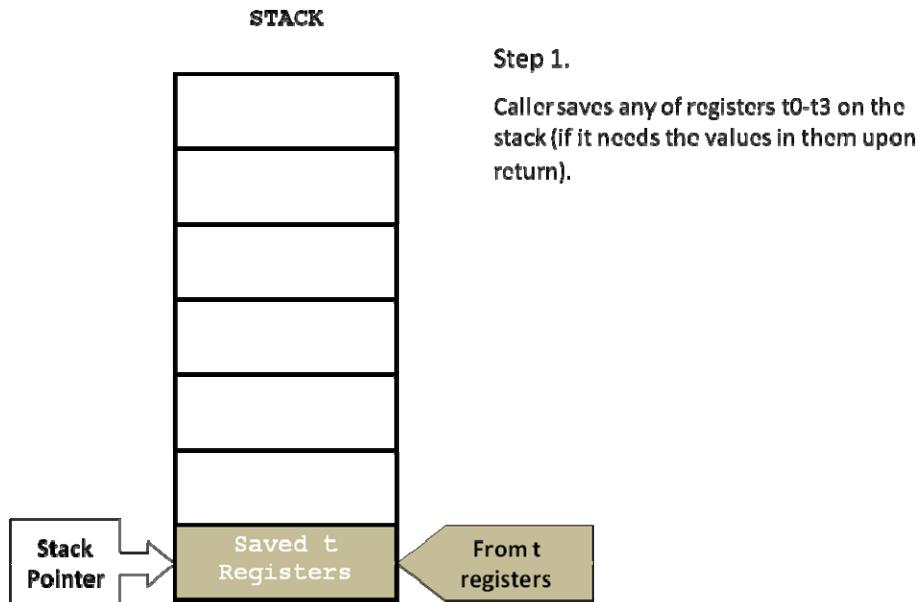


Figure 2.9: Procedure call/return - Step 1

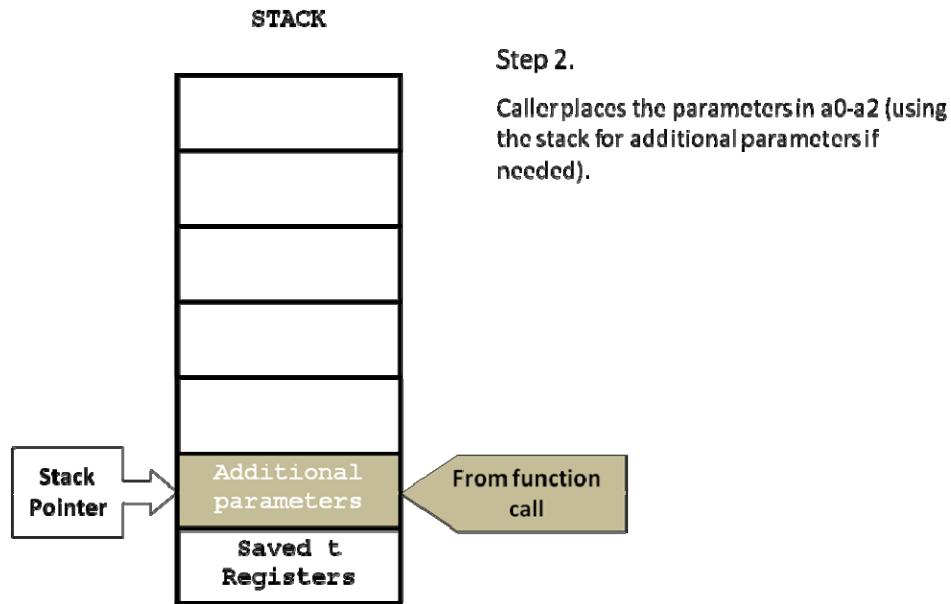


Figure 2.10: Procedure call/return - Step 2

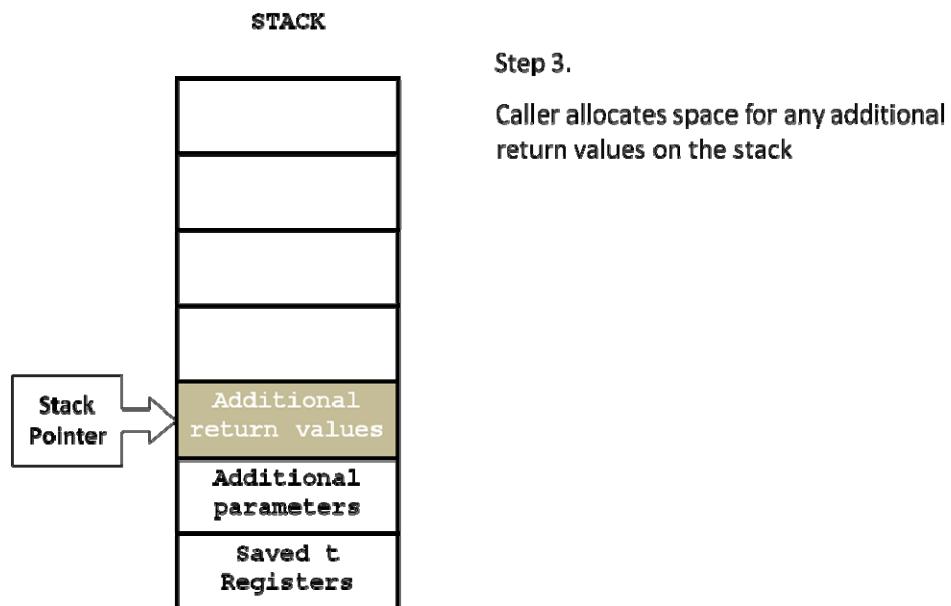


Figure 2.11: Procedure call/return - Step 3

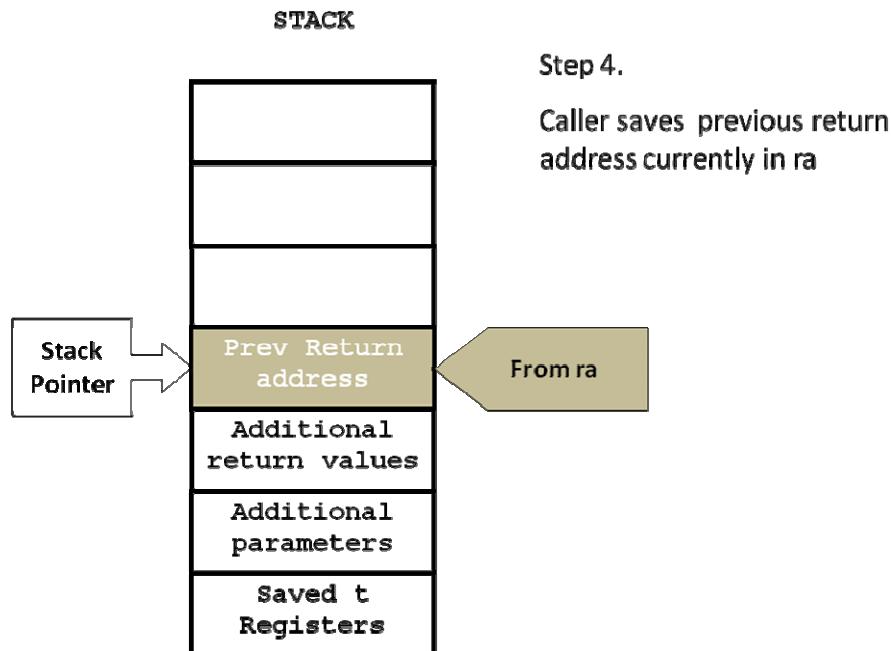


Figure 2.12: Procedure call/return - Step 4

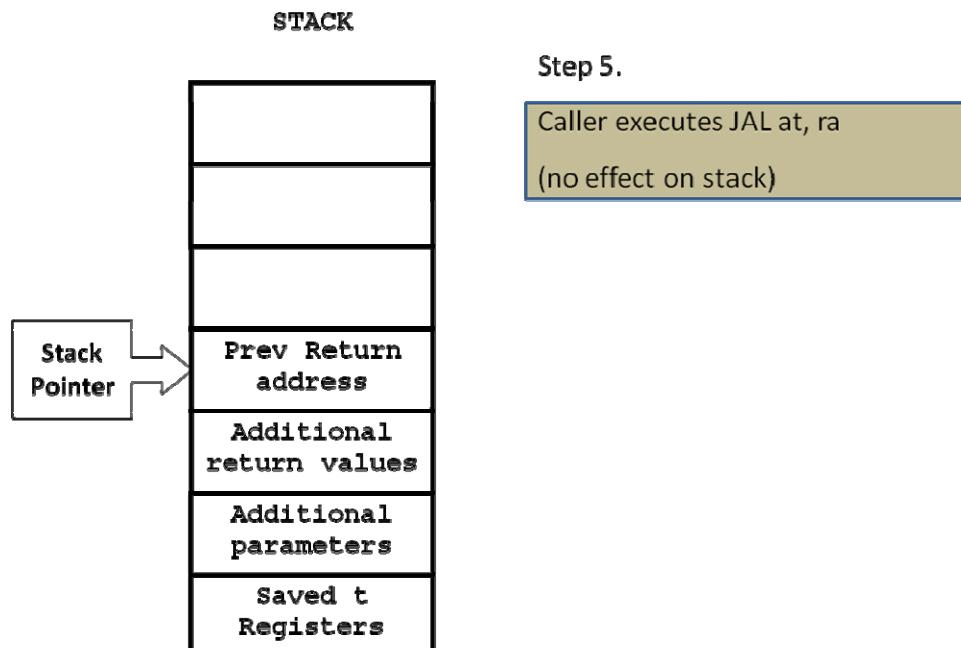


Figure 2.13: Procedure call/return - Step 5

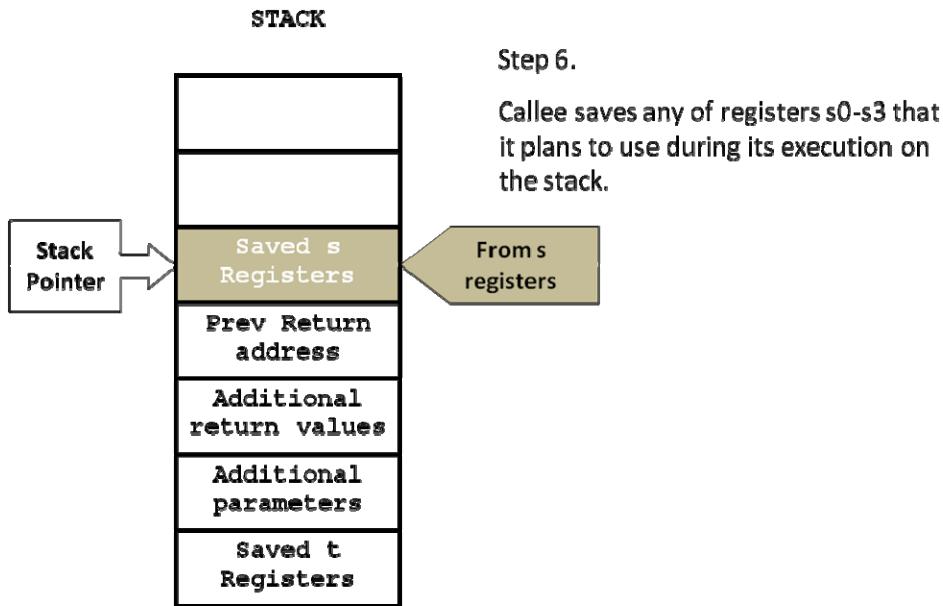


Figure 2.14: Procedure call/return - Step 6

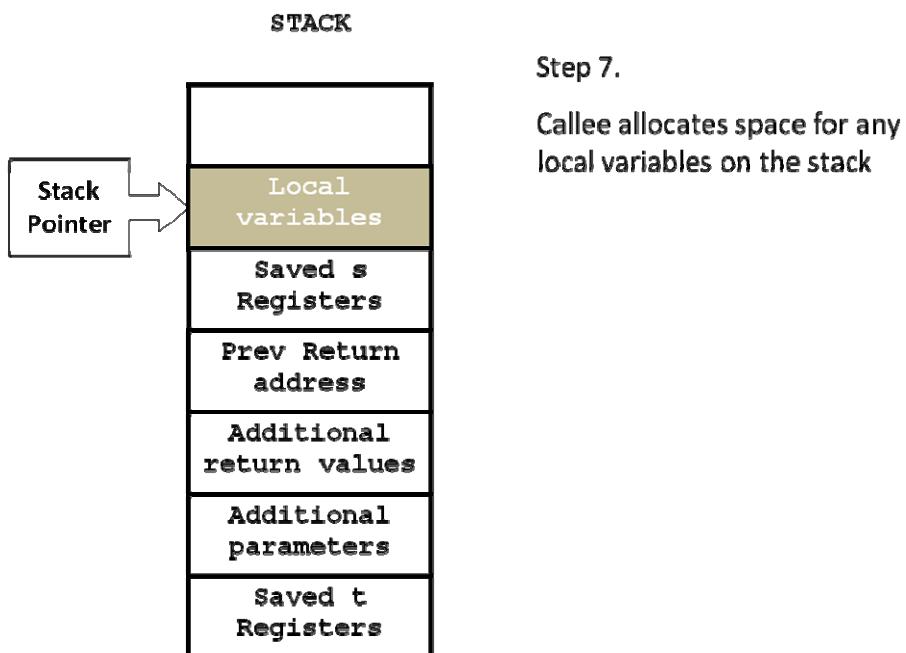


Figure 2.15: Procedure call/return - Step 7

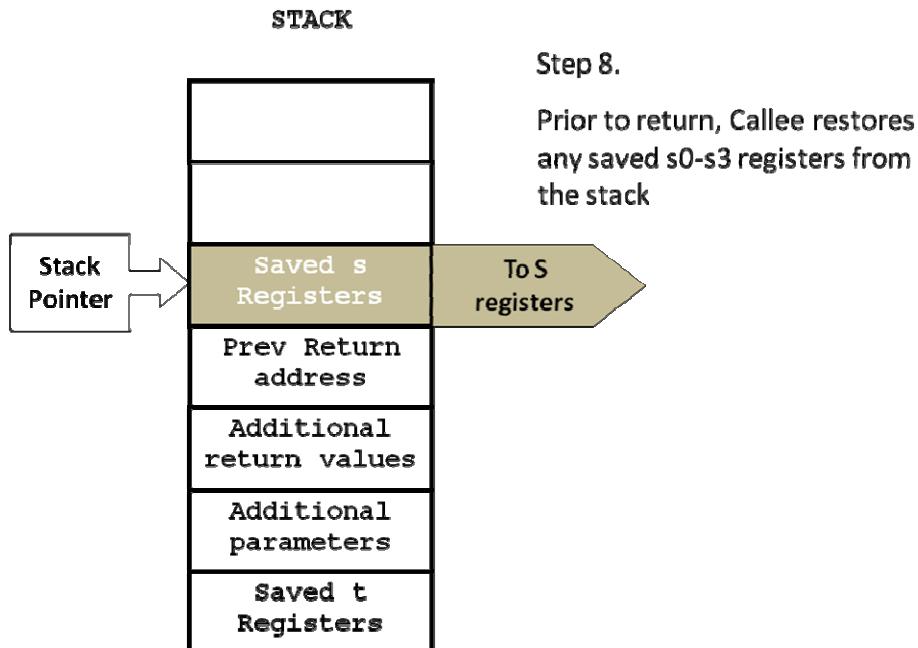


Figure 2.16: Procedure call/return - Step 8

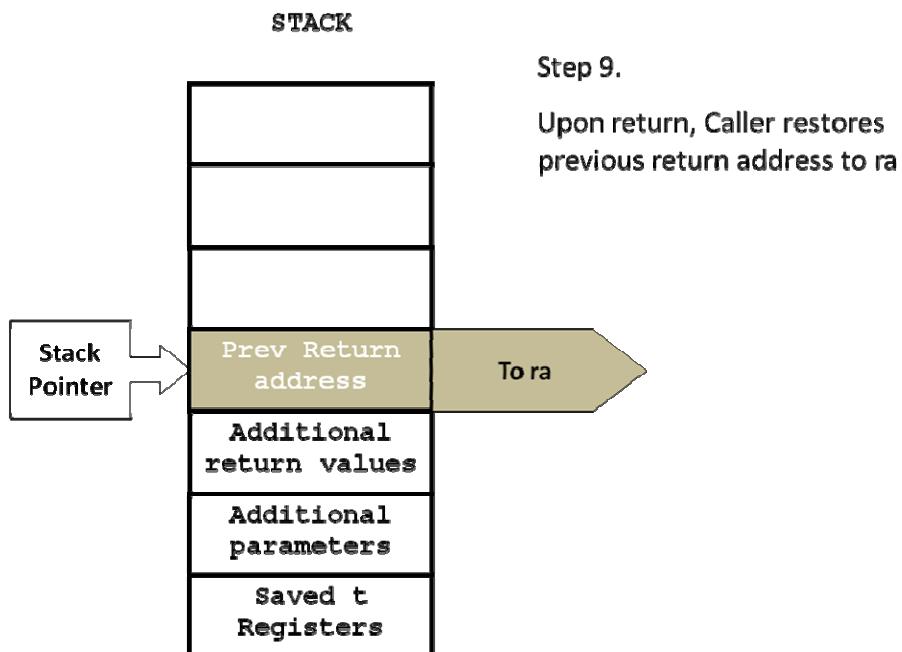


Figure 2.17: Procedure call/return - Step 9

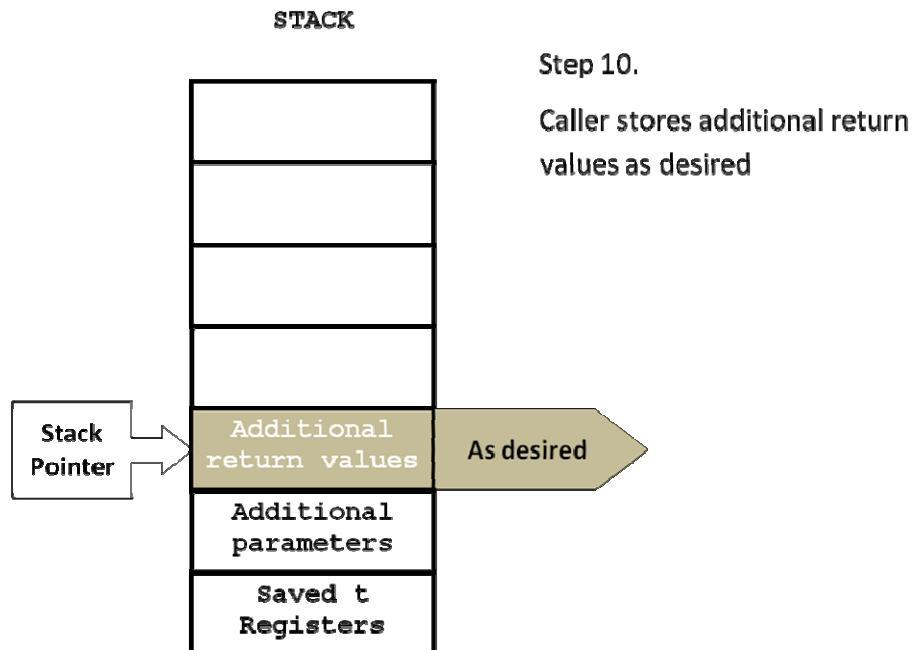


Figure 2.18: Procedure call/return - Step 10

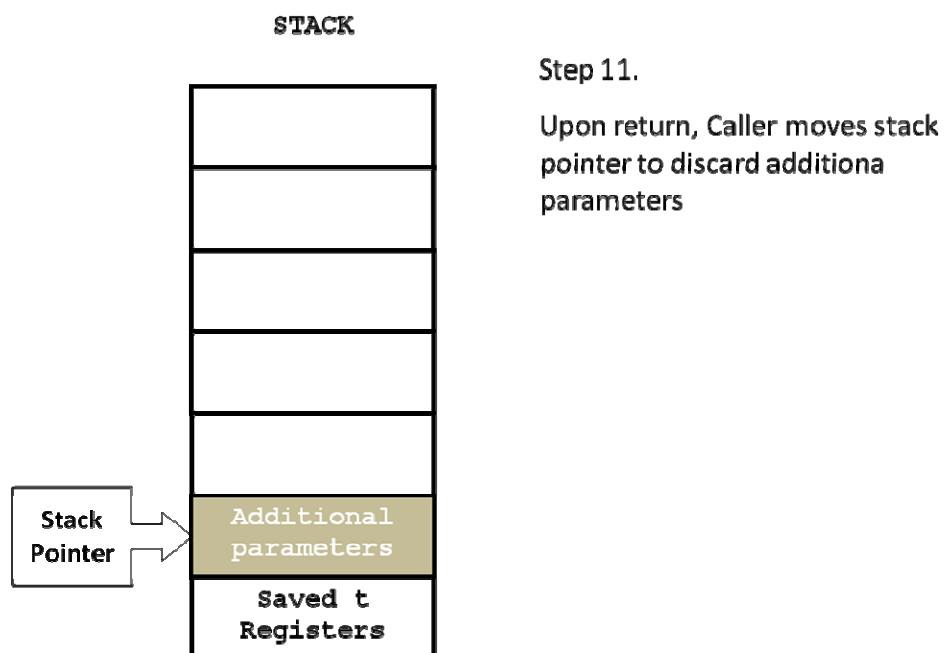


Figure 2.19: Procedure call/return - Step 11

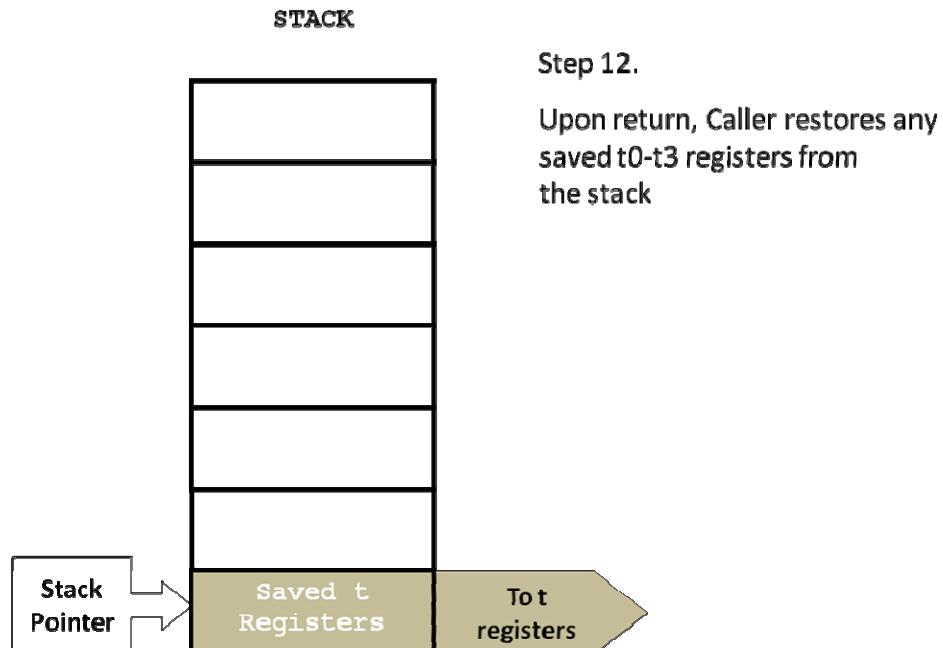


Figure 2.20: Procedure call/return - Step 12

2.8.4 Activation Record

The portion of the stack that is relevant to the currently executing procedure is called the *activation record* for that procedure. An activation record is the communication area between the caller and the callee. The illustrations in Figure 2.9 to 2.19 show how the activation record is built up by the caller and the callee, used by the callee, and dismantled (by the caller and callee) when control is returned back to the caller. Depending on the nesting of the procedure calls, there could be multiple activation records on the stack. However, at any point of time exactly one activation record is active pertaining to the currently executing procedure.

Consider,

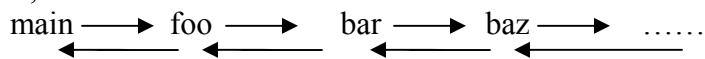


Figure 2.21 shows the stack and the activation records for the above sequence of calls.

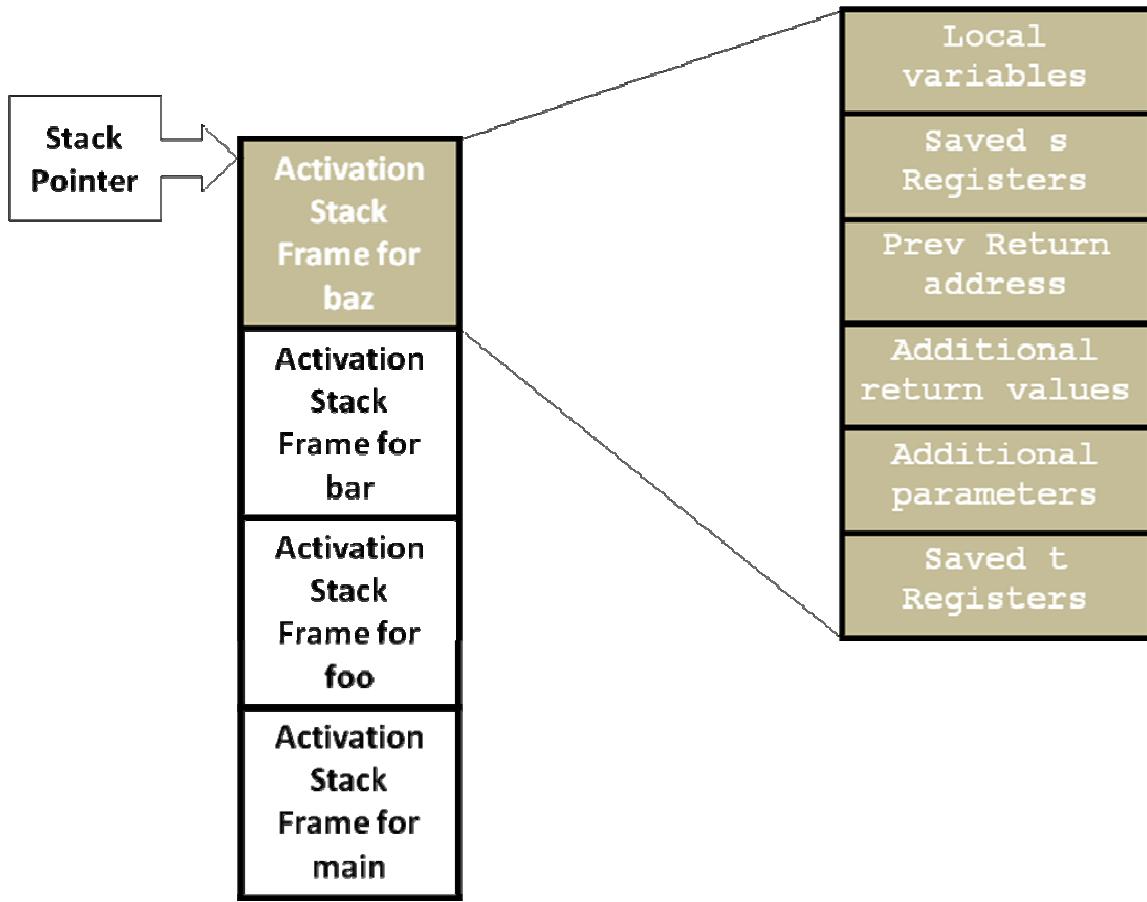


Figure 2.21: Activation records for a sequence of calls

2.8.5 Recursion

One of the powerful tools to a programmer is recursion. It turns out that we do not need anything special in the instruction-set architecture to support recursion. The stack mechanism ensures that every instantiation of a procedure, whether it is the same procedure or a different one, gets a new activation record. Of course, a procedure that could be called recursively has to be coded in a manner that supports recursion. This is in the purview of the programmer. The instruction-set need not be modified in any way to specially support recursion.

2.8.6 Frame Pointer

During execution of a program, it is obviously essential to be able to locate all of the items that are stored on the stack by the program. Their absolute location cannot be known until the program is executing. It might seem obvious that they can be referenced as being at some offset from the stack pointer but there is a problem with this approach. Certain compilers may generate code that will move the stack pointer during the execution of the function (after the stack frame is constructed). For example, a number of languages permit dynamic allocation on the stack. While it would be possible to keep track of this stack movement the extra bookkeeping and execution time penalties would make this a poor choice. The common solution is to designate one of the general purpose

registers as a *frame pointer*. This contains the address of a known point in the activation record for the function and it will never change during execution of the function. An example will help to understand the problem and the solution. Consider the following procedure:

```

int foo(formal-parameters)
{
    int a, b;

    /* some code */
    if (a > b) {                                (1)
        int c = 1;                                (2)
        a = a + b + c;                            (3)
    }

    printf("%d\n, a);                           (4)

    /* more code for foo */

    /*
     * return from foo
     */
    return(0);
}

```

Let us assume that the stack pointer (denoted as \$sp) contains 100 after step 6 (Figure 2.14) in a call sequence to foo. In step 7, the callee allocates space for the local variables (a and b). Assuming the usual convention of the stack growing from high addresses to low addresses, a is allocated at address 96 and b is allocated at address 92. Now \$sp contains 92. This situation is shown in Figure 2.22.

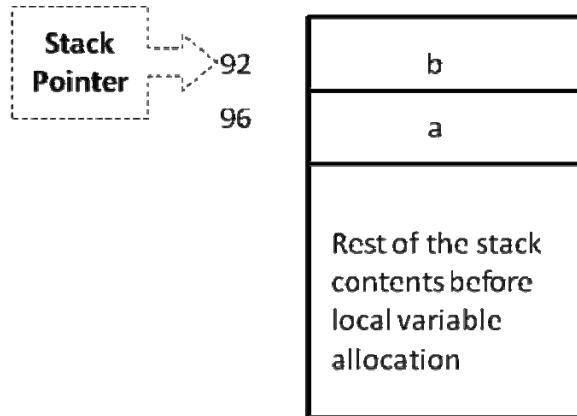


Figure 2.22: State of the stack after the local variables are allocated in the procedure. Note that the stack pointer value is 92 at this time.

Procedure foo starts executing. The compiled code for the if statement needs to load a and b into processor registers. This is quite straightforward for the compiler. The following two instructions

```
ld      r1, 4($sp);    /* load r1 with a; address of a = $sp+offset = 92+4 = 96 */
ld      r2, 0($sp);    /* load r2 with b; address of b = $sp+offset = 92+0 = 92 */
loads a and b into processor registers r1 and r2 respectively.
```

Note what happens next in the procedure if the predicate calculation turns out to be true. The program allocates a new variable c. As one might guess, this variable will also be allocated on the stack. Note however, this is not part of the local variable allocation (step 7, Figure 2.15) prior to beginning the execution of foo. This is a conditional allocation subject to the predicate of the “if” statement evaluating to true. The address of c is 88. Now \$sp contains 88. This situation is shown in Figure 2.23.

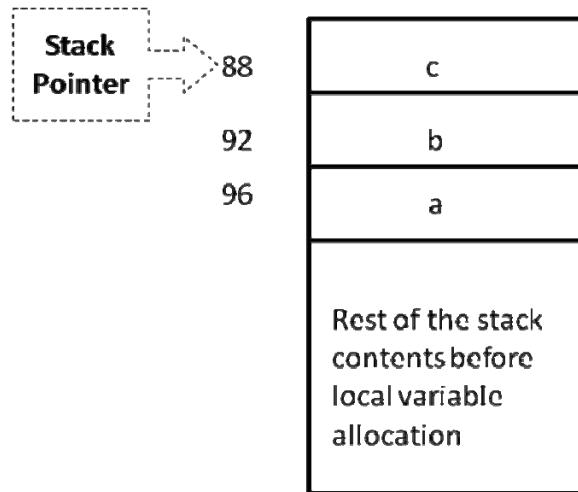


Figure 2.23: State of the stack after the variable *c* is allocated corresponding to statement (2) in the procedure. Note that the stack pointer value is 88 at this time.

Inside the code block for the if statement, we may need to load/store the variable *a* and *b* (see statement 3 in the code block). Generating the correct addresses for *a* and *b* is the tricky part. The variable *a* is used to be at an offset of 4 with respect to \$sp. However, it is at an offset of 8 with respect to the current value in \$sp. Thus, for loading variables *a* and *b* in statement (3), the compiler will have to generate the following code:

```
ld      r1, 8($sp);    /* load r1 with a; address of a = $sp+offset = 88+8 = 96 */
ld      r2, 4($sp);    /* load r2 with b; address of b = $sp+offset = 88+4 = 92 */
```

Once the *if* code block execution completes, *c* is de-allocated from the stack and the \$sp changes to 92. Now *a* is at an offset of 4 with respect to the current value in \$sp. This situation same as shown in Figure 2.22.

The reader can see that from the point of writing the compiler, the fact that the stack can grow and shrink makes the job harder. The offset for local variables on the stack changes depending on the current value of the stack pointer.

This is the reason for dedicating a register as a frame pointer. The frame pointer contains the first address on the stack that pertains to the activation record of the called procedure and never changes while this procedure is in execution. Of course, if a procedure makes another call, then its frame pointer has to be saved and restored by the callee. Thus, the very first thing that the callee does is to save the frame pointer on the stack and copy the current value of the stack pointer into the frame pointer. This is shown in Figure 2.24.

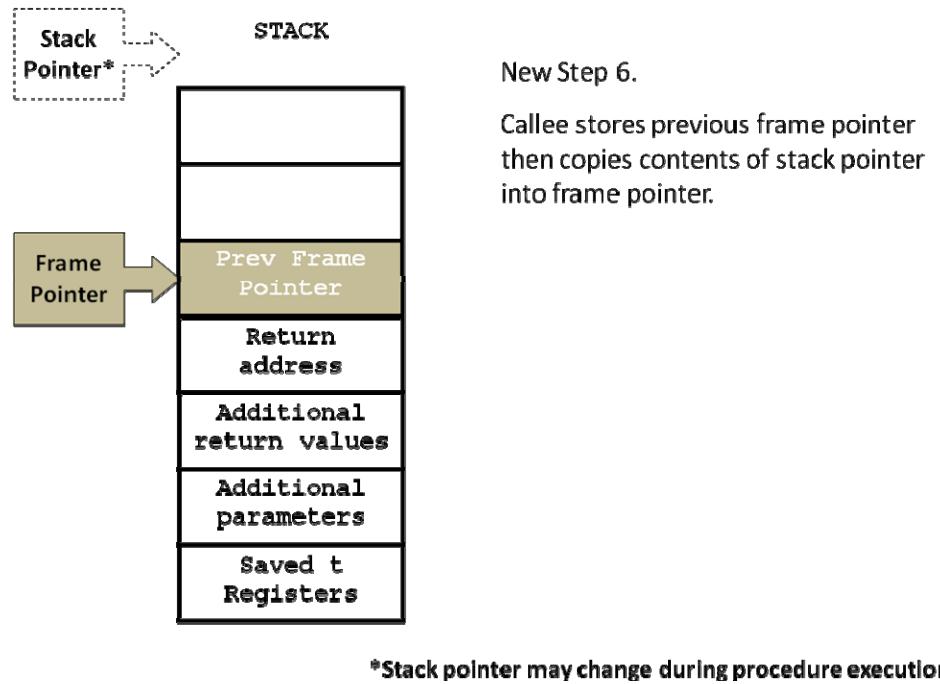


Figure 2.24: Frame Pointer

Thus, the frame pointer is a fixed harness on the stack (for a given procedure) and points to the first address of the activation record (AR) of the currently executing procedure.

2.9 Instruction-Set Architecture Choices

In this section, we summarize architectural choices in the design of instruction-sets. The choices range from specific set of arithmetic and logic instructions in the repertoire, the addressing modes, architectural style, and the actual memory layout (i.e., format) of the instruction. Sometimes these choices are driven by technology trends at that time and implementation feasibility, while at other times by the goal of elegant and/or efficient support for high-level language constructs.

2.9.1 Additional Instructions

Some architectures provide additional instructions to improve the space and time efficiency of compiled code.

- For example, in the MIPS architecture, loads and stores are always for an entire word of 32 bits. Nevertheless, once the instruction is brought into a processor

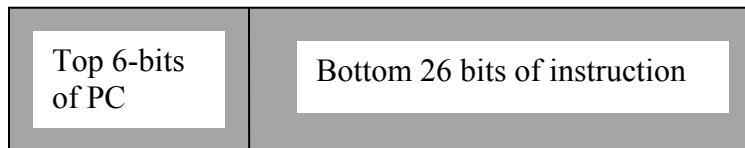
register, special instructions are available for *extracting* a specific byte of the word into another register; similarly, a specific byte of the word can be *inserted* into the word.

- In the DEC Alpha architecture, there are instructions for loading and storing at different operand sizes: byte, half-word, word, and quad-word.
- Some architectures may have some pre-defined immediate values (e.g., 0, 1, and such small integers) available for direct use in instructions.
- DEC VAX architecture has a single instruction to load/store all the registers in the register-file to/from memory. As one might guess, such an instruction would be useful for storing and restoring registers at the point of procedure call and return. The reader should think about the utility of such an instruction in light of the discussion on procedure calling convention that we discussed in this chapter.

2.9.2 Additional addressing modes

In addition to the addressing modes we already discussed, some architectures provide fancier addressing modes.

- Many architectures provide an indirect addressing mode.
 - **ld @ (ra)**
In this instruction, the contents of the register ra will be used as the address of the address of the actual memory operand.
- Pseudo-direct addressing
 - MIPS provides this particular addressing mode where the effective address is computed by taking the top 6-bits of the PC and concatenating it with the bottom 26-bits of the instruction word to get a 32-bit absolute address as shown below:



Early architectures such as IBM 360, PDP-11, and VAX 11 supported many more addressing modes than what we covered in this chapter. Most modern architectures have taken a minimalist approach towards addressing modes for accessing memory. This is mainly because over the years it has been seen that very few of the more complex addressing modes are actually used by the compiler. Even base+index is not found in MIPS architecture, though IBM PowerPC and Intel Pentium do support this addressing mode.

2.9.3 Architecture styles

Historically, there have been several different kinds of architecture styles.

- **Stack oriented:** Burroughs Computers introduced the stack-oriented architecture where all the operands are on a stack. All instructions work on operands that are on the stack.
- **Memory oriented:** IBM 360 series of machines focused on memory-oriented architectures wherein most (if not all) instructions work on memory operands.
- **Register oriented:** As has been discussed in most of this chapter, most instructions in this architecture deal with operands that are in registers. With the maturation of compiler technology, and the efficient use of registers within the processor, this style of architecture has come to stay as the instruction-set architecture of choice. DEC Alpha, and MIPS are modern day examples of this style of architecture.
- **Hybrid:** As is always the case, one can make a case for each of the above styles of architecture with specific applications or set of applications in mind. So naturally, a combination of these styles is quite popular. Both the IBM PowerPC and the Intel x 86 families of architectures are a hybrid of the memory-oriented and register-oriented styles.

2.9.4 Instruction Format

Depending on the instruction repertoire, instructions may be grouped into the following classes:

1. Zero operand instructions

Examples include

- HALT (halts the processor)
- NOP (does nothing)

Also, if the architecture is stack-oriented then it only has implicit operands for most instructions (except for pushing and popping values explicitly on the stack). Instructions in such an architecture would look like:

- ADD (pop top two elements of the stack, add them, and push the result back on to the stack)
- PUSH <operand> (pushes the operand on to the stack)
- POP <operand> (pops the top element of the stack into the operand)

2. One operand instructions

Examples include instructions that map to unary operations in high-level languages:

- INC/DEC <operand> (increments or decrements the specified operand by a constant value)
- NEG <operand> (2's complement of the operand)
- NOT <operand> (1's complement of the operand)

Also, unconditional jump instructions usually have only one operand

- J <target> (PC <- target)

Also, some older machines (such as DEC's PDP-8) used 1 implicit operand (called the *accumulator*) and one 1 explicit operand. Instructions in such architecture would look like

- ADD <operand> (ACC <- ACC+operand)
- STORE <operand> (operand <- ACC)
- LOAD <operand> (ACC <- operand)

3. Two operand instructions

Examples include instructions that map to binary operations in a high level language. The basic idea is one of the operands is used as both source and destination in a binary operation.

- ADD R1, R2 (R1 <- R1 + R2)

Data movement instructions also fall into this class:

- MOV R1, R2 (R1 <- R2)

4. Three operand instructions

This is the most general kind and we have seen examples of this throughout this chapter. Examples include:

- ADD R_{dst}, R_{src1}, R_{src2} (R_{dst} <- R_{src1}+R_{src2})
- LOAD R, Rb, offset (R <- MEM[Rb + offset])

Instruction format deals with how the instruction is laid out in memory. An architecture may have a mixture of the styles that we discussed earlier, and may consist of various types of instructions that need different number of operands in every instruction.

An instruction typically has the following generic format



At the point of designing the instruction format, we are getting to the actual implementation. This is because the choice of the instruction format has a bearing on the space and time efficiency of the design. A related choice is the actual *encoding* of the fields within an instruction, which pertains to the semantic meaning associated with each bit pattern for a given field. For example, the bit pattern corresponding to ADD, etc.

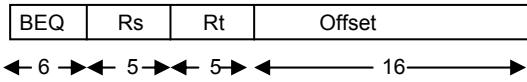
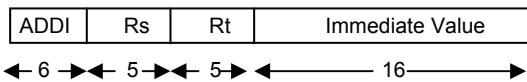
Broadly, instruction formats may be grouped under two categories:

- **All instructions are of the same length**
In this format, all instructions have the same length (e.g. one memory word). What this means is that the same bit position in an instruction may have a different meaning depending on the instruction.
 - *Pros:*
 - This simplifies the implementation due to the fixed size of the instruction.
 - Interpretation of the fields of the instruction can start (speculatively) as soon as the instruction is available as all instructions have fixed length.
 - *Cons:*

- Since all instructions do not need all the fields (for example, single operand versus multiple operand instructions), there is potential for wasted space in each instruction.
- We may need additional glue logic (such as decoders and multiplexers) to apply the fields of the instruction to the datapath elements.
- The instruction-set designer is limited by the fact that all instructions have to fit within a fixed length (usually one word). The limitation becomes apparent in the size of the immediate operands and address offsets specifiable in the instruction.

MIPS is an example of an architecture that uses instructions of the same length.

Here are examples of instructions from MIPS:



For example, in the above ADD instruction 5-bit field following Rd serves no purpose in the instruction but is there due to the fixed word length of each instruction.

- **Instructions may be of variable length**

In this format instructions are of variable length, i.e., an instruction may occupy multiple words.

- *Pros:*

- There is no wasted space since an instruction occupies exactly the space required for it.
- The instruction-set designer is not constrained by limited sizes (for example size of immediate fields).
- There is an opportunity to choose different sizes and encoding for the opcodes, addressing modes, and operands depending on their observed usage by the compiler.

- *Cons:*

- This complicates the implementation since the length of the instruction can be discerned only after decoding the opcode. This leads to a sequential interpretation of the instruction and its operands.

DEC VAX 11 family and the Intel x86 family are examples of architectures with variable length instructions. In VAX 11, the instructions can vary in size from 1 byte to 53 bytes.

It should be emphasized that our intent here is not to suggest that all such architectural styles and instruction formats covered in this section are feasible today. It is to give the reader an exposure to the variety of choices that have been explored in instruction-set design. For example, there have been commercial offerings in the past of stack-oriented architectures (with zero operand instruction format), and accumulator-based machines (with one operand instruction format). However, such architectures are no longer in the mainstream of general-purpose processors.

2.10 LC-2200 Instruction Set

As a concrete example of a simple architecture we define the LC-2200. This is a 32-bit register-oriented little-endian architecture with a fixed length instruction format. There are 16 general-purpose registers as well as a separate program counter (PC) register. All addresses are word addresses. The purpose of introducing this instruction set is three-fold:

- It serves as a concrete example of a simple instruction-set that can cater to the needs of any high-level language.
- It serves as a concrete architecture to discuss implementation issues in Chapters 3 and 5.
- Perhaps most importantly, it also serves as a simple unencumbered vehicle to add other features to the processor architecture in later chapters when we discuss interrupts, virtual memory and synchronization issues. This is particularly attractive since it exposes the reader to the process by which a particular feature may be added to the processor architecture to serve a specific functionality.

2.10.1 Instruction Format

LC-2200 supports four instruction formats. The R-type instruction includes **add** and **nand**. The I-type instruction includes **addi**, **lw**, **sw**, and **beq**. The J-type instruction is **jalr**, and the O-type instruction is **halt**. Thus, totally LC-2200 has only 8 instructions. Table 2.1 summarizes the semantics of these instructions.

R-type instructions (add, nand):

- | | | |
|-------------|---------------------------|-------------------------------------|
| bits 31-28: | opcode | 31 28 27 24 23 20 19 |
| bits 27-24: | reg X | 4 3 0 |
| bits 23-20: | reg Y | |
| bits 19-4: | unused (should be all 0s) | |
| bits 3-0: | reg Z | |

Opcode	Reg X	Reg Y	Unused	Reg Z
--------	-------	-------	--------	-------



I-type instructions (addi, lw, sw, beq):

- bits 31-28: opcode
- bits 27-24: reg X
- bits 23-20: reg Y
- bits 19-0: Immediate value or address offset (a 20-bit, 2s complement number with a range of -524288 to +524287)



J-type instructions (jalr):⁶

- bits 31-28: opcode
- bits 27-24: reg X (target of the jump)
- bits 23-20: reg Y (link register)
- bits 19-0: unused (should be all 0s)



O-type instructions (halt):

- bits 31-28: opcode
- bits 27-0: unused (should be all 0s)

⁶ LC-2200 does not have a separate unconditional jump instruction. However, it should be easy to see that we can realize such an instruction using JALR R_{link}, R_{don't-care}; where R_{link} contains the address to jump to, and R_{don't-care} is a register whose current value you don't mind trashing.

Mnemonic Example	Format	Opcode	Action Register Transfer Language
add add \$v0, \$a0, \$a1	R	0 0000 ₂	Add contents of reg Y with contents of reg Z, store results in reg X. RTL: $\$v0 \leftarrow \$a0 + \$a1$
nand nand \$v0, \$a0, \$a1	R	1 0001 ₂	Nand contents of reg Y with contents of reg Z, store results in reg X. RTL: $\$v0 \leftarrow \sim (\$a0 \&\& \$a1)$
addi addi \$v0, \$a0, 25	I	2 0010 ₂	Add Immediate value to the contents of reg Y and store the result in reg X. RTL: $\$v0 \leftarrow \$a0 + 25$
lw lw \$v0, 0x42(\$fp)	I	3 0011 ₂	Load reg X from memory. The memory address is formed by adding OFFSET to the contents of reg Y. RTL: $\$v0 \leftarrow \text{MEM}[\$fp + 0x42]$
sw sw \$a0, 0x42(\$fp)	I	4 0100 ₂	Store reg X into memory. The memory address is formed by adding OFFSET to the contents of reg Y. RTL: $\text{MEM}[\$fp + 0x42] \leftarrow \$a0$
beq beq \$a0, \$a1, done	I	5 0101 ₂	Compare the contents of reg X and reg Y. If they are the same, then branch to the address PC+1+OFFSET, where PC is the address of the beq instruction. RTL: if($\$a0 == \$a1$) PC \leftarrow PC+1+OFFSET
Note: For programmer convenience (and implementer confusion), the assembler computes the OFFSET value from the number or symbol given in the instruction and the assembler's idea of the PC. In the example, the assembler stores done-(PC+1) in OFFSET so that the machine will branch to label "done" at run time.			
jalr jalr \$at, \$ra	J	6 0110 ₂	First store PC+1 into reg Y, where PC is the address of the jalr instruction. Then branch to the address now contained in reg X. Note that if reg X is the same as reg Y, the processor will first store PC+1 into that register, then end up branching to PC+1. RTL: $\$ra \leftarrow \text{PC} + 1$; PC $\leftarrow \$at$ Note that an unconditional jump can be realized using jalr \$ra, \$t0 , and discarding the value stored in \$t0 by the instruction. This is why there is no separate jump instruction in LC-2200.
nop	n.a.	n.a.	Actually a pseudo instruction (i.e. the assembler will emit: add \$zero, \$zero, \$zero)
halt halt	O	7 0111 ₂	

Table 2.1: LC-2200 Instruction Set

2.10.2 LC-2200 Register Set

As we mentioned already, LC-2200 has 16 programmer visible registers. It turns out that zero is a very useful small integer in compiling high-level language programs. For example, it is needed for initializing program variables. For this reason, we dedicate register R0 to always contain the value 0. Writes to R0 are ignored by the architecture.

We give mnemonic names to the 16 registers consistent with the software convention that we introduced in Section 2.8.3. Further, since the assembler needs it, we introduce a ‘\$’ sign in front of the mnemonic name for a register. The registers, their mnemonic names, their intended use, and the software convention are summarized in Table 2.2

Reg #	Name	Use	callee-save?
0	\$zero	always zero (by hardware)	n.a.
1	\$at	reserved for assembler	n.a.
2	\$v0	return value	No
3	\$a0	argument	No
4	\$a1	argument	No
5	\$a2	argument	No
6	\$t0	Temporary	No
7	\$t1	Temporary	No
8	\$t2	Temporary	No
9	\$s0	Saved register	YES
10	\$s1	Saved register	YES
11	\$s2	Saved register	YES
12	\$k0	reserved for OS/traps	n.a.
13	\$sp	Stack pointer	No
14	\$fp	Frame pointer	YES
15	\$ra	return address	No

Table 2.2: Register convention

2.11 Issues influencing processor design

2.11.1 Instruction-set

Throughout this chapter, we focused on instruction-set design. We also made compiling high-level language constructs into efficient machine code as the over-arching concern in the design of an instruction-set. This concern is definitely true up to a point. However, compiler technology and instruction-set design have evolved to a point where this concern is not what is keeping the computer architects awake at nights in companies such as Intel or AMD. In fact, the 80’s and 90’s saw the emergence of many ISAs, some more elegant than the others, but all driven by the kinds of concerns we articulated in the earlier sections of this chapter. Perhaps one of the favorites from the point of view of elegance and performance is Digital Equipment Corporation’s Alpha architecture. The architects of DEC Alpha gave a lot of thought to intuitive and efficient code generation from the point of the compiler writer as well as an ISA design that will lend itself to an

efficient implementation. With the demise of DEC, a pioneer in mini computers throughout the 80's and 90's, the Alpha architecture also met its end.

The decade of the 80's saw the debate between *Complex Instruction Set Computers (CISC)* and *Reduced Instruction Set Computers (RISC)*. With CISC-style ISA, the task of a compiler writer is complicated by the plethora of choices that exist for compiling high-level constructs into machine code. Further, the complexity of the ISA makes efficient hardware implementation a significant challenge. Choices for the compiler writer are good in general of course, but if the programmer does not know the implication in terms of performance *a priori*, then such choices become questionable. With the maturation of compiler technology, it was argued that a RISC-style ISA is both easier to use for the compiler writer and would lead to better implementation efficiency than a CISC-style ISA.

As we all know, an ISA that has stood the test of time is Intel's x86. It represents a CISC-style ISA. Yet, x86 is still the dominant ISA while many elegant ones such DEC Alpha have disappeared. The reason is there are too many other factors (market pressure⁷ being a primary one) that determine the success or failure of an ISA. Performance is an important consideration of course, but the performance advantage of a really good ISA such as Alpha to that of x86 is not large enough to make that a key determinant. Besides, despite the implementation challenge posed by the x86 ISA, smart architects at Intel and AMD have managed to come up with efficient implementation, which when coupled with the prevailing high clock speeds of processors, make the performance advantage of a "good" ISA not that significant. At least not significant *enough* to displace a well entrenched ISA such as x86.

Ultimately, the success or failure of an ISA largely depends on market adoption. Today, computer software powers everything from commerce to entertainment. Therefore, adoption of an ISA by major software vendors (such as Microsoft, Google, IBM, and Apple) is a key factor that determines the success of an ISA. An equally important factor is the adoption of processors embodying the ISA by "box makers" (such as Dell, HP, Apple, and IBM). In addition to the traditional markets (laptops, desktops, and servers), embedded systems (such as game consoles, cell phones, PDAs, and automobiles) have emerged as dominant players in the computing landscape. It is not easy to pinpoint why an ISA may or may not be adopted by each of these segments (software giants, box makers, and builders of embedded systems). While it is tempting to think that such decisions are based solely on the elegance of an ISA, we know from the history of computing that this is not exactly true. Such decisions are often based on pragmatics: availability of good compilers (especially for C) for a given ISA, need to support legacy software, etc.

⁷ Part of the reason for this market pressure is the necessity to support legacy code, i.e., software developed on older versions of the same processor. This is called *backward compatibility* of a processor, and contributes to the bloated nature of the Intel x86 ISA.

2.11.2 Influence of applications on instruction-set design

Applications have in the past and continue to date, influence the design of instruction-set. In the 70's and perhaps into the 80's, computers were primarily used for number crunching in scientific and engineering applications. Such applications rely heavily on floating-point arithmetic. While high-end computers (such as IBM 370 series and Cray) included such instructions in their ISA, the so-called *mini* computers of that era (such as DEC PDP 11 series) did not include them in their ISA. There were successful companies (e.g., *Floating Point Systems Inc.*) that made attached processors for accelerating floating point arithmetic for such mini computers. Now floating-point instructions are a part of any general-purpose processor. Processors (e.g., StrongARM, ARM) that are used in embedded applications such as cellphones and PDAs may not have such instructions. Instead, they realize the effect of floating-point arithmetic using integer instructions for supporting math libraries.

Another example of applications' influence on the ISA is the MMX instructions from Intel. Applications that process audio, video, and graphics deal with *streaming* data, i.e., continuous data such as a movie or music. Such data would be represented as arrays in the memory. The MMX instructions, first introduced by Intel in 1997 in their Pentium line of processors, aimed at dealing with streaming data efficiently by the CPU. The intuition behind these instructions is pretty straightforward. As the name "stream data" suggests, audio, video and graphics applications require the same operation (such as addition) to be applied to corresponding elements of two or more streams. Therefore, it makes sense to have instructions that mimic this behavior. The MMX instructions originally introduced in the Pentium line and its successors, do precisely that. Each instruction (there are 57 of them grouped into categories such as arithmetic, logical, comparison, conversion, shift, and data transfer) takes two operands (each of which is not a scalar but a vector of elements). For example, an add instruction will add the corresponding elements of the two vectors⁸.

A more recent example comes from the gaming industry. Interactive video games have become very sophisticated. The graphics and animation processing required to be done in real-time in such gaming consoles have gotten to the point where it is beyond the capability of the general-purpose processors. Of course, you wouldn't want to lug around a supercomputer on your next vacation trip to play video games! Enter *Graphic Processing Units* (GPUs for short). These are special-purpose attached processors that perform the needed arithmetic for such gaming consoles. Basically, the GPU comprises a number of functional units (implementing primitive operations required in graphics rendering applications) that operate in parallel on a stream of data. A recent partnership between Sony, IBM, and Toshiba unveiled the Cell processor that takes the concept of GPUs a step further. The Cell processor comprises several processing elements on a

⁸ As a historical note, the MMX instructions evolved from a style of parallel architectures called Single Instruction Multiple Data (SIMD for short), which was prevalent until the mid 90's for catering to the needs of image processing applications. See Chapter 11 for an introduction to the different parallel architecture styles.

single chip, each of which can be programmed to do some specialized task. The Cell processor architecture has made its way into PlayStation (PS3).

2.11.3 Other issues driving processor design

ISA is only one design issue, and perhaps not the most riveting one, in the design of modern processors. We will list some of the more demanding issues in this section, some of which will be elaborated in later chapters.

1. **Operating system:** We already mentioned that operating system plays a crucial role in the processor design. One manifestation of that role is giving to the programmer an illusion of memory space that is larger than the actual amount of memory that is present in the system. Another manifestation is the responsiveness of the processor to external events such as interrupts. As we will see in later chapters, to efficiently support such requirements stemming from the operating system, the processor may include new instructions as well as new architectural mechanisms that are not necessarily visible via the ISA.
2. **Support for modern languages:** Most modern languages such as Java, C++, and C# provide the programmer with the ability to dynamically grow and shrink the data size of the program. Dubbed *dynamic memory allocation*, this is a powerful feature both from the point of view of the application programmer and from the point of view of resource management by the operating system. Reclaiming memory when the data size shrinks, referred to as *garbage collection* is crucial from the point of view of resource management. Mechanisms in the processor architecture for automatic garbage collection are another modern day concern in processor design.
3. **Memory system:** As you very well know, processor speeds have been appreciating exponentially over the past decade. For example, a Sun 3/50 had a processor speed of 0.5 MHz circa 1986. Today circa 2007, laptops and desktops have processor speeds in excess of 2 GHz. Memory density has been increasing at an exponential rate but memory speed has not increased at the same rate as processor speed. Often this disparity between processor and memory speeds is referred to as the *memory wall*. Clever techniques in the processor design to overcome the memory wall are one of the most important issues in processor design. For example, the design of cache memories and integrating them into the processor design is one such technique. We will cover these issues in a later chapter on memory hierarchies.
4. **Parallelism:** With increasing density of chips usually measured in millions of transistors on a single piece of silicon, it is becoming possible to pack more and more functionality into a single processor. In fact, the chip density has reached a level where it is possible to pack multiple processors on the same piece of silicon. These architectures, called *multi-core* and *many-cores* bring a whole new set of processor design issues⁹. Some of these issues, such as parallel programming, and memory consistency, are carried over from traditional multiprocessors (a box comprising several processors), which we discuss in a later chapter.

⁹ Architecturally, there is not a major difference between multi- and many-cores. However, the programming paradigm needs a radical rethinking if there are more than a few (up to 8 or 16) cores. Hence the distinction: multi-core may have up to 8 or 16 cores; anything beyond that is a many-core architecture.

5. **Debugging:** Programs have become complex. An application such as a web server, in addition to being parallel and having a large memory footprint, may also have program components that reach into the network and databases. Naturally, developing such applications is non-trivial. A significant concern in the design of modern processors is support for efficient debugging, especially for parallel program.
6. **Virtualization:** As the complexity of applications increases, their needs have become complex as well. For example, an application may need some services available only in one particular operating system. If we want to run multiple applications simultaneously, each having its own unique requirements, then there may be a need to simultaneously support multiple application execution environments. You may have a dual boot laptop, and your own unique reason for having this capability. It would be nice if you can have multiple operating systems co-existing simultaneously without you having to switch back and forth. *Virtualization* is the system concept for supporting multiple distinct execution environments in the same computer system. Architects are now paying attention to efficiently supporting this concept in modern processor design.
7. **Fault tolerance:** As the hardware architecture becomes more complex with multi- and many-cores and large memory hierarchies, the probability of component failures also increases. Architects are increasingly paying attention to processor design techniques that will hide such failures from the programmer.
8. **Security:** Computer security is a big issue in this day and age. We normally think of network attacks when we are worried about protecting the security of our computer. It turns out that the security can be violated even within a box (between the memory system and the CPU). Architects are increasingly paying attention to alleviate such concerns by incorporating encryption techniques for processor-memory communication.

2.12 Summary

Instruction-set serves as a contract between hardware and software. In this chapter, we started from basics to understand the issues in the design of an instruction set. The important points to take away from the chapter are summarized below:

- Influence of high-level language constructs in shaping the ISA
- Minimal support needed in the ISA for compiling arithmetic and logic expressions, conditional statements, loops, and procedure calls
- Pragmatic issues (such as addressing and access times) that necessitate use of registers in the ISA
- Addressing modes for accessing memory operands in the ISA commensurate with the needs of efficient compilation of high level language constructs
- Software conventions that guide the use of the limited register set available within the processor
- The concept of a software stack and its use in compiling procedure calls
- Possible extensions to a minimal ISA
- Other important issues guiding processor design in this day and age.

2.13 Review Questions

1. Having a large register-file is detrimental to the performance of a processor since it results in a large overhead for procedure call/return in high-level languages. Do you agree or disagree? Give supporting arguments.
2. Distinguish between the frame pointer and the stack pointer.
3. In the LC-2200 architecture, where are operands normally found for an add instruction?
4. This question pertains to endianness. Let's say you want to write a program for comparing two strings. You have a choice of using a 32-bit byte-addressable Big-endian or Little-endian architecture to do this. In either case, you can pack 4 characters in each word of 32-bits. Which one would you choose and how will you write such a program? [Hint: Normally, you would do string comparison one character at a time. If you can do it a word at a time instead of a character at a time, that implementation will be faster.]
5. An ISA may support different flavors of conditional branch instructions such as BZ (branch on Zero), BN (branch on negative), and BEQ (branch on equal). Figure out the predicate expressions in an “if” statement that may be best served by these different flavors of conditional branch instructions. Give examples of such predicates in an “if” statement, and how you will compile them using these different flavors of branch instructions.
6. We said that endianness will not affect your program performance or correctness so long as the use of a (high level) data structure is commensurate with its declaration. Are there situations where even if your program does not violate the above rule, you could be bitten by the endianness of the architecture? [Hint: Think of programs that cross network boundaries.]
7. Work out the details of implementing the *switch* statement of C using jump tables in assembly using any flavor of conditional branch instruction. [Hint: After ensuring that the value of the switch variable is within the bounds of valid case values, jump to the start of the appropriate code segment corresponding to the current switch value, execute the code and finally jump to exit.]
8. Procedure A has important data in both S and T registers and is about to call procedure B. Which registers should A store on the stack? Which registers should B store on the stack?
9. Consider the usage of the stack abstraction in executing procedure calls. Do all actions on the stack happen only via pushes and pops to the top of the stack? Explain circumstances that warrant reaching into other parts of the stack during program execution. How is this accomplished?

10. Answer True/False with justification: Procedure call/return cannot be implemented without a frame pointer.
11. DEC VAX has a single instruction for loading and storing all the program visible registers from/to memory. Can you see a reason for such an instruction pair? Consider both the pros and cons.
12. Show how you can simulate a subtract instruction using the existing LC-2200 ISA?
13. The BEQ instruction restricts the distance you can branch to from the current position of the PC. If your program warrants jumping to a distance larger than that allowed by the offset field of the BEQ instruction, show how you can accomplish such “long” branches using the existing LC-2200 ISA.
14. What is an ISA and why is it important?
15. What are the influences on instruction set design?
16. What are conditional statements and how are they handled in the ISA?
17. Define the term addressing mode.
18. In Section 2.8, we mentioned that local variables in a procedure are allocated on the stack. While this description is convenient for keeping the exposition simple, modern compilers work quite differently. This exercise is for you to search the Internet and find out how exactly modern compilers allocate space for local variables in a procedure call. [Hint: Recall that registers are faster than memory. So, the objective should be to keep as many of the variables in registers as possible.]
19. We use the term *abstraction* to refer to the stack. What is meant by this term? Does the term abstraction imply how it is implemented? For example, is a stack used in a procedure call/return a hardware device or a software device?
20. Given the following instructions

BEQ Rx, Ry, offset ;	if ($Rx == Ry$) $PC = PC + offset$
SUB Rx, Ry, Rz ;	$Rx \leftarrow Ry - Rz$
ADDI Rx, Ry, Imm ;	$Rx \leftarrow Ry + \text{Immediate value}$
AND Rx, Ry, Rz ;	$Rx \leftarrow Ry \text{ AND } Rz$

Show how you can realize the effect of the following instruction:

BGT Rx, Ry, offset ;	if ($Rx > Ry$) $PC = PC + offset$
----------------------	-------------------------------------

Assume that the registers and the Immediate fields are 8-bits wide. You can ignore overflow that may be caused by the SUB instruction.

21. Given the following load instruction

LW Rx, Ry, OFFSET ; Rx <- MEM[Ry + OFFSET]

Show how to realize a new addressing mode, called *indirect*, for use with the load instruction that is represented in assembly language as:

LW Rx, @(Ry) ;

The semantics of this instruction is that the contents of register Ry is the address of a pointer to the memory operand that must be loaded into Rx.

22. Convert this statement:

g = h + A[i];

into an LC-2200 assembler with the assumption that the Address of A is located in \$t0, g is in \$s1, h is in \$s2, and, i is in \$t1

23. Suppose you design a computer called the Big Looper 2000 that will never be used to call procedures and that will automatically jump back to the beginning of memory when it reaches the end. Do you need a program counter? Justify your answer.

24. Consider the following program and assume that for this processor:

- All arguments are passed on the stack.
- Register V0 is for return values.
- The S registers are expected to be saved, that is a calling routine can leave values in the S registers and expect it to be there after a call.
- The T registers are expected to be temporary, that is a calling routine must not expect values in the T registers to be preserved after a call.

```
int bar(int a, int b)
{
    /* Code that uses registers T5, T6, S11-S13; */
    return(1);
}

int foo(int a, int b, int c, int d, int e)
{
    int x, y;
    /* Code that uses registers T5-T10, S11-S13; */
    bar(x, y); /* call bar */
    /* Code that reuses register T6 and arguments a, b, and c;
     * return(0);
    */

main(int argc, char **argv)
{
    int p, q, r, s, t, u;
    /* Code that uses registers T5-T10, S11-S15; */
    foo(p, q, r, s, t); /* Call foo */
    /* Code that reuses registers T9, T10; */
}
```

Shown below is the stack when bar is executing, clearly indicate in the spaces provided which procedure (main, foo, bar) saved specific entries on the stack.

main	foo	bar	
_____	_____	_____	p
_____	_____	_____	q
_____	_____	_____	r
_____	_____	_____	s
_____	_____	_____	t
_____	_____	_____	u
_____	_____	_____	T9
_____	_____	_____	T10
_____	_____	_____	p
_____	_____	_____	q
_____	_____	_____	r
_____	_____	_____	s
_____	_____	_____	t
_____	_____	_____	x
_____	_____	_____	y
_____	_____	_____	S11
_____	_____	_____	S12
_____	_____	_____	S13
_____	_____	_____	S14
_____	_____	_____	S15
_____	_____	_____	T6
_____	_____	_____	x
_____	_____	_____	y
_____	_____	_____	S11
_____	_____	_____	S12
_____	_____	_____	S13

<----- Top of Stack

Chapter 3 Processor Implementation

(Revision number 20)

The previous chapter dealt with issues involved in deciding the instruction-set architecture of the processor. This chapter deals with the implementation of the processor once we have decided on an instruction set. The instruction-set is not a description of the implementation of the processor. It serves as a contract between hardware and software. For example, once the instruction-set is decided, a compiler writer can generate code for different high-level languages to execute on a processor that implements this contract. Naturally, we can have different implementations of the same instruction set. As we will see in this chapter, a number of factors go into deciding the implementation choice that one may make.

3.1 Architecture versus Implementation

First, let us understand why this distinction between architecture and implementation is important.

1. Depending on cost/performance, several implementations of the same architecture may be possible and even necessary to meet market demand. For example, it may be necessary to have a high performance version of the processor for a server market (such as a web server); at the same time, there may be a need for lower performance version of the same processor for an embedded application (such as a printer). This is why we see a *family* of processors adhering to a particular architecture description, some of which may even be unveiled by a vendor simultaneously (e.g., Intel Xeon series, IBM 360 series, DEC PDP 11 series, etc.).
2. Another important reason for decoupling architecture from implementation is to allow parallel development of system software and hardware. For example, it becomes feasible to validate system software (such as compilers, debuggers, and operating systems) for a new architecture even prior to an implementation of the architecture is available. This cuts down the time to market a computer system drastically.
3. Customers of high-performance servers make a huge investment in software. For example, Oracle database is a huge and complex database system. Such software systems evolve more slowly compared to generations of processors. Intel co-founder Gordon Moore predicted in 1965 that the number of transistors on a chip would double every two years. In reality, the pace of technology evolution has been even faster, with processor speed doubling every 18 months. This means that a faster processor can hit the market every 18 months. You would have observed this phenomenon if you have been following the published speeds of processors appearing in the market year after year. Software changes more slowly compared to hardware technology; therefore, it is important that *legacy* software run on new releases of processors. This suggests that we want to maintain the contract (i.e., the instruction-set) the same so that much of the software base (such as compilers and related tool

sets, as well as fine-tuned applications) remain largely unchanged from one generation of processor to the next. Decoupling architecture from implementation allows this flexibility to maintain binary compatibility for legacy software.

3.2 What is involved in Processor Implementation?

There are several factors to consider in implementing a processor: price, performance, power consumption, cooling considerations, operating environment, etc. For example, a processor for military applications may require a more rugged implementation capable of withstanding harsh environmental conditions. The same processor inside a laptop may not require as rugged an implementation.

There are primarily two aspects to processor implementations:

1. The first aspect concerns the organization of the electrical components (ALUs, buses, registers, etc.) commensurate with the expected price/performance characteristic of the processor.
2. The second aspect concerns thermal and mechanical issues including cooling and physical geometry for placement of the processor in a printed circuit board (often referred to as *motherboard*).

The above two issues are specifically for a single-chip processor. Of course, the hardware that is “inside a box” is much more than a processor. There is a host of other issues to consider for the box as a whole including printed circuit boards, backplanes, connectors, chassis design, etc. In general, computer system design is a tradeoff along several axes. If we consider just the high-end markets (supercomputers, servers, and desktops) then the tradeoff is mostly one of *price/performance*. However, in embedded devices such as cell phones, a combination of three dimensions, namely, *power consumption*, *performance*, and *area* (i.e., size), commonly referred to as *PPA*, has been the primary guiding principle for design decisions.

Super Computers	Servers	Desktops & Personal Computers	Embedded
<i>High performance primary objective</i>	<i>Intermediate performance and cost</i>	<i>Low cost primary objective</i>	<i>Small size, performance, and low power consumption primary objectives</i>

Computer design is principally an empirical process, riddled with tradeoffs along the various dimensions as mentioned above.

In this chapter, we will focus on processor implementation, in particular the design of the datapath and control for a processor. The design presented in this chapter is a basic one. In Chapter 5, we explore pipelined processor implementation.

We will first review some key hardware concepts that are usually covered in a first course on logic design.

3.3 Key hardware concepts

3.3.1 Circuits

Combinational logic: The output of such a circuit is a Boolean combination of the inputs to the circuit. That is, there is no concept of a *state* (i.e., memory). Basic logic gates (AND, OR, NOT, NOR, NAND) are the building blocks for realizing such circuits. Another way of thinking about such circuits is that there is no *feedback* from output of the circuit to the input.

Consider a patch panel that mixes different microphone inputs and produces a composite output to send to the speaker. The output of the speaker depends on the microphones selected by the patch panel to produce the composite sound. The patch panel is an example of a combinational logic circuit. Examples of combinational logic circuits found in the datapath of a processor include multiplexers, de-multiplexers, encoders, decoders, ALU's.

Sequential logic: The output of such a circuit is a Boolean combination of the current inputs to the circuit and the current *state* of the circuit. A memory element, called a *flip-flop*, is a key building block of such a circuit in addition to the basic logic gates that make up a combinational logic circuit.

Consider a garage door opener's control circuit. The “input” to the circuit is the clicker with a single push-button and some switches that indicate if the door is all the way up or all the way down. The “output” is a signal that tells the motor to raise or lower the door. The direction of motion depends on the “current state” of the door. Thus, the garage door opener's control circuit is a sequential logic circuit. Examples of sequential logic circuits usually found in the datapath of a processor include registers and memory.

3.3.2 Hardware resources of the datapath

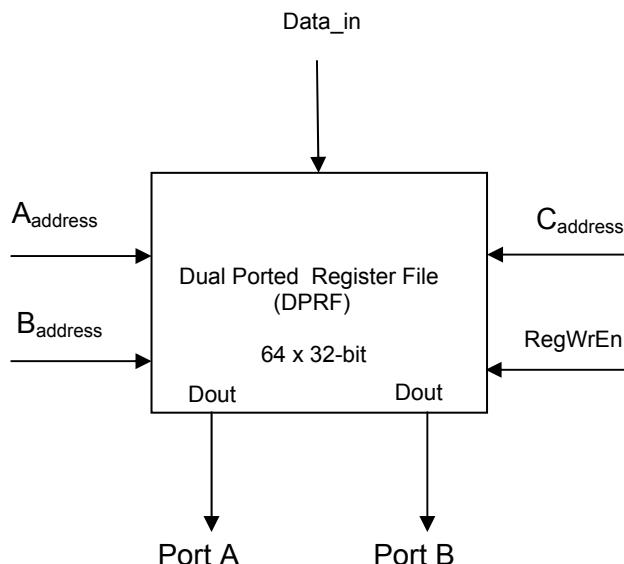
The datapath of a processor consists of combinational and sequential logic elements. With respect to the instruction-set of LC-2200 that we summarized in Chapter 2, let us identify the datapath resources we will need.

We need **MEMORY** to store the instructions and operands. We need an **Arithmetic Logic Unit (ALU)** to do the arithmetic and logic instructions. We need a **REGISTER-FILE** since they are the focal point of action in most instruction-set architectures. Most instructions use them. We need a **Program Counter** (henceforth referred to simply as **PC**) in the datapath to point to the current instruction and for implementing the branch and jump instructions we discussed in Chapter 2. When an instruction is brought from memory it has to be kept somewhere in the datapath; so we will introduce an **Instruction Register (IR)** to hold the instruction.

As the name suggests, a register-file is a collection of architectural registers that are visible to the programmer. We need control and data lines to manipulate the register file. These include address lines to name a specific register from that collection, and data lines for reading from or writing into that register. A register-file that allows a single register to be read at a time is called a *single ported register file (SPRF)*. A register-file that allows two registers to be read simultaneously is referred to as a *dual ported register file (DPRF)*. The following example sheds light on all the control and data lines needed for a register-file.

Example 1:

Shown below is a dual-ported register file (DPRF) containing 64 registers. Each register has 32 bits. $A_{address}$ and $B_{address}$ are the register addresses for reading the 32-bit register contents on to Ports A and B, respectively. $C_{address}$ is the register address for writing Data_in into a chosen register in the register file. RegWrEn is the write enable control for writing into the register file. Fill in the blanks.



Answer:

- Data_in has 32 wires
- Port A has 32 wires
- Port B has 32 wires
- $A_{address}$ has 6 wires
- $B_{address}$ has 6 wires
- $C_{address}$ has 6 wires
- RegWrEn has 1 wires

3.3.3 Edge Triggered Logic

The contents of a register changes from its current *state* to its new *state* in response to a clock signal.

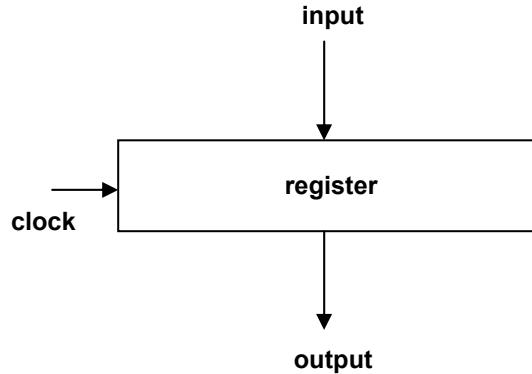


Figure 3.1: Register

The precise moment when the output changes state with respect to a change in the input depends on whether a storage element works on *level logic*¹ or *edge-triggered logic*. With level logic, the change happens as long as the clock signal is *high*. With edge-triggered logic, the change happens only on the *rising* or the *falling* edge of the clock. If the state change happens on the rising edge, we refer to it as *positive-edge-triggered logic*; if the change happens on the falling edge, we refer to it as *negative-edge-triggered logic*.

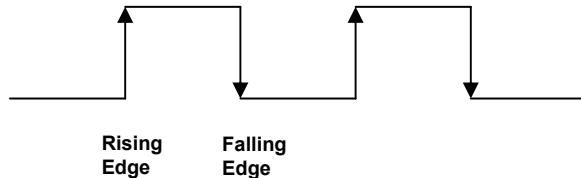


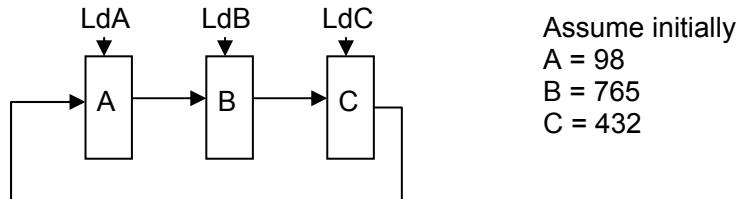
Figure 3.2: Clock

From now on in our discussion, we will assume positive edge triggered logic. We will discuss the details of choosing the width of the clock cycle in Section 3.4.2.

¹ It is customary to refer to a storage device that works with level logic as a *latch*. Registers usually denote edge-triggered storage elements.

Example 2:

Given the following circuit with three registers A, B, and C connected as shown:



LdA, LdB, and LdC are the clock signals for registers A, B, and C, respectively. In the above circuit if LdA, LdB, and LdC are enabled in a given clock cycle. What are the contents of A, B, and C in the next clock cycle?

Answer:

Whatever is at the input of the registers will appear at their respective outputs.

Therefore,

$$A = 432; B = 98; C = 765$$

The **MEMORY** element is special. As we saw in the organization of the computer system in Chapter 1, the memory subsystem is in fact completely separate from the processor. However, for the sake of simplicity in developing the basic concepts in processor implementation, we will include memory in the datapath design. For the purposes of this discussion, **we will say that memory is not edge-triggered**.

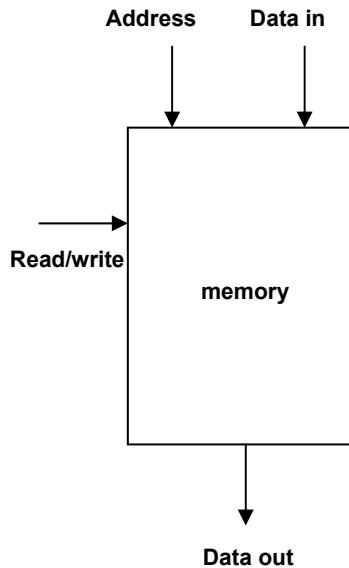


Figure 3.3: Memory

For example, to read a particular memory location, you supply the “Address” and the “Read” signal to the memory; after a finite amount of time (called the read access time of the memory) the **contents of the particular address is available on the “Data out” lines**.

Similarly, to write to a particular memory location, you supply the “Address”, “Data in”, and the “Write” signal to the memory; after a finite amount of time (the write access time) the **particular memory location would have the value supplied via “Data in”**. We will deal with memory systems in much more detail in Chapter 9.

3.3.4 Connecting the datapath elements

Let us consider what needs to happen to execute an **ADD** instruction of LC-2200 and from that derive how the datapath elements ought to be interconnected.

1. **Step 1:** We need to use the PC to **specify to the memory what location contains the instruction** (Figure 3.4).

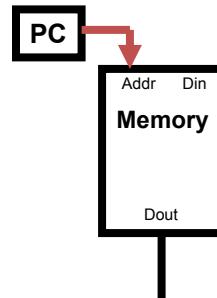


Figure 3.4: Step 1 – PC supplies instruction address to memory

2. **Step 2:** Once the instruction is read from the memory then it has to be stored in **the IR** (Figure 3.5).

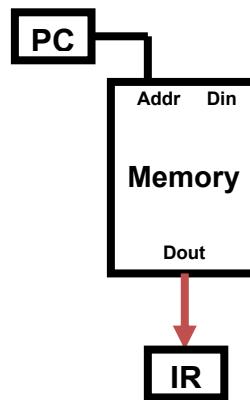


Figure 3.5: Step 2 – Instruction is read out of memory and **clocked into IR**

3. **Step 3:** Once the instruction is available in IR, then we **can use the register numbers specified in the instruction (contained in IR) to read the appropriate registers from the REGISTER-FILE** (dual ported similar to the one in Example 1), perform the addition using the ALU, and write the appropriate register into the REGISTER-FILE (Figure 3.6).

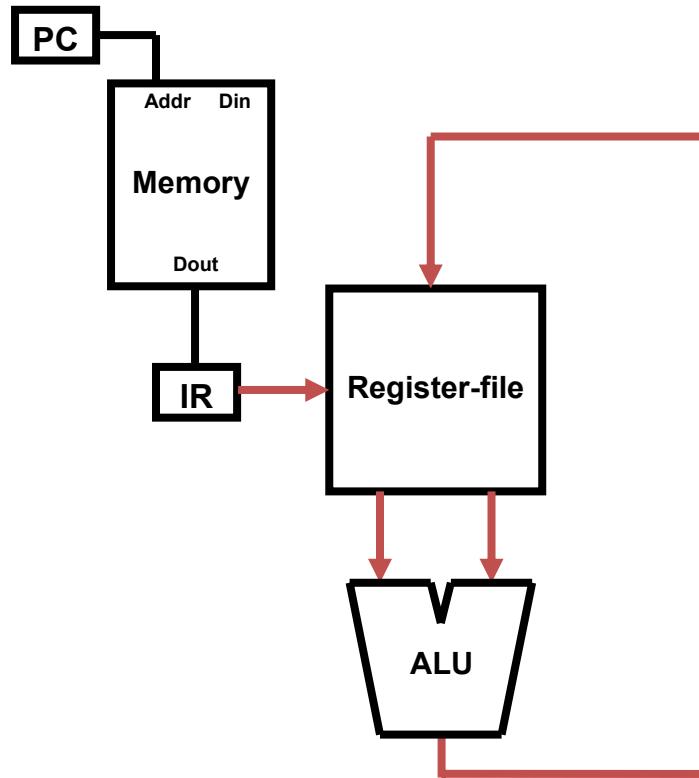


Figure 3.6: Step 3 – perform ADD two register values and store in third register

The above steps show the roadmap of the ADD instruction execution.

Let us see if all of the above steps can be completed in one clock cycle. As mentioned earlier, all the storage elements (except memory) are positive-edge triggered. What this means is that in one clock cycle we can transfer information from one storage element to another (going through combinational logic elements and/or memory) so long as the clock signal is long enough to account for all the intervening latencies of the logic elements. So for instance, Steps 1 and 2 can be completed in one clock cycle; but step 3 cannot be completed in the same clock cycle. For step 3, we have to supply the register number bits from IR to the register-file. However, due to the edge-triggered nature of IR, the instruction is available in IR only at the beginning of the next clock cycle (see Figure 3.7).

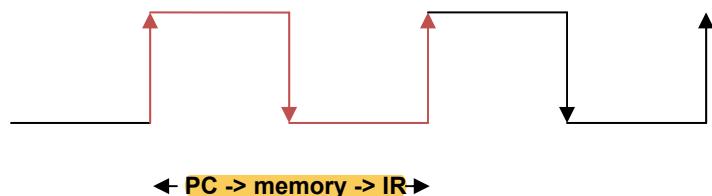


Figure 3.7: Steps 1 and 2 (first clock cycle)

It turns out that step 3 can be done in one clock cycle. At the beginning of the next clock cycle, the output of IR can be used to index the specific source registers needed for reading out of the register-file. The registers are read (similar to how memory is read given an address in the same cycle), passed to the ALU, the ADD operation is performed, and the results are written into the destination register (pointed to by IR again). Figure 3.8 illustrates the completion of step 3 in the second clock cycle.

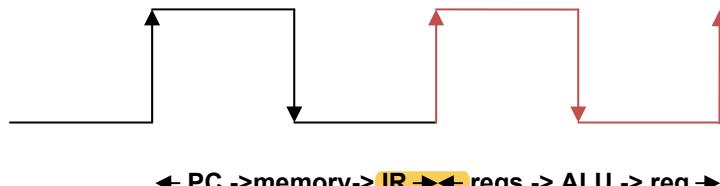


Figure 3.8: Step 3 (second clock cycle)

Determining the clock cycle time: Let us re-examine steps 1 and 2, which we said could be completed in one clock cycle. How wide should the clock cycle be to accomplish these steps? With reference to Figure 3.7, from the first rising edge of the clock, we can enumerate all the combinational delays encountered to accomplish steps 1 and 2:

- time that has to elapse for the output of PC to be stable for reading ($D_{r\text{-output-stable}}$);
- wire delay for the address to propagate from the output of PC to the Addr input of the memory ($D_{\text{wire-PC-Addr}}$);
- access time of the memory to read the addressed location ($D_{\text{mem-read}}$);
- wire delay for the value read from the memory to propagate to the input of IR ($D_{\text{wire-Dout-IR}}$);
- time that has to elapse for the input of IR to be stable before the second rising edge in Figure 3.7, usually called the setup time ($D_{r\text{-setup}}$);
- time that the input of IR has to be stable after the second rising edge, usually called the hold time ($D_{r\text{-hold}}$).

The width of the clock for accomplishing steps 1-2 should be greater than the sum of all the above delays:

$$\text{Clock width} > D_{r\text{-output-stable}} + D_{\text{wire-PC-Addr}} + D_{\text{mem-read}} + D_{\text{wire-Dout-IR}} + D_{r\text{-setup}} + D_{r\text{-hold}}$$

We do such an analysis for each of the potential paths of signal propagation in every clock cycle. Thereafter, we choose the clock width to be greater than the worse *case delay for signal propagation* in the entire datapath. We will formally define the terms involved in computing the clock cycle time in Section 3.4.2.

Example 3:

Given the following parameters (all in picoseconds), determine the minimum clock width of the system. Consider only steps 1-3 of the datapath actions.

$D_{r\text{-output-stable}}$	(PC output stable)	- 20 ps
$D_{\text{wire-PC-Addr}}$	(wire delay from PC to Addr of Memory)	- 250 ps
$D_{\text{mem-read}}$	(Memory read)	- 1500 ps
$D_{\text{wire-Dout-IR}}$	(wire delay from Dout of Memory to IR)	- 250 ps
$D_{r\text{-setup}}$	(setup time for IR)	- 20 ps
$D_{r\text{-hold}}$	(hold time for IR)	- 20 ps
$D_{\text{wire-IR-regfile}}$	(wire delay from IR to Register file)	- 250 ps
$D_{\text{regfile-read}}$	(Register file read)	- 500 ps
$D_{\text{wire-regfile-ALU}}$	(wire delay from Register file to input of ALU)	- 250 ps
$D_{\text{ALU-OP}}$	(time to perform ALU operation)	- 100 ps
$D_{\text{wire-ALU-regfile}}$	(wire delay from ALU output to Register file)	- 250 ps
$D_{\text{regfile-write}}$	(time for writing into a Register file)	- 500 ps

Answer:

Steps 1 and 2 are carried out in one clock cycle.

Clock width needed for steps 1 and 2

$$C_{1-2} > D_{r\text{-output-stable}} + D_{\text{wire-PC-Addr}} + D_{\text{mem-read}} + D_{\text{wire-Dout-IR}} + D_{r\text{-setup}} + D_{r\text{-hold}} \\ > 2060 \text{ ps}$$

Step 3 is carried out in one clock cycle.

Clock width needed for step 3

$$C_3 > D_{\text{wire-IR-regfile}} + D_{\text{regfile-read}} + D_{\text{wire-regfile-ALU}} + D_{\text{ALU-OP}} + D_{\text{wire-ALU-regfile}} + D_{\text{regfile-write}} \\ > 1850 \text{ ps}$$

Minimum clock width > worse case signal propagation delay

$$> \text{MAX } (C_{1-2}, C_3)$$

$$> 2060 \text{ ps}$$

The above parameters are fairly accurate circa 2007. What should be striking about these numbers is the fact that wire delays dominate.

3.3.5 Towards bus-based Design

We made up ad hoc connections among the datapath elements to get this one instruction executed. To implement another instruction (e.g., LD), we may have to create a path from the memory to the register file. If we extrapolate this line of thought, we can envision every datapath element connected to every other one. As it turns out, this is neither necessary nor the right approach. Let us examine what is involved in connecting the ALU to the register file. We have to run as many wires as the width of the datapath between the two elements. For a 32-bit machine, this means 32 wires. You can see wires

quickly multiply as we increase the connectivity among the datapath elements. Wires are expensive in terms of taking up space on silicon and we want to reduce the number of wires so that we can use the silicon real estate for active datapath elements. Further, just because we have more wires we do not necessarily improve the performance. For example, the fact that there are wires from the memory to the register file does not help the implementation of ADD instruction in any way.

Therefore, it is clear that we have to think through the issue of connecting datapath elements more carefully. In particular, the above discussion suggests that perhaps instead of dedicating a set of wires between every pair of datapath elements, we should think of designing the datapath sharing the wires among the datapath elements. Let us investigate how many sets of wires we need and how we can share them among the datapath elements.

Single bus design: One extreme is to have a single set of wires and have all the datapath elements share this single set of wires. This is analogous to what happens in a group meeting: one person talks and the others listen. Everyone takes a turn to talk during the meeting if he/she has something to contribute to the discussion. If everyone talks at the same time, there will be chaos of course. This is exactly how a *single bus system* – a single set of wires shared by all the datapath elements – works. Figure 3.9 shows such a system.

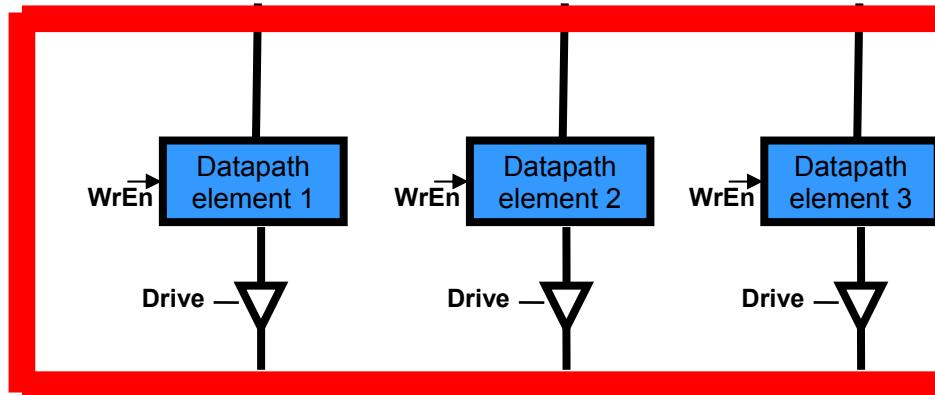


Figure 3.9 Single bus design

Bus denotes that the set of wires is shared. The first thing to notice is that the red line is a single bus (electrically), i.e., any value that is put on the bus becomes available on all segments of the wire. The second thing to notice is that there are the triangular elements that sit between the output of a datapath element and the bus. These are called *drivers* (also referred to as *tristate*² buffers). There is one such driver gate for each wire coming out from a datapath element that needs to be connected to a bus, and they isolate electrically the datapath element from the bus. Therefore, to electrically “connect”

² A binary signal is in one of two states: 0 or 1. The output of a driver when not enabled is in a third state, which is neither a 0 nor a 1. This is a high impedance state where the driver electrically isolates the bus from the datapath element that it is connected to. Hence the term tristate buffer.

datapath element 1 to the bus the associated driver must be “on”. This is accomplished by selecting the “Drive” signal associated with this driver. When this is done, we say datapath element 1 is “driving” the bus. It will be a mistake to have more than one datapath element “driving” the bus at a time. So the designer of the control logic has to ensure that only one of the drivers connected to the bus is “on” in a given clock cycle. If multiple drivers are “on” at the same time, apart from the fact that the value on the bus becomes unpredictable, there is potential for seriously damaging the circuitry. On the other hand, multiple data elements may choose to grab what is on the bus in any clock cycle. To accomplish this, the WrEn (write enable) signal associated with the respective data elements (shown in Figure 3.9) has to be turned “on.”

Two-bus design: Figure 3.10 illustrates a two-bus design. In this design, the register file is a dual ported one similar to that shown in Example 1. That is, two registers can be read and supplied to the ALU in the same clock cycle. Both the red (top) and purple (bottom) buses may carry address or data values depending on the need in a particular cycle. However, nominally, the red bus carries address values and the purple bus carries data values between the datapath elements.

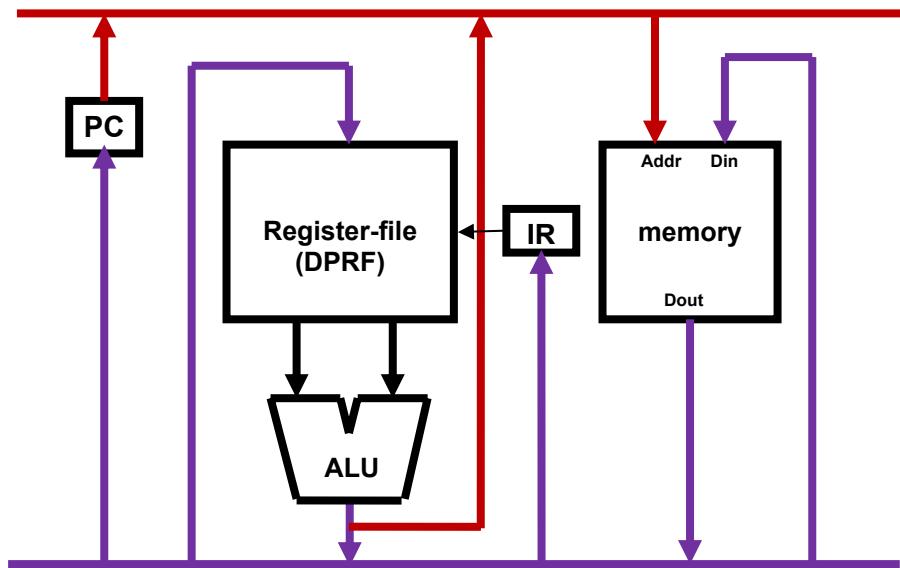


Figure 3.10 Two bus design

Although not shown in the Figure, it should be clear that there is a driver at the output of each of the datapath elements that are connected to either of the buses. How many cycles will be needed to carry out the Steps 1-3 mentioned mentioned at the beginning of Section 3.3.4? What needs to happen in each cycle?

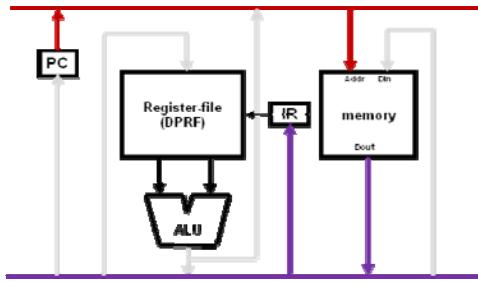
Let us explore these two questions.

First clock cycle:

- PC to Red bus (note: no one else can drive the Red bus in this clock cycle)
- Red bus to Addr of Memory
- Memory reads the location specified by Addr

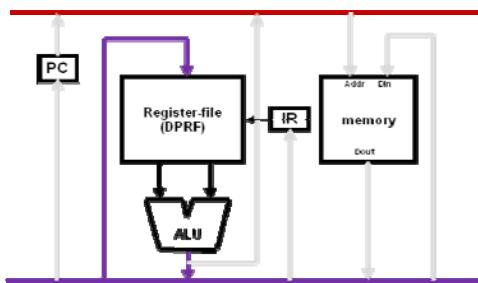
- Data from Dout to Purple bus (note: no one else can drive the Purple bus in this clock cycle)
- Purple bus to IR
- Clock IR

We have accomplished all that is needed in **Steps 1 and 2 in one clock cycle.**



Second clock cycle:

- IR supplies register numbers to Register file (see the dedicated wires represented by the arrow from IR to Register file); two source registers; one destination register
- Read the Register file and pull out the data from the two source registers
- Register file supplies the data values from the two source registers to the ALU (see the dedicated wires represented by the arrows from the Register file to the ALU)
- Perform the ALU ADD operation
- ALU result to Purple bus (note: no one else can drive the Purple bus in this clock cycle)
- Purple bus to Register file
- **Write to the register file at the destination register number specified by IR**



We have accomplished all that is needed for **Step 3 in one clock cycle.**

The key thing to take away is that we have accomplished steps 1-3 without having to run dedicated wires connecting every pair of datapath elements (except for the register-file to ALU connections, and the register selection wires from IR) by using the two shared buses.

3.3.6 Finite State Machine (FSM)

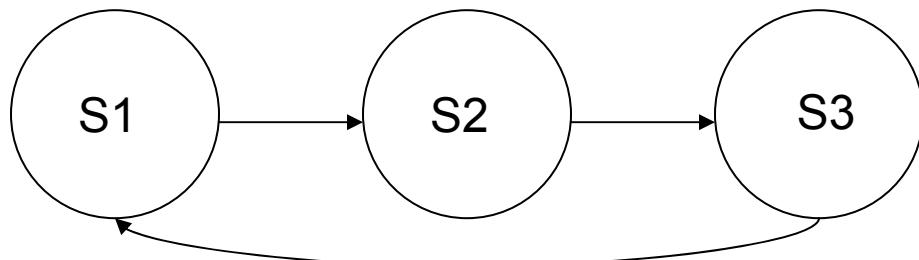


Figure 3.11: A Finite State Machine (FSM)

Thus far, we have summarized the circuit elements and how they could be assembled into a datapath for a processor. That is just one part of the processor design. The other

equally important aspect of processor design is the control unit. It is best to understand the control unit as a finite state machine, since it takes the datapath through successive stages to accomplish instruction execution.

A finite state machine, as the name suggests, has a *finite* number of states.

In Figure 3.11, S1, S2, and S3 are the *states* of the FSM. The arrows are the *transitions* between the states. **FSM** is an *abstraction* for any sequential logic circuit. It captures the desired behavior of the logic circuit. The state of the **FSM** corresponds to some physical state of the **sequential logic circuit**. Two things characterize a transition: (1) the external *input* that triggers that state change in the actual circuit, and (2) the *output* control signals generated in the actual circuit during the state transition. Thus, the **FSM** is a convenient way of capturing all the hardware details in the actual circuit.

For example, the simple FSM shown in Figure 3.12 represents the sequential logic circuit for the garage door opener that we introduced earlier. Table 3.1 shows the state transition table for this FSM with inputs that cause the transitions and the corresponding outputs they produce.

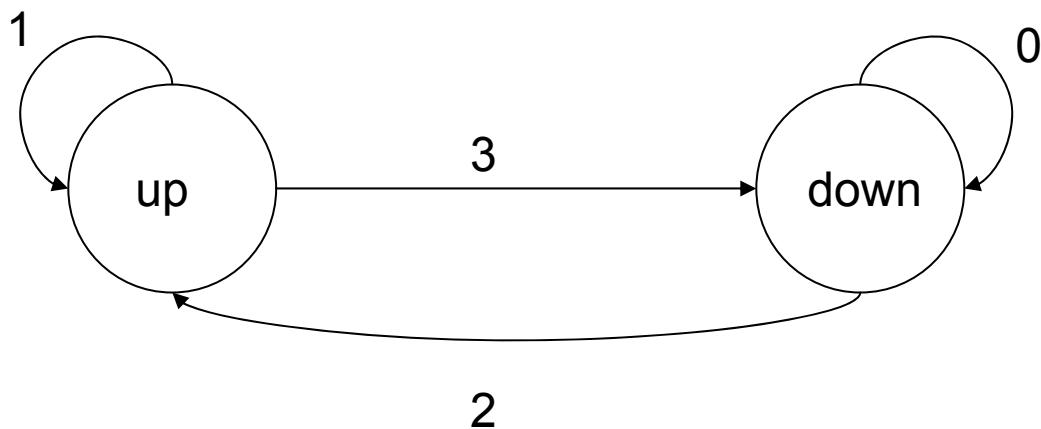


Figure 3.12: An FSM for garage door opener

Transition number	Input	State Current	State Next	Output
0	None	Down	Down	None
1	None	Up	Up	None
2	Clicker	Down	Up	Up motor
3	Clicker	Up	Down	Down motor

Table 3.1: State Transition Table for the FSM in Figure 3.12-(a)

The “up” state corresponds to the door being up, while the “down” state corresponds to the door being “down”. The input is the clicker push-button. The outputs are the control signals to the up and down motors, respectively. The transition labeled “0” and “1” correspond to there being no clicker action. The transition labeled “2” and “3” correspond to the clicker being pushed. The former state transition (“2”) is accompanied

with an “output” control signal to the up motor, while the latter (“3”) is accompanied with an “output” control signal to the down motor. Anyone who has had a logic design course knows that it is a straightforward exercise to design the sequential logic circuit given the FSM and the state transition diagram (see the two exercise problems at the end of this chapter that relate to the garage door opener).

We know that sequential logic circuits can be either *synchronous* or *asynchronous*. In the former, a state transition occurs synchronized with a clock edge, while in the latter a transition occurs as soon as the input is applied.

The control unit of a processor is a sequential logic circuit as well. Therefore, we can represent the control unit by an FSM shown in Figure 3.13.

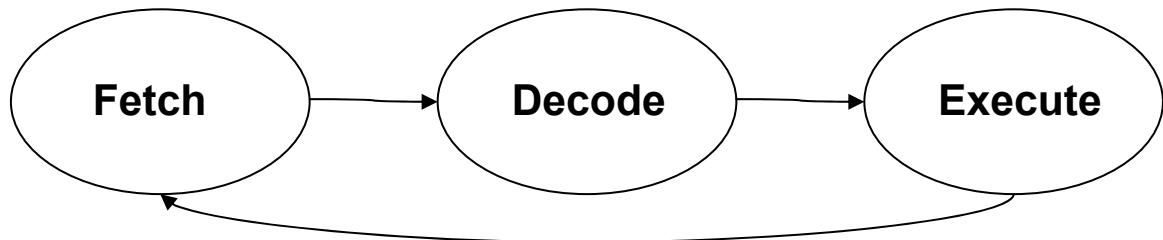


Figure 3.13: An FSM for controlling the CPU datapath

Fetch: This state corresponds to fetching the instruction from the memory.

Decode: This state corresponds to decoding the instruction brought from the memory to figure out the operands needed and the operation to be performed.

Execute: This state corresponds to carrying out the instruction execution.

We will return to the discussion of the control unit design shortly in Section 3.5.

3.4 Datapath Design

The Central Processing Unit (CPU) consists of the Datapath and the Control Unit. The datapath has all the logic elements and the control unit supplies the control signals to orchestrate the datapath commensurate with the instruction-set of the processor.

Datapath is the combination of the hardware resources and their connections. Let us review how to decide on the hardware resources needed in the datapath. As we already mentioned, the instruction-set architecture itself explicitly decides some of the hardware resources. In general, we would need more hardware resources than what is apparent from the instruction-set, as we will shortly see.

To make this discussion concrete, let us start by specifying the hardware resources needed by the instruction-set of LC-2200.

1. ALU capable of ADD, NAND, SUB,
2. register file with 16 registers (32-bit) shown in Figure 3.14
3. PC (32-bit)

4. Memory with $2^{32} \times 32$ bit words

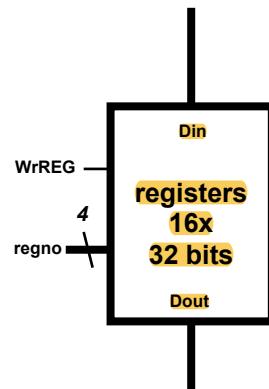


Figure 3.14: Register file with a single output port

Memory is a hardware resource that we need in LC-2200 for storing instructions and data. The size of memory is an implementation choice. The architecture only specifies the maximum size of memory that can be accommodated based on addressability. Given 32-bit addressing in LC-2200, the maximum amount of memory that can be addressed is 2^{32} words of 32-bits.

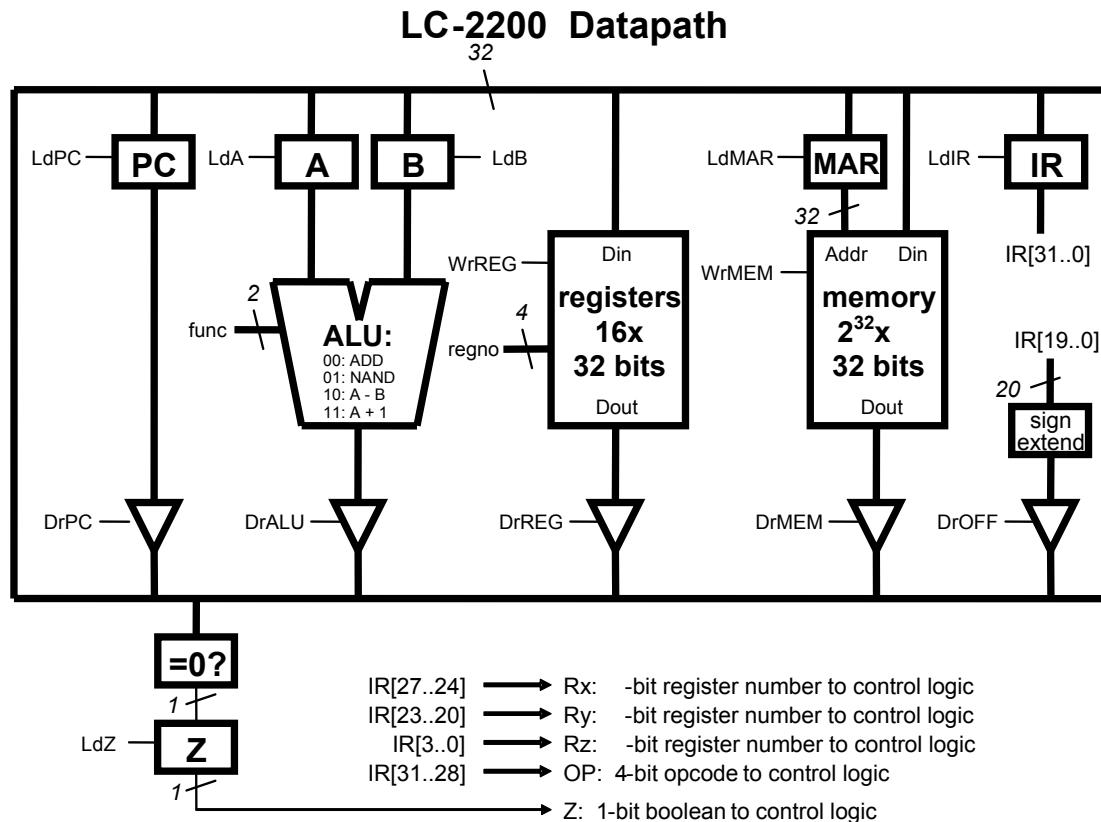


Figure 3.15: LC-2200 Datapath

Let us figure out what additional hardware resources may be needed. We already mentioned that when an instruction is brought from the memory it has to be kept in some

place in the datapath. IR serves this purpose. Let us assume we want a single bus to connect all these datapath elements. Regardless of the number of buses, one thing should be very evident right away looking at the register file. We can only get one register value out of the register file since there is only one output port (Dout). ALU operations require two operands. Therefore, we need some temporary register in the datapath to hold one of the registers. Further, with a single bus, there is exactly one channel of communication between any pair of datapath elements. This is the reason why we have **A** and **B** registers in front of the ALU. By similar reasoning, we need a place to hold the address sent by the ALU to the memory. **Memory Address Register (MAR)** serves this purpose. The purpose of the **Z** register (a 1-bit register) will become evident later on when we discuss the implementation of the instruction set. The zero-detect combination logic in front of the Z register (Figure 3.15) checks if the value on the bus is equal to zero. Based on the resources needed by the instruction-set, the limitations of the datapath, and implementation requirements of the instruction-set, we end up with a single bus design as shown in Figure 3.15.

3.4.1 ISA and datapath width

We have defined LC-2200 to be a 32-bit instruction-set architecture. It basically means that all instructions, addresses, and data operands are 32-bits in width. We will now explore the implication of this architectural choice on the datapath design. Let us understand the implication on the size of the buses and other components such as the ALU.

Purely from the point of view of logic design, it is conceivable to implement higher precision arithmetic and logic operations with lower precision hardware. For example, you could implement 32-bit addition using a 1-bit adder if you so choose. It will be slow but you can do it.

By the same token, you could have the bus in Figure 3.15 to be smaller than 32 bits. Such a design choice would have implications on instruction execution. For example, if the bus is just 8-bits wide, then you may have to make 4 trips to the memory to bring all 32-bits of an instruction or a memory operand. Once again, we are paying a performance penalty for making this choice.

It is a cost-performance argument that we would want to use lower precision hardware or narrower buses than what the ISA requires. The lesser the width of the buses the cheaper it is to implement the processor since most of the real estate on a chip is taken up by interconnects. The same is true for using lower precision hardware, since it will reduce the width of the wiring inside the datapath as well.

Thus, the datapath design represents a price/performance tradeoff. This is why, as we mentioned in Section 3.1, a chipmaker may bring out several versions of the same processor each representing a different point in the price/performance spectrum.

For the purposes of our discussion, we will assume that the architecture visible portions of the datapath (PC, register file, IR, and memory) are all 32-bits wide.

3.4.2 Width of the Clock Pulse

We informally discussed how to calculate the clock cycle width earlier in Section 3.3 (see Example 3). Let us formally define some of the terms involved in computing the clock cycle time.

- Every combinational logic element (for example the ALU or the drive gates in Figure 3.15) has latency for propagating a value from its input to the output, namely, *propagation delay*.
- Similarly, there is latency (called *access time*) from the time a register is enabled for reading (for example by applying a particular *regno* value to the register file in Figure 3.15) until the contents of that register appears on the output port (Dout).
- To write a value into a register, the *input to the register* has to be *stable* (meaning the value does not change) *for some amount of time (called *set up time*) before the rising edge of the clock*.
- Similarly, the *input to the register* has to continue to be *stable for some amount of time (called *hold time*) after the rising edge of the clock*.
- Finally, there is a *transmission delay* (also referred to as *wire delay*) for a value placed at the output of a logic element to traverse on the wire and appear at the input of another logic element (for example from the output of a drive gate to the input of PC in Figure 3.15).

Thus if in a single clock cycle we wish to perform a datapath action that reads a value from the register file and puts it into the A register, we have to add up all the constituent delays. We compute the worst-case delay for any of the datapath actions that needs to happen in a single clock cycle. This gives us a *lower bound* for the clock cycle time.

3.4.3 Checkpoint

So far, we have reviewed the following hardware concepts:

- Basics of logic design including combinational and sequential logic circuits
- Hardware resources for a datapath such as register file, ALU, and memory
- Edge-triggered logic and arriving at the width of a clock cycle
- Datapath interconnection and buses
- Finite State Machines

We have used these concepts to arrive at a datapath for LC-2200 instruction-set architecture.

3.5 Control Unit Design

Take a look at the picture in Figure 3.16. The role of the conductor of the orchestra is to pick out the members of the orchestra who should be playing/singing at any point of time. Each member knows what he/she has to play, so the conductor's job is essentially

keeping the timing and order, and not the content itself. If the datapath is the orchestra, then the control unit is the conductor. The control unit gives cues for the various datapath elements to carry out their respective functions. For example, if the DrALU line is asserted (i.e., if a 1 is placed on this line) then the corresponding set of driver gates will place whatever is at the output of the ALU on the corresponding bus lines.



Figure 3.16: An Orchestral arrangement³

Inspecting the datapath, we can list the control signals needed from the control unit:

- **Drive signals:** DrPC, DrALU, DrREG, DrMEM, DrOFF
- **Load signals:** LdPC, LdA, LdB, LdMAR, LdIR
- **Write Memory signal:** WrMEM
- **Write Registers signal:** WrREG
- **ALU function selector:** func
- **Register selector:** regno

There are several possible alternate designs to generate the control signals, all of which are the hardware realization of the FSM abstraction for the control unit of the processor.

3.5.1 ROM plus state register

³ Source: http://www.sanyo.org.za/img/IMG_8290_nys.jpg

Let us first look at a very simple design. First, we need a way of knowing what state the processor is in. Earlier we introduced an FSM for the control unit consisting of the states FETCH, DECODE, and EXECUTE. These are the **macro** states of the processor in the FSM abstraction. In a real implementation, several **microstates** may be necessary to carry out the details of each of the macro states depending on the capabilities of the datapath. For example, let us assume that it takes three microstates to implement the FETCH macro states. We can then encode these microstates as

ifetch1	0000
ifetch2	0001
ifetch3	0010

We introduce a **state register** the contents of which hold the encoding of these microstates. So, at any instant, the contents of this register shows the state of the processor.

The introduction of the state register brings us a step closer to hardware implementation from the FSM abstraction. Next, to control the datapath elements in each microstate, we have to generate the control signals (listed above). Let us discuss how to accomplish the generation of control signals.

One simple way is to use the state register as an index into a table. Each entry of the table contains the control signals needed in that state. Returning to the analogy of an orchestra, the conductor has the music score in front of her. She maintains the “state” of the orchestra with respect to the piece they are playing. Each line of the music score tells her who all should be playing at any point of time, just as an entry in the table of the control unit signifies the datapath elements that should participate in the datapath actions in that state. The individual player knows what he/she has to play. In the same manner, each datapath element knows what it has to do. In both cases, they need someone to tell them “when” to do their act. So the job of the conductor and the control unit is exactly similar in that they give the timing necessary (the “when” question) for the players and the datapath elements to do their part at the right time, respectively. This appears quite straightforward. So let us represent each control signal by **one bit** in this table entry. If the value of the bit is one then the control signal is generated; if it is 0 then it is not generated. Of course, the number of bits in **func** and **regno** fields corresponds to their width in the datapath (2 and 4, respectively, see Figure 3.15). Figure 3.17 shows a layout of the table entry for the control signals:

Current State	Drive Signals					Load Signals				Write Signals			func	regno
	PC	ALU	Reg	MEM	OFF	PC	A	B	MAR	IR	MEM	REG		

Figure 3.17: An entry of the table of control signals

The control unit has to transition from one state to another. For example, if the FETCH macro state needs 3 micro states, then

<u>Current state</u>	<u>Next state</u>
ifetch1	ifetch2
ifetch2	ifetch3

It is relatively straightforward to accomplish this next state transition by making the next state part of the table entry. So now, our table looks as shown in Figure 3.18.

Current State	Drive Signals						Load Signals				Write Signals				Next State
	PC	ALU	Reg	MEM	OFF	PC	A	B	MAR	IR	MEM	REG	Func	regno	
...															

Figure 3.18: Next State field added to the control signals table

Let us investigate how to implement this table in hardware.

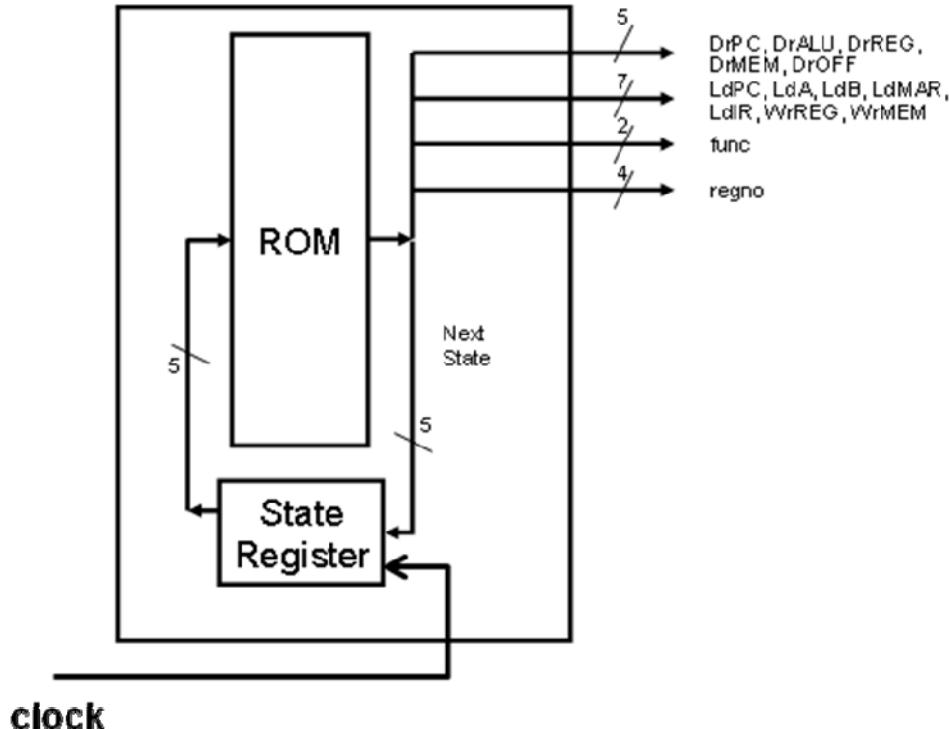


Figure 3.19: Control Unit

The table is nothing but a memory element. The property of this memory element is that once we have determined the control signals for a particular state then we can *freeze* the

contents of that table entry. We refer to this kind of memory as *Read-Only-Memory* or *ROM*.

Therefore, our hardware for the control unit looks as shown in Figure 3.19. On every clock tick, the state register advances to the next state as specified by the output of the ROM entry accessed in the current clock cycle. This is the same clock that drives all the edge-triggered storage elements in the datapath (see Figure 3.15). All the load signals coming out of the ROM (LdPC, LdMAR, etc.) serve as *masks* for the clock signal in that they determine if the associated storage element they control should be clocked in a given clock cycle.

The next thing to do is to combine the datapath and control. This is accomplished by simply connecting the correspondingly named entities in the datapath (see Figure 3.15) to the ones coming out of the ROM.

The way this control unit works is as follows:

1. The state register names the state that the processor is in for this clock cycle.
2. The ROM is accessed to retrieve the contents at the address pointed to by the state register.
3. The output of the ROM is the current set of control signals to be passed on to the datapath.
4. The datapath carries out the functions as dictated by these control signals in this clock cycle.
5. The *next state* field of the ROM feeds the state register so that it can transition to the next state at the beginning of the next clock cycle.

The above five steps are repeated in every clock cycle.

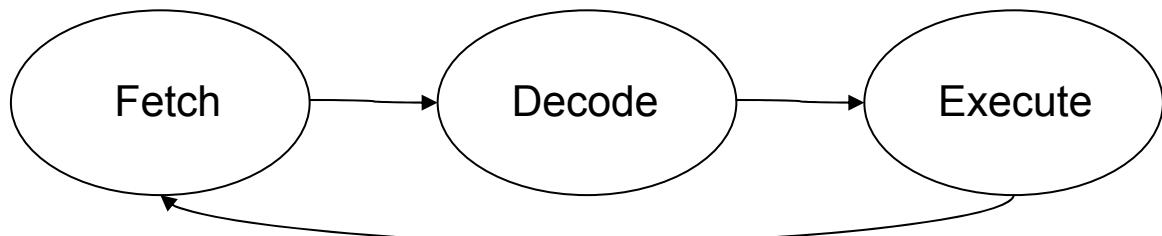


Figure 3.20: FSM for the CPU datapath reproduced

Figure 3.13 introduced the control unit of the processor as an FSM. Figure 3.20 is a reproduction of the same figure for convenience. Now let us examine what needs to happen in every macro state represented by the above FSM (Figure 3.20) and how our control unit can implement this. For each microstate, we will show the datapath actions alongside.

3.5.2 FETCH macro state

The FETCH macro state fetches an instruction from memory at the address pointed to by the Program Counter (PC) into the Instruction Register (IR); it subsequently increments the PC in readiness for fetching the next instruction.

Now let us list what needs to be done to implement the FETCH macro state:

- We need to send PC to the memory
- Read the memory contents
- Bring the memory contents read into the IR
- Increment the PC

With respect to the datapath, it is clear that with a single-bus datapath all of these steps cannot be accomplished in one clock cycle. We can break this up into the following microstates (each microstate being executed in one clock cycle):

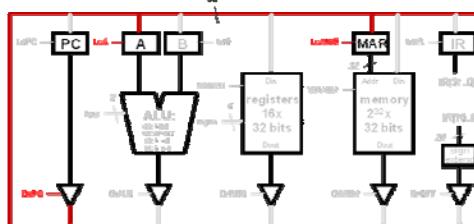
- **ifetch1**
 $\text{PC} \rightarrow \text{MAR}$
- **ifetch2**
 $\text{MEM}[\text{MAR}] \rightarrow \text{IR}$
- **ifetch3**
 $\text{PC} \rightarrow \text{A}$
- **ifetch4**
 $\text{A+1} \rightarrow \text{PC}$

With a little bit of reflection, we will be able to accomplish the actions of the FETCH macro state in fewer than 4 cycles. Observe what is being done in **ifetch1** and **ifetch3**. The content of PC is transferred to MAR and A registers, respectively. These two states can be collapsed into a single state since the contents of PC once put on the bus can be clocked into both the registers in the same cycle. Therefore, we can simplify the above sequence to:

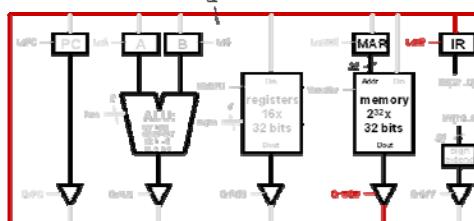
- **ifetch1**
 $\text{PC} \rightarrow \text{MAR}$
 $\text{PC} \rightarrow \text{A}$
- **ifetch2**
 $\text{MEM}[\text{MAR}] \rightarrow \text{IR}$
- **ifetch3**
 $\text{A+1} \rightarrow \text{PC}$

Now that we have identified what needs to be done in the datapath for the microstates that implement the FETCH macro state, we can now write down the control signals needed to effect the desired actions in each of these microstates. For each microstate, we highlight the datapath elements and control lines that are activate.

- **ifetch1**
 $\text{PC} \rightarrow \text{MAR}$
 $\text{PC} \rightarrow \text{A}$
Control signals needed:
DrPC
LdMAR
LdA

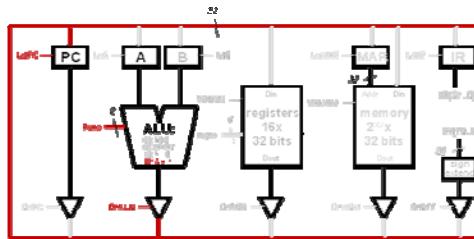


- **ifetch2**
 $\text{MEM}[\text{MAR}] \rightarrow \text{IR}$
Control signals needed:
DrMEM
LdIR



Note: with reference to the datapath, the default action of the memory is to read (i.e. when WrMEM is 0). Also the memory implicitly reads the contents of the memory at address contained in MAR and has the result available at *Dout* in ifetch2.

- **ifetch3**
 $\text{A}+1 \rightarrow \text{PC}$
Control signals needed:
func = 11
DrALU
LdPC



Note: If the func selected is 11 then the ALU implicitly does A+1 (see Figure 3.15).

Now we can fill out the contents of the ROM for the addresses associated with the microstates **ifetch1**, **ifetch2**, and **ifetch3**. (X denotes “don’t care”). The next-state field of **ifetch3** is intentionally marked TBD (To Be Determined), and we will come back to that shortly.

Current State	State Num	Drive Signals					Load Signals				Write Signals			Func	regno	Next State
		PC	ALU	Reg	MEM	OFF	PC	A	B	MAR	IR	MEM	REG			
Ifetch1	00000	1	0	0	0	0	0	1	0	1	0	0	0	xx	xxxx	00001
Ifetch2	00001	0	0	0	1	0	0	0	0	0	1	0	0	xx	xxxx	00010
Ifetch3	00010	0	1	0	0	0	1	0	0	0	0	0	0	11	xxxx	TBD

Figure 3.21: ROM with some entries filled in with control signals

This is starting to look like a *program* albeit at a much lower level than what we may have learned in a first computer-programming course. Every ROM location contains a set of commands that actuate different parts of the datapath. We will call each table entry a *microinstruction* and we will call the entire contents of the ROM a *micro program*. Each microinstruction also contains the (address of the) next microinstruction to be executed. Control unit design has now become a programming exercise. It is the

ultimate concurrent program since we are exploiting all the hardware concurrency that is available in the datapath in every microinstruction.

You can notice that there is a structure to each microinstruction. For example, all the drive signals can be grouped together; similarly, all the load signals can be grouped together. There are opportunities to reduce the space requirement of this table. For example, it may be possible to combine some of the control signals into one encoded field. Since we know that only one entity can drive the bus at any one time, we could group all the drive signals into one 3-bit field and each unique code of this 3-bit field signifies the entity selected to put its value on the bus. While this would reduce the size of the table, it adds a decoding step, which adds to the delay in the datapath for generating the drive control signals. We cannot group all the load signals into one encoded field since multiple storage elements may need to be clocked in the same clock cycle.

3.5.3 DECODE macro state

Once we are done with fetching the instruction, we are ready to decode it. So from the `ifetch3` microstate we want to transition to the DECODE macro state.

In this macro state, we want to examine the contents of `IR` (bits 31-28) to figure out what the instruction is. Once we know the instruction, then we can go to that part of the micro program that implements that particular instruction. So we can think of the DECODE process as a multi-way branch based on the `OPCODE` of the instruction. So we will redraw the control unit FSM depicting the DECODE as a multi-way branch. Each leg of the multi-way branch takes the FSM to the macro state corresponding to the execution sequence for a particular instruction. To keep the diagram simple, we show the multi-way branch taking the FSM to a particular class of instruction.

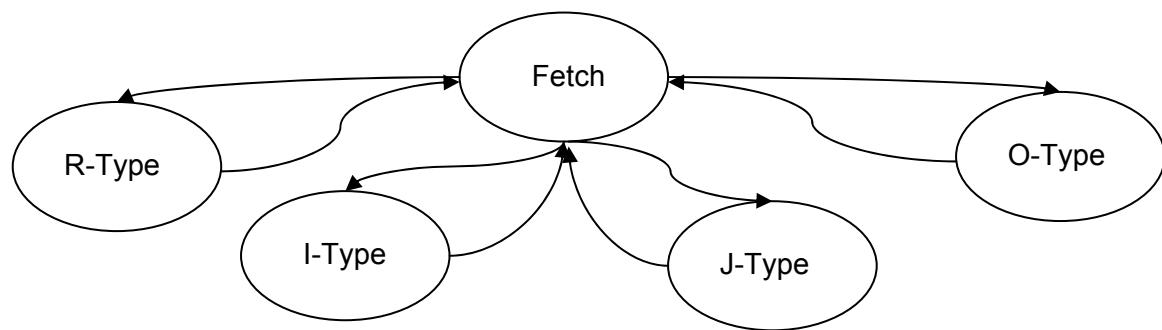
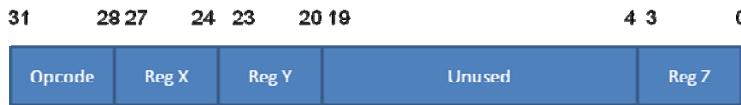


Figure 3.22: Extending the FSM with DECODE macro state fleshed out

We will come back to the question of implementing the multi-way branch in the control unit shortly. Let us first address the simpler issue of implementing each of the instructions.

3.5.4 EXECUTE macro state: ADD instruction (part of R-Type)

R-type has the following format:



Recall that the ADD instruction does the following:

$$R_X \leftarrow R_Y + R_Z$$

To implement this instruction, we have to read two registers from the register file and write to a third register. The registers to be read are specified as part of the instruction and are available in the datapath as the contents of IR. However, as can be seen from the datapath there is no path from the IR to the register file. There is a good reason for this omission. Depending on whether we want to read one of the source registers or write to a destination register, we need to send different parts of the IR to the **regno** input of the register-file. As we have seen **multiplexer** is the logic element that will let us do such selection.

Therefore, we add the following element (shown in Figure 3.23) to the datapath.

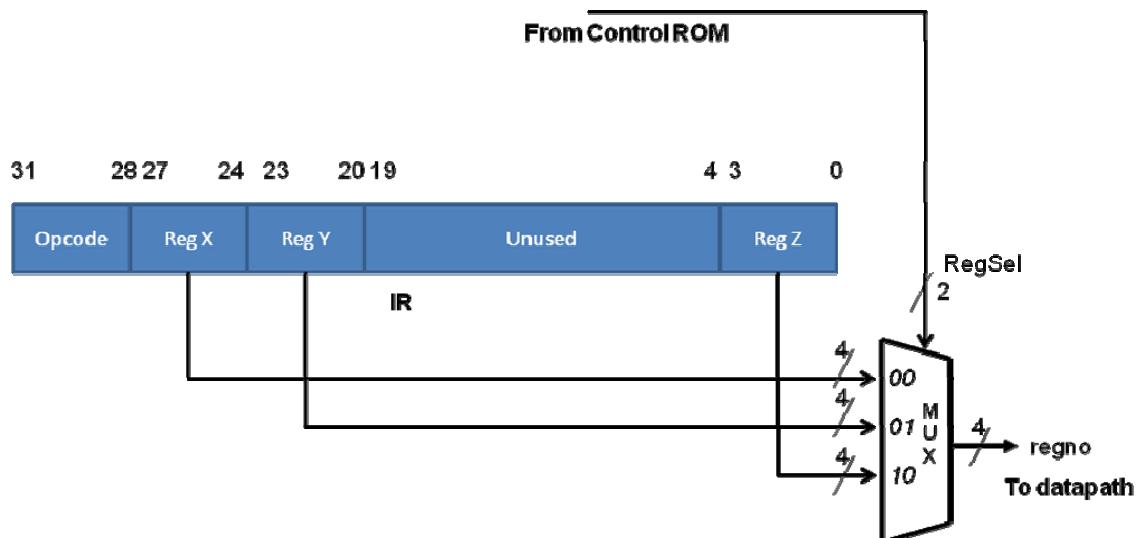


Figure 3.23: Using the IR bit fields to specify register selection

The **RegSel** control input (2-bits) comes from the **microinstruction**. The inputs to the multiplexer are the three different register-specifier fields of the IR (see Chapter 2 for the format of the LC-2200 instructions). It turns out that the register file is never addressed directly from the microinstruction. Therefore, we replace the 4-bit **regno** field of the microinstruction with a 2-bit **RegSel** field.

Current State	Drive Signals					Load Signals				Write Signals			Func	RegSel	Next State
	PC	ALU	Reg	MEM	OFF	PC	A	B	MAR	IR	MEM	REG			
...															

Figure 3.24: RegSel field added to the ROM control signals

Now we can write the datapath actions and the corresponding control signals needed in the microstates to implement the ADD execution macro state:

- **add1**

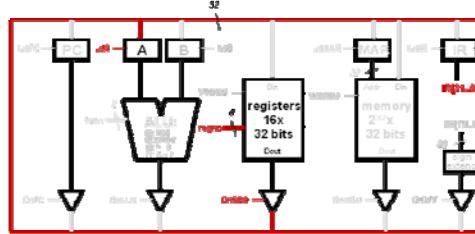
$Ry \rightarrow A$

Control signals needed:

RegSel = 01

DrREG

LdA



Note: The default action of the register-file is to read the contents of the register-file at the address specified by **regno** and make the data available at **Dout**.

- **add2**

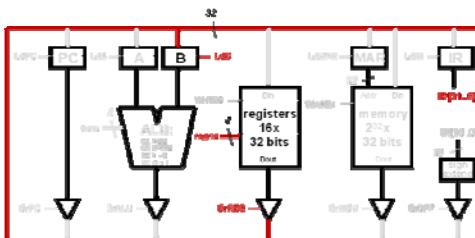
$Rz \rightarrow B$

Control signals needed:

RegSel = 10

DrREG

LdB



- **add3**

$A+B \rightarrow Rx$

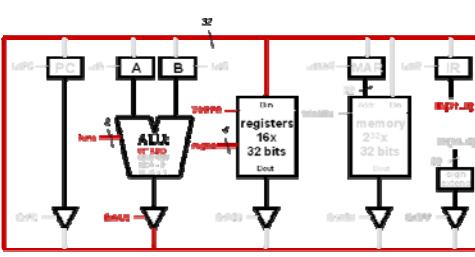
Control signals needed:

func = 00

DrALU

RegSel = 00

WrREG



The ADD macro state is implemented by the control unit sequencing through the microstates **add1**, **add2**, **add3** and then returning to the FETCH macro state:

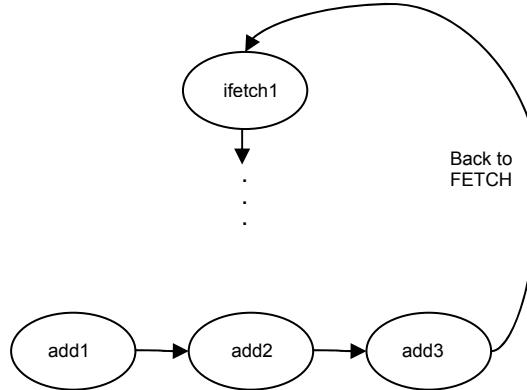


Figure 3.25: ADD macro state fleshed out in the FSM

3.5.5 EXECUTE macro state: NAND instruction (part of R-Type)

Recall that the NAND instruction does the following:

$$R_X \leftarrow R_Y \text{ NAND } R_Z$$

The NAND macro state is similar to ADD and consists of **nand1**, **nand2**, and **nand3** microstates. We leave it as an exercise to the reader to figure out what changes are needed in those microstates compared to the corresponding states for ADD.

3.5.6 EXECUTE macro state: JALR instruction (part of J-Type)

J-type instruction has the following format:



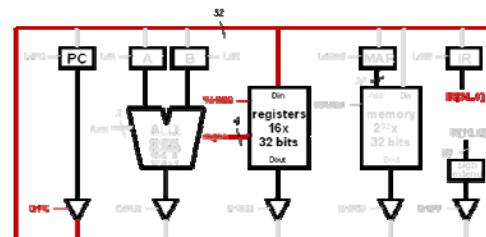
Recall that JALR instruction was introduced in LC-2200 for supporting the subroutine calling mechanism in high-level languages. JALR, stashes the return address in a register, and transfers control to the subroutine by doing the following:

$$R_Y \leftarrow PC + 1$$

$$PC \leftarrow R_X$$

Given below are the microstates, datapath actions, and control signals for the JALR instruction:

- **jalr1**
- $PC \rightarrow Ry$
- Control signals needed:
DrPC
RegSel = 01
WrREG



Note: PC+1 needs to be stored in Ry. Recall that we already incremented PC in the FETCH macro state.

- jalr2

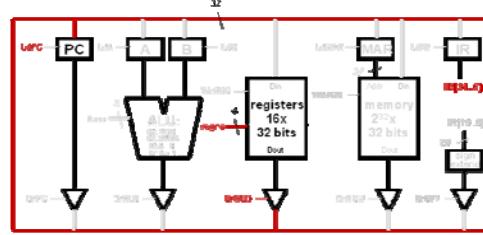
Rx → PC

Control signals needed:

RegSel = 00

DrREG

LdPC



3.5.7 EXECUTE macro state: LW instruction (part of I-Type)

I-type instruction has the following format:



Recall that LW instruction has the following semantics:

$$Rx \leftarrow \text{MEMORY}[Ry + \text{signed address-offset}]$$

In the I-Type instruction, the signed address offset is given by an immediate field that is part of the instruction. The immediate field occupies IR 19-0. As can be seen from the datapath, there is a **sign-extend** hardware that converts this 20-bit 2's complement value to a 32-bit 2's complement value. The **DrOFF** control line enables this sign-extended offset from the IR to be placed on the bus.

Given below are the microstates, datapath actions, and control signals for LW instruction:

- lw1

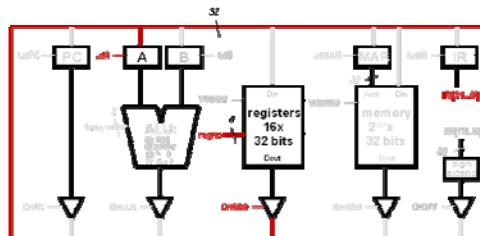
Ry → A

Control signals needed:

RegSel = 01

DrREG

LdA



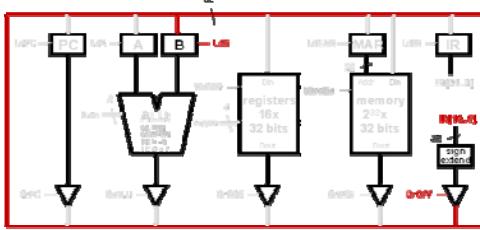
- lw2

Sign-extended offset → B

Control signals needed:

DrOFF

LdB



- lw3

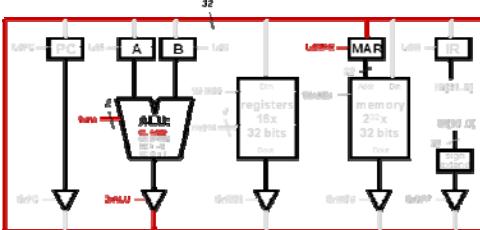
A+B → MAR

Control signals needed:

func = 00

DrALU

LdMAR



- **Iw4**

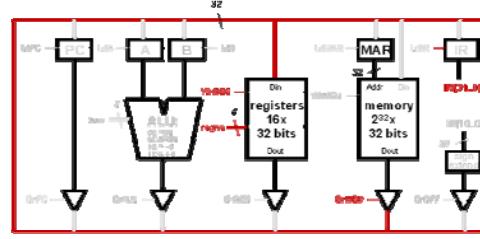
MEM[MAR] → Rx

Control signals needed:

DrMEM

RegSel = 00

WrREG



Example 4:

We have decided to add another addressing mode **autoincrement** to LC-2200. This mode comes in handy for LW/SW instructions. The semantics of this addressing mode with LW instruction is as follows:

LW Rx, (Ry)+ ; Rx <- MEM[Ry];
 ; Ry <- Ry + 1;

The instruction format is as shown below:

31	28 27	24 23	20 19	0
OPCODE	Rx	Ry	UNUSED	

Write the sequence for implementing the LW instruction with this addressing mode (you need to write the sequence for the execution macro state of the instruction). For each microstate, show the datapath action (in register transfer format such as A← Ry) along with the control signals you need to enable for the datapath action (such as DrPC).

Answer:

LW1: Ry → A, MAR
 Control Signals:
 RegSel=01; DrReg; LdA; LdMAR

LW2: MEM[Ry] → Rx
 Control Signals:
 DrMEM; RegSel=00; WrREG

LW3: A + 1 → Ry
 Control Signals:
 Func=11; DrALU; RegSel=01; WrREG

3.5.8 EXECUTE macro state: SW and ADDI instructions (part of I-Type)

Implementation of the SW macro state is similar to the LW macro state. Implementation of the ADDI macro state is similar to the ADD macro state with the only difference that the second operand comes from the immediate field of the IR as opposed to another

register. The development of the microstates for these two instructions is left as an exercise to the reader.

3.5.9 EXECUTE macro state: BEQ instruction (part of I-Type)

BEQ instruction has the following semantics:

If ($R_X == R_Y$) then $PC \leftarrow PC + 1 + \text{signed address-offset}$
else nothing

This instruction needs some special handling. The semantics of this instruction calls for comparing the contents of two registers (R_X and R_Y) and branching to a target address generated by adding the sign-extended offset to $PC+1$ (where PC is the address of the BEQ instruction) if the two values are equal.

In the datapath, there is hardware to detect if the value on the bus is a zero. The microstates for BEQ use this logic to set the Z register upon comparing the two registers.

Given below are the microstates, datapath actions, and control signals for the BEQ macro state:

- **beq1**

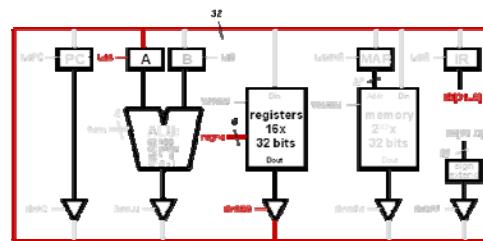
$R_X \rightarrow A$

Control signals needed:

RegSel = 00

DrREG

LdA



- **beq2**

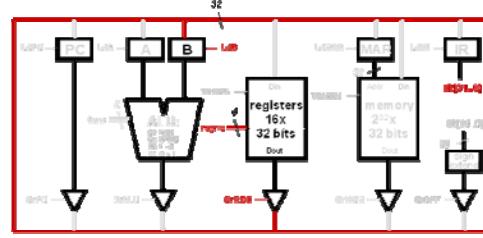
$R_Y \rightarrow B$

Control signals needed:

RegSel = 01

DrREG

LdB



- **beq3**

$A - B$

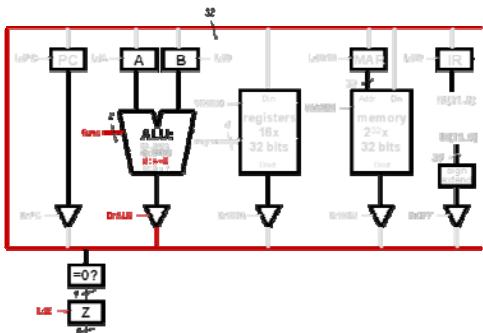
Load Z register with result of zero detect logic

Control signals needed:

func = 10

DrALU

LdZ



Note: The zero-detect combinational logic element in the datapath is always checking if the value on the bus is zero. By asserting LdZ, the Z register (1-bit register) is capturing the result of this detection for later use.

The actions following this microstate get tricky compared to the micro states for the other **instructions**. In the other instructions, we simply sequenced through all the microstates for that instruction and then return to the FETCH macro state. However, BEQ instruction causes a control flow change depending on the outcome of the comparison. **If the Z register is not set (i.e., Rx != Ry) then we simply return to ifetch1 to continue execution with the next sequential instruction (PC is already pointing to that instruction).** On the other hand, if Z is set then we want to continue with the microstates of BEQ to compute the target address of branch.

First let us go ahead and complete the microstates for BEQ assuming a branch has to be taken.

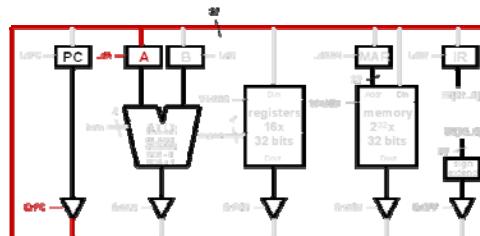
- **beq4**

PC → A

Control signals needed:

DrPC

LdA



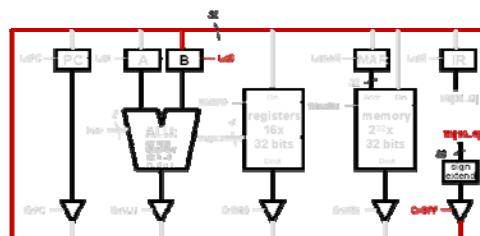
- **beq5**

Sign-extended offset → B

Control signals needed:

DrOFF

LdB



- **beq6**

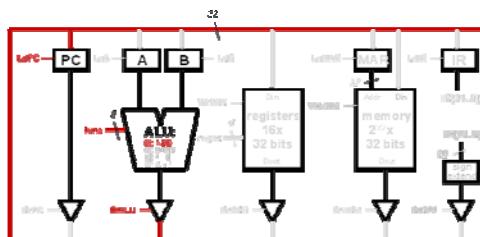
A+B → PC

Control signals needed:

func = 00

DrALU

LdPC



Note: In the FETCH macro state itself, PC has been incremented. Therefore, we simply add the sign-extended offset to PC to compute the target address.

3.5.10 Engineering a conditional branch in the microprogram

Figure 3.26 shows the desired state transitions in the BEQ macro state. The **next-state** field of the **beq3** microinstruction will contain **beq4**. With only one **next-state** field in the microinstruction, we need to engineer the transition from **beq3** to **ifetch1** or **beq4** depending on the state of the Z register. A time-efficient way of accomplishing this task is by using an additional location in the ROM to duplicate the microinstruction corresponding to **ifetch1**. We will explain how this is done.

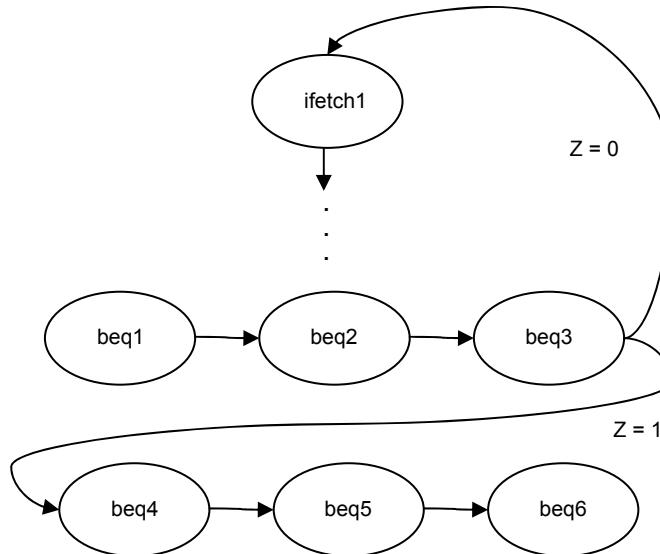


Figure 3.26: Desired state transitions in the BEQ macro state

Let us assume the state register has 5 bits, **beq4** has the binary encoding **01000**, and the next state field of the **beq3** microinstruction is set to **beq4**. We will prefix this encoding with the contents of the Z register to create a 6-bit address to the ROM. If the Z bit is 0 then the address presented to the ROM will be **001000** and if the Z bit is 1 then the address will be **101000**. The latter address (**101000**) is the one where we will store the microinstruction corresponding to the **beq4** microstate. In the location **001000** (let us call this **ifetch1-clone**) we will store the exact same microinstruction as in the original **ifetch1** location. Figure 3.27 shows this pictorially.

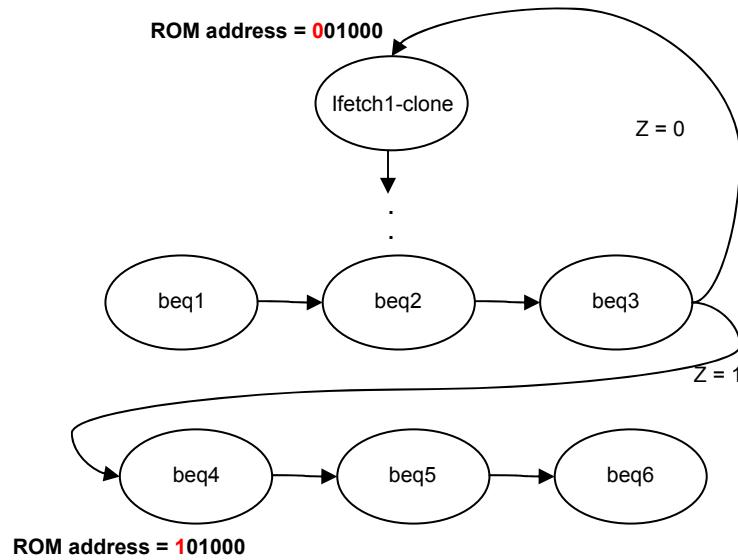


Figure 3.27: Engineering the conditional micro branch

We can extend this idea (of cloning a microinstruction) whenever we need to take a conditional branch in the micro program.

3.5.11 DECODE macro state revisited

Let us return to the DECODE macro state. Recall that this is a multi-way branch from **ifetch3** to the macro state corresponding to the specific instruction contained in IR. We adopt a trick similar to the one with the Z register for implementing the 2-way branch for the BEQ instruction.

Let us assume that 10000 is the encoding for the *generic* EXECUTE macro state.

The **next-state** field of **ifetch3** will have this generic value. We prefix this generic value with the contents of the OPCODE (IR bits 31-28) to generate a 9-bit address to the ROM. Thus, the ADD macro state will start at ROM address **000010000**, the NAND macro state will start at ROM address **000110000**, the ADDI at ROM address **001010000**, and so on.

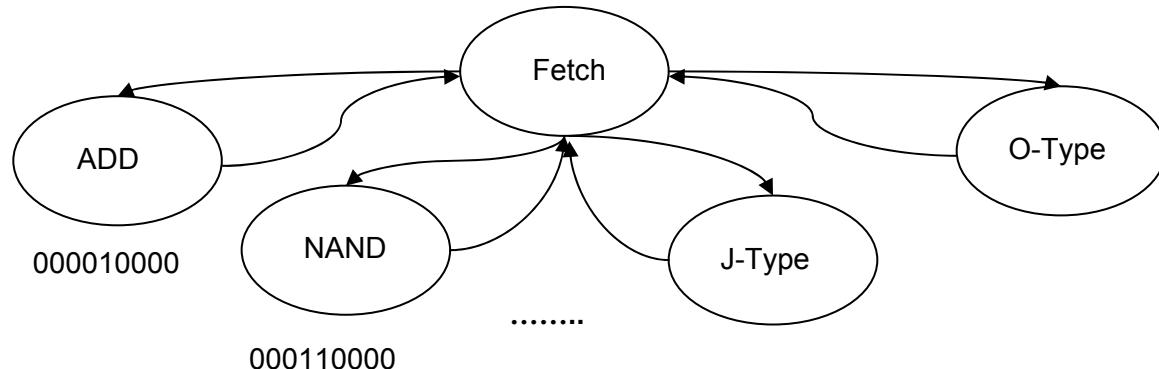


Figure 3.28: Effecting the multi-way branch of DECODE macro state

Putting all this together, the control logic of the processor has a 10-bit address (as shown in Figure 3.29):

- top 4 bits from IR 31-28
- next bit is the output of the Z register
- the bottom 5 bits are from the 5-bit state register

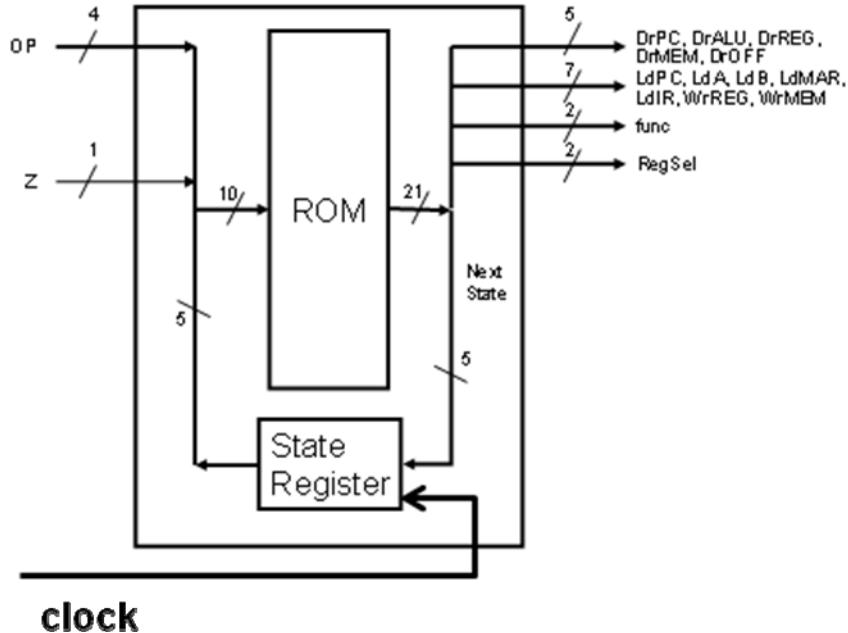


Figure 3.29: LC-2200 Control Unit

The lines coming in and out of the above control unit connect directly to the corresponding signal lines of the datapath in Figure 3.15.

3.6 Alternative Style of Control Unit Design

Let us consider different styles of implementing the control unit of a processor.

3.6.1 Microprogrammed Control

In Section 3.5, we presented the microprogrammed style of designing a control unit. This style has a certain elegance, simplicity, and ease of maintenance. The advantage we get with the micro programmed design is the fact that the control logic appears as a program all contained in a ROM and lends itself to better maintainability. There are two potential sources of inefficiency with microprogrammed design. The first relates to **time**. To generate the control signals in a particular clock cycle, an address has to be presented to the ROM and after a delay referred to as the **access time** of the ROM; the control signals are available for operating on the datapath. This time penalty is in the critical path of the clock cycle time and hence is a source of performance loss. However, the time penalty can be masked by employing prefetching of the next microinstruction while the current one is being executed. The second source of inefficiency relates to **space**. The design presented in the previous section represents one specific style of microprogramming called *horizontal microcode*, wherein there is a bit position in each microinstruction for every control signal needed in the entire datapath. From the sequences we developed so

far, it is clear that in most microinstructions most of the bits are 0 and only a few bits are 1 corresponding to the control signals that are needed for that clock cycle. For example, in **ifetch1** only **DrPC**, **LdMAR**, and **LdA** are 1. All the other bits in the microinstruction are 0. The space inefficiency of horizontal microcode could be overcome by a technique called *vertical microcode* that is akin to writing assembly language programming. Basically, the bit position in each microinstruction may represent different control signals depending on an opcode field in each vertical microinstruction. Vertical microcode is trickier since the control signals corresponding to a particular bit position in the microinstruction have to be mutually exclusive.

3.6.2 Hardwired control

It is instructive to look at what exactly the ROM represents in the horizontal microcode developed in Section 3.5. It really is a **truth table** for all the control signals needed in the datapath. The rows of the ROM represent the states and the columns represent the functions (one for each control signal). From previous exposure to logic design, the reader may know how to synthesize the minimal Boolean logic function for each column. Such logic functions are more efficient than a truth table.

We can implement the Boolean logic functions corresponding to the control signals using combinational logic circuits. The terms of this function are the conditions under which that control signal needs to be asserted.

For example, the Boolean function for **DrPC** will look as follows:

$$\text{DrPC} = \text{ifetch1} + \text{jalr1} + \text{beq4} + \dots$$

Using AND/OR gates or universal gates such as NAND/NOR all the control signals can be generated. We refer to this style of design as **hardwired control** since the control signals are implemented using combinational logic circuits (and hence hardwired, i.e, they are not easy to change). Such a design leads to both time (no access time penalty for ROM lookup) and space (no wasted space for signals that are NOT generated in a clock cycle) efficiency. There has been some criticism in the past that such a design leads to a maintenance nightmare since the Boolean functions are implemented using random logic.

However, the advent of **Programmable Logic Arrays (PLAs)** and **Field Programmable Gate Arrays (FPGAs)** largely nullifies this criticism. For example, PLAs give a structure to the random logic by organizing the required logic as a structured 2-dimensional array. We show this organization in Figure 3.30.

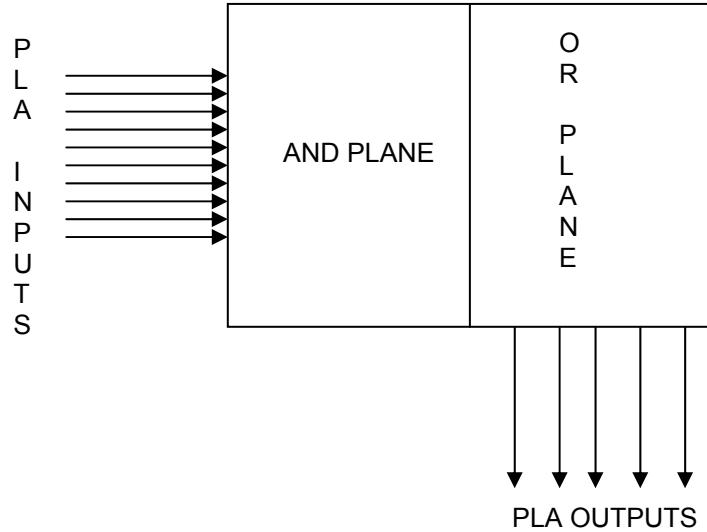


Figure 3.30: PLA

The outputs are the control signals needed to actuate the datapath (**DrPC**, **DrALU**, etc.). The inputs are the state the processor is in (**ifetch1**, **ifetch2**, etc.) and the conditions generated in the datapath (contents of the IR, Z, etc.). Every gate in the AND plane has ALL the inputs (true and complement versions). Similarly every gate in the OR plane has as inputs ALL the product terms generated by the AND plane. This is called a PLA because the logic can be “programmed” by selecting or not selecting the inputs to the AND and OR planes. Every PLA OUTPUT is a **SUM-of-PRODUCT** term. The PRODUCT terms are implemented in the AND plane. The SUM terms are implemented in the OR plane. This design concentrates all the control logic in one place and thus has the structural advantage of the micro programmed design. At the same time since the control signals are generated using combinational logic circuits, there is no penalty to this hardwired design. There is a slight space disadvantage compared to a true random logic design since every AND gate has a fan-in equal to ALL the PLA INPUTS and every OR gate has a fan-in equal to the number of AND gates in the AND plane. However, the structural advantage and the regularity of the PLA lend itself to compact VLSI design and far out-weigh any slight disadvantage.

More recently, FPGAs have become very popular as a way of quickly prototyping complex hardware designs. An FPGA is really a successor to the PLA and contains logic elements and storage elements. The connections among these elements can be programmed “in the field” and hence the name. This flexibility allows any design bugs to be corrected more easily even after deployment thus improving the maintainability of hardwired design.

3.6.3 Choosing between the two control design styles

The choice for control design between the two styles depends on a variety of factors.. We have given the pros and cons of both control styles. It would appear with the advent of FPGAs, much of the maintainability argument against hardwired control has

disappeared. Nevertheless, for basic implementation of processors (i.e., non-pipelined) as well as for the implementation of complex instructions (such as those found in the Intel x86 architecture), microprogrammed control is preferred due to its flexibility and amenability to quick changes. On the other hand, as we will see in the chapter on pipelined processor design, hardwired control is very much the preferred option for high-end pipelined processor implementation. Table 3.2 summarizes the pros and cons of the two design approaches.

Control Regime	Pros	Cons	Comment	When to use	Examples
Microprogrammed	Simplicity, maintainability, flexibility Rapid prototyping	Potential for space and time inefficiency	Space inefficiency may be mitigated with vertical microcode Time inefficiency may be mitigated with prefetching	For complex instructions, and for quick non-pipelined prototyping of architectures	PDP 11 series, IBM 360 and 370 series, Motorola 68000, complex instructions in Intel x86 architecture
Hardwired	Amenable for pipelined implementation Potential for higher performance	Potentially harder to change the design Longer design time	Maintainability can be increased with the use of structured hardware such as PLAs and FPGAs	For High performance pipelined implementation of architectures	Most modern processors including Intel Xeon series, IBM PowerPC, MIPS

Table 3.2: Comparison of Control Regimes

3.7 Summary

In this chapter, we got a feel for implementing a processor given an instruction set. The first step in the implementation is the choice of the datapath and the components that comprise the datapath. We reviewed the basic digital logic elements in Section 3.3 and the datapath design in Section 3.4. Once the datapath is fixed, we can turn our attention to the design of the control unit that drives the datapath to realize the instruction set. Assuming a microprogrammed control unit, Section 3.5 walked us through the microstates for implementing the instructions found in LC-2200 ISA. In Section 3.6, we reviewed the difference between hardwired control and microprogrammed control.

3.8 Historical Perspective

It is instructive to look back in time to see how the performance tradeoffs relate to economic and technological factors.

In the 1940's & 1950's, logic elements for constructing the hardware and memory were extremely expensive. Vacuum tubes and later discrete transistors served as the implementation technology. The instruction-set architecture was very simple, and typically featured a single register called an *accumulator*. Examples of machines from this period include EDSAC and IBM 701.

In the 1960's, we started seeing the first glimmer of "integration." IBM 1130 that was introduced in 1965 featured *Solid Logic Technology (SLT)* that was a precursor to integrated circuits. This decade saw a drop in hardware prices but memory was still implemented using magnetic cores (called *core memory*) and was a dominant cost of a computer system.

The trend continued in the 1970's with the initial introduction of SSI (Small Scale Integrated) and then MSI (Medium Scale Integrated) circuits as implementation technologies for the processor. Semiconductor memories started making their way in the 70's as a replacement for core memory. It is interesting to note that circa 1974 the price per bit for core memory and semiconductor memory were about equal (\$0.01 per bit). Semiconductor memory prices started dropping rapidly from then on, and the rest is history!

The 70's was an era for experimenting with a number of different architecture approaches (such as stack-oriented, memory-oriented, and register-oriented). Examples of machines from this era include IBM 360 and DEC PDP-11 for a hybrid memory- and register-oriented architecture; and Burroughs B-5000 for a stack-oriented approach. All of these are variants of a *stored program computer*, which is often referred to as *von Neumann* architecture, named after John von Neumann, a computer pioneer.

In parallel with the development of the stored program computer, computer scientists were experimenting with radically new architectural ideas. These include *dataflow* and *systolic* architectures. These architectures were aimed at giving the programmer control over performing several instructions in parallel, breaking the mold of sequential execution of instructions that is inherent in the stored program model. Both dataflow and systolic architectures focus on data rather than instructions. The dataflow approach allows all instructions whose input data is ready and available to execute and pass the results of their respective executions to other instructions that are waiting for these results. The systolic approach allows parallel streams of data to flow through an array of functional units pre-arranged to carry out a specific computation on the data streams (e.g., matrix multiplication). While the dataflow architecture is a realization of a general model of computation, systolic is an algorithm-specific model of synthesizing architectures. While these alternative styles of architectures did not replace the stored program computer, they had tremendous impact on computing as a whole all the way from algorithm design to processor implementation.

In the 1980's, there were several interesting developments. In the first place, high-speed LSI (Large Scale Integrated) circuits using bipolar transistors were becoming

commonplace. This was the implementation technology of choice for high-end processors such as IBM 370 and DEC VAX 780. In parallel with this trend, VLSI (Very Large Scale Integrated) circuits using CMOS transistors (also called Field Effect Transistors or FETs) were making their way as vehicles for single-chip microprocessors. By the end of the decade, and in the early 1990's these *killer micros* started posing a real threat to high-end machines in terms of price/performance. This decade also saw rapid advances in compiler technologies and the development of a true partnership between system software (as exemplified by compilers) and instruction-set design. This partnership paved the way for RISC (Reduced Instruction Set Computer) architectures. IBM 801, Berkeley RISC, and Stanford MIPS processors led the way in the RISC revolution. Interestingly, there was still a strong following for the CISC (Complex Instruction Set Computer) architecture as exemplified by Motorola 68000 and Intel x86 series of processors. The principle of pipelined processor design (see Chapter 5) which until now was reserved for high-end processors made its way into microprocessors as the level of integration allowed placing more and more transistors in one piece of silicon. Interestingly, the debate over RISC versus CISC petered out and the issue became one of striving to achieve an instruction throughput of one per clock cycle in a pipelined processor.

The decade of the 1990 has firmly established the Microchip (single chip microprocessors based on CMOS technology) as the implementation technology of choice in the computer industry. By the end of the decade, Intel x86 and Power PC instruction-sets (curiously, both are CISC style architectures that incorporated a number of the implementation techniques from the RISC style architectures) became the industry standard for making "boxes," be they desktops, servers, or supercomputers. It is important to note that one of the most promising architectures of this decade was DEC Alpha. Unfortunately, due to the demise of DEC in the late 90's, the Alpha architecture also rests in peace!

With the advent of personal communication devices (cell phones, pagers, and PDAs) and gaming devices, embedded computing platforms have been growing in importance from the 80's. Interestingly, a significant number of embedded platforms use descendants of a RISC architecture called ARM (Acorn RISC Machine, originally designed by Acorn Computers). Intel makes XScale processors that are derived from the original ARM architecture. A point to note is that ARM processors were originally designed with PCs and workstations in mind.

Superscalar and *VLIW* (Very Large Instruction Word architectures) processors represent technologies that grew out of the RISC revolution. Both are attempts to increase the throughput of the processor. They are usually referred to as *multiple issue* processors. Superscalar processor relies on the hardware to execute a fixed number of mutually independent adjacent instructions in parallel. In the VLIW approach, as the name suggests, an instruction actually contains a number of operations that are strung together in a single instruction. VLIW relies heavily on compiler technology to reduce the hardware design complexity and exploit parallelism. First introduced in the late 80's, the VLIW technology has gone through significant refinement over the years. The most

recent foray of VLIW architecture is the IA-64 offering from Intel targeting the supercomputing marketplace. VLIW architectures are also popular in the high-end embedded computing space as exemplified by DSP (Digital Signal Processing) applications.

We are reaching the end of the first decade of the new millennium. There is little debate over instruction-sets these days. Most of the action is at the level of micro-architecture, namely, how to improve the performance of the processor by various hardware techniques. Simultaneously, the level of integration has increased to allow placing multiple processors on a single piece of silicon. Dubbed *multicore*, such chips have started appearing now in most computer systems that we purchase today.

3.9 Review Questions

1. What is the difference between level logic and edge-triggered logic? Which do we use in implementing an ISA? Why?
2. Given the FSM and state transition diagram for a garage door opener (Figure 3.12-a and Table 3.1) implement the sequential logic circuit for the garage door opener (Hint: The sequential logic circuit has 2 states and produces three outputs, namely, next state, up motor control and down motor control).
3. Re-implement the above logic circuit using the ROM plus state register approach detailed in this chapter.
4. Compare and contrast the various approaches to control logic design.
5. One of the optimizations to reduce the space requirement of the control ROM based design is to club together independent control signals and represent them using an encoded field in the control ROM. What are the pros and cons of this approach? What control signals can be clubbed together and what cannot be? Justify your answer.
6. What are the advantages and disadvantages of a bus-based datapath design?
7. Consider a three-bus design. How would you use it for organizing the above datapath elements? How does this help compared to the two-bus design?
8. Explain why internal registers such as the instruction register (IR), and memory address register (MAR) may not be usable for temporary storage of values in implementing the ISA by the control unit.
9. An engineer would like to reduce the number of microstates for implementing the FETCH macrostate down to 2. How would she able to accomplish that goal?

10. How many words of memory will this code snippet require when assembled? Is space allocated for “L1”?

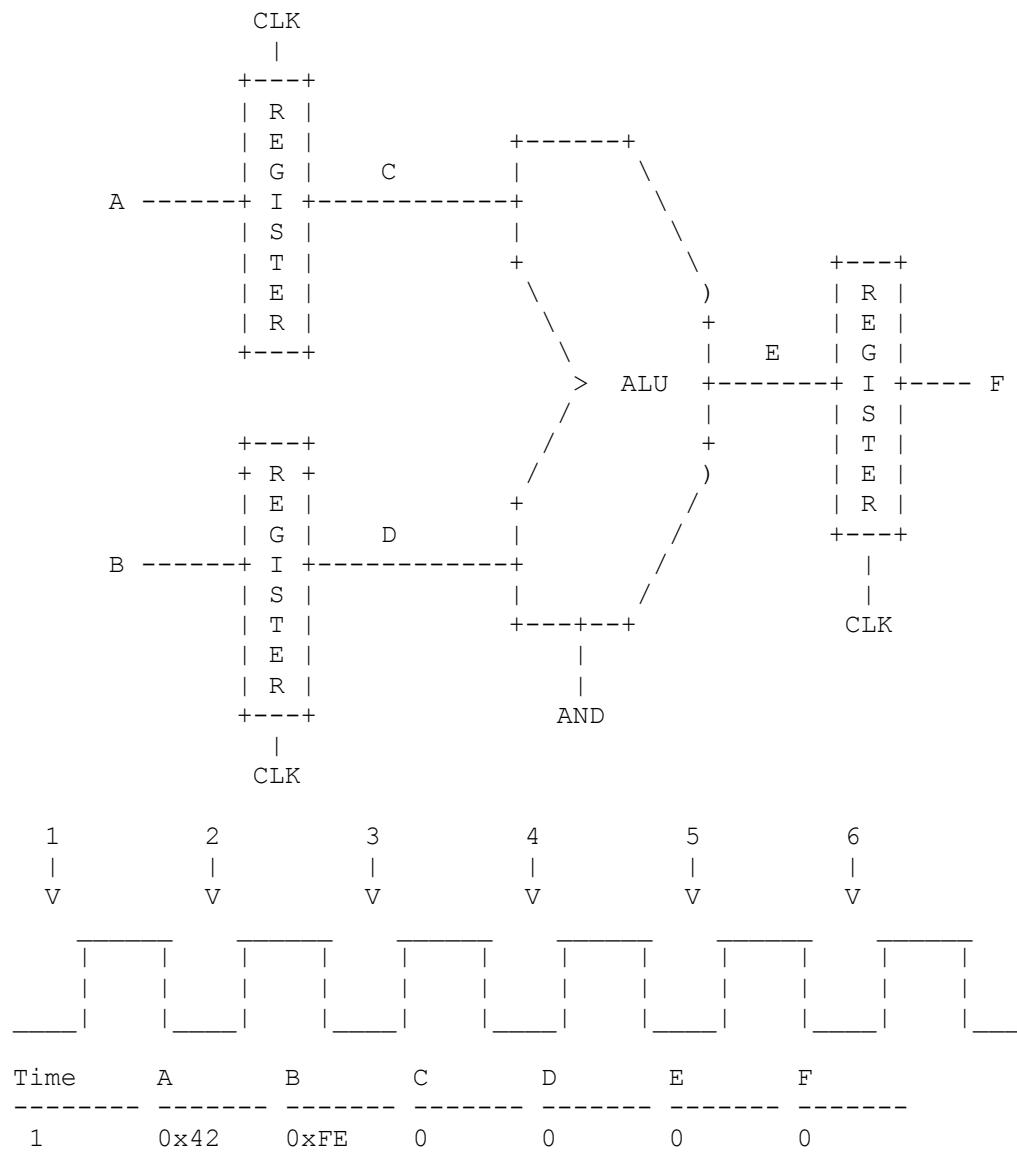
```

beq $s3, $s4, L1
add $s0, $s1, $s2
L1: sub $s0, $s0, $s3
    
```

11. What is the advantage of fixed length instructions?

12. What is a leaf procedure?

13. Assume that for the portion of the datapath shown below that all the lines are 16 bits wide. Fill in the table below.



2	0	0	_____	_____	_____	_____
3	0xCAFE	0x1	_____	_____	_____	_____
4	0	0	_____	_____	_____	_____
5	0	0	_____	_____	_____	_____
6	0	0	_____	_____	_____	_____

14. In the LC-2200 processor, why is there not a register after the ALU?
15. Extend the LC-2200 ISA to include a subtract instruction. Show the actions that must be taken in microstates of the subtract instruction assuming the datapath shown in Figure 3.15.
16. In the datapath diagram shown in Figure 3.15, why do we need the A and B registers in front of the ALU? Why do we need MAR? Under what conditions would you be able to do without with any of these registers? [Hint: Think of additional ports in the register file and/or buses.]
17. Core memory used to cost \$0.01 per bit. Consider your own computer. What would be a rough estimate of the cost if memory cost is \$0.01/bit? If memory were still at that price what would be the effect on the computer industry?
18. If computer designers focused entirely on speed and ignored cost implications, what would the computer industry look like today? Who would the customers be? Now consider the same question reversed: If the only consideration was cost what would the industry be like?
19. (Design Question)
 Consider a CPU with a stack-based instruction set. Operands and results for arithmetic instructions are stored on the stack; the architecture contains no general-purpose registers.

The data path shown on the next page uses two separate memories, a 65,536 (2^{16}) byte memory to hold instructions and (non-stack) data, and a 256 byte memory to hold the stack. The stack is implemented with a conventional memory and a stack pointer register. The stack starts at address 0, and grows upward (to higher addresses) as data are pushed onto the stack. The stack pointer points to the element on top of the stack (or is -1 if the stack is empty). You may ignore issues such as stack overflow and underflow.

Memory addresses referring to locations in program/data memory are 16 bits. All data are 8 bits. Assume the program/data memory is byte addressable, i.e., each address refers to an 8-bit byte. Each instruction includes an 8-bit opcode. Many instructions

also include a 16-bit address field. The instruction set is shown below. Below, "memory" refers to the program/data memory (as opposed to the stack memory).

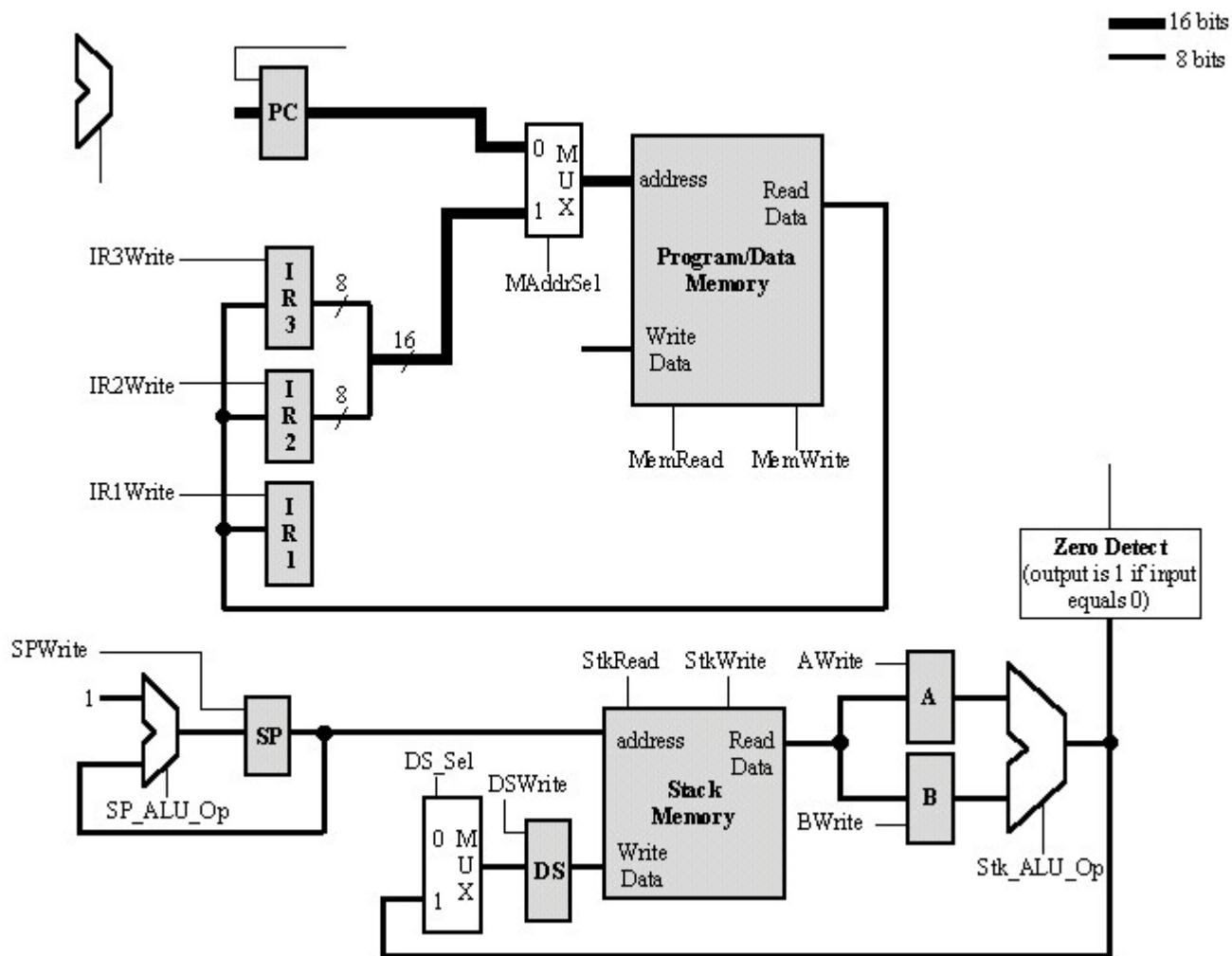
OPCODE	INSTRUCTION	OPERATION
00000000	PUSH <addr>	push the contents of memory at address <addr> onto the stack
00000001	POP <addr>	pop the element on top of the stack into memory at location <addr>
00000010	ADD	Pop the top two elements from the stack, add them, and push the result onto the stack
00000100	BEQ <addr>	Pop top two elements from stack; if they're equal, branch to memory location <addr>

Note that the ADD instruction is only 8 bits, but the others are 24 bits. Instructions are packed into successive byte locations of memory (i.e., do NOT assume all instructions use 24 bits).

Assume memory is 8 bits wide, i.e., each read or write operation to main memory accesses 8 bits of instruction or data. This means the instruction fetch for multi-byte instructions requires multiple memory accesses.

Datapath

Complete the partial design shown on the next page.



Assume reading or writing the program/data memory or the stack memory requires a single clock cycle to complete (actually, slightly less to allow time to read/write registers). Similarly, assume each ALU requires slightly less than one clock cycle to complete an arithmetic operation, and the zero detection circuit requires negligible time.

Control Unit

Show a state diagram for the control unit indicating the control signals that must be asserted in each state of the state diagram.

Chapter 4 Interrupts, Traps and Exceptions

(Revision number 16)

In the previous chapter, we discussed the implementation of the processor. In this chapter, we will discuss how the processor can handle discontinuities in program execution. In a sense, branch instructions and subroutine calls are discontinuities as well. However, the programmer consciously introduced such discontinuities that are part of the program. The discontinuities we will look into in this chapter are those that are potentially unplanned for and often may not even be part of the program that experiences it.

Let us look at a simple analogy, a *classroom*. The professor is giving a lecture on computer architecture. To encourage class participation, he wants students to ask questions. He could do one of two things: (1) periodically, he could stop lecturing and poll the students to see if they have any questions; or (2) he could tell the students whenever they have a question, they should put up their hands to signal that they have a question. Clearly, the first approach takes away the spontaneity of the students. By the time the professor gets around to polling the students, they may have forgotten that they had a question. Worse yet, they had so many questions one after another that they are now completely lost! The second approach will ensure that the students remain engaged and they do not lose their train of thought. However, there is a slight problem. When should the professor take a student's question? He could take it immediately as soon as someone puts up a hand, but he may be in mid-sentence. Therefore, he should finish his train of thought and then take the question. He has to be careful to remember where he was in his lecture so that after answering the student's question he can return to where he left off in the lecture. What if another student asks a question while the professor is in the middle of answering the first student's question? That will soon spiral out of control, so as a first order principle the professor should not take another question until he is finished with answering the first one. So two things to take away from the classroom analogy: remember where to return in the lecture, and disable further questions.



The processor that we designed executes instructions but unless it can talk to the outside world for I/O it is pretty much useless. Building on the above analogy, we can have the processor poll an input device (such as a keyboard) periodically. On the one hand, polling is error prone since the device could be generating data faster than the polling rate. On the other hand, it is extremely wasteful for the processor to be doing the polling if the device has no data to deliver. Therefore, we can apply the classroom analogy and let the device *interrupt* the processor to let it know that it has something to say to the processor. As in the classroom analogy, the processor should remember where it is in its current program execution and disable further interruptions until it services the current one.

4.1 Discontinuities in program execution

In Chapter 3, we defined the terms synchronous and asynchronous in the context of logic circuit behavior. Consider a real life example. Let's say you walk to the fridge and pick up a soda. That's a *synchronous* event. It is part of your intended activities. On the other hand, while you are working on your homework in your room, your roommate comes in and gives you a soda. That's an *asynchronous* event, since it was not part of your intended activities. Making a phone call is a synchronous event; receiving one is an asynchronous event.

We can generalize the definition of synchronous and asynchronous events observed in a system, be it in hardware or software. A *synchronous* event is one, which occurs (if it does at all) at well-defined points of time aligned with the intended activities of the system. The state changes from one microstate to the next in the sequences we discussed in Chapter 3 are all examples of synchronous events in a hardware system. Similarly, opening a file in your program is a synchronous software event.

An *asynchronous* event is one, which occurs (if it does at all) unexpectedly with respect to other ongoing activities in the system. As we will see shortly, interrupts are asynchronous hardware events. An e-mail arrival notification while you are in the middle of your assignment is an asynchronous software event.

A system may compose synchronous and asynchronous events. For example, the hardware may use polling (which is synchronous) to detect an event and then generate an asynchronous software event.

Now we are ready to discuss discontinuities in program execution that come in three forms: *interrupts*, *exceptions*, and *traps*.

1. Interrupts

An *interrupt* is the mechanism by which devices catch the attention of the processor. This is an unplanned discontinuity for the currently executing program and is asynchronous with the processor execution. Furthermore, the device I/O may be intended for an altogether different program from the current one. For the purposes of clarity of discourse, we will consider only discontinuities caused by external devices as interrupts.

2. Exceptions

Programs may unintentionally perform certain illegal operations (for example *divide by zero*) or follow an execution path unintended in the program specification. In such cases, once again it becomes necessary to discontinue the original sequence of instruction execution of the program and deal with this kind of unplanned discontinuity, namely, *exception*. Exceptions are **internally generated conditions** and are **synchronous with the processor execution**. They are usually unintended by the current program and are the result of some erroneous condition encountered during execution. However, programming languages such as Java define an exception mechanism to allow error propagation through layers of software. In this case, the program *intentionally* generates an exception to signal some unexpected program behavior. In either case, we define exception to be some condition (intentional or unintentional) that deviates from normal program execution.

3. Traps

Programs often make *system calls* to **read/write files** or for other such services from the system. Such **system calls** are like **procedure calls** but they need some **special handling** since, the user program will be accessing parts of the system whose integrity affects a whole community of users and not just this program. Further, the user program may not know where in memory the procedure corresponding to this service exists and may have to discern that information at the point of call. Trap, as the name suggests, **allows the program to fall into the operating system**, which will then decide **what the user program wants**. Another term often used in computer literature for program generated traps is **software interrupts**. For the purposes of our discussion, we consider software interrupts to be the same as traps. Similar to exceptions, traps are **internally generated conditions and are synchronous with the processor execution**. Some traps could be **intentional**, for example, as manifestation of a program making a system call. Some traps could be unintentional as far as the program is concerned. We will see examples of such unintentional traps in later chapters when we discuss memory systems.

While the understanding of the term “interrupt” is fairly standardized, the same cannot be said about the other two terms in the literature. In this book, we have chosen to adopt a particular definition of these three terms, which we use consistently, but acknowledge that the definition and use of these terms may differ in other books. Specifically, in our definition, a “trap” is an internal condition that the currently running program has no way of dealing with on its own, and if anything, the system has to handle it. On the other hand, it is the currently running program’s responsibility to handle an “exception”.

Table 4.1 summarizes the characteristics of these three types of program discontinuities. The second column characterizes whether it is **synchronous or asynchronous**; the third column identifies the source of the discontinuity as **internal or external** to the currently running program; the fourth column identifies if the discontinuity was **intentional irrespective of the source**; and the last column gives examples of each type of discontinuity.

Type	Sync/Async	Source	Intentional?	Examples
Exception	Sync	Internal	Yes and No	Overflow, Divide by zero, Illegal memory address, Java exception mechanism
Trap	Sync	Internal	Yes and No	System call, Software interrupts, Page fault, Emulated instructions
Interrupt	Async	External	Yes	I/O device completion

Table 4.1: Program Discontinuities

4.2 Dealing with program discontinuities

As it turns out, program discontinuity is a powerful tool. First, it allows the computer system to provide input/output capabilities; second, it allows the computer system to perform resource management among competing activities; third, it allows the computer system to aid the programmer with developing correct programs. Interrupts, traps, and exceptions, serve the three functions, respectively.

Dealing with program discontinuities is a partnership between the processor architecture and the operating system. Let us understand the division of labor. Detecting program discontinuity is the processor's responsibility. Redirecting the processor to execute code to deal with the discontinuity is the operating system's responsibility. As we will see throughout this book, such a partnership exists between the hardware and the system software for handling each subsystem.

Let us now figure out what needs to happen to deal with program discontinuities, specifically the actions that need to happen implicitly in hardware, and those that need to happen explicitly in the operating system.

It turns out that most of what the processor has to do to deal with program discontinuity is the same regardless of the type. Recall that a processor is simply capable of executing instructions. To deal with any of these program discontinuities, the processor has to start

executing a different set of instructions than the ones it is currently executing. A *Handler* is the procedure that is executed when a discontinuity occurs. The code for the handler is very much like any other procedure that you may write. In this sense, such discontinuities are very much like a **procedure call** (see Figure 4.1). However, it is an unplanned procedure call. Moreover, the control may or may not return to the interrupted program (depending on the nature of the discontinuity). Yet, it is necessary to observe all the formalities (the procedure calling convention) for this unplanned procedure call and subsequent resumption of normal program execution. Most of what needs to happen is straightforward, similar to normal procedure call/return.

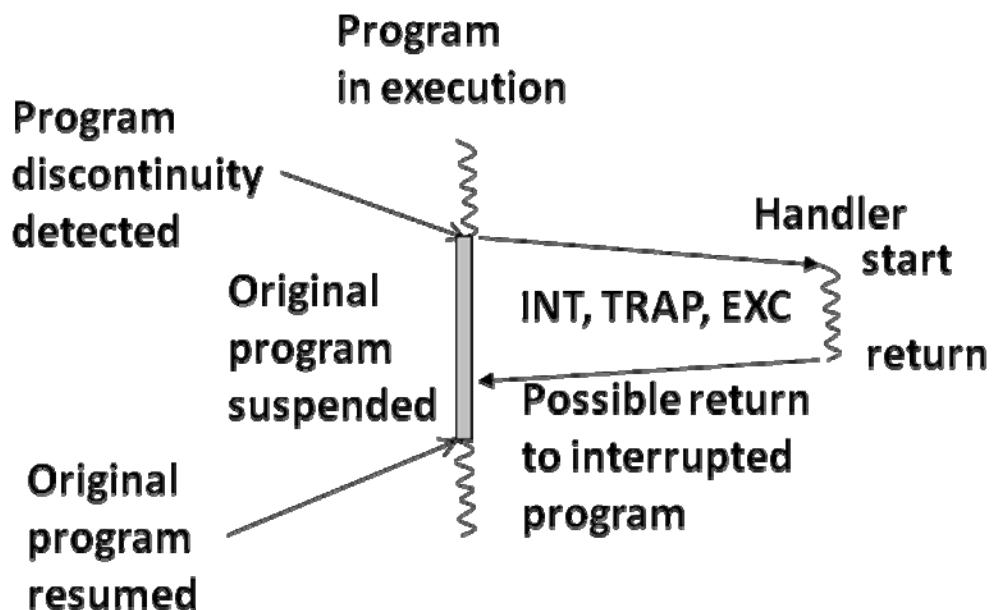


Figure 4.1: Program Discontinuity

Four things are tricky about these discontinuities.

1. They can happen anywhere during the instruction execution. That is the discontinuity can happen in the middle of an instruction execution.
2. The discontinuity is unplanned for and quite possibly completely unrelated to the current program in execution. Therefore, the hardware has to save the program counter value implicitly before the control goes to the handler.
3. At the point of detecting the program discontinuity, the hardware has to determine the address of the handler to transfer control from the currently executing program to the handler.
4. Since the hardware saved the PC implicitly, the handler has to discover how to resume normal program execution.

What makes it possible to address all of the above issues is the partnership between the operating system and the processor architecture. The basis for this partnership is a **data structure** maintained by the operating system somewhere in memory that is known to the processor. This data structure is a **fixed-size table of handler addresses**, one for each type

of anticipated program discontinuity (see Figure 4.2). The size of the table is architecture dependent. Historically, the name given to this data structure is *interrupt vector table (IVT)*¹. Each discontinuity is given a *unique* number often referred to as a *vector*. This number serves as a unique index into the IVT. The operating system sets up this table at boot time. This is the explicit part of getting ready for dealing with program discontinuities. Once set up, the processor uses this table during normal program execution to look up a specific handler address upon detecting a discontinuity.

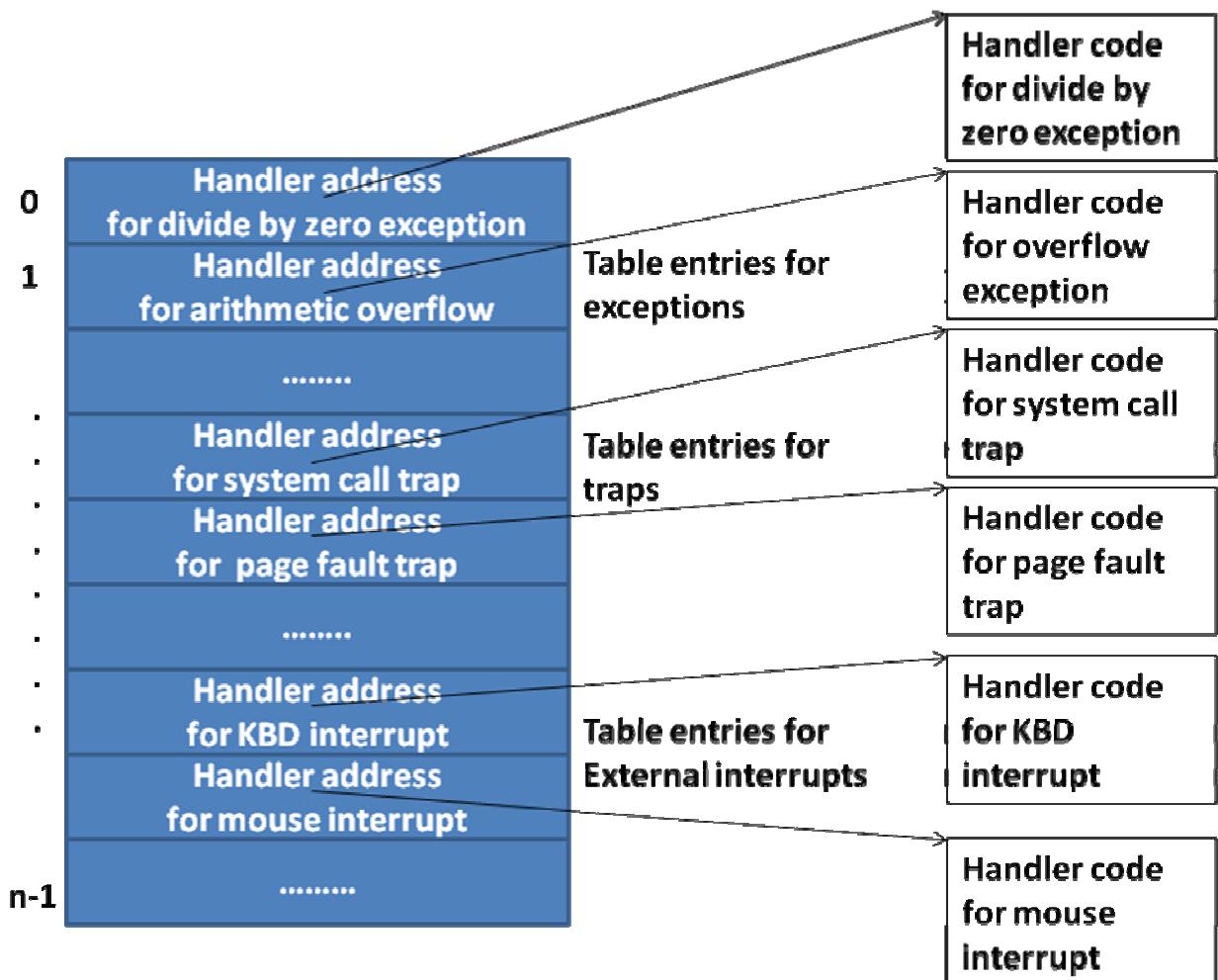


Figure 4.2: Interrupt Vector Table (IVT) – OS sets up this table at boot time

In the case of traps and exceptions, the hardware generates this vector internally. We introduce an *exception/trap register (ETR)*, internal to the processor, for storing this vector (Figure 4.3). Upon an exception or a trap, the unique number associated with that exception or trap will be placed in ETR. For example, upon detecting a “divide by zero” exception, the *FSM for divide instruction will place the vector corresponding to this exception in ETR*. Similarly, a *system call may manifest as a “trap instruction”*

¹ Different vendors give this data structure a different name. Intel calls this data structure Interrupt Descriptor Table (IDT), with 256 entries.

supported by the processor architecture. In this case, the FSM for the trap instruction will place the vector corresponding to the specific system call in ETR.



Figure 4.3: Exception/Trap Register— Number set by the processor upon detecting an exception/trap; used by the processor to index into the IVT to get the handler address

To summarize, the essence of the partnership between the operating system and the hardware for dealing with program discontinuities is as follows:

1. The architecture may itself define a set of exceptions and specify the numbers (vector values) associated with them. These are usually due to runtime errors encountered during program execution (such as arithmetic overflow and divide by zero).
2. The operating system may define its own set of exceptions (software interrupts) and traps (system calls) and specify the numbers (vector values) associated with them.
3. The operating system sets up the IVT at boot time with the addresses of the handlers for dealing with different kinds of exceptions, traps, and interrupts.
4. During the normal course of execution, the hardware detects exceptions/traps and stashes the corresponding vector values in ETR.
5. During normal course of execution, the hardware detects external interrupts and receives the vector value corresponding to the interrupting device.
6. The hardware uses the vector value as an index into the IVT to retrieve the handler address to transfer control from the currently executing program.

In the case of external interrupt, the processor has to do additional work to determine vector corresponding to the device that is interrupting to dispatch the appropriate device handling program. Section 4.3 discusses the enhancements to the processor architecture and instruction-set to deal with program discontinuities. Section 4.4 deals with hardware design considerations for dealing with program discontinuities.

4.3 Architectural enhancements to handle program discontinuities

Let us first understand the architectural enhancements needed to take care of these program discontinuities. Since the processor mechanism is the same regardless of the type of discontinuity, we will henceforth simply refer to these discontinuities as interrupts.

1. When should the processor entertain an interrupt? This is analogous to the classroom example. We need to leave the processor in a clean state before going to the handler. Even if the interrupt happened in the middle of an instruction execution, the processor should wait until the instruction execution is complete before checking for an interrupt.

2. How does the processor know there is an interrupt? We can add a *hardware line* on the datapath bus of the processor. At the end of each instruction execution, the processor samples this line to see if there is an interrupt pending.
3. How do we save the return address? How do we manufacture the handler address? Every instruction execution FSM enters a *special* macro state, INT, at the end of instruction execution if there is an interrupt pending.
4. How do we handle multiple cascaded interrupts? We will discuss the answer to this question in Section 4.3.3.
5. How do we return from the interrupt? We will present ideas for this question in Section 4.3.4.

4.3.1 Modifications to FSM

In Chapter 3, the basic FSM for implementing a processor consisted of three macro states Fetch, Decode, and Execute, as shown below.

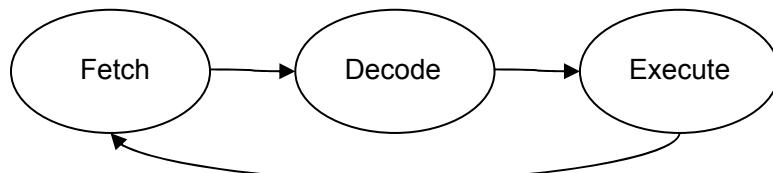


Figure 4.4-(a): Basic FSM of a processor

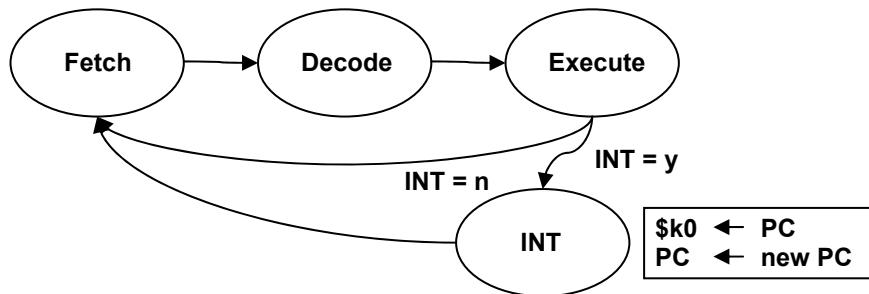


Figure 4.4-(b): Modified FSM for handling interrupts

Figure 4.4-(b) shows the modified FSM that includes a new macro state for handling interrupts. As we mentioned in Section 4.2, an interrupt may have been generated anytime during the execution of the current instruction. The FSM now checks at the point of completion of an instruction if there is a pending interrupt. If there is ($\text{INT}=y$), then the FSM transitions to the INT macro state; if there is no interrupt pending ($\text{INT}=n$) then next instruction execution resumes by returning to the Fetch macro state. A possibility is to check for interrupts after each macro state. This is analogous to the professor completing his/her thought before recognizing a student with a question in the classroom example. However, checking for interrupts after each macro state is problematic. In Chapter 3, we saw that the datapath of a processor includes several internal registers that are not visible at the level of the instruction set architecture of the processor. We know that once an instruction execution is complete, the values in such

internal registers are no longer relevant. Thus, deferring the check for interrupt until the completion of the current instruction leaves the processor in a clean state. To resume the interrupted program execution after servicing the interrupt, the two things that are needed are the state of the program visible registers and the point of program resumption.

Example 1:

Consider the following program:

```
100    ADD  
101    NAND  
102    LW  
103    NAND  
104    BEQ
```

An interrupt occurs when the processor is executing the ADD instruction. What is the PC value that needs to be preserved to resume this program after the interrupt?

Answer:

Even though the interrupt occurs during the execution of the ADD instruction, the interrupt will be taken only after the instruction execution is complete. Therefore, the PC value to be preserved to resume this program after the interrupt is 101.

Let us now discuss what needs to happen in the INT macro state. To make the discussion concrete, we will make enhancements to the LC-2200 processor for handling interrupts.

1. We have to save the current PC value somewhere. We reserve one of the processor registers \$k0 (general purpose register number 12 in the register file) for this purpose. The INT macro state will save PC into \$k0.
2. We receive the PC value of the handler address from the device, load it into the PC and go to the FETCH macro state. We will elaborate on the details of accomplishing this step shortly.

4.3.2 A simple interrupt handler

Figure 4.5 shows a simple interrupt handler. The save/restore of processor registers is exactly similar to the procedure calling convention discussed in Chapter 2.

```
Handler:  
  save processor registers;  
  execute device code;  
  restore processor registers;  
  return to original program;
```

Figure 4.5: A Simple Interrupt Handler

Example 2:

Consider the following program:

```
100    ADD  
101    NAND  
102    LW
```

103 NAND
104 BEQ

An interrupt occurs when the processor is executing the ADD instruction. At this time, the only registers in use by the program are R2, R3, and R4. What registers are saved and restored by the interrupt handler?

Answer:

Unfortunately, since an interrupt can happen at any time, the interrupt handler has no way of knowing which registers are currently in use by the program. Therefore, it saves and restores **ALL** the program visible registers though this program only needs R2, R3, and R4 to be saved and restored.

4.3.3 Handling cascaded interrupts

There is a problem with the simple handler code in Figure 4.5. If there is another interrupt while servicing the current one then we will lose the PC value of the original program (currently in \$k0). This essentially would make it impossible to return to the original program that incurred the first interrupt. Figure 4.6 depicts this situation.

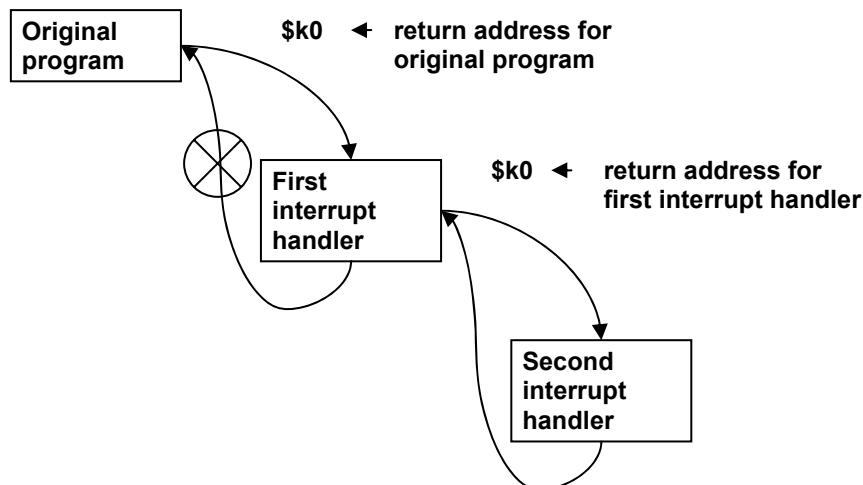


Figure 4.6: Cascaded interrupts

By the time we get to the second interrupt handler, we have lost the return address to the original program. This situation is analogous to the classroom example when the professor has to handle a second question before completing the answer to the first one. In that analogy, we simply took the approach of **disallowing a second questioner before the answer to the first questioner was complete**. May be the second question needs to be answered right away to help everyone understand the answer to the original question. Not being able to entertain multiple interrupts is just not a viable situation in a computer system. **Devices are heterogeneous in their speed. For example, the data rate of a disk is much higher than that of a keyboard or a mouse.** Therefore, we cannot always afford to turn off interrupts while servicing one. At the same time, there has to be a window

devoid of interrupts available to any handler wherein it can take the necessary actions to avoid the situation shown in Figure 4.6.

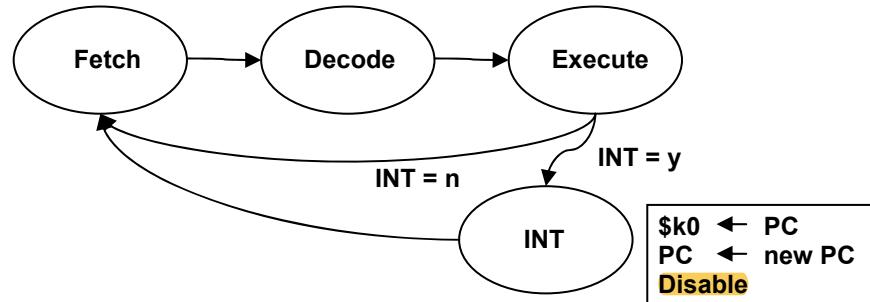


Figure 4.7: Modified FSM with disable interrupts added

Therefore, two things become clear in terms of handling cascaded interrupts.

1. A new instruction to turn off interrupts: **disable interrupts**
2. A new instruction to turn on interrupts: **enable interrupts**

Further, the hardware should implicitly turn off interrupts while in the INT state and hand over control to the handler. Figure 4.7 shows the modified FSM with the disable interrupt added to the INT macro state.

Let us now investigate what the handler should do to avoid the situation shown in Figure 4.6. It should save the return address to the original program contained in \$k0 while the interrupts are still disabled. Once it does that, it can then enable interrupts to ensure that the processor does not miss interrupts that are more important. **Before leaving the handler, it should restore \$k0, with interrupts disabled.** Figure 4.8 shows this modified interrupt handler. As in the procedure calling convention discussed in Chapter 2, saving/restoring registers is on the stack.

```

Handler:
/* The interrupts are disabled when we enter */
save $k0;
enable interrupts;
save processor registers;
execute device code;
restore processor registers;
disable interrupts;
restore $k0;
return to original program;
  
```

Figure 4.8: Modified Interrupt Handler

Example 3:

Consider the following program:

```

100  ADD
101  NAND
102  LW
103  NAND
104  BEQ
  
```

An interrupt occurs when the processor is executing the ADD instruction. The handler code for handling the interrupt is as follows:

```
1000 Save $k0
1001 Enable interrupts
1002 /* next several instructions save processor registers */
.....
1020 /* next several instructions execute device code */
.....
1102 /* next several instructions restore processor registers */
.....
1120 restore $k0
1121 return to original program
```

Assume that a second interrupt occurs at the instruction “restore \$k0” (PC = 1120). When will the original program be resumed?

Answer:

The original program **will never be resumed**. Note that the second interrupt will be taken immediately on completion of “restore \$k0”. Upon completion of this instruction, the handler \$k0 = 101, which is the point of resumption of the original program. Now the second interrupt is taken. Unfortunately, the second interrupt (see Figure 4.7) will store the point of resumption of the first handler (memory address = 1121) into \$k0. Thus the point of resumption of the original program (memory address = 101) is lost forever. The reason is that the interrupt handler in this example does not have the crucial “disable interrupts” instruction in Figure 4.8.

It should be noted, however, that it might not always be prudent to entertain a second interrupt while servicing the current one. For example, in Section 4.4.1, we will introduce the notion of multiple interrupt levels. **Based on their relative speeds, devices will be placed on different interrupt priority levels.** For example, **a high-speed device such as disk will be placed on a higher priority level compared to a low-speed device such as a keyboard.** **When the processor is serving an interrupt from the disk, it may temporarily ignore an interrupt coming from a keyboard.**

The role of the hardware is to provide the necessary mechanisms for the processor to handle cascaded interrupts correctly. It is a partnership between the handler code (which is part of the **operating system**) and the processor hardware to determine how best to handle multiple simultaneous interrupts depending on what the processor is doing at that point of time.

Basically, the choice is two-fold:

- ignore the interrupt for a while (if the **operating system** is currently handling a higher priority interrupt), or
- attend to the interrupt immediately as described in this sub-section.

Ignoring an interrupt temporarily may be implicit using the hardware priority levels or explicit via the “disable interrupt” instruction that is available to the handler program.

4.3.4 Returning from the handler

Once the handler completes execution, it can return to the original program by using the PC value stored in \$k0. At first glance, it appears that we should be able to use the mechanism used for returning from a procedure call to return from the interrupt as well. For example, in Chapter 2, to return from a procedure call we introduced the instruction²,

J rlink

Naturally, we are tempted to use the same instruction to return from the interrupt,

J \$k0

However, there is a problem. Recall that the interrupts should be in the enabled state when we return to the original program. Therefore, we may consider the following sequence of instructions to return from the interrupt:

Enable interrupts;

J \$k0;

There is a problem with using this sequence of instructions as well to return from an interrupt. Recall that we check for interrupts at the end of each instruction execution. Therefore, between “Enable Interrupts” and “J \$k0”, we may get a new interrupt that will trash \$k0.

Therefore, we introduce a new instruction

Return from interrupt (RETI)

The semantics of this instruction are as follows:

Load PC from \$k0;

Enable interrupts;

The important point to note is that this instruction is *atomic*, that is, this instruction executes fully before any new interrupts can occur. With this new instruction, Figure 4.9 shows the correct interrupt handler that can handle nested interrupts.

```
Handler:  
/* The interrupts are disabled when we enter */  
save $k0;  
enable interrupts;  
save processor registers;  
execute device code;  
restore processor registers;  
disable interrupts;  
restore $k0;  
return from interrupt;  
/* interrupts will be enabled by return from interrupt */
```

Figure 4.9: Complete Interrupt Handler

² Recall that LC-2200 does not have a separate unconditional jump instruction. We can simulate it with the JALR instruction available in LC-2200.

4.3.5 Checkpoint

To summarize, we have made the following architectural enhancements to LC-2200 to handle interrupts:

1. Three new instructions to LC-2200:
Enable interrupts
Disable interrupts
Return from interrupt
2. Upon an interrupt, store the current PC implicitly into a special register \$k0.

We now turn to the discussion of the hardware needed to accommodate these architectural enhancements. We already detailed the changes to the FSM at the macro level. We will leave working out the details of the INT macro state in the context of the LC-2200 datapath as an exercise to the reader. To complete this exercise, we will need to figure out the microstates needed to carry out all the actions needed in the INT macro state, and then generate the control signals needed in each microstate.

4.4 Hardware details for handling program discontinuities

As we mentioned earlier in Section 4.2, the architectural enhancements discussed thus far are neutral to the type of discontinuity, namely, exception, trap, or interrupt. In this section, we discuss the hardware details for dealing with program discontinuities in general, and handling external interrupts in particular. We already introduced the interrupt vector table (IVT) and exception/trap register (ETR). We investigate the datapath modifications for interrupts as well as the ability to receive the interrupt vector from an external device. We intentionally keep this discussion simple. The interrupt architecture in modern processors is very sophisticated and we review this briefly in the chapter summary (Section 4.6).

4.4.1 Datapath details for interrupts

We will now discuss the implementation details for handling interrupts. In Chapter 3, we introduced and discussed the concept of a bus for connecting the datapath elements. Let us expand on that concept since it is necessary to understand the datapath extensions for handling interrupts. Specifically, let us propose a bus that connects the processor to the memory and other I/O devices. For the processor to talk to the memory, we need address and data lines. Figure 4.10 shows the datapath with additional lines on the bus for supporting interrupts. There is a wire labeled INT on the bus. Any device that wishes to interrupt the CPU asserts this line. In Chapter 3, we emphasized the importance of ensuring that only one entity accesses a shared bus at any point of time. The INT line on the other hand is different. Any number of devices can simultaneously assert this line to indicate their intent to talk to the processor (analogous to multiple students putting up their hands in the classroom example). An electrical circuit concept called *wired-or* logic makes it possible for several devices to assert the INT line simultaneously. The details of

this electrical trick are outside the scope of this course, and the interested reader should refer to a textbook³ on logic design for more details.

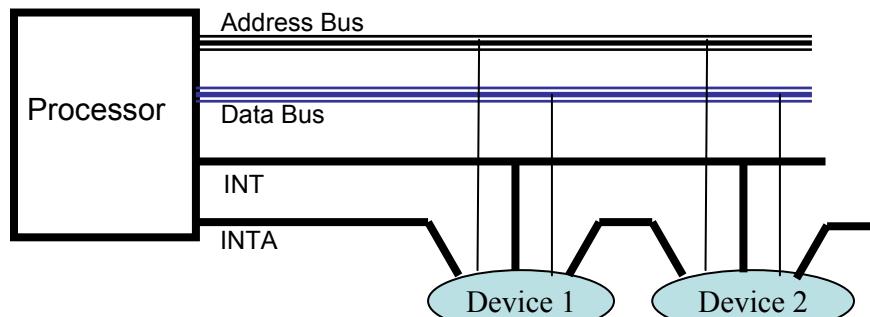


Figure 4.10: Datapath enhancements for handling interrupts

Upon an interrupt, the processor asserts the INTA line (in the INT macro state). Exactly one device should get this acknowledgement, though multiple devices may have signaled their intent to interrupt the processor. Notice the wiring of the INTA. It is not a shared line (like INT) but a chain from one device to another, often referred to as a *daisy chain*. The electrically closest device (Device 1 in Figure 4.10) gets the INTA signal first. If it has requested an interrupt then it knows the processor is ready to talk to it. If it has not requested an interrupt, then it knows that some other device is waiting to talk to the processor and passes the INTA signal down the chain. Daisy chain has the virtue of simplicity but it suffers from latency for the propagation of the acknowledgement signal to the interrupting device, especially in this day and age of very fast processors. For this reason, this method is not used in modern processors.

A generalization of this design principle allows multiple INT and INTA lines to exist on the bus. Each distinct pair of INT and INTA lines corresponds to a *priority level*. Figure 4.11 shows an 8-level priority interrupt scheme. Notice that there is still exactly one device that can talk to the processor at a time (the INTA line from the processor is routed to the INTA line corresponding to the highest priority pending interrupt). Device priority is linked to the speed of the device. The higher the speed of the device the greater the chance of data loss, and hence greater is the need to give prompt attention to that device. Thus, the devices will be arranged on the interrupt lines based on their relative priorities. For example, a disk would be at a higher priority compared to a keyboard. However, despite having multiple interrupt levels, they are insufficient to accommodate all the devices, due to the sheer number, on dedicated interrupt lines. Besides, by definition a device on a higher priority level is more important than the one on a lower priority level. However, there may be devices with similar speed characteristics that belong to the same equivalence class as far as priority goes. Consider a keyboard and a mouse, for instance. Thus, it may still be necessary to place multiple devices on the same priority level as shown in Figure 4.11.

³ Please see: Yale Patt and Sanjay Patel, "Introduction to Computing Systems," McGraw-Hill.

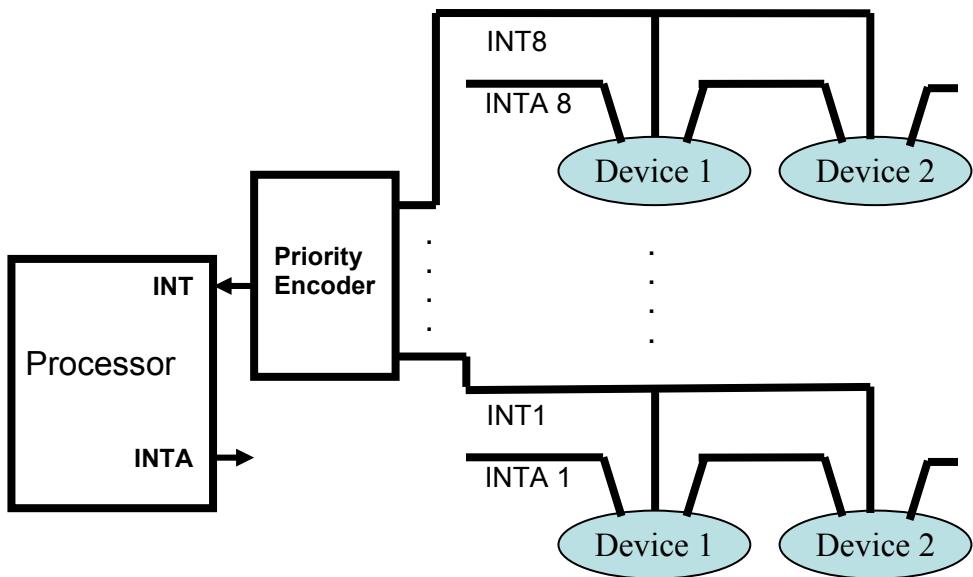


Figure 4.11: Priority Interrupt

As we mentioned earlier, **daisy chaining** the acknowledgement line through the devices may not be desirable **due to the latency associated with the signal propagation**. Besides, processor is a precious resource and every attempt is made nowadays to shift unnecessary burden away from the processor to supporting glue logic outside the processor.

Therefore, in modern processors, **the burden of determining which device needs attention is shifted to an external hardware unit called *interrupt controller***. The interrupt controller takes care of fielding the interrupts, determining the highest priority one to report to the processor, and handling the basic handshake as detailed above for responding to the devices. Instead of daisy chaining the devices in hardware, **the operating system chains the interrupt service routines for a given priority level in a linked list**. Servicing an interrupt now involves walking through the linked list to determine the first device that has an interrupt pending and choosing that for servicing.

You are probably wondering how all of this relates to the way you plug in a device (say a memory stick or a pair of headphones) into your laptop. Actually, it is no big magic. The position of the device (and hence its priority level) **is already pre-determined in the I/O architecture of the system** (as in Figure 4.11), and all you are seeing is the external manifestation of the device slot where you have to plug in your device. We will discuss more about computer buses in a later chapter on Input/Output (See Chapter 10).

4.4.2 Details of receiving the address of the handler

Let us now look at how the **processor receives the vector from an external device**. As we mentioned earlier (see Section 4.2), **the interrupt vector table (IVT)**, set up by the **operating system at boot time**, contains the **handler addresses for all the external interrupts**. While a **device does not know where in memory its handler code is located**, it knows the **table entry that will contain it**. For example, the keyboard may know its vector

is 80 and the mouse may know that its vector is 82⁴. Upon receiving the INTA signal from the processor (see Figure 4.12), the device puts its vector on the data bus. Recall that the processor is still in the INT macro state. The processor uses this vector as the index to look up in the vector table and retrieve the handler address, which is then loaded into the PC. The operating system reserves a portion of the memory (usually low memory addresses) for housing this vector table. The size of the vector table is a design choice of the operating system. Thus, with this one level of indirection through the interrupt vector table, the processor determines the handler address where the code for executing the procedure associated with this specific interrupt is in memory. Figure 4.12 pictorially shows this interaction between the device and the processor for receiving the handler address.

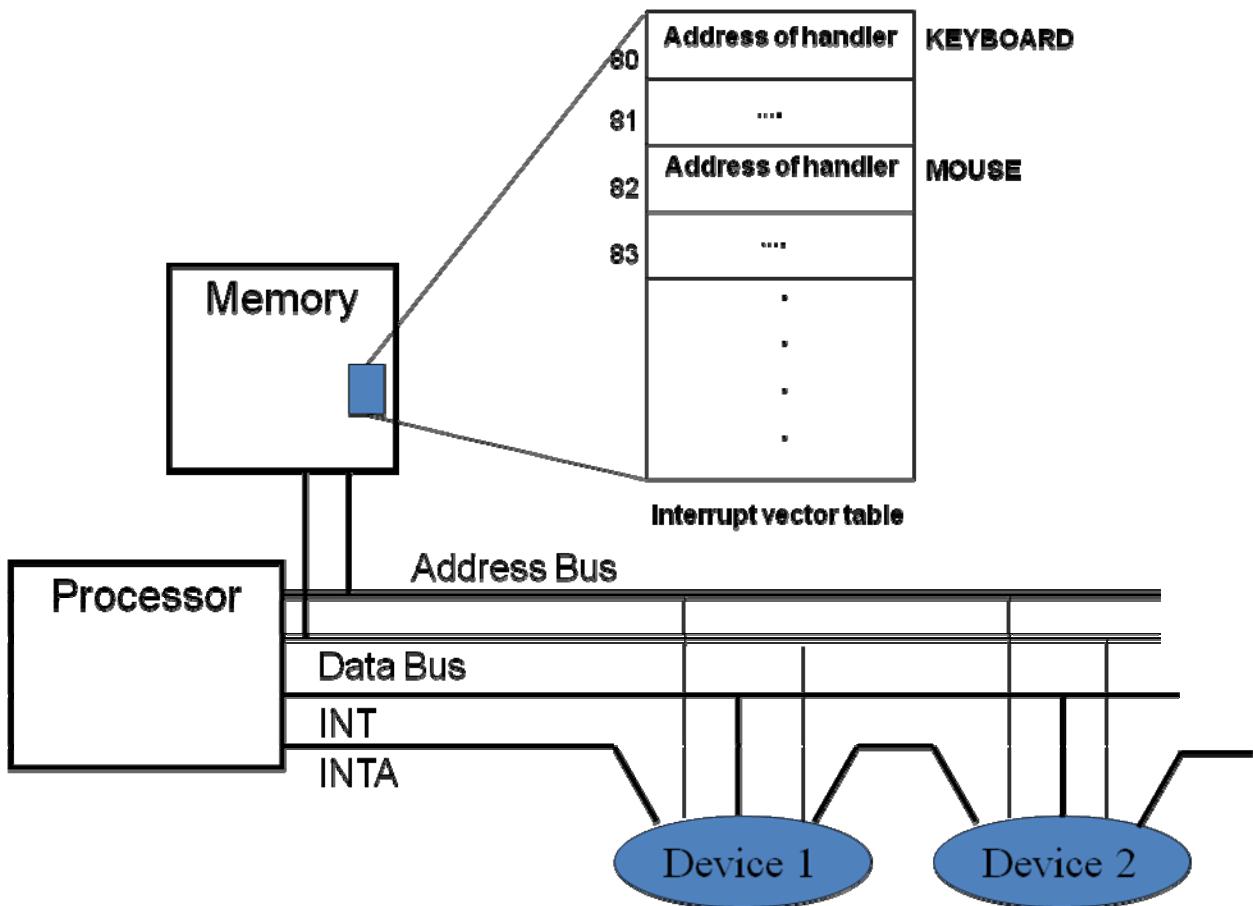


Figure 4.12: Processor-device interaction to receive the interrupt vector

The handshake between the processor and the device is summarized below:

- The device asserts the INT line whenever it is ready to interrupt the processor.

⁴ The OS typically decides the table entry for each device that is then “programmed” into the device interface. We will revisit this topic again in Chapter 9 that deals with Input/Output.

- The processor upon completion of the current instruction (Execute macro-state in Figure 4.4-(b)), checks the INT line (in Figure 4.4-(b) we show this check as INT=y/n in the FSM) for pending interrupts.
- If there is a pending interrupt (INT=y in the FSM shown in Figure 4.4-(b)), then the processor enters the INT macro-state and asserts the INTA line on the bus.
- The device upon receiving the INTA from the processor, places its vector (for example, keyboard will put out 80 as its vector) on the data bus.
- The processor receives the vector and looks up the entry in the interrupt vector table corresponding to this vector. Let us say, the address of the handler found in this entry is 0x5000. This is the PC value for the procedure that needs to be executed for handling this interrupt.
- The processor (which is still in the INT macro state), completes the action in the INT macro-state as shown in Figure 4.4-(b), saving the current PC in \$k0, and loading the PC with the value retrieved from the interrupt vector table.

4.4.3 Stack for saving/restoring

Figure 4.9 shows the handler code which includes saving and restoring of registers (similar to the procedure calling convention). The stack seems like an obvious place to store the processor registers. However, there is a problem. How does the handler know which part of the memory is to be used as a stack? The interrupt may not even be for the currently running program after all.

For this reason, it is usual for the architecture to have two stacks: *user stack* and *system stack*. Quite often, the architecture may designate a particular register as the stack pointer. Upon entering the INT macro state, the FSM performs *stack switching*.

Let us see what hardware enhancements are needed to facilitate this stack switching.

1. Duplicate stack pointer:

Really, all that needs to be done is to duplicate the register designated by the architecture as the stack pointer. In Chapter 2, we designated an architectural register \$sp as the stack pointer. We will duplicate that register: one for use by the user program and the other for use by the system. The state saving in the interrupt handler will use the system version of \$sp while the user programs will use the user version of \$sp. This way, the state saving will not disturb the user stack since all the saving/restoring happens on the system stack. At system start up (i.e., boot time), the system initializes the system version of \$sp with the address of the system stack with sufficient space allocated to deal with interrupts (including nested interrupts).

2. Privileged mode:

Remember that the interrupt handler is simply a program. We need to let the processor know which version of \$sp to use at any point of time. For this reason, we introduce a *mode* bit in the processor. The processor is in *user* or *kernel* mode depending on the value of this bit. If the processor is in user mode the hardware

implicitly uses the user version of \$sp. If it is in kernel mode it uses the kernel version of \$sp. The FSM sets this bit in the INT macro state. Thus, the handler will run in kernel mode and hence use the system stack. Before returning to the user program, RETI instruction sets the mode bit back to “user” to enable the user program to use the user stack when it resumes execution.

The mode bit also serves another important purpose. We introduced three new instructions to support interrupt. It would not be prudent to allow any program to execute these instructions. For example, the register \$k0 has a special connotation and any arbitrary program should not be allowed to write to it. Similarly, any arbitrary program should not be allowed to enable and disable interrupts. Only the operating system executes these instructions, referred to as *privileged instructions*. We need a way to prevent normal user programs from trying to execute these privileged instructions, either accidentally or maliciously. An interrupt handler is part of the operating system and runs in the “kernel” mode (the FSM sets the mode bit in the INT macro state to “kernel”). If a user program tries to use these instructions it will result in an illegal instruction trap.

We need to address two more subtle points of detail to complete the interrupt handling. First, recall that *interrupts could be nested*. Since all interrupt handlers run in the kernel mode, we need to do the mode switching (and the implicit stack switching due to the mode bit) in the INT macro state only when the processor is going from a user program to servicing interrupts. Further, we need to remember the current mode the processor is in to take the appropriate action (whether to return to user or kernel mode) when the handler executes RETI instruction. The system stack is a convenient vehicle to remember the current mode of the processor.

The INT macro state and the RETI instruction push and pop the current mode of the processor on to the system stack, respectively. The INT macro state and the RETI instruction take actions commensurate with the current mode of the processor. Figure 4.13 summarizes all the actions taken in the INT macro state; and Figure 4.14 summarizes the semantics of the RETI instruction.

INT macro state:

```
$k0 ← PC;  
ACK INT by asserting INTA;  
Receive interrupt vector from device on the data bus;  
Retrieve address of the handler from the interrupt vector table;  
PC ← handler address retrieved from the vector table;  
Save current mode on the system stack;  
mode = kernel; /* noop if the mode is already kernel */  
Disable interrupts;
```

Figure 4.13: Actions in the INT Macro State

```

RETI:
    Load PC from $k0;
    /* since the handler executes RETI, we are in the kernel mode */
    Restore mode from the system stack; /* return to previous mode */
    Enable interrupts;

```

Figure 4.14: Semantics of RETI instruction

4.5 Putting it all together

4.5.1 Summary of Architectural/hardware enhancements

To deal with program discontinuities, we added the following architectural/hardware enhancements to LC-2200:

1. An interrupt vector table (IVT), to be initialized by the operating system with handler addresses.
2. An exception/trap register (ETR) that contains the vector for internally generated exceptions and traps.
3. A Hardware mechanism for receiving the vector for an externally generated interrupt.
4. User/kernel mode and associated mode bit in the processor.
5. User/system stack corresponding to the mode bit.
6. A hardware mechanism for storing the current PC implicitly into a special register \$k0, upon an interrupt, and for retrieving the handler address from the IVT using the vector (either internally generated or received from the external device).
7. Three new instructions to LC-2200:
 Enable interrupts
 Disable interrupts
 Return from interrupt

4.5.2 Interrupt mechanism at work

We present a couple of examples to bring all these concepts together and give the reader an understanding of how the interrupt mechanism works. For clarity of presentation, we will call the system version of \$sp as SSP and the user version of \$sp as USP. However, architecturally (i.e., from the point of view of the instruction set) they refer to the same register. The hardware knows (via the mode bit) whether to use USP or SSP as \$sp.

Example 4:

The example sketched in Figure 4.15 (A-D) illustrates the sequence of steps involved in interrupt handling. Some program **foo** is executing (Figure 4.15-A).

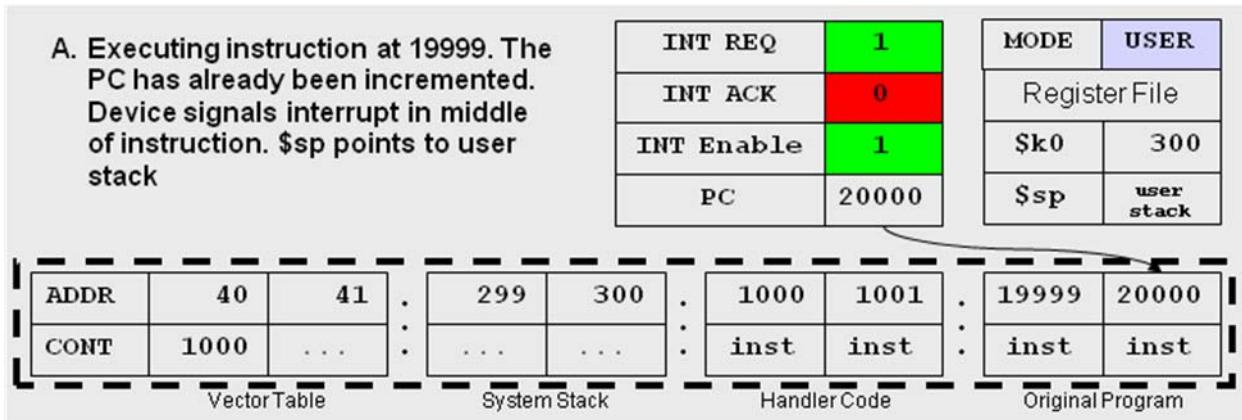


Figure 4.15-A: Interrupt Handling (INT received)

A keyboard device interrupts the processor. The processor is currently executing an instruction at location 19999. It waits until the instruction execution completes. It then goes to the INT macro state (Figure 4.15-B) and saves the current PC (whose value is 20000) in \$k0. Upon receipt of INTA from the processor, the device puts out its vector on the data bus. The processor receives the vector from the device on the data bus as shown in Figure 4.15-B.

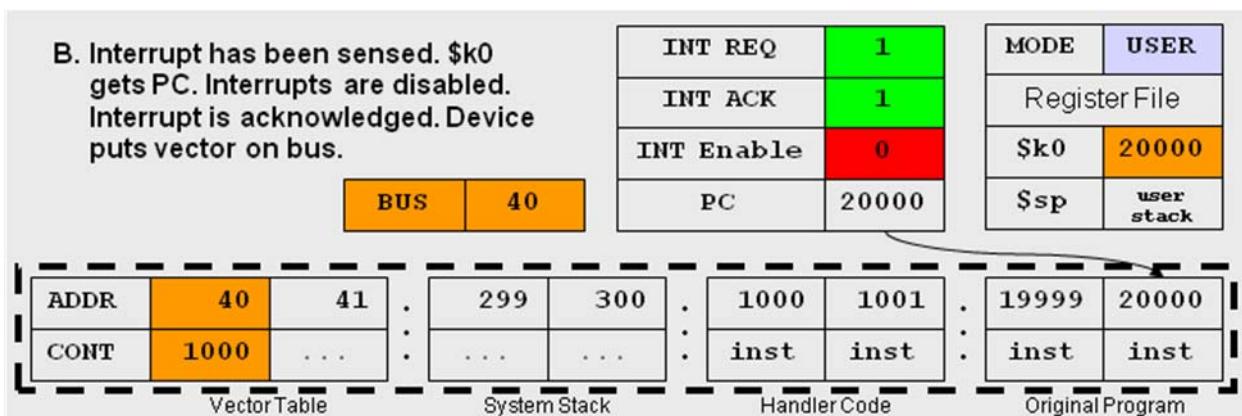


Figure 4.15-B: Interrupt Handling (INT macro state – receive vector)

Let us say the value received is 40. The processor looks up memory location 40 to get the handler address (let this be 1000). Let the contents of the SSP be 300. In the INT macro state, the FSM loads 1000 into PC, 300 into \$sp, saves the current mode on the system stack, and goes back to the FETCH macro state. The handler code at location 1000 (similar to Figure 4.9) starts executing using \$sp = 299 as its stack (Figure 4.15-C).

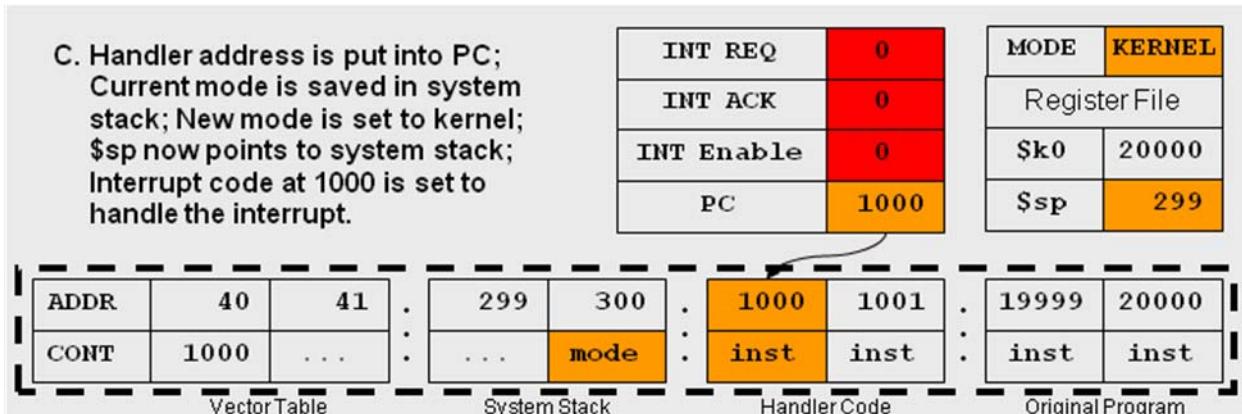


Figure 4.15-C: Interrupt Handling (Transfer control to handler code)

The original program will resume execution at PC = 20000 when the handler executes return from interrupt (Figure 4.15-D).

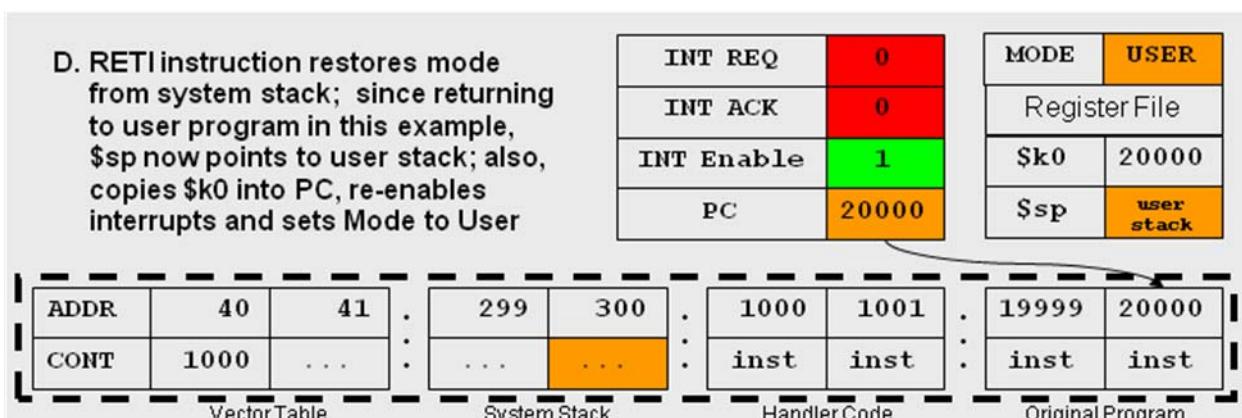


Figure 4.15-D: Interrupt Handling (return to original program)

Example 5:

Consider the following:

Assume memory addresses are consecutive integers

User program executing instruction at memory location 7500

User Stack pointer (\$sp) value 18000

SSP value 500

Vector for keyboard = 80

Vector for disk = 50

Handler address for keyboard interrupt = 3000

Handler address for disk interrupt = 5000

(a) Pictorially represent the above information similar to Figure 4.15

(b) An interrupt occurs from the keyboard. Show the relevant state of the processor (similar to Figure 4.15-C) when the interrupt handler for the keyboard is about to start executing.

- (c) A higher priority interrupt from the disk occurs while the keyboard handler is running after it has re-enabled interrupts. Assume that it is executing an instruction at memory location 3023, and the stack pointer (\$sp) value is 515. Show the relevant state of the processor (similar to Figure 4.15-C) when the interrupt handler for the disk is about to start executing.
- (d) Show the relevant state of the processor when the disk handler executes RETI instruction (similar to Figure 4.15-D).
- (e) Show the relevant state of the processor when the keyboard handler executes RETI instruction (similar to Figure 4.15-D).

This example is left as an exercise for the reader.

4.6 Summary

In this Chapter, we introduced an important concept, interrupts, that allows a processor to communicate with the outside world. An interrupt is a specific instance of program discontinuity. We discussed the minimal hardware enhancements needed inside the processor as well as at the bus level to handle nested interrupts.

- The processor enhancements included three new instructions, a user stack, a system stack, a mode bit, and a new macro state called INT.
- At the bus level, we introduced special control lines called INT and INTA for the device to indicate to the processor that it wants to interrupt and for the processor to acknowledge the interrupt, respectively.

We reviewed traps and exceptions that are synchronous versions of program discontinuities as well. The interesting thing is that the software mechanism needed to handle all such discontinuities is similar. We discussed how to write a generic interrupt handler that can handle nested interrupts.

We have intentionally simplified the presentation of interrupts in this chapter to make it accessible to students in a first systems course. Interrupt mechanisms in modern processors are considerably more complex. For example, modern processors categorize interrupts into two groups: *maskable* and *non-maskable*.

- The former refers to interrupts that can be temporarily turned off using the disable interrupt mechanism (e.g., a device interrupt).
- The latter corresponds to interrupts that cannot be turned off even with the disable interrupt mechanism (e.g., an internal hardware error detected by the system).

We presented a mechanism by which a processor learns the starting address of the interrupt handler (via the vector table) and the use of a dedicated register for stashing the return address for the interrupted program. We also presented a simple hardware scheme by which a processor may discover the identity of the interrupting device and acknowledge the interrupt. The main intent is give confidence to the reader that designing such hardware is simple and straightforward.

We presented mode as a characterization of the internal state of a processor. This is also an intentionally simplistic view. The processor state may have a number of other attributes available as discrete bits of information (similar to the mode bit). Usually, a processor aggregates all of these bits into one register called *processor status word* (*PSW*). Upon an interrupt and its return, the hardware implicitly pushes and pops, respectively, both the PC and the PSW on the system stack⁵.

We also presented a fairly simple treatment of the interrupt handler code to understand what needs to be done in the processor architecture to deal with interrupts. The handler would typically do a lot more than save processor registers. In later Chapters, we will revisit interrupts in the context of operating system functionalities such as processor scheduling (Chapter 6) and device drivers (Chapter 10).

Interrupt architecture of modern processors is much more sophisticated. First of all, since the processor is a precious resource, most of the chores associated with interrupt processing except for executing the actual handler code is kept outside the processor. For example, a device called *programmable interrupt controller* (*PIC*) aids the processor in dealing with many of the nitty-gritty details with handling external interrupts including:

- Dealing with multiple interrupt levels,
- Fielding the actual interrupts from the devices,
- Selecting the highest priority device among the interrupting devices,
- Getting the identity (vector table index) of the device selected to interrupt the processor, and
- Acknowledging the selected device

The PIC provides processor-readable registers, one of which contains the identity of the device selected to interrupt the processor. The use of PIC simplifies what the processor has to do on an interrupt. Upon an interrupt, the return address either is placed on a system stack or is made available in a special processor register, and control is simply transferred to a well-defined address set by the operating system, which corresponds to a generic first level interrupt handler of the operating system. This handler simply saves the return address for the interrupted program, reads the identity of the interrupting device from the PIC, and jumps to the appropriate handler code. Usually this first level operating system handler is non-interruptible and may run in a special mode called *interrupt mode*, so the operating system does not have to worry about nested interrupts. In general, it is important that a device driver – software that actually manipulates a device – do as little work in an interrupt handling code as possible. This is to make sure that the processor is not tied up forever in interrupt processing. Device drivers are written such that only the time sensitive code is in the interrupt handler. For example, Linux operating system defines *top-half* and *bottom-half* handlers. By definition, the bottom-half handlers do not have the same sense of urgency as the top-half handlers. A device that has significant amount of work to do that is not time critical will do it in the bottom-half handler.

⁵ In LC-2200, we designate a register \$k0 for saving PC in the INT macro state. An alternative approach adopted in many modern processors is to save the PC directly on the system stack.

We have only scratched the surface of issues relating to interrupt architecture of processors and operating system mechanisms for efficiently dealing with them. The interested reader is referred to more advanced textbooks on computer architecture⁶ for details on how the interrupt architecture is implemented in modern processors, as well as to books on operating system implementation⁷ to get a deeper knowledge on interrupt handling.

4.7 Review Questions

1. Upon an interrupt, what has to happen implicitly in hardware before control is transferred to the interrupt handler?
2. Why not use JALR to return from the interrupt handler?
3. Put the following steps in the correct order

Actual work of the handler

Disable interrupt
Enable interrupt
Restore ko from stack
Restore state
Return from interrupt
Save ko on stack
Save state

4. How does the processor know which device has requested an interrupt?
5. What instructions are needed to implement interruptible interrupts? Explain the function and purpose of each along with an explanation of what would happen if you didn't have them.
6. In the following interrupt handler code select the **ONES THAT DO NOT BELONG.**

_____ disable interrupts;
_____ save PC;
_____ save \$k0;
_____ enable interrupts;
_____ save processor registers;
_____ execute device code;
_____ restore processor registers;
_____ disable interrupts;
_____ restore \$k0;

⁶ Hennessy and Patterson, “Computer Architecture: A Quantitative Approach,” Morgan Kaufmann publishers.

⁷ Alessandro Rubini & Jonathan Corbet , “Linux Device Drivers,” 2nd Edition, O'Reilly & Associates

- _____ disable interrupts;
_____ restore PC;
_____ enable interrupts;
_____ return from interrupt;
7. In the following actions in the INT macrostate, select the **ONES THAT DO NOT BELONG**.

_____ save PC;
_____ save SP;
_____ $\$k0 \leftarrow PC$;
_____ enable interrupts;
_____ save processor registers;
_____ ACK INT by asserting INTA;
_____ Receive interrupt vector from device on the data bus;
_____ Retrieve PC address from the interrupt vector table;
_____ Retrieve SP value from the interrupt vector table;
_____ disable interrupts
_____ $PC \leftarrow PC$ retrieved from the vector table;
_____ $SP \leftarrow SP$ value retrieved from the vector table;
_____ disable interrupts;

Chapter 5 Processor Performance and Rudiments of Pipelined Processor Design (Revision number 14)

In Chapter 3, we gave a basic design of a processor to implement the LC-2200 ISA. We hinted at how to achieve good performance when we talked about selecting the width of the clock cycle time and reducing the number of microstates in implementing each macro state (Fetch, Decode, and instruction-specific Execute states).

Processor design and implementation is a quantitative exercise. An architect is constantly evaluating his/her decision to include an architectural feature in terms of its impact on performance. Therefore, we will first explore the performance metrics that are of relevance in processor design. We will then look at ways to improve processor performance first in the context of the simple design we presented in Chapter 3, and then in the context of a new concept called pipelining.

First, let's introduce some metrics to understand processor performance.

5.1 Space and Time Metrics

Let's say you are building an airplane. You want to ferry some number of passengers every time, so you have to provide adequate space inside the aircraft for accommodating the passengers, their baggage, and food for them en route. In addition, you want to engineer the aircraft to take the passengers from point A to point B in a certain amount of time. The number of passengers to be accommodated in the plane has a bearing on the time it takes to ferry them, since the horsepower of the engine to fly the plane will depend on the total weight to be carried.

Let's see how this analogy applies to processor performance. Due to the hype about processor speed, we always think of “MHz” “GHz” and “THz” whenever we think of processor performance. These terms, of course, refer to the **clock rate of the processor**. We know from Chapter 3 that the clock cycle time (inverse of clock rate) is determined by **estimating the worst-case delay in carrying out the datapath actions in a single clock cycle**.

Let us understand why **processor speed is not the only determinant of performance**. Let us say you wrote a program “foo” and are running it on a processor. The two performance metrics that you would be interested in are, **how much memory foo occupies (space metric)** and **how long does foo take to run (time metric)**. *Memory footprint* quantifies the space metric, while *execution time* quantifies the time metric. We define the former as the space occupied by a given program and the latter as the running time of the program. Let us relate these metrics to what we have seen about processor design so far.

The instruction-set architecture of the processor has a bearing on the memory footprint of the program. First, let us understand the **relationship between the two metrics**. There is a belief that the **smaller the footprint the better the execution time**. The conventional

wisdom in the 70's was that this premise is true and led to the design of *Complex Instruction Set Computer* (CISC) architectures. The criterion in CISC architecture was to make the datapath and control unit do more work for each instruction fetched from memory. This gave rise to a set of instructions, in which different instructions took different number of microstates to complete execution depending on their complexity. For example, an *add* instruction will take less number of microstates to execute, while a *multiply* instruction will take more. Similarly, an *add* instruction that uses register operands will take less microstates to execute while the same instruction that uses memory to operands will take more. With instructions that are more complex it is conceivable that the intended program logic can be contained in fewer instructions leading to a smaller memory footprint.

Common sense reasoning and advances in computer technology weakened the premise regarding the connection between memory footprint and the execution time.

- This is where the airplane analogy breaks down. The passengers in the analogy are akin to instructions in memory. All the passengers in the airplane need to be ferried to the destination, and therefore contribute to the weight of the airplane, which in turn determines the flying time to the destination. However, not all instructions that comprise a program are necessarily executed. For example, it is well known that in many production programs a significant percentage of the program deals with error conditions that may arise during program execution. You know from your own programming experience that one of the good software engineering practices is to check for return codes on systems calls. The error-handling portion of these checks in these programs may rarely be executed. Just to drive home that point, as a pathological example, consider a program that consists of a million instructions. Suppose there is a tight loop in the program with just 10 instructions where 99% of the execution time is spent; then the size of the program does not matter.
- Second, advances in processor implementation techniques (principally the idea of pipelining instruction execution, which we will discuss in this chapter) blurred the advantage of a complex instruction over a sequence of simple instructions that accomplish the same end result.
- Third, with the shift in programming in assembly language to programming in high-level languages, the utility of an instruction-set was measured by how useful it is to the compiler writer.
- Fourth, with the advent of cache memories, which we will cover in detail in Chapter 9, trips to the main memory from the processor is reduced, thus weakening one of the fundamental premise with CISC that execution time is minimized by fetching a complex instruction as opposed to a set of simple instructions.

These arguments gave rise to the *Reduced Instruction Set Computer (RISC)* Architecture in the late 70's and early 80's. While there was a lot of debate on CISC vs. RISC in the 80's, such arguments have become largely irrelevant today. The reality is both sides have shifted to include good features from the other camp. As we will see shortly, when we discuss pipelining, the ISA is not as important as ensuring that the processor maintains a

throughput of one instruction every clock cycle. For example, the dominant ISA today is Intel x86, a CISC architecture, but implemented with the RISC philosophy as far as implementation goes. Another influential ISA is MIPS, which is a RISC architecture but includes instructions traditionally found in CISC architectures.

We will have more detailed discussion on memory footprint itself in a later chapter that discusses memory hierarchy. At this point, let us simply observe that there is not a strong correlation between the memory footprint of a program and its execution time.

Let us try to understand what determines the execution time of the program. The number of instructions executed by the processor for this program is one component of the execution time. The second component is the number of microstates needed for executing each instruction. Since each microstate executes in one CPU clock cycle, the execution time for each instruction is measured by the clock cycles taken for each instruction (usually referred to as *CPI – clocks per instruction*). The third component is the clock cycle time of the processor.

So, if n is the total number of instructions executed by the program, then,

$$\text{Execution time} = (\sum \text{CPI}_j) * \text{clock cycle time}, \text{ where } 1 \leq j \leq n \quad (1)$$

Sometimes, it is convenient to think of an average CPI for the instructions that are executed by the program. So, if CPI_{Avg} is the average CPI for the set of instructions executed by the program, we can express the execution time as follows:

$$\text{Execution time} = n * \text{CPI}_{\text{Avg}} * \text{clock cycle time} \quad (2)$$

Of course, it is difficult to quantify what an average CPI is, since this really depends on the frequency of execution of the instructions. We discuss this aspect in more detail in the next section. Execution time of a program is the key determinant of processor performance. Perhaps more accurately, since the clock cycle time changes frequently, *cycle count* (i.e., the total number of clock cycles for executing a program) is a more appropriate measure of processor performance. In any event, it should be clear that the processor performance is much more than simply the processor speed. Processor speed is no doubt important but the cycle count, which is the product of the number of instructions executed and the CPI for each instruction, is an equally if not more important metric in determining the execution time of a program.

Example 1:

A processor has three classes of instructions:

$$A, B, C; \text{CPI}_A = 1; \text{CPI}_B = 2; \text{CPI}_C = 5.$$

A compiler produces two different but functionally equivalent code sequences for a program:

Code sequence 1 executes:

$$\begin{aligned} A\text{-class instructions} &= 5 \\ B\text{-class instructions} &= 3 \end{aligned}$$

C-class instructions = 1

Code sequence 2 executes:

A-class instructions = 3

B-class instructions = 2

C-class instructions = 2

Which is faster?

Answer:

Code sequence 1 results in executing 9 instructions, and takes a total of 16 cycles

Code sequence 2 results in executing 7 instructions but takes a total of 17 cycles

Therefore, code sequence 1 is faster.

5.2 Instruction Frequency

It is useful to know how often a particular instruction occurs in programs. *Instruction frequency* is the metric for capturing this information. *Static* instruction frequency refers to the number of times a particular instruction occurs in the compiled code. *Dynamic* instruction frequency refers to the number of times a particular instruction is executed when the program is actually run. Let us understand the importance of these metrics. Static instruction frequency has impact on the memory footprint. So, if we find that a particular instruction appears a lot in a program, then we may try to optimize the amount of space it occupies in memory by clever instruction encoding techniques in the instruction format. Dynamic instruction frequency has impact on the execution time of the program. So, if we find that the dynamic frequency of an instruction is high then we may try to make enhancements to the datapath and control to ensure that the CPI taken for its execution is minimized.

Static instruction frequency has become less important in general-purpose processors since reducing the memory footprint is not as important a consideration as increasing the processor performance. In fact, techniques to support static instruction frequency such as special encoding will have detrimental effect on performance since it destroys the uniformity of the instructions, which is crucial for a pipelined processor, as we will see later in this chapter (please see Section 5.10). However, static instruction frequency may still be an important factor in embedded processors, wherein it may be necessary to optimize on the available limited memory space.

Example 2:

Consider the program shown below that consists of 1000 instructions.

I₁:

I₂:

..

..

```

..  

I10:  

I11: ADD  

I12:  

I13:  

I14: COND BR I10 } loop  

..  

..  

I1000:

```

ADD instruction occurs exactly once in the program as shown. Instructions I₁₀-I₁₄ constitute a loop that gets executed 800 times. All other instructions execute exactly once.

- (a) What is the static frequency of ADD instruction?

Answer:

The memory footprint of the program is 1000 instructions. Out of these 1000 instructions, Add occurs exactly once.

Hence the static frequency of Add instruction = $1/1000 * 100 = 0.1\%$

- (b) What is the dynamic frequency of ADD instruction?

Answer:

$$\begin{aligned}
 \text{Total number of instructions executed} &= \text{loop execution} + \text{other instruction execution} \\
 &= (800 * 5) + (1000 - 5) * 1 \\
 &= 4995
 \end{aligned}$$

Add is executed once every time the loop is executed.

So, the number of Add instructions executed = 800

Dynamic frequency of Add instruction

$$\begin{aligned}
 &= (\text{number of Add instructions executed} / \text{total number of instructions executed}) * 100 \\
 &= (800 / (4995 + 4000)) * 100 = 16\%
 \end{aligned}$$

5.3 Benchmarks

Let us discuss how to compare the performance of machines. One often sees marketing hype such as “processor X is 1 GHz” or “processor Y is 500 MHz”. How do we know which processor is better given that **execution time is not entirely determined by processor speed**. **Benchmarks** are a set of programs that are representative of the **workload for a processor**. For example, for a processor used in a gaming console, a video game may be the benchmark program. For a processor used in a scientific application, matrix operations may be benchmark programs. Quite often, **kernels** of real programs are used as benchmarks. For example, matrix multiply may occur in several scientific applications and may be biggest component of the execution time of such applications. In that case, it makes sense to benchmark the processor on the matrix multiply routine.

The performance of the processor on such kernels is a good indicator of the expected performance of the processor on an application as a whole.

It is often the case that a set of programs constitute the benchmark. There are several possibilities for deciding how best to use these benchmark programs to evaluate processor performance:

1. Let's say you have a set of programs and all of them have to be run one after the other to completion. In that case, a summary metric that will be useful is **total execution time**, which is the cumulative **total of the execution times** of the individual programs.
2. Let's say you have a set of programs and it is equally likely that you would want to **run them at different times**, but not **all at the same time**. In this case, **arithmetic mean (AM)** would be a useful metric, which is **simply an average of all the individual program execution times**. It should be noted, however that this metric may bias the summary value towards a time-consuming benchmark program (e.g., execution times of programs: P1 = 100 secs; P2 = 1 secs; AM = 50.5 secs).
3. Let's say you have a similar situation as above, but you have an idea of the **frequency with which you may run the individual programs**. In this case, **weighted arithmetic mean (WAM)** would be a useful metric, which is a weighted average of the execution times of all the individual programs. This metric takes into account the **relative frequency of execution of the programs in the benchmark mix** (e.g., for the same programs P1 = 100 secs; P2 = 1 secs; $f_{P1} = 0.1$; $f_{P2} = 0.9$; WAM = $0.1 * 100 + 0.9 * 1 = 10.9$ secs).
4. Let's say you have a situation similar to (2) above but you have no idea of the relative frequency with which you may want to run one program versus another. In this case, using arithmetic mean may give a biased view of the processor performance. Another summary metric that is useful in this situation is **geometric mean (GM)**, which is the p^{th} root of the product of p values. This metric removes the bias present in arithmetic mean (e.g. for the same programs P1 = 100 secs; P2 = 1 secs; GM = $\sqrt{100*1} = 10$ secs).
5. **Harmonic mean (HM)** is another useful composite metric. Mathematically, it is computed by taking the arithmetic mean of the reciprocals of the values being considered, and then taking the reciprocal of the result. This also helps the bias towards high values present in the arithmetic mean. The HM for the example we are considering (execution times of programs: P1 = 100 secs; P2 = 1 secs),

$$\begin{aligned} \text{HM} &= 1/(\text{arithmetic mean of the reciprocals}) \\ &= 1/((1/100) + (1/1)/2) = 1.9801. \end{aligned}$$

Harmonic means are especially considered useful when the **dataset consists of ratios**.

If the data set consists of all equal values then all three composite metrics (AM, GM, and HM) will yield the same result. In general, AM tends to bias the result towards high values in the data set, HM tends to bias the result towards low values in the data set, and GM tends to be in between. The moral of the story is one has to be very cautious about the use of a single composite metric to judge an architecture.

Over the years, several benchmark programs have been created and used in architectural evaluations. The most widely accepted one is the SPEC benchmarks developed by an independent non-profit body called *Standard Performance Evaluation Corporation* (SPEC), whose goal is “to establish, maintain and endorse a standardized set of relevant benchmarks that can be applied to the newest generation of high-performance computers¹. ” SPEC benchmark consists of a set of generic applications including scientific, transaction processing, and web server that serves as the workload for general-purpose processors².

What makes **benchmarking hard** is the fact that processor performance is not just determined by **the clock cycle rating of the processor**. For example, the organization of the memory system and the processor-memory bus bandwidth are key determinants beyond the clock cycle time. Further, the behavior of each benchmark application poses different demands on the overall system. Therefore, when we compare two processors with comparable or even same CPU clock cycle rating, we may find that one does better on some benchmark programs while the second does better on some others. This is the reason a composite index is useful sometimes when we want to compare two processors without knowing the exact kind of workload we may want to run on them. One has to be very cautious on the over-use of metrics as has been noted in a very famous saying, “Lies, damned lies, and statistics³.”

Example 3:

The SPECint2006 integer benchmark consists of 12 programs for quantifying the performance of processors on integer programs (as opposed to floating point arithmetic). The following table⁴ shows the performance of Intel Core 2 Duo E6850 (3 GHz) processor on SPECint2006 benchmark.

Program name	Description	Time in seconds
400.perlbench	Applications in Perl	510
401.bzip2	Data compression	602
403.gcc	C Compiler	382
429.mcf	Optimization	328
445.gobmk	Game based on AI	548
456.hmmr	Gene sequencing	593
458.sjeng	Chess based on AI	679
462.libquantum	Quantum computing	422
464.h264ref	Video compression	708
471.omnetpp	Discrete event simulation	362
473.astar	Path-finding algorithm	466
483.xalancbmk	XML processing	302

¹ Source: <http://www.spec.org/>

² Please see <http://www.spec.org/cpu2006/publications/CPU2006benchmarks.pdf> for a description of SPEC2006 benchmark programs for measuring the integer and floating point performance of processors.

³ Please see: <http://www.york.ac.uk/depts/mathsc/histstat/lies.htm>

⁴ Source: <http://www.spec.org/cpu2006/results/res2007q4/cpu2006-20071112-02562.pdf>

(a) Compute the arithmetic mean and the geometric mean.

Answer:

$$\begin{aligned}\text{Arithmetic mean} &= (510+602+\dots+302)/12 &= \mathbf{491.8 \text{ secs}} \\ \text{Geometric mean} &= (510 * 602 * \dots * 302)^{1/12} &= \mathbf{474.2 \text{ secs}}\end{aligned}$$

Note how arithmetic mean biases the result towards the larger execution times in the program mix.

(b) One intended use of the system, has the following frequencies for the 12 programs:

- 10% video compression
- 10% XML processing
- 30% path finding algorithm
- 50% all the other programs

Compute the weighted arithmetic mean for this workload

Answer:

The workload uses 9 programs equally 50% of the time.

The average execution time of these nine programs

$$\begin{aligned}&= (510 + 602 + 382 + 328 + 548 + 593 + 679 + 422 + 362)/9 \\ &= 491.8 \text{ secs}\end{aligned}$$

$$\begin{aligned}\text{Weighted arithmetic mean} &= (0.1 * 708 + 0.1 * 302 + 0.3 * 466 + 0.5 * 491.8) \\ &= \mathbf{486.7 \text{ secs}}\end{aligned}$$

One of the usefulness of metrics is that it gives a basis for comparison of machines with different architectures, implementation, and hardware specifications. However, raw numbers as reported in Example 3, makes it difficult to compare different machines. For this reason, SPEC benchmark results are expressed as ratios with respect to some reference machine. For example, if the time for a benchmark on the target machine is x secs, and the time on the reference machine for the same benchmark is y secs, then the SPECratio for this benchmark on the target machine is defined as

SPECratio

$$\begin{aligned}&= \text{execution time on reference machine / execution time on target machine} \\ &= y/x\end{aligned}$$

SPEC organization chose Sun Microsystems's Ultra5_10 workstation with a 300-MHz SPARC processor and 256-MB of memory as a reference machine for SPEC CPU 2000 performance test results. The same is used for SPEC CPU 2006 as well.

The SPECratios for different benchmarks may be combined using one of the statistical measures (arithmetic, weighted arithmetic, geometric, or harmonic) to get a single composite metric. Since we are dealing with ratios when using SPEC benchmarks as the standard way of reporting performance, it is customary to use harmonic mean.

The nice thing about SPECratio is that it serves as a basis for comparison of machines. For example, if the mean SPECratios of two machines A and B are R_A and R_B , respectively, then we can directly make relative conclusions regarding the performance capabilities of the two machines.

5.4 Increasing the Processor Performance

To explore avenues for increasing the processor performance, a good starting point is the equation for execution time. Let us look at each term individually and understand the opportunity for improving the performance that each offers.

- **Increasing the clock speed:** The processor clock speed is determined by the worst-case delay in the datapath. We could rearrange the datapath elements such that the worst-case delay is reduced (for example bringing them closer together physically in the layout of the datapath). Further, we could reduce the number of datapath actions taken in a single clock cycle. However, such attempts at reducing the clock cycle time have an impact on the number of CPIs needed to execute the different instructions. To get any more reduction in clock cycle time beyond these ideas, the feature size of the individual datapath elements should be shrunk. This avenue of optimization requires coming up with new chip fabrication processes and device technologies that help in reducing the feature sizes.
- **Datapath organization leading to lower CPI:** In Chapter 3, the implementation uses a single bus. Such an organization limits the amount of hardware concurrency among the datapath elements. We alluded to designs using multiple buses to increase the hardware concurrency. Such designs help in reducing the CPI for each instruction. Once again any such attempt may have a negative impact on the clock cycle time, and therefore requires careful analysis. Designing the microarchitecture of a processor and optimizing the implementation to maximize the performance is a fertile area of research both in academia and in industries.

The above two bullets focus on reducing the *latency* of individual instructions so that cumulatively we end up with a lower execution time.

Another opportunity for reducing the execution time is reducing the number of instructions.

- **Reduction in the number of executed instructions:** One possibility for reducing the number of instructions executed in the program is to replace simple instructions by more complex instructions. This would reduce the number of instructions executed by the program overall. We already saw a counter example to this idea. Once again any attempt to introduce new complex instructions has to be carefully balanced against the CPI, the clock cycle time, and the dynamic instruction frequencies.

It should be clear from the above discussion that all three components of the execution time are inter-related and have to be optimized simultaneously and not in isolation.

Example 4:

An architecture has three types of instructions that have the following CPI:

Type	CPI
A	2
B	5
C	1

An architect determines that he can reduce the CPI for B to three, with no change to the CPIs of the other two instruction types, but with an increase in the clock cycle time of the processor. What is the maximum permissible increase in clock cycle time that will make this architectural change still worthwhile? Assume that all the workloads that execute on this processor use 30% of A, 10% of B, and 60% of C types of instructions.

Answer:

Let C_o and C_n be the clock cycle times of the old (M_o) and new (M_n) machines, respectively. Let N be the total number of instructions executed in a program.

Execution time of Old Machine

$$ET_{M_o} = N * (F_A * CPI_{A_o} + F_B * CPI_{B_o} + F_C * CPI_{C_o}) * C_o$$

Where F_A , CPI_{A_o} , F_B , CPI_{B_o} , F_C , CPI_{C_o} are the dynamic frequencies and CPIs of each type of instruction, respectively.

Execution time of the Old Machine

$$\begin{aligned} ET_{M_o} &= N * (0.3 * 2 + 0.1 * 5 + 0.6 * 1) * C_o \\ &= N * 1.7 C_o \end{aligned}$$

Execution time of New Machine

$$\begin{aligned} ET_{M_n} &= N * (0.3 * 2 + 0.1 * 3 + 0.6 * 1) * C_n \\ &= N * 1.5 C_n \end{aligned}$$

For design to be viable

$$\begin{aligned} ET_{M_n} &< ET_{M_o} \\ N * 1.5 C_n &< N * 1.7 C_o \\ C_n &< 1.7/1.5 * C_o \\ C_n &< 1.13 C_o \end{aligned}$$

Max Permissible increase in clock cycle time = 13%

5.5 Speedup

Comparing the execution times of processors for the same program or for a benchmark suite is the most obvious method for understanding the performance of one processor relative to another. Similarly, we can compare the improvement in execution time before and after a proposed modification to quantify the performance improvement that can be attributed to that modification.

We define,

$$Speedup_{A \text{ over } B} = \frac{\text{Execution Time on Processor B}}{\text{Execution Time on Processor A}} \quad (3)$$

Speedup of processor A over processor B is the ratio of execution time on processor B to the execution time on processor A.

Similarly,

$$Speedup_{improved} = \frac{\text{Execution Time Before Improvement}}{\text{Execution Time After Improvement}} \quad (4)$$

Speedup due to a modification is the ratio of the execution time before improvement to that after the improvement.

Example 5:

Given below are the CPIs of instructions in an architecture:

Instruction	CPI
Add	2
Shift	3
Others	2 (average of all instructions including Add and Shift)

Profiling the performance of a programs, an architect realizes that the sequence ADD followed by SHIFT appears in 20% of the dynamic frequency of the program. He designs a new instruction, which is an ADD/SHIFT combo that has a CPI of 4.

What is the speedup of the program with all {ADD, SHIFT} replaced by the new combo instruction?

Answer:

Let N be the number of instructions in the original program.

Execution time of the original program

$$\begin{aligned} &= N * \text{frequency of ADD/SHIFT} * (2+3) + N * \text{frequency of others} * 2 \\ &= N * 0.2 * 5 + N * 0.8 * 2 = 2.6 N \end{aligned}$$

With the combo instruction replacing {ADD, SHIFT}, the number of instructions in the new program shrinks to 0.9 N. In the new program, the frequency of the combo instruction is 1/9 and the other instructions are 8/9.

Execution time of the new program

$$\begin{aligned} &= (0.9 N) * \text{frequency of combo} * 4 + (0.9 N) * \text{frequency of others} * 2 \\ &= (0.9 N) * (1/9) * 4 + (0.9 N) * (8/9) * 2 \\ &= 2 N \end{aligned}$$

Speedup of program = old execution time/new execution time
 $= (2.6 N) / (2 N)$
 $= 1.3$

Another useful metric is the performance improvement due to a modification:

$$\text{Improvement in execution time} = \frac{\text{old execution time} - \text{new execution time}}{\text{old execution time}} \quad (5)$$

Example 6:

A given program takes 1000 instructions to execute with an average CPI of 3 and a clock cycle time of 2 ns. An architect is considering two options. (1) She can reduce the average CPI of instructions by 25% while increasing the clock cycle time by 10%; or (2) She can reduce the clock cycle time by 20% while increasing the average CPI of instructions by 15%.

- (a) You are the manager deciding which option to pursue. Give the reasoning behind your decision.

Answer:

Let E0, E1, and E2, denote the execution times with base machine, first option and second option respectively.

$$\begin{aligned} E0 &= 1000 * 3 * 2 \text{ ns} \\ E1 &= 1000 * (3 * 0.75) * 2 (1.1) \text{ ns} = 0.825 E0 \\ E2 &= 1000 * (3 * 1.15) * 2 (0.8) \text{ ns} = 0.920 E0 \end{aligned}$$

Option 1 is better since it results in lesser execution time than 2.

- (b) What is the improvement in execution time of the option you chose compared to the original design?

Answer:

$$\begin{aligned} \text{Improvement of option 1 relative to base} &= (E0 - E1)/E0 \\ &= (E0 - 0.825 E0)/E0 \\ &= 0.175 \\ &\text{17.5 \% improvement} \end{aligned}$$

Another way of understanding the effect of an improvement is to take into perspective the extent of impact that change has on the execution time. For example, the change may have an impact only on a portion of the execution time. A law associated with Gene Amdahl, a pioneer in parallel computing can be adapted for capturing this notion:

Amdahl's law:

$$\text{Time}_{\text{after}} = \text{Time}_{\text{unaffected}} + \text{Time}_{\text{affected}}/x \quad (6)$$

In the above equation, Time_{after} the total execution time as a result of a change is the sum of the execution time that is unaffected by the change (Time_{unaffected}), and the ratio of the affected time (Time_{affected}) to the extent of the improvement (denoted by x). Basically, Amdahl's law suggests an engineering approach to improving processor performance, namely, spend resources on critical instructions that will have the maximum impact on the execution time.

Name	Notation	Units	Comment
Memory footprint	-	Bytes	Total space occupied by the program in memory
Execution time	$(\sum CPI_j) * \text{clock cycle time, where } 1 \leq j \leq n$	Seconds	Running time of the program that executes n instructions
Arithmetic mean	$(E_1+E_2+\dots+E_p)/p$	Seconds	Average of execution times of constituent p benchmark programs
Weighted Arithmetic mean	$(f_1*E_1+f_2*E_2+\dots+f_p*E_p)$	Seconds	Weighted average of execution times of constituent p benchmark programs
Geometric mean	$p^{\text{th}} \text{ root } (E_1*E_2*\dots*E_p)$	Seconds	p^{th} root of the product of execution times of p programs that constitute the benchmark
Harmonic mean	$1/\left(\left(\frac{1}{E_1} + \frac{1}{E_2} + \dots + \frac{1}{E_p} \right) / p \right)$	Seconds	Arithmetic mean of the reciprocals of the execution times of the constituent p benchmark programs
Static instruction frequency		%	Occurrence of instruction i in compiled code
Dynamic instruction frequency		%	Occurrence of instruction i in executed code
Speedup (M_A over M_B)	E_B/E_A	Number	Speedup of Machine A over B
Speedup (improvement)	$E_{\text{Before}}/E_{\text{After}}$	Number	Speedup After improvement
Improvement in Exec time	$(E_{\text{old}}-E_{\text{new}})/E_{\text{old}}$	Number	New Vs. old
Amdahl's law	$\text{Time}_{\text{after}} = \text{Time}_{\text{unaffected}} + \text{Time}_{\text{affected}}/x$	Seconds	x is amount of improvement

Table 5.1: Summary of Performance Metrics

Table 5.1 summarizes all the processor-related performance metrics that we have discussed so far.

Example 7:

A processor spends 20% of its time on add instructions. An engineer proposes to improve the *add* instruction by 4 times. What is the speedup achieved because of the modification?

Answer:

The improvement only applies for the Add instruction, so **80% of the execution time is unaffected by the improvement**.

Original normalized execution time = 1

$$\begin{aligned}\text{New execution time} &= (\text{time spent in add instruction}/4) + \text{remaining execution time} \\ &= 0.2/4 + 0.8 \\ &= 0.85\end{aligned}$$

$$\begin{aligned}\text{Speedup} &= \text{Execution time before improvement}/\text{Execution time after improvement} \\ &= 1/0.85 = 1.18\end{aligned}$$

5.6 Increasing the Throughput of the Processor

Thus far, we focused on techniques to reduce the latency of individual instructions to improve processor performance. A radically different approach to improving processor performance is not to focus on the *latency* for individual instructions (i.e., the CPI metric) but on *throughput*. That is, the number of instructions executed by the processor per unit time. Latency answers the question, how many clock cycles does the processor take to execute an individual instruction (i.e., CPI). On the other hand, throughput answers the question, how many instructions does the processor execute in each clock cycle (i.e., **IPC** or *instructions per clock cycle*). The concept called **pipelining** is the focus of the rest of this chapter.

5.7 Introduction to Pipelining

Welcome to Bill's Sandwich shop! He has a huge selection of breads, condiments, cheese, meats, and veggies to choose from in his shop, all neatly organized into individual stations. When Bill was starting out in business, he was a one-man team. He took the orders, went through the various stations, and made the sandwich to the specs. Now his business has grown. He has 5 employees one for each of the five stations involved with the sandwich assembly: taking the order, bread and condiments selection, cheese selection, meat selection, and veggies selection. The following figure shows the sandwich assembly process.

<u>Station 1 (place order)</u>	<u>station II (select bread)</u>	<u>station III (cheese)</u>	<u>station IV (meat)</u>	<u>station V (veggies)</u>
New (5 th order)	4 th order	3 rd order	2 nd order	1 st order

Each station is working on a different order; while the last station (station V) is working on the very first order, the first station is taking a new order for a sandwich. Each station after doing “its thing” for the sandwich passes the partially assembled sandwich to the next station along with the order. Bill is a clever manager. He decided against dedicating an employee to any one customer. That would require each employee to have his/her own stash of *all* the ingredients needed to make a sandwich and increase the inventory of raw materials unnecessarily since each customer may want only a subset of the ingredients. Instead, he carefully chose the work to be done in each station to be roughly the same, so that no employee is going to be twiddling his thumbs. Of course, if a particular sandwich order does not need a specific ingredient (say cheese) then the corresponding station simply passes on the partially assembled sandwich on to the next station. Nevertheless, most of the time (especially during peak time) all his employees are kept busy rolling out sandwiches in rapid succession.

Bill has quintupled the rate at which he can serve his customers.

5.8 Towards an instruction processing assembly line

You can see where we are going with this...in the simple implementation of LC-2200, the FSM executes one instruction at a time taking it all the way through the FETCH, DECODE, EXECUTE macro states before turning its attention to the next instruction. The problem with that approach is that the datapath resources are under-utilized. This is because, for each macro state, not all the resources are needed. Let us not worry about the specific datapath we used in implementing the LC-2200 ISA in Chapter 3. Regardless of the details of the datapath, we know that any implementation of LC-2200 ISA would need the following datapath resources: Memory, PC, ALU, Register file, IR, and sign extender. Figure 5.1 shows the hardware resources of the datapath that are in use for each of the macro states for a couple of instructions:

Macro State	Datapath Resources in Use				
FETCH	IR	ALU	PC	MEM	
DECODE	IR				
EXECUTE (ADD)	IR	ALU	Reg-file		
EXECUTE (LW)	IR	ALU	Reg-file	MEM	Sign extender

Figure 5.1: Datapath resources in use for different macro states

We can immediately make the following two observations:

1. The IR is in use in every macro state. This is not surprising since IR contains the instruction, and parts of the IR are used in different macro states of its execution. IR is equivalent to the “order” being passed from station to station in the sandwich assembly line.
2. At the macro level, it can be seen that a different amount of work is being done in the three-macro states. Each state of the FSM is equivalent to a station in the sandwich assembly line.

What we want to do is to use all the hardware resources in the datapath all the time if possible. In our sandwich assembly line, we kept all the stations busy by assembling multiple sandwiches simultaneously. We will try to apply the sandwich assembly line idea to instruction execution in the processor.

A program is a sequence of instructions as shown in Figure 5.2:

```

I1: Ld R1, MEM [1000]; R1 <- Memory at location 1000
I2: Ld R2, MEM [2000]; R2 <- Memory at location 2000
I3: Add R3, R5, R4; R3 <- R4 + R5
I4: Nand R6, R7, R8; R6 <- R7 NAND R8
I5: St R9, MEM [3000]; R9 -> Memory at location 3000
I6: .....
I7: .....
I8
I9
I10
I11
I12
I13

```

Figure 5.2: A Program is a sequence of instructions

With the simple implementation of the FSM, the time line for instruction execution in the processor will look as shown in Figure 5.3-(a):

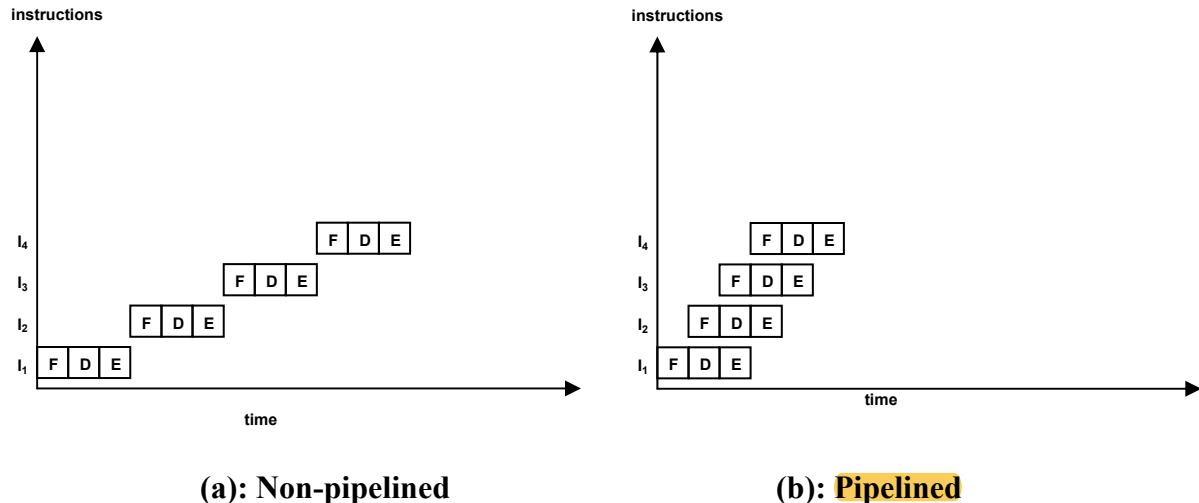


Figure 5.3: Execution Timeline

A new instruction processing starts only after the previous one is fully completed. Using our sandwich assembly line idea, to maximize the utilization of the datapath resources, we should have multiple instructions in execution in our instruction *pipeline* as shown in Figure 5.3-(b). A question that arises immediately is if this is even possible. In a

sandwich assembly line, assembling your sandwich is completely independent of what your predecessor or your successor in the line wants for his/her sandwich. Are the successive instructions in a program similarly independent of one another? “No” is the immediate answer that comes to mind since the **program is sequential**. However, look at the sequence of instructions shown in Figure 5.2. Even though it is a sequential program, one can see that instructions I_1 through I_5 happen to be independent of one another. That is, the **execution of one instruction does not depend on the results of the execution of the previous instruction in this sequence**. We will be quick to point out this happy state of affairs is not the norm, and we will deal with such **dependencies in a processor pipeline in a later section** (please see [Section 5.13](#)). For the moment, it is convenient to think that the **instructions in sequence are independent of one another to illustrate the possibility of adopting the sandwich assembly line idea for designing the processor pipeline**.

Notice that in the pipeline execution timeline shown in Figure 5.3-(b), we are starting the processing of a new instruction as soon as the previous instruction moves to the next macro state. If this were possible, **we can triple the throughput of instructions processed**. The key observation is that the datapath resources are akin to individual ingredients in the sandwich assembly line, and the macro states are akin to the employees at specific stations of the sandwich assembly line.

5.9 Problems with a simple-minded instruction pipeline

Notice several problems with the above simple-minded application of the assembly line idea to the instruction pipeline.

1. **The different stages often need the same datapath resources (e.g. ALU, IR).**
2. The amount of work done in the **different stages is not the same**. For example, compare the work done in DECODE and EXECUTE-LW states in Figure 5.1. The former is simply a combinational function to determine the type of instruction and the resources needed for it. On the other hand, the latter involves address arithmetic, memory access, and writing to a register file. **In general, the work done in the EXECUTE state will far outweigh the work done in the other stages.**

The first point implies that **we have resource contention among the stages**. This is often referred to as a **structural hazard**, and is a result of the **limitations in the datapath such as having a single IR, a single ALU, and a single bus to connect the datapath elements**. In the sandwich assembly line the **individual order** (equivalent of the contents of IR) is on a piece of paper that is passed from station to station; and the partially assembled sandwich (equivalent of an instruction in partial execution) is passed directly from one station to the next (i.e., there is no central “bus” as in our simple datapath). We can use similar ideas to fix this problem **with our instruction pipeline**. For example, **if we add an extra ALU, an extra IR, and an extra MEM to the datapath dedicated to the FETCH stage then that stage will become independent of the other stages**. An extra ALU is understandable since both the FETCH and EXECUTE stages need to use it, but we need to understand what it means to have an extra IR. Just as the order is passed from station to station in the sandwich assembly line, we are passing the **contents of the IR from the FETCH to the DECODE stage** and so on to **keep the stages independent of one another**. Similarly, we split the total memory into an **I-MEM** for instruction memory and **D-MEM** for data

memory to make the **FETCH** stage independent of the **EXECUTION** stage. As should be evident, **instructions of the program** come from the **I-MEM** and the **data structures** that the program manipulates come from the **D-MEM**. This is a reasonable split since (a) it is a good programming practice to keep the two separate, and (b) most modern processors (e.g., memory segmentation in Intel x86 architecture) separate the memory space into distinct regions to ensure inadvertent modifications to instruction memory by a program.

The second point implies that the time needed for each stage of the instruction pipeline is different. Recall, that Bill carefully engineered his sandwich assembly line to ensure that each employee did the same amount of work in each station. The reason is the slowest member of the pipeline limits the throughput of the assembly line. Therefore, for our instruction assembly line to be as efficient, **we should break up the instruction processing so that each stage does roughly the same amount of work.**

5.10 Fixing the problems with the instruction pipeline

Let us look at the **DECODE** stage. This stage does the **least amount of work in the current setup**. To spread the work more evenly, we have to assign more work to this stage. We have a dilemma here. We cannot really do anything until we know what the instruction is which is the focus of this stage. However, we can do something opportunistically so long as the semantics of the actual instruction is not affected.

We expect most **instructions to use register contents**. Therefore, we can go ahead and read the register contents from the register file **without actually knowing what the instruction is**. In the worst case, we may end up **not using the values read from the registers**. However, to do this we need to know which registers to read. This is where the instruction format chosen during instruction-set design becomes crucial. If you go back (Chapter 2) and look at the instructions that use registers in LC-2200 (ADD, NAND, BEQ, LW, SW, and JAL) you will find the source register specifiers for arithmetic/logic operations or for address calculations occupy **always the same bit positions in the instruction format**. We can **exploit this fact and opportunistically read the registers as we are decoding what the actual instruction is**. In a similar vein, we can break up the **EXECUTE** stage which potentially does a lot of work into smaller stages.

Using such reasoning, let us **subdivide** the processing of an instruction into the following five functional components or stages:

- **IF:** This stage fetches the instruction pointed to by the PC from I-MEM and places it into IR. It also increments the current PC in readiness for fetching the next instruction.
- **ID/RR:** This stage decodes the instruction and reads the register files to pull out two source operands (more than what may actually be needed depending on the instruction). To enable this functionality, the register file has to be **dual-ported**, meaning that two register addresses can be given simultaneously and two register contents can be read out in the **same clock cycle**. We will call such a register file dual-ported register file (DPRF). Since this stage contains the logic for decoding the instruction as well as reading the register file, we have given it a hybrid name.

- **EX:** This is the stage that does all the arithmetic and/or logic operations that are needed for processing the instruction. We will see shortly (Section 5.11) that one ALU may not suffice in the EX stage to cater to the needs of all the instructions.
- **MEM:** This stage either reads from or writes to the D-MEM for LW and SW instructions, respectively. Instructions that do not have a memory operand will not need the operations performed in this stage.
- **WB:** This stage writes the appropriate destination register (R_x) if the instruction requires it. Instructions in LC-2200 that require writing to a destination register include arithmetic and logic operations as well as load.

Pictorially the passage of an instruction through the pipeline is shown in Figure 5.4.

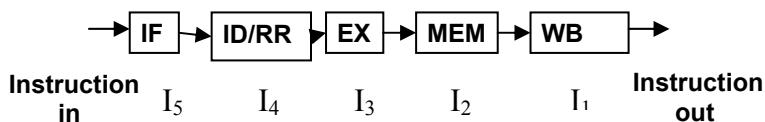


Figure 5.4: Passage of instructions through the pipeline

Similar to the sandwich assembly line, we expect every instruction to pass through each of these stages in the course of the processing. At any point of time, five instructions are in execution in the pipeline: when Instruction I_1 is in the WB stage, instruction I_5 is in the IF stage. This is a synchronous pipeline in the sense that on each clock pulse the partial results of the instruction execution is passed on to the next stage. The inherent assumption is that the clock pulse is wide enough to allow the slowest member of the pipeline to complete its function within the clock cycle time.

Every stage works on the partial results generated in the *previous clock cycle* by the preceding stage. Not every instruction needs every stage. For example, an ADD instruction does not need the MEM stage. However, this simply means that the MEM stage does nothing for one clock cycle when it gets the partial results of an ADD instruction (one can see the analogy to the sandwich assembly line). One might think that this is an inefficient design if you look at the passage of specific instructions. For example, this design adds one more cycle to the processing of an ADD instruction.

However, the goal of the pipelined design is to increase the throughput of instruction processing and not reduce the latency for each instruction. To return to our sandwich assembly line example, the analogous criterion in that example was to keep the customer line moving. The number of customers served per unit time in that example is analogous to the number of instructions processed per unit time in the pipelined processor.

Since each stage is working on a different instruction, once a stage has completed its function it has to place the results of its function in a well-known place for the next stage to pick it up in the next clock cycle. This is called buffering the output of a stage. Such buffering is essential to give the independence for each stage. Figure 5.5 shows the instruction pipeline with the buffers added between the stages. *Pipeline register* is the term commonly used to refer to the buffer between the stages. We will use pipeline

register and buffer interchangeably in this chapter. In our sandwich assembly line example, the partially assembled sandwich serves as the buffer and gives independence and autonomy to each stage.

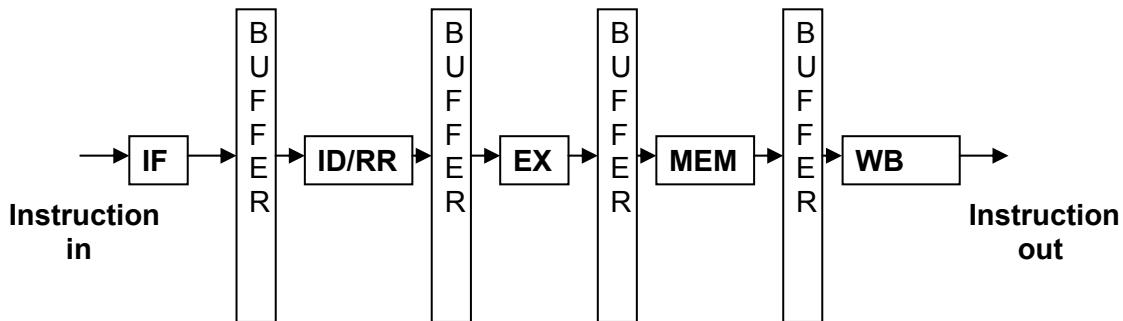


Figure 5.5: Instruction pipeline with buffers between stages

5.11 Datapath elements for the instruction pipeline

The next step is to decide the datapath elements needed for each stage of the instruction pipeline and the contents of the buffers between the stages to provide the isolation between them.

- In the **IF stage** we need a **PC**, an **ALU**, and **I-MEM**. The output of this stage is the **instruction fetched from the memory**; therefore, the pipeline register between IF and ID/RR stages should contain the **instruction**.
- In the **ID/RR stage**, we need the **DPRF**. The output of this stage are the contents of the **registers read from the register file** (call them A and B); and the results of **decoding the instruction** (**opcode** of the instruction, and **offset** if needed by the instruction). These constitute the contents of the pipeline register between ID/RR and EX stages.
- The **EX stage** performs any needed **arithmetic** for an instruction. Since this is the only stage that performs all the arithmetic for an instruction execution, we need to determine the **worst-case resource need for this stage**. This depends on the **instruction-set repertoire**.

In our case (LC-2200), the only instruction that required more than one arithmetic operation is the **BEQ instruction**. BEQ requires one ALU to do the comparison ($A == B$) and another to do the **effective address computation (PC + signed-offset)**. Therefore, **we need two ALU's for the EX stage**.

BEQ instruction also brings out another requirement. The address arithmetic requires the **value of the PC** that corresponds to the **BEQ instruction** (Incidentally, the value of PC is required for another instruction as well, namely, JALR). Therefore, the **value of PC should also be communicated from stage to stage** (in addition to the other things) through the pipeline.

The output of the EX stage is the result of the arithmetic operations carried out and therefore instruction specific. The contents of the pipeline register between EX and

MEM stage depends on such specifics. For example, if the instruction is an ADD instruction, then the pipeline register will contain the result of the ADD, the opcode, and the destination register specifier (Rx). We can see that the PC value is not required after the EX stage for any of the instructions. Working out the details of the pipeline register contents for the other instructions is left as an exercise to the reader.

- The MEM stage requires the D-MEM. If the instruction processed by this stage in a clock cycle is not LW or SW, then the contents of the input buffer is simply copied to the output buffer at the end of the clock cycle. For LW instruction, the output buffer of the stage contains the D-MEM contents read, the opcode, and the destination register specifier (Rx); while for SW instruction the output buffer contains the opcode. With a little of reflection, it is easy to see that no action is needed in the WB stage if the opcode is SW.
- The WB stage requires the register file (DPRF). This stage is relevant only for the instructions that write to a destination register (such as LW, ADD, and NAND). This poses an interesting dilemma. In every clock cycle we know every stage is working on a different instruction. Therefore, with reference to Figure 5.4, at the same time that WB is working on I_1 , ID/RR is working on I_4 . Both these stages need to access the DPRF simultaneously on behalf of different instructions (for example: I_1 may be ADD R1, R3, R4 and I_4 may be NAND R5, R6, R7). Fortunately, WB is writing to a register while ID/RR is reading registers. Therefore, there is no conflict in terms of the logic operation being performed by these two stages, and both can proceed simultaneously as long as the same register is not being read and written to in a given cycle. It is a semantic conflict when the same register is being read and written to in the same cycle (for example consider I_1 as ADD R1, R3, R4 and I_4 as ADD R4, R1, R6). We will address such semantic conflicts shortly (Section 5.13.2).

Pictorially, we show the new organization of the resources for the various stages in Figure 5.6. Note that DPRF in ID/RR and WB stages refer to the same logic element in the datapath. In Figure 5.6, both the stages include DPRF just for clarity of the resources needed in those stages.

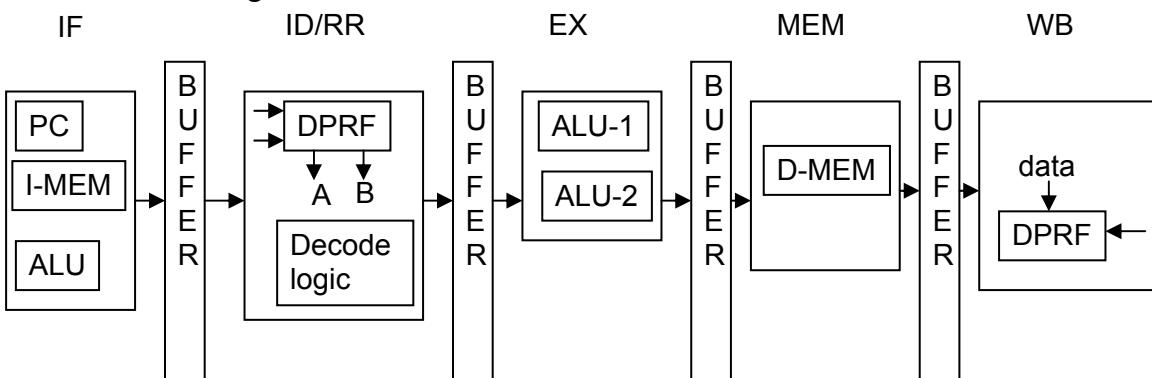


Figure 5.6: Organization of hardware resources for various stages

Some things to note with our pipelined processor design. In the steady state, there are 5 instructions **in different stages of processing in the pipeline**. We know in the simple design of LC-2200, the FSM transitioned through microstates such as **ifetch1**, **ifetch2**, etc. In a given clock cycle, the processor was in **exactly one state**. In a pipelined implementation, the processor is simultaneously in all the states represented by the stages of the pipelined design. **Every instruction takes 5 clock cycles to execute**. Each instruction enters the pipeline at the IF stage and is *retired* (i.e., completed successfully) from the WB stage. **In the ideal situation, the pipelined processor retires one new instruction in every clock cycle**. Thus, the effective CPI of the pipelined processor is 1. Inspecting Figure 5.6, one might guess that MEM stage could be the slowest. However, the answer to this question is more involved. We discuss considerations in modern processors with respect to reducing the clock cycle time in Section 5.15.

The performance of pipelined processor is crucially dependent on the memory system. Memory access time is the most dominant latency in a pipelined processor. To hide this latency, processors employ caches. Recall the toolbox and tool tray analogy from Chapter 2. We mentioned that registers serve the role of the tool tray in that we *explicitly* bring the memory values the processor needs into the registers using the load instruction. Similarly, caches serve as *implicit* tool trays. In other words, whenever the processor brings something from the memory (instruction or data), it is implicitly placed into a high-speed storage inside the processor called caches. By creating an implicit copy of the memory location in the cache, the processor can subsequently re-use values in these memory locations without making trips to the memory. We will discuss caches in much more detail in a later chapter (please see Chapter 9) together with their effect on pipelined processor implementation. For the purpose of this discussion on pipelined processor implementation, we will simply say that caches allow hiding the memory latency and make pipelined processor implementation viable. In spite of the caches and high-speed registers, the combinational logic delays (ALU, multiplexers, decoders, etc.) are significantly smaller than access to caches and general-purpose registers. In the interest of making sure that all stages of the pipeline have roughly the same latency, modern processor implementation comprises much more than five stages. For example, access to storage elements (cache memory, register file) may take multiple cycles in the processor pipeline. We will discuss such issues in a later section (please see Section 5.15). Since, this is the first introduction to pipelined implementation of a processor, we will keep the discussion simple.

5.12 Pipeline-conscious architecture and implementation

The key points to note in designing a pipeline-conscious architecture are:

- **Need for a symmetric instruction format:** This has to do with ensuring that the locations of certain fields in the instruction (e.g. register specifiers, size and position of offset, etc.) remain unchanged independent of the instruction. As we saw, this was a key property we exploited in the ID/RR stage for LC-2200.
- **Need to ensure equal amount of work in each stage:** This property ensures that the clock cycle time is optimal since the slowest member of the pipeline determines it.

Figure 5.6a shows the full datapath for a pipelined LC-2200 implementation.

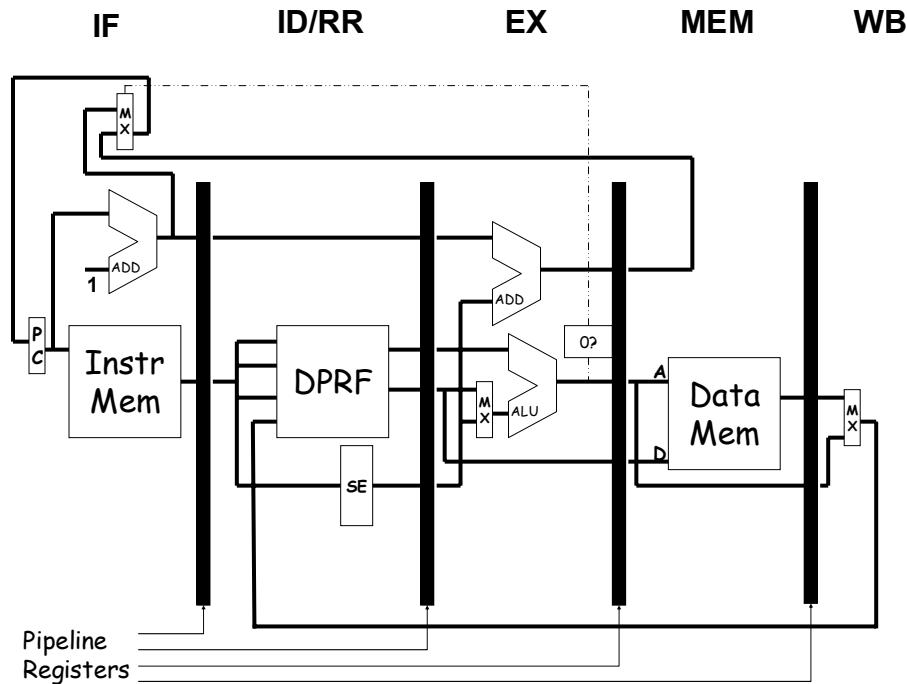


Figure 5.6a: Datapath for a pipelined LC 2200

5.12.1 Anatomy of an instruction passage through the pipeline

In this subsection, we will trace the passage of an instruction through the 5-stage pipeline. We will denote the buffers between the stages with unique names as shown in Figure 5.6b.

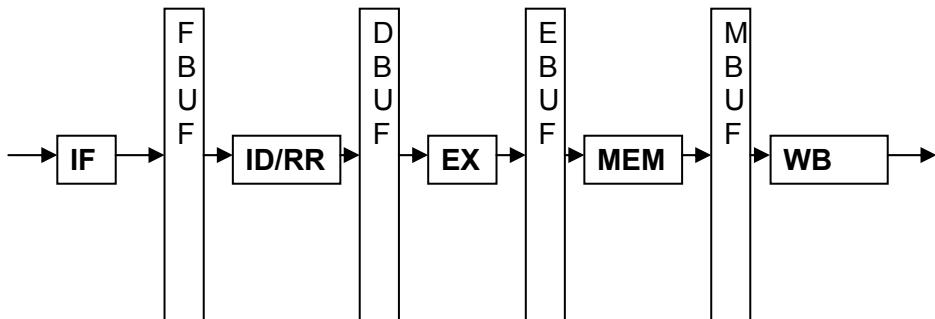


Figure 5.6b: Pipeline registers with unique names

Table 5.2 summarizes the function of each of the pipeline buffers between the stages.

Name	Output of Stage	Contents
FBUF	IF	Primarily contains instruction read from memory
DBUF	ID/RR	Decoded IR and values read from register file
EBUG	EX	Primarily contains result of ALU operation plus other parts of the instruction depending on the instruction specifics
MBUF	MEM	Same as EBUG if instruction is not LW or SW; If instruction is LW, then buffer contains the contents of memory location read

Table 5.2: Pipeline Buffers

Let us consider the passage of the Add instruction that has the following syntax and format:

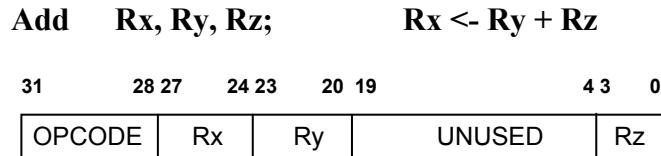


Figure 5.6c: Syntax and format of LC 2200 Add instruction

Each stage performs the actions summarized below to contribute towards the execution of the Add instruction:

IF stage (cycle 1):

```
I-MEM[PC] -> FBUF // The instruction at memory address given by PC is
                      fetched and placed in FBUF (which is essentially
                      the IR); the contents of FBUF after this action will
                      be as shown in Figure 5.6c
PC + 1 -> PC // Increment PC
```

ID/RR stage (cycle 2):

```
DPRF[ FBUF[Ry] ] -> DBUF [A]; // read Ry into DBUF[A]
DPRF[ FBUF[Rz] ] -> DBUF [B]; // read Rz into DBUF[B]
FBUF[OPCODE] -> DBUF[OPCODE]; // copy opcode from FBUF to
                                DBUF
```

FBUF[Rx] → DBUF[Rx]; // copy Rx register specifier from FBUF to DBUF

EX stage (cycle 3):

DBUF[A] + DBUF [B] → EBUF[Result]; // perform addition
DBUF[OPCODE] → EBUF[OPCODE]; // copy opcode from DBUF to EBUF
DBUF[Rx] → EBUF[Rx]; // copy Rx register specifier from DBUF to EBUF

MEM stage (cycle 4):

DBUF → MBUF; // The MEM stage has nothing to contribute towards the execution of the Add instruction; so simply copy the DBUF to MBUF

WB stage (cycle 5):

MBUF[Result] → DPRF [MBUF[Rx]]; // write back the result of the addition into the register specified by Rx

Example 8:

Considering only the Add instruction, quantify the sizes of the various buffers between the stages of the above pipeline.

Answer:

Size of FBUF (same as the size of an instruction in LC-2200) = **32 bits**

Size of DBUF:

Size of contents of Ry register in DBUF[A] = 32 bits
Size of contents of Rz register in DBUF[B] = 32 bits
Size of opcode in DBUF[opcode] = 4 bits
Size of Rx register specifier in DBUF[Rx] = 4 bits
Total size (sum of the above fields) = **72 bits**

Size of EBUF:

Size of result of addition in EBUF[result] = 32 bits
Size of opcode in EBUF[opcode] = 4 bits
Size of Rx register specifier in EBUF[Rx] = 4 bits
Total size (sum of the above fields) = **40 bits**

Size of MBUF (same as EBUF) = **40 bits**

5.12.2 Design of the Pipeline Registers

While we looked at the passage of a single instruction through the pipeline in the previous section, it should be clear that each stage of the pipeline is working on a different instruction in each clock cycle. The reader should work out the actions taken by each stage of the pipeline for the different instructions in LC-2200 (see problems at the end of this chapter). This exercise is similar to the design of the FSM control sequences that we undertook in Chapter 3 for the sequential implementation of LC-2200. Once such a design is complete, then the size of the pipeline register for each stage can be determined as the maximum required for the passage of any instruction. The pipeline register at the output of the ID/RR stage will have the maximal content since we do not yet know what the instruction is. The interpretation of the contents of a pipeline register will depend on the stage and the opcode of the instruction that is being acted upon by that stage.

A generic layout of the pipeline register is as shown in Figure 5.6d. Opcode will always occupy the same position in each pipeline register. In each clock cycle, each stage interprets the remaining fields of the pipeline register at its input based on the opcode and takes the appropriate datapath actions (similar to what we detailed for the Add instruction in Section 5.12.1).

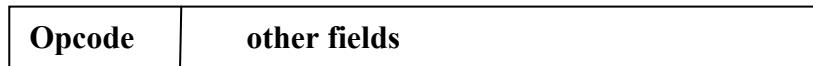


Figure 5.6d: A generic layout of a pipeline register

Example 9:

Design the DBUF pipeline register for LC-2200. Do not attempt to optimize the design by overloading the different fields of this register.

Answer:

DBUF has the following fields:

Opcode (needed for all instructions)	4 bits
A (needed for R-type)	32 bits
B (needed for R-type)	32 bits
Offset (needed for I-type and J-type)	20 bits
PC value (needed for BEQ)	32 bits
Rx specifier (needed for R-, I-, and J-type)	4 bits

Layout of the DBUF pipeline register:

Opcode	A	B	Offset	PC	Rx
4 bits	32 bits	32 bits	20 bits	32 bits	4 bits

5.12.3 Implementation of the stages

In a sense, the design and implementation of a pipeline processor may be simpler than a non-pipelined processor. This is because the pipelined implementation modularizes the design. This modularity leads to the same design advantages that accrue in writing a large software system as a composite of several smaller modules. The layout and interpretation of the pipeline registers are analogous to well-defined interfaces between components of a large software system. Once we complete the layout and interpretation of the pipeline registers, we can get down to the datapath actions needed for each stage in complete isolation of the other stages. Further, since the datapath actions of each stage happen in one clock cycle, the design of each stage is purely combinational. At the beginning of each clock cycle, each stage interprets the input pipeline register, carries out the datapath actions using the combinational logic for this stage, and writes the result of the datapath action into its output pipeline register.

Example 10:

Design and implement the datapath for the ID/RR stage of the pipeline to implement the LC-2200 instruction set. You can use any available logic design tool to do this exercise.

Answer:

Figures 5.6, 5.6a, and 5.6b are the starting points for solving this problem. The datapath elements have to be laid out. Using the format of the instruction the register files have to be accessed and put into the appropriate fields of DBUF. The offset and the opcode fields of the instruction have to be copied from FBUF to DBUF. The PC value has to be put into the appropriate field of DBUF. Completing this example is left as an exercise for the reader.

5.13 Hazards

Though the sandwich assembly line serves as a good analogy for the pipelined processor, there are several twists in the instruction pipeline that complicate its design. These issues are summed up as *pipeline hazards*. Specifically, there are three types of hazards: **structural, data, control**.

As we will see shortly, the effect of all these hazards is the same, namely, **reduce** the pipeline efficiency. In other words, the pipeline will execute less than one instruction every clock cycle. Recall, however, that the pipeline is synchronous. That is, in each clock cycle every stage is working on an instruction that has been placed on the pipeline register by the preceding stage. Just like air bubbles in a water pipe, if a stage is not ready to send a valid instruction to the next stage, it should place the equivalent of an “air bubble”, a dummy instruction that does nothing, in the pipeline register. We refer to this as a *NOP (No-Operation)* instruction.

We will include such a NOP instruction in the repertoire of the processor. In the subsequent discussions on hazards, we will see how the hardware automatically generates such NOPs when warranted by the hazard encountered. However, this need not be the case always. By exposing the structure of the pipeline to the software, we can make the system software, namely, the compiler, responsible for including such NOP instructions in the source code itself. We will return to this possibility later (see Section 5.13.4).

Another way of understanding the effect of bubbles in the pipeline is that the average CPI of instructions goes above 1. It is important to add a cautionary note to the use of the CPI metric. Recall that the execution time of a program in clock cycles is the product of CPI and the number of instructions executed. Thus, CPI by itself does not tell the whole story of how good an architecture or the implementation is. In reality, the compiler and the architecture in close partnership determine the program execution time. For example, unoptimized code that a compiler generates may have a lower CPI than the optimized code. However, the execution time may be much more than the optimized code. The reason is the optimization phase of the compiler may have gotten rid of a number of useless instructions in the program thus reducing the total number of instructions executed. But this may have come at the cost of increasing the three kinds of hazards and hence the average CPI of the instructions in the optimized code. Yet, the net effect of the optimized code may be a reduction in program execution time.

5.13.1 Structural hazard

We already alluded to the structural hazard. This comes about primarily due to the limitations in the hardware resources available for concurrent operation of the different stages. For example, a single data bus in the non-pipelined version is a structural hazard for a pipelined implementation. Similarly, a single ALU is another structural hazard. There are two solutions to this problem: live with it or fix it. If the hazard is likely to occur only occasionally (for some particular combinations of instructions passing through the pipeline simultaneously) then it may be prudent not to waste additional hardware resources to fix the problem. As an example, let us assume that our pointy-haired manager has told us we can use only one ALU in the EX unit. So every time we encounter the BEQ instruction, we spend two clock cycles in the EX unit to carry out the two arithmetic operations that are needed, one for the comparison and the other for the address computation. Of course, the preceding and succeeding stages should be made aware that the EX stage will occasionally take two cycles to do its operation. Therefore, the EX unit should tell the stages preceding it (namely, IF and ID/RR) not to send a new instruction in the next clock cycle. Basically there is a *feedback line* that each of the preceding stages looks at to determine if they should just “pause” in a given cycle or do something useful. If a stage decides to “pause” in a clock cycle it just does not change anything in the output buffer.

The succeeding stages need to be handled differently from the preceding stages. Specifically, the EX stage will write a NOP opcode in its output buffer for the opcode field. NOP instruction is a convenient way of making the processor execute a “dummy” instruction that has no impact on the actual program under execution. Essentially, via the

NOP instruction, we have introduced a “bubble” in the pipeline between the BEQ and the instruction that preceded it.

The passage of a BEQ instruction is pictorially shown in a series of timing diagrams in Figure 5.7. We pick up the action from cycle 2, when the BEQ instruction is in the ID/RR stage. A value of “STAY” (equal to binary 1) on the feedback line tells the preceding stage to remain in the same instruction and not to send a new instruction at the end of this cycle.

Cycle 2

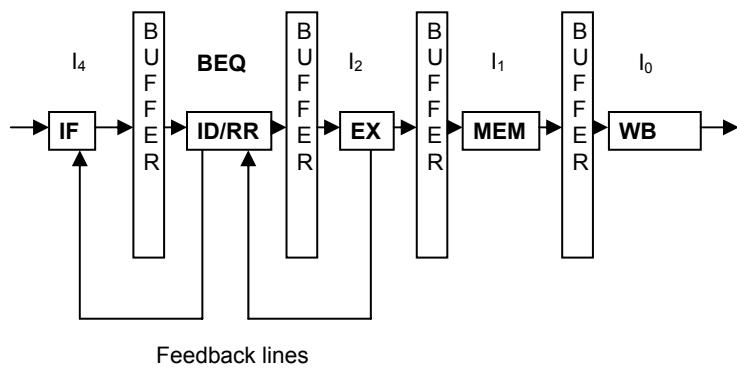


Figure 5.7 (a)

Cycle 3

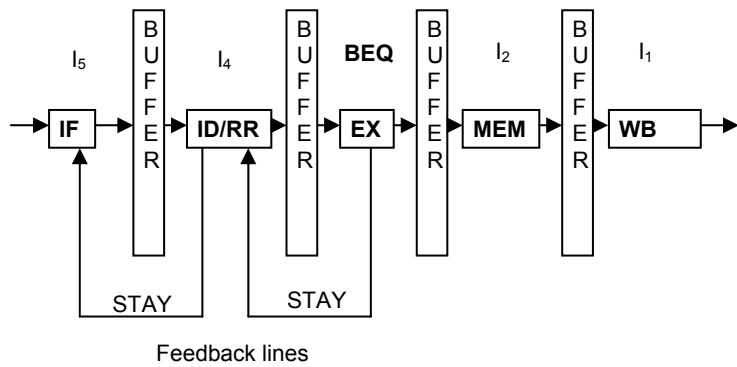


Figure 5.7 (b)

Cycle 4

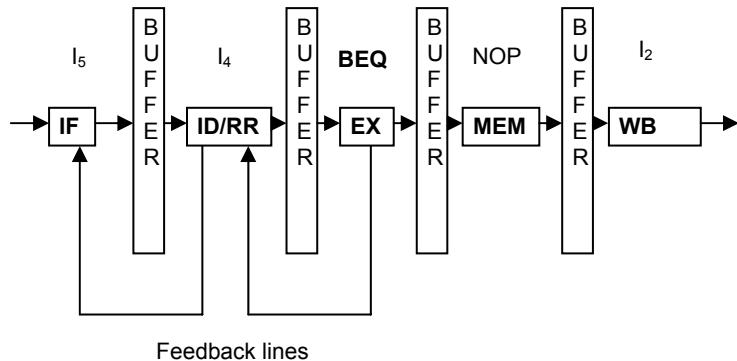


Figure 5.7 (c)

Cycle 5

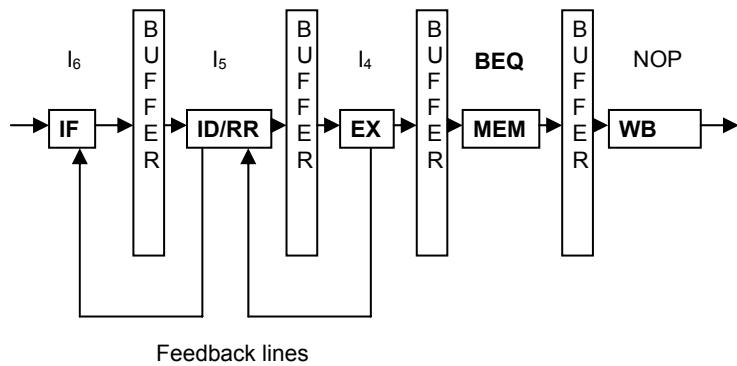


Figure 5.7 (d)

Figure 5.7: Illustration of structural Hazard

Essentially, such a “bubble” in the pipeline brings down the pipeline efficiency since the effective CPI goes above 1 due to the bubble.

If on the other hand, it is determined that such an inefficiency is unacceptable then we can fix the structural hazard by throwing hardware at the problem. This is what we did in our earlier discussion when we added an extra ALU in the EX stage to overcome this structural hazard.

A quick note on terminology:

- The pipeline is *stalled* when an instruction cannot proceed to the next stage.
- The result of such a stall is to introduce a *bubble* in the pipeline.
- A NOP instruction is the manifestation of the bubble in the pipeline. Essentially a stage executing a NOP instruction does nothing for one cycle. Its output buffer remains unchanged from the previous cycle.

You may notice that we use stalls, bubbles, and NOPs interchangeably in the textbook. They all mean the same thing.

5.13.2 Data Hazard

Consider the following two instructions occurring in the execution order of a program:

$I_1: R1 \leftarrow R2 + R3$  $I_2: R4 \leftarrow R1 + R5$

(7)

Figure 5.8a: RAW hazard

I_1 reads the values in registers $R2$ and $R3$ and writes the result of the addition into register $R1$. The next instruction I_2 reads the values in registers $R1$ (written into by the previous instruction I_1) and $R5$ and writes the result of the addition into register $R4$. The situation presents a *data hazard* since there is a dependency between the two instructions. In particular, this kind of hazard is called a *Read After Write (RAW)* data hazard. Of course, the two instructions need not occur strictly next to each other. So long as there is a data dependency between any two instructions in the execution order of the program it amounts to a data hazard.

There are two other kinds of data hazards.

 $I_1: R4 \leftarrow R1 + R5$  $I_2: R1 \leftarrow R2 + R3$

(8)

Figure 5.8b: WAR hazard

The situation presented above is called a *Write After Read (WAR)* data hazard. I_2 is writing a new value to a register while I_1 is reading the old value in the same register.

 $I_1: R1 \leftarrow R4 + R5$  $I_2: R1 \leftarrow R2 + R3$

(9)

Figure 5.8c: WAW hazard

The situation presented above is called a *Write After Write (WAW)* data hazard. I_2 is writing a new value to a register that is also the target of a previous instruction.

These data hazards pose no problem if the instructions execute one at a time in program order, as would happen in a non-pipelined processor. However, they could lead to problems in a pipelined processor as explained in the next paragraph. It turns out that for the simple pipeline we are considering, WAR and WAW hazards do not pose much of a problem, and we will discuss solutions to deal with them in Section 5.13.2.4. First let us deal with the RAW hazard.

5.13.2.1 RAW Hazard

Let us trace the flow of instructions represented by (7) (Figure 5.8(a)) through the pipeline.



I₂ I₁

When I₁ is in the EX stage of the pipeline, I₂ is in ID/RR stage and is about to read the values in registers R1 and R5. This is problematic since I₁ has not computed the new value of R1 yet. In fact, I₁ will write the newly computed value into R1 only in the WB stage of the pipeline. If I₂ is allowed to read the value in R1 as shown in the above picture, it will read to an erroneous execution of the program. We refer to this situation as *semantic inconsistency*, when the intent of the programmer is different from the actual execution. Such a problem would never occur in a non-pipelined implementation since the processor executes one instruction at a time.

The problem may not be as severe if the two instructions are not following one another. For example, consider:

I₁: R1 <- R2 + R3
I_x: R8 <- R6 + R7
I₂: R4 <- R1 + R5

In this case, the pipeline looks as follows:



I₂ I_x I₁

I₁ has completed execution. However, the new value for R1 will be written into the register only when I₁ reaches the WB stage. Thus if there is a RAW hazard for any of the subsequent three instructions following I₁ it will result in a semantic inconsistency.

Example 11:

Consider

I₁: Id R1, MEM

<2 unrelated instructions>

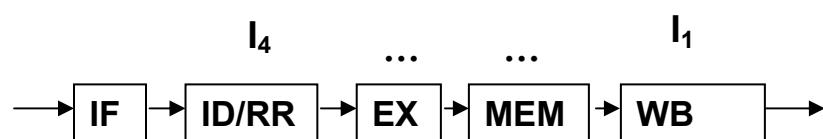
I₄: R4 <- R1 + R5



I₄ and I₁ are separated by two unrelated instructions in the pipeline as shown above. How many bubbles will result in the pipeline due to the above execution?

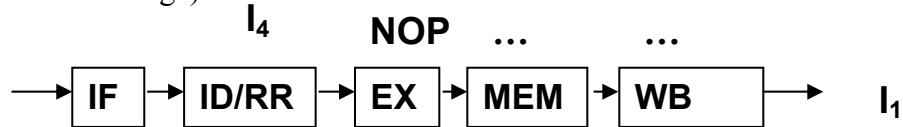
Answer:

The state of the pipeline when I₄ gets to ID/RR stage is shown below:



I_1 will write the value into R1 only at the end of the current cycle. Therefore, I_4 cannot read the register in this cycle and get the correct value.

Thus there is a 1 cycle delay leading to one bubble (NOP instruction passed down from the ID/RR stage to the EX stage).



Example 12:

$I_1: R1 \leftarrow R2 + R3$
 $I_2: R4 \leftarrow R4 + R3$
 $I_3: R5 \leftarrow R5 + R3$
 $I_4: R6 \leftarrow R1 + R6$



As shown above, the sequence of instructions I_1 through I_4 are just about to enter the 5-stage pipeline.

(a)

In the table below, show the passage of these instructions through the pipeline until all 4 instructions have been completed and retired from the pipeline; “retired” from the pipeline means that the instruction is not in any of the 5-stages of the pipeline.

Answer:

Cycle Number	IF	ID/RR	EX	MEM	WB
1	I_1	-	-	-	-
2	I_2	I_1	-	-	-
3	I_3	I_2	I_1	-	-
4	I_4	I_3	I_2	I_1	-
5	-	I_4	I_3	I_2	I_1
6	-	I_4	NOP	I_3	I_2
7	-	-	I_4	NOP	I_3
8	-	-	-	I_4	NOP
9	-	-	-	-	I_4

(b) Assuming that the program contains just these 4 instructions, what is the average CPI achieved for the above execution?

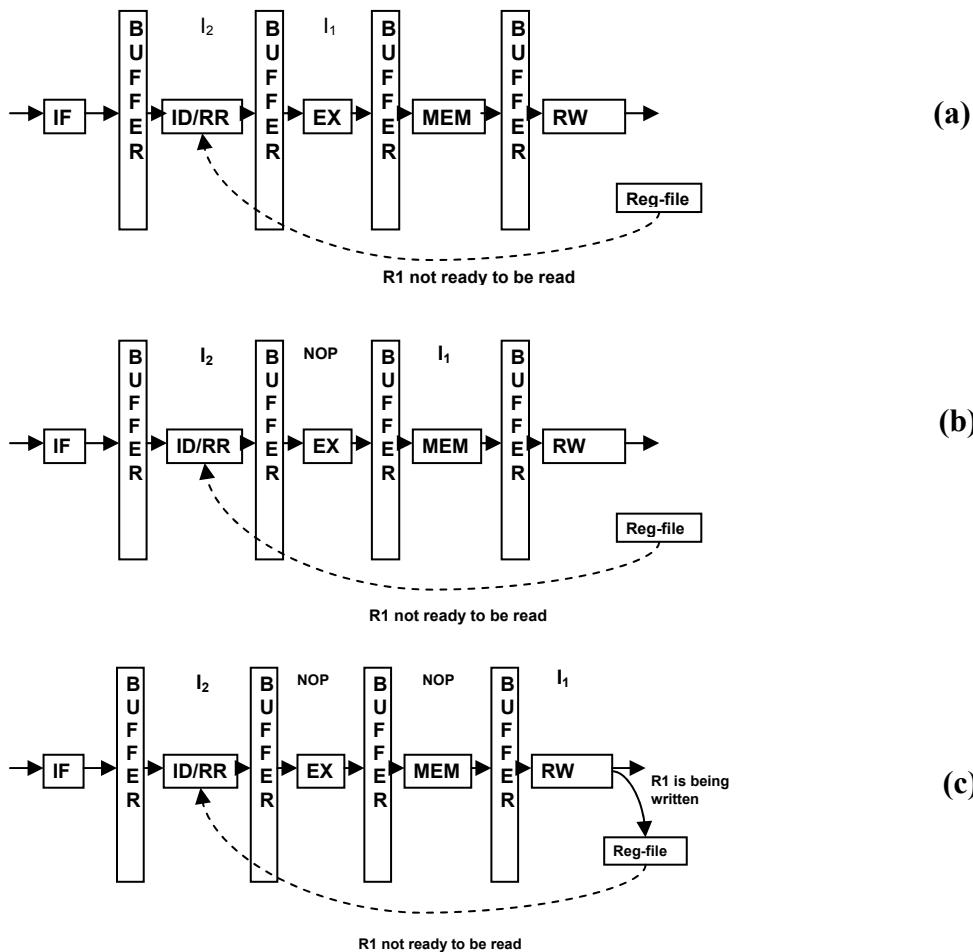
Answer:

Four instructions are retired from the pipeline in 9 cycles. Therefore the average CPI experienced by these instructions:

$$\text{Avg CPI} = 9/4 = 2.25$$

5.13.2.2 Solving the RAW Data Hazard Problem: Data Forwarding

A simple solution to this problem is similar to the handling of the structural hazard. We simply stall the instruction that causes the RAW hazard in the ID/RR stage until the register value is available. In the case of (7) shown in Figure 5.8(a), the ID/RR stage holds I₂ for 3 cycles until I₁ retires from the pipeline. For those three cycles, bubbles (in the form of NOP instructions manufactured by the ID/RR stage) are sent down the pipeline. For the same reason, the preceding stage (IF) is told to stay on the same instruction and not fetch new instructions.



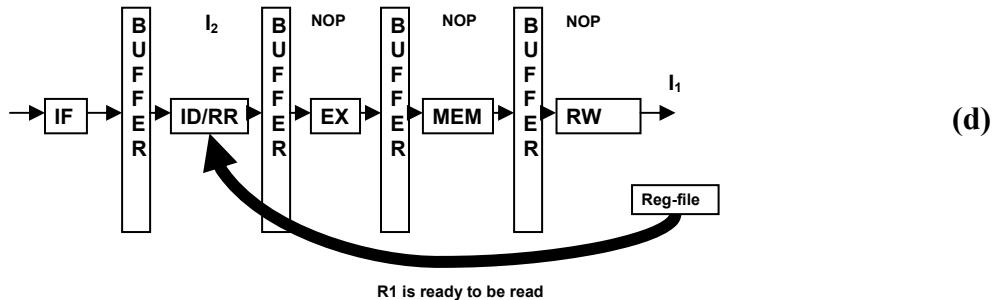
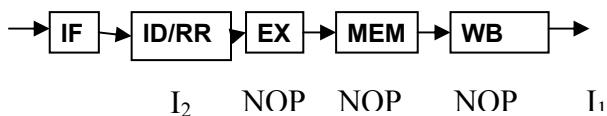


Figure 5.9: RAW hazard due to (7) shown in Figure 5.8a

The series of pictures shown in Figure 5.9 illustrate the stalling of the pipeline due to the data hazard posed by (7) (Figure 5.8a).

In the clock cycle following the picture shown below, normal pipeline execution will start with the IF stage starting to fetch new instructions.



The ID/RR needs some hardware help to know that it has to stall. The hardware for detecting this is very simple. In the register file shown below (Figure 5.10), the **B** bit is the register **busy** bit, and there is one bit per processor register. The ID/RR stage determines the destination register for an instruction and sets the appropriate B bit for that register in the DPRF. I₁ would have set the B bit for register R1. It is the responsibility of the WB stage to clear this bit when it writes the new value into the register file. Therefore, when I₂ gets to ID/RR stage it will find B bit set for R1 and hence stall until the busy condition goes away (3 cycles later).

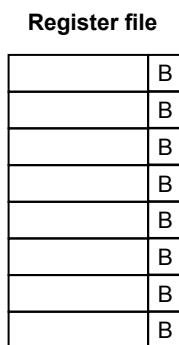


Figure 5.10: Register file with busy bits for each register

Let us see how we can get rid of the stalls induced by the RAW hazard. It may not be possible to get rid of stalls altogether, but the number of stalls can certainly be minimized with a little bit of hardware complexity. The original algorithm, which bears the name of its inventor, Tomasulo, made its appearance in the IBM 360/91 processor in the 60's. The idea is quite simple. In general, a stage that generates a new value for a register

looks around to see if any other stage is awaiting this new value. If so, it will *forward* the value to the stages that need it⁵.

With respect to our simple pipeline, the only stage that reads a register is the ID/RR stage. Let us examine what we need to do to enable this data forwarding. We add a bit called RP (read pending) to each of the registers in the register file (see Figure 5.11).

Register file		
	B	RP

Figure 5.11: Register file with busy and read-pending bits for each register

When an instruction hits the ID/RR stage, if the B bit is set for a register it wants to read, then it sets the RP bit for the same register. If any of the EX, MEM, or WB stages sees that the RP bit is set for a register for which it is generating a new value then it supplies the generated value to the ID/RR stage. The hardware complexity comes from having to run wires from these stages back to the ID/RR stage. One might wonder why not let these stages simply write to the register file if the RP bit is set. In principle, they could; however, as we said earlier, writing to the register file and clearing the B bit is the responsibility of the WB stage of the pipeline. Allowing any stage to write to the register file will increase the hardware complexity. Besides, this increased hardware complexity does not result in any performance advantage since the instruction (in the ID/RR stage) that may need the data is getting it via the forwarding mechanism.

Revisiting the RAW hazard shown in Figure 5.8(a), data forwarding completely eliminates any bubbles as shown in Figure 5.12.

⁵ The solution we describe for RAW hazard for our simple pipeline is inspired by the Tomasulo algorithm but it is nowhere near the generality of that original algorithm.

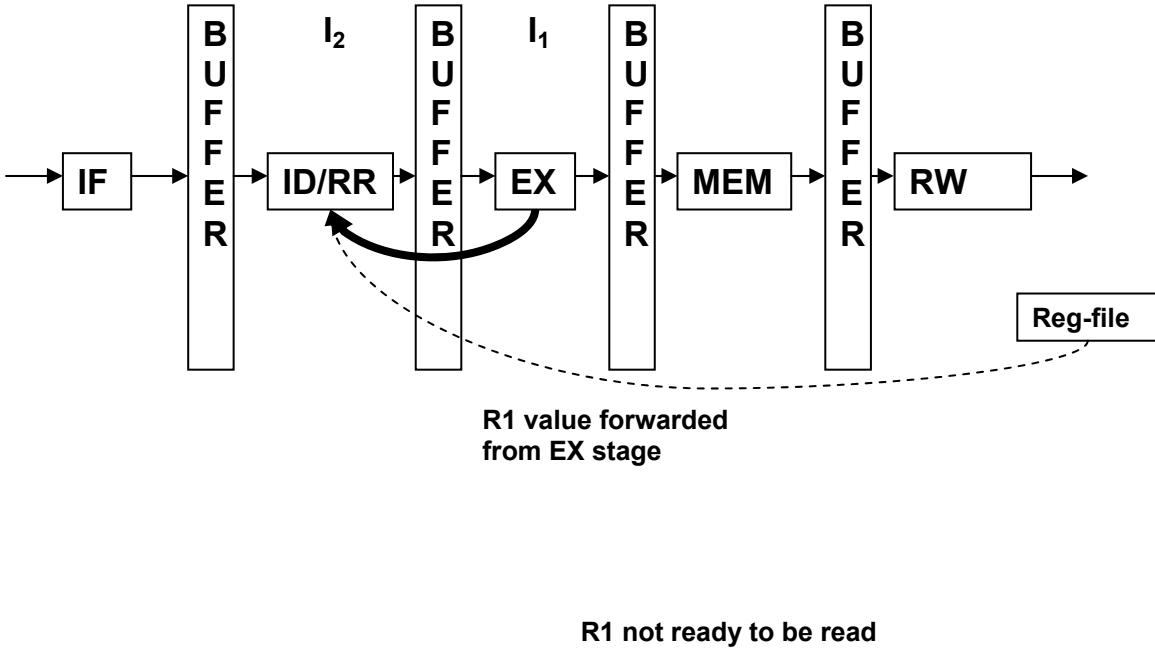


Figure 5.12: Data forwarding⁶ to solve RAW hazard shown in Figure 5.8a

Consider the following stream of instructions:

I_1
 I_2
 I_3
 I_4

If in the above sequence of instructions, I_1 is an arithmetic/logic instruction that generates a new value for a register, and any of the subsequent three instructions (I_2 , I_3 , or I_4) need the same value, then this forwarding method will eliminate any stalls due to the RAW hazard. Table 5.3 summarizes the number of bubbles introduced in the pipeline with and without data forwarding for RAW hazard shown in Figure 5.8a, as a function of the number of unrelated instructions separating I_1 and I_2 in the execution order of the program.

Number of unrelated instructions Between I_1 and I_2	Number of bubbles Without forwarding	Number of bubbles With forwarding
0	3	0
1	2	0
2	1	0
3 or more	0	0

Table 5.3: Bubbles created in pipeline due to RAW Hazard shown in Figure 5.8a

⁶ “Forwarding” seems counter-intuitive given the arrow is going backwards! I_1 is ahead of I_2 in the execution order of the program and it “forwards” the value it computed to I_2 .

Example 13:

$I_1: R1 \leftarrow R2 + R3$
 $I_2: R4 \leftarrow R4 + R3$
 $I_3: R5 \leftarrow R5 + R3$
 $I_4: R6 \leftarrow R1 + R6$



This is the same sequence of instructions as in Example 12. Assuming data forwarding, show the passage of instructions in the chart below. What is the average CPI experienced by these instructions?

Answer:

As we see in Table 5.3, with data forwarding, there will be no more bubbles in the pipeline due to RAW hazard posed by arithmetic and logic instructions since there is data forwarding from every stage to the preceding stage. Thus, as can be seen in the chart below, there are no more bubbles in the pipeline for the above sequence.

Cycle Number	IF	ID/RR	EX	MEM	WB
1	I_1	-	-	-	-
2	I_2	I_1	-	-	-
3	I_3	I_2	I_1	-	-
4	I_4	I_3	I_2	I_1	-
5	-	I_4	I_3	I_2	I_1
6	-	-	I_4	I_3	I_2
7	-	-	-	I_4	I_3
8	-	-	-	-	I_4

Average CPI = $8/4 = 2$.

5.13.2.3 Dealing with RAW Data Hazard introduced by Load instructions

Load instructions introduce data hazard as well. Consider the following sequence:

$I_1: \text{Load } R1, 0(R2)$



(4)

$I_2: \text{Add } R4, R1, R4$

In this case, the new value for R1 is not available until the MEM stage. Therefore, even with forwarding a 1 cycle stall is inevitable if the RAW hazard occurs in the immediate next instruction as shown above in (4). The following exercise goes into the details of bubbles experienced due to a load instruction in the pipeline.

Example 14:

Consider

$I_1: \text{Id } R1, \text{ MEM}$
 $I_2: R4 \leftarrow R1 + R5$



(a)

I_2 immediately follows I_1 as shown above. Assuming there is register forwarding from each stage to the ID/RR stage, how many bubbles will result with the above execution?

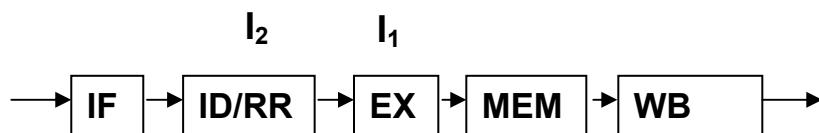
(b)

How many bubbles will result with the above execution if there was no register forwarding?

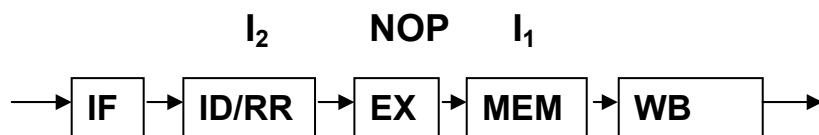
Answer:

(a)

The state of the pipeline when I_2 gets to ID/RR stage is shown below:



I_1 will have a value for R1 only at the end of the MEM cycle. Thus there is a 1 cycle delay leading to one bubble (NOP instruction passed down from the ID/RR stage to the EX stage) as shown below.



The MEM stage will simultaneously write to its output buffer (MBUF) and forward the value it read from the memory for R1 to the ID/RR stage, so that the ID/RR stage can use this value (for I_2) to write into the pipeline register (DBUF) at the output of the ID/RR stage. Thus, there is no further delay in the pipeline. This is despite the fact that I_1 writes the value into R1 only at the end of WB stage. The chart below shows the progression of the two instructions through the pipeline.

Cycle Number	IF	ID/RR	EX	MEM	WB
1	I_1	-	-	-	-
2	I_2	I_1	-	-	-

3	-	I₂	I₁	-	-
4	-	I₂	NOP	I₁	-
5	-	-	I₂	NOP	I₁
6	-	-	-	I₂	NOP
7	-	-	-	-	I₂

So, total number of bubbles for this execution with forwarding = **1**.

(b)

Without the forwarding, I₂ cannot read R1 until I₁ has written the value into R1. Writing into R1 happens at the end of the WB stage. Note that the value written into R1 is available for reading only in the following cycle. The chart below shows the progression of the two instructions through the pipeline.

Cycle Number	IF	ID/RR	EX	MEM	WB
1	I₁	-	-	-	-
2	I₂	I₁	-	-	-
3	I₃	I₂	I₁	-	-
4	I₄	I₂	NOP	I₁	-
5	-	I₂	NOP	NOP	I₁
6	-	I₂	NOP	NOP	NOP
7	-	-	I₂	NOP	NOP
8	-	-	-	I₂	NOP
9	-	-	-	-	I₂

So, total number of bubbles for this execution without forwarding = **3**.

Table 5.4 summarizes the number of bubbles introduced in the pipeline with and without data forwarding for RAW hazard due to Load instruction, as a function of the number of unrelated instructions separating I₁ and I₂ in the execution order of the program.

Number of unrelated instructions Between I₁ and I₂	Number of bubbles Without forwarding	Number of bubbles With forwarding
0	3	1
1	2	0
2	1	0
3 or more	0	0

**Table 5.4: Bubbles created in pipeline due to
Load instruction induced RAW Hazard**

5.13.2.4 Other types of Data Hazards

Other types of hazards, WAR and WAW pose their own set of problems as well for the pipelined processor. However, these problems are much less severe compared to the RAW problem in terms of affecting pipeline processor performance. For example, WAR is not a problem at all since the instruction that needs to read the data has already copied the register value into the pipeline buffer when it was in ID/RR stage. For the WAW problem, a simple solution would be to stall an instruction that needs to write to a register in the ID/RR stage if it finds the B bit set for the register. This instruction will be able to proceed once the preceding instruction that set the B bit clears it in the WB stage.

Let us understand the source of a WAW hazard. WAW essentially means that a register value that was written by one instruction is being over-written by a subsequent instruction with no intervening read of that register. One can safely conclude that the first write was a useless one to start with. After all, if there were a read of the register after the first write, then the ensuing RAW hazard would have over-shadowed the WAW hazard. This begs the question as to why a compiler would generate code with a WAW hazard. There are several possible answers to this question. We will defer discussing this issue until later (please see Section 5.15.4). Suffice it to say at this point that WAW hazards could occur in practice, and it is the responsibility of the hardware to deal with them.

5.13.3 Control Hazard

This hazard refers to breaks in the sequential execution of a program due to branch instructions. Studies on the dynamic frequency of instructions in benchmark programs show that conditional branch instructions occur once in every 3 or 4 instructions. Branches cause disruption to the normal flow of control and are detrimental to pipelined processor performance. The problem is especially acute with conditional branches since the outcome of the branch is typically not known until much later in the pipeline.

Let us assume that the stream of instructions coming into the pipeline in program order is as follows:

BEQ

ADD

NAND

Cycle	IF*	ID/RR	EX	MEM	WB
1	BEQ				
2	ADD	BEQ			
3	ADD+	NOP	BEQ		
4	NAND	ADD	NOP	BEQ	
5	LW	NAND	ADD	NOP	BEQ

* We do not actually know what the instruction is in the Fetch Stage
+ ADD instruction is stalled in the IF stage until the BEQ is resolved
The above schedule assumes that the branch was unsuccessful allowing instructions in the sequential path to enter the IF stage once BEQ is resolved. There will be one more cycle delay to access the non-sequential path, if the branch was successful.

Figure 5.13: Conservative Approach to Handling Branches

One conservative way of handling branches is to stop new instructions from entering the pipeline when the decode stage encounters a branch instruction. Once the branch is resolved, normal pipeline execution can resume either along the sequential path of control or along the target of the branch. Figure 5.13 shows the flow of instructions for such a conservative arrangement. For BEQ instruction, we know the outcome of the branch at the end of the EX cycle. If the *branch is not taken* (i.e. we continue along the sequential path) then the IF stage already has the right instruction fetched from memory and ready to be passed on to the ID/RR stage. Therefore, we stall the pipeline for 1 cycle and continue with the ADD instruction as shown in Figure 5.13. However, if the *branch is taken*, then we have to start fetching the instruction from the newly computed address of the PC which is clocked into PC only at the end of the EX cycle for the BEQ instruction. So, in cycle 4 we will start fetching the correct next instruction from the target of the branch. Hence, there will be a 2-cycle stall if the branch is taken with the above scheme.

Example 15:

Given the following sequence of instructions:

BEQ	L1
ADD	
LW	
....	
L1	NAND
	SW

The hardware uses a conservative approach to handling branches.

- (a) Assuming that the branch is not taken, show the passage of instructions through the pipeline by filling the chart below until three instructions have successfully retired from the pipeline. What is the observed CPI for these three instructions?
- (b) Assuming that the branch is taken, show the passage of instructions through the pipeline by filling the chart below until three instructions have successfully retired from the pipeline. What is the observed CPI for these three instructions?

Answer :

(a) The time chart for the above sequence when the *branch is not taken* is shown below. Note that ADD instruction is stalled in IF stage for 1 cycle until BEQ instruction is resolved at the end of the EX stage.

Cycle Number	IF	ID/RR	EX	MEM	WB
1	BEQ	-	-	-	-
2	ADD	BEQ	-	-	-
3	ADD	NOP	BEQ	-	-
4	LW	ADD	NOP	BEQ	-
5	-	LW	ADD	NOP	BEQ
6	-	-	LW	ADD	NOP
7			-	LW	ADD
8				-	LW

The average CPI for these three instructions = $8/3 = 2.666$.

(b) The time chart for the above sequence when the *branch is taken* is shown below. Note that ADD instruction in the IF stage has to be converted into a NOP in cycle 4 since the branch is taken. A new fetch has to be instantiated from the target of the branch in cycle 4, thus resulting in a 2 cycle stall of the pipeline.

Cycle Number	IF	ID/RR	EX	MEM	WB
1	BEQ	-	-	-	-
2	ADD	BEQ	-	-	-
3	ADD	NOP	BEQ	-	-
4	NAND	NOP	NOP	BEQ	-
5	SW	NAND	NOP	NOP	BEQ

6	-	SW	NAND	NOP	NOP
7	-	-	SW	NAND	NOP
8			-	SW	NAND
9				-	SW

The average CPI for the three instructions = $9/3 = 3$.

Of course, we can do even better if we are ready to throw some hardware at the problem. In fact, there are several different approaches to solving hiccups in pipelines due to branches. This has been a fertile area of research in the computer architecture community. Especially since modern processors may have deep pipelines (with more than 20 stages), it is extremely important to have a speedy resolution of branches to ensure high performance.

In this chapter, we will present a small window into the range of possibilities for solving this problem. You will have to wait to take a senior level course in computer architecture to get a more detailed treatment of this subject.

5.13.3.1 Dealing with branches in the pipelined processor

1. **Delayed branch:** The idea here is to assume that the instruction following the branch executes irrespective of the outcome of the branch. This simplifies the hardware since there is no need to terminate the instruction that immediately follows the branch instruction. The responsibility of ensuring the semantic correctness underlying this assumption shifts to the compiler from the hardware. The default is to stash a NOP instruction in software (i.e. in the memory image of the program) following a branch. You may ask what the use of this approach is. The answer is a smart compiler will be able to do program analysis and find a useful instruction that does not affect the program semantics instead of the NOP. The instruction slot that immediately follows the branch is called a **delay slot**. Correspondingly, this technique is called **delayed branch**.

The code fragments shown in Figure 5.14 and 5.15 show how a compiler may find useful instructions to stick into delay slots. In this example, every time there is branch instruction, the instruction that immediately follows the branch is executed regardless of the outcome of the branch. If the branch is unsuccessful, the pipeline continues without a hiccup. If the branch is successful, then (as in the case of branch misprediction) the instructions following the branch except the immediate next one are terminated.

```

;      Code before optimization to fill branch delay slots
;      Add 7 to each element of a ten-element array
;      whose address is in a0
addi t1, a0, 40           ; When a0=t1 we are done
loop: beq a0, t1, done
    nop                  ; branch delay slot [1]
    lw   t0, 0(a0)        [2]
    addi t0, t0, 7
    sw   t0, 0(a0)
    addi a0, a0, 4
    beq zero, zero, loop
    nop                  ; branch delay slot [4]
done: halt

```

Figure 5.14: Delayed branch: delay slots with NOPs

```

;      Code after optimization to fill branch delay slots
;      Add 7 to each element of a ten-element array
;      whose address is in a0
addi t1, a0, 40           ; When a0=t1 we are done
loop: beq a0, t1, done
    lw   t0, 0(a0)        ; branch delay slot [2]
    addi t0, t0, 7
    sw   t0, 0(a0)
    beq zero, zero, loop
    addi a0, a0, 4         ; branch delay slot [3]
done: halt

```

Figure 5.15: Delayed branch: NOPs in delay slots replaced with useful instructions

With reference to Figure 5.14, the compiler knows that the instructions labeled [1] and [4] are always executed. Therefore, the compiler initially sticks NOP instructions as placeholders in those slots. During optimization phase, the compiler recognizes that instruction [2] is benign⁷ from the point of view of the program semantics, since it simply loads a value into a temporary register. Therefore, the compiler replaces the NOP instruction [1] by the load instruction [2] in Figure 5.15. Similarly, the last branch instruction in the loop is an unconditional branch, which is independent of the preceding add instruction [3]. Therefore, the compiler replaces the NOP instruction [4] by the add instruction [3] in Figure 5.15.

⁷ This is for illustration of the concept of delayed branch. Strictly speaking, executing the load instruction after the loop has terminated may access a memory location that is non-existent.

Some machines may use even multiple delay slots to increase pipeline efficiency. The greater the number of delay slots the lesser the number of instructions that need terminating upon a successful branch. However, this increases the burden on the compiler to find useful instructions to fill in the delay slot, which may not always be possible.

Delayed branch seems like a reasonable idea especially for shallow pipelines (less than 10 stages) since it simplifies the hardware. However, there are several issues with this idea. The most glaring problem is that it exposes the details of the microarchitecture to the compiler writer thus making a compiler not just ISA-specific but processor-implementation specific. Either it necessitates re-writing parts of the compiler for each generation of the processor or it limits evolution of the microarchitecture for reasons of backward compatibility. Further, modern processors use deep pipelining. For example, Intel Pentium line of processors has pipeline stages in excess of 20. While the pipelines have gotten deeper, the sizes of basic blocks in the program that determine the frequency of branches have not changed. If anything, the branches have become more frequent with object-oriented programming. Therefore, delayed branch has fallen out of favor in modern processor implementation.

2. **Branch prediction:** The idea here is to assume the outcome of the branch to be one way and start letting instructions into the pipeline even upon detecting a branch. For example, for the same sequence shown in Figure 5.13, let us predict that the outcome is negative (i.e., the sequential path is the winner). Figure 5.16 shows the flow of instructions in the pipeline with this prediction. As soon as the outcome is known (when BEQ is in the EX stage), the result is fed back to the preceding stages. Figure 5.16 shows the happy state when the outcome is as predicted. The pipeline can continue without a hiccup.

Cycle	IF*	ID/RR	EX	MEM	WB
1	BEQ				
2	ADD	BEQ			
3	NAND	ADD	BEQ		
4	LW	NAND	ADD	BEQ	
5	...	LW	NAND	ADD	BEQ

* We do not actually know what the instruction is in the Fetch Stage

Figure 5.16: Branch Prediction

Of course, there are chances of misprediction and we need to be able to recover from such mispredictions. Therefore, we need a hardware capability to terminate the instructions that are in partial execution in the preceding stages of the pipeline, and start fetching from the alternate path. This termination capability is often referred to as *flushing*. Another feedback line labeled “flush”, shown pictorially in Figure 5.17, implements this hardware capability. Upon receiving this flush signal, both the IF and

ID/RR stages abandon the partial execution of the instructions they are currently dealing with and start sending “bubbles” down the pipeline. There will be 2 bubbles (i.e., 2-cycle stall) corresponding to the ADD and NAND instructions in Figure 5.17 before normal execution can begin.

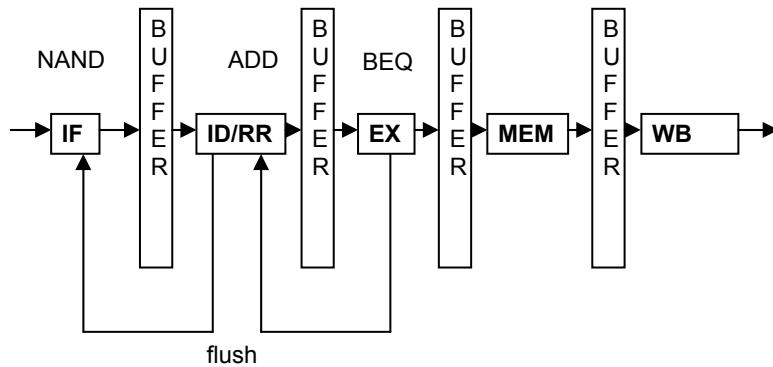


Figure 5.17: Pipeline with flush control lines

Example 16:

Given the following sequence of instructions:

```

BEQ    L1
ADD
LW
.....
L1    NAND
SW

```

The hardware uses branch prediction (branch will not be taken).

- Assuming that the prediction turns out to be true, show the passage of these instructions through the pipeline by filling the chart below until three instructions have successfully retired from the pipeline. What is the observed CPI for these three instructions?
- Assuming that the prediction turns out to be false, show the passage of these instructions through the pipeline by filling the chart below until three instructions have successfully retired from the pipeline. What is the observed CPI for these three instructions?

Answer:

- The time chart for the above sequence when the prediction is true is shown below. The branch prediction logic starts feeding in the instructions from the sequential path into the pipeline and they all complete successfully.

Cycle Number	IF	ID/RR	EX	MEM	WB
1	BEQ	-	-	-	-
2	ADD	BEQ	-	-	-
3	LW	ADD	BEQ	-	-
4	-	LW	ADD	BEQ	-
5	-	-	LW	ADD	BEQ
6	-	-	-	LW	ADD
7	-	-	-	-	LW

At the end of 7 cycles, 3 instructions complete,
yielding an average CPI of $7/3 = 2.333$

- (b) The time chart for the above sequence when the prediction turns out to be false is shown below. Note that ADD and LW instructions are flushed in their respective stages as soon as BEQ determines that the outcome of the branch has been mispredicted. This is why in cycle 4, ADD and LW have been replaced by NOPs in the successive stages. The PC is set at the end of the EX cycle by the BEQ instruction (cycle 3) in readiness to start fetching the correct instruction from the target of the branch in cycle 4.

Cycle Number	IF	ID/RR	EX	MEM	WB
1	BEQ	-	-	-	-
2	ADD	BEQ	-	-	-
3	LW	ADD	BEQ	-	-
4	NAND	NOP	NOP	BEQ	-
5	SW	NAND	NOP	NOP	BEQ
6	-	SW	NAND	NOP	NOP
7	-	-	SW	NAND	NOP
8			-	SW	NAND
9				-	SW

The average CPI for the three instructions = $9/3 = 3$.

Note that this average CPI is the same as what we obtained earlier with no branch prediction and the branch is taken.

You may be wondering how one can predict the outcome of a branch since on the surface the odds seem evenly split between taking and not taking. However, programs have a lot more structure that aids prediction. For example, consider a loop. This usually consists of a series of instructions; at the end of the loop, a conditional branch takes control back to the top of the loop or out of the loop. One can immediately see that the outcome of this conditional branch is heavily biased towards going back to the top of the loop. Thus, the branch prediction technique relies on such structural properties of programs.

There has been considerable research invested in understanding the properties of branches that occur in programs and designing prediction schemes to support them. As we already mentioned, loops and conditional statements are the high-level constructs that lead to conditional branch instructions. One strategy for branch prediction is to predict the branch is likely to be taken if the target address is *lower* than the current PC value. The flip side of this statement is that if the target address is *higher* than the current PC value, then the prediction is that the branch will not be taken. The motivation behind this strategy is that loops usually involve a backward branch to the top of the loop (i.e., to a lower address), and there is a higher likelihood of this happening since loops are executed multiple times. On the other hand, forward branches are usually associated with conditional statements, and they are less likely to be taken.

Branch prediction is in the purview of the compiler based on program analysis. The ISA has to support the compiler by providing a mechanism for conveying the prediction to the hardware. To this end, modern processors include two versions of many if not all branch instructions, the difference being a prediction on whether the branch will be taken or not. The compiler will choose the version of a given branch instruction that best meets its need.

3. **Branch prediction with branch target buffer:** This builds on branch prediction that we mentioned earlier. It uses a hardware device called **Branch Target Buffer (BTB)**⁸ to improve the prediction of branches. Every entry of this table contains three fields as shown in Figure 5.18.

Address of branch instruction	Taken/Not taken	Address of Target of Branch Instruction
-------------------------------	-----------------	---

Figure 5.18: An entry in the Branch Target Buffer

This table records the history of the branches encountered for a particular program during execution. The table may have some small number of entries (say 100). Every time a branch instruction is encountered, the table is looked up by the hardware. Let us assume the PC value of this branch instruction is not present in the table (this is the first encounter). In that case, once the branch outcome is determined,

⁸ We already introduced the idea of caches earlier in this Chapter (see Section 5.11). In Chapter 9, we discuss caches in detail. BTB is essentially a cache of branch target addresses matched to addresses of branch instructions. The hardware needed to realize a BTB is similar to what would be found in a cache.

a table entry with the address of this branch instruction, the target address of the branch, and the direction of the branch (taken/not taken) is created. Next time the same branch instruction is encountered this history information helps in predicting the outcome of the branch. The BTB is looked up in the IF stage. Thus, if the lookup is successful (i.e., the table contains the address of the branch instruction currently being fetched and sent down the pipeline), then the IF stage starts fetching the target of the branch immediately. In this case, there will be no bubbles in the pipeline due to the branch instruction. Of course, there could be mispredictions. Typically, such mispredictions occur due to a branch instruction that is executed in either the first or the last iteration of a loop. The flush line in the datapath takes care of such mispredictions. We can make the history mechanism more robust by providing more than one bit of history.

5.13.3.2 Summary of dealing with branches in a pipelined processor

Implementation detail of an ISA is commonly referred to as the *microarchitecture* of the processor. Dealing with branches in the microarchitecture is a key to achieving high performance, especially with the increased depth of the pipelines in modern processors. We already mentioned that the frequency of branches in compiled code could be as high as 1 in every 3 or 4 instructions. Thus, with a pipeline that has 20 stages, the instructions in partial execution in the various stages of the pipe are unlikely to be sequential instructions. Therefore, taking early decisions on the possible outcome of branches so that the pipeline can be filled with useful instructions is a key to achieving high performance in deeply pipelined processors. We will discuss the state-of-the-art in pipelined processor design in a later section (see Section 5.15). Table 5.5 summarizes the various techniques we have discussed in this chapter for dealing with branches and processors that have used these techniques.

Name	Pros	Cons	Use cases
Stall the pipeline	Simple strategy, no hardware needed for flushing instructions	Loss of performance	Early pipelined machines such as IBM 360 series
Branch Prediction (branch not taken)	Results in good performance with small additional hardware since the instruction is anyhow being fetched from the sequential path already in IF stage	Needs ability to flush instructions in partial execution in the pipeline	Most modern processors such as Intel Pentium, AMD Athlon, and PowerPC use this technique; typically they also employ sophisticated branch target buffers; MIPS R4000 uses a combination 1-delay slot plus a 2-cycle branch-not-taken prediction
Branch	Results in good	Since the new PC value	-

Prediction (branch taken)	performance but requires slightly more elaborate hardware design	that points to the target of the branch is not available until the branch instruction is in EX stage, this technique requires more elaborate hardware assist to be practical	
Delayed Branch	No need for any additional hardware for either stalling or flushing instructions; It involves the compiler by exposing the pipeline delay slots and takes its help to achieve good performance	With increase in depth of pipelines of modern processors, it becomes increasingly difficult to fill the delay slots by the compiler; limits microarchitecture evolution due to backward compatibility restrictions; it makes the compiler not just ISA-specific but implementation specific	Older RISC architectures such as MIPS, PA-RISC, SPARC

Table 5.5: Summary of Techniques for Handling Branches

5.13.4 Summary of Hazards

We discussed structural, data, and control hazards and the most basic mechanisms for dealing with them in the microarchitecture. We presented mostly hardware assists for detecting and overcoming such hazards. When hazards are detected and enforced by the hardware, it is often referred to in the literature as *hardware interlocks*. However, it is perfectly feasible to shift this burden to the software, namely, the compiler. The trick is to expose the details of the microarchitecture to the compiler such that the compiler writer can ensure that either (a) these hazards are eliminated in the code as part of program optimization, or (b) by inserting NOPs into the code explicitly to overcome such hazards. For example, with the 5-stage pipeline that we have been discussing, if the compiler always ensures that a register being written to is not used by at least the following three instructions, then there will be no RAW hazard. This can be done by either placing other useful instructions between the definition of a value and its subsequent use, or in the absence of such useful instructions, by placing explicit NOP instructions. The advantage of shifting the burden to the compiler is that the hardware is simplified, since it has neither to do hazard detection nor incorporate techniques such as data forwarding. Early versions of the MIPS architecture eliminated hardware interlocks and depended on the compiler to resolve all such hazards. However, the problem is it is not always possible to know such dependencies at compile time. The problem gets worse with deeper pipelines. Therefore, all modern processors use hardware interlocks to eliminate hazards. However, in the spirit of the partnership between hardware and software, the chip vendors publish the details of the microarchitecture to aid the compiler writer in using such details in writing efficient optimizers.

Table 5.6 summarizes the LC-2200 instructions that can cause potential stalls in the pipeline, and the extent to which such stalls can be overcome with hardware solutions.

Instruction	Type of Hazard	Potential Stalls	With Data forwarding	With branch prediction (branch not taken)
ADD, NAND	Data	0, 1, 2, or 3	0	Not Applicable
LW	Data	0, 1, 2, or 3	0 or 1	Not Applicable
BEQ	Control	1 or 2	Not Applicable	0 (success) or 2 (mispredict)

Table 5.6: Summary of Hazards in LC-2200

5.14 Dealing with program discontinuities in a pipelined processor

Earlier when we discussed interrupts, we mentioned that we wait for the FSM to be in a clean state to entertain interrupts. In a non-pipelined processor, instruction completion offers such a clean state. In a pipelined processor since several instructions are in flight (i.e., under partial execution) at any point of time, it is difficult to define such a clean state. This complicates dealing with interrupts and other sources of program discontinuities such as exceptions and traps.

There is one of two possibilities for dealing with interrupts. One possibility is as soon as an external interrupt arrives, the processor can:

1. Stop sending new instructions into the pipeline (i.e., the logic in the IF stage starts manufacturing NOP instructions to send down the pipeline).
2. Wait until the instructions that are in partial execution complete their execution (i.e., **drain** the pipe).
3. Go to the interrupt state as we discussed in Chapter 4 and do the needful (See Example 17).

The downside to draining the pipe is that the response time to external interrupts can be quite slow especially with deep pipelining, as is the case with modern processors. The other possibility is to **flush** the pipeline. As soon as an interrupt arrives, send a signal to all the stages to abandon their respective instructions. This allows the process to switch to the interrupt state immediately. Of course, in this case the memory address of the last completed instruction will be used to record in the PC, the address where the program needs to be resumed after interrupt servicing. There are subtle issues with implementing such a scheme to ensure that no permanent program state has been changed (e.g., register values) by any instruction that was in partial execution in any of the pipeline stages.

In reality, processors do not use either of these two extremes. Interrupts are caught by a specific stage of the pipe. The program execution will restart at the instruction at that stage. The stages ahead are allowed to complete while the preceding stages are flushed.

In each of these cases, one interesting question that comes up is what should happen implicitly in hardware to support interrupts in a pipelined processor. The pipeline registers are part of the internal state of the processor. Each of the above approaches gets the processor to a clean state, so we have to worry only about saving a single value of the PC for the correct resumption of the interrupted program. The value of the PC to be saved depends on the approach chosen.

If we want to be simplistic and drain the pipe before going to the INT macro state (see Chapter 4), then it is sufficient to freeze the PC value pointing to the next instruction in program order in the IF stage. Upon an interrupt, the hardware communicates this PC value to the INT macro state for saving.

Example 17:

Enumerate the steps taken in hardware from the time an interrupt occurs to the time the processor starts executing the handler code in a pipelined processor. Assume the pipeline is drained upon an interrupt. (An English description is sufficient.)

Answer:

1. Allow instructions already in the pipeline (useful instructions) to complete execution
 2. Stop fetching new instructions
 3. Start sending NOPs from the fetch stage into the pipeline
 4. Once all the useful instructions have completed execution, record the address where the program needs to be resumed in the PC (this will be memory address of last completed useful instruction + 1);
 5. The next three steps are interrupt state actions that we covered in Chapter 4.
 6. Go to INT state; sent INTA; receive vector; disable interrupts
 7. Save current mode in system stack; change mode to kernel mode;
 8. Store PC in \$k0; Retrieve handler address using vector; Load PC; resume pipeline execution
-

Flushing the instructions (either fully or partially) would require carrying the PC value of each instruction in the pipeline register since we do not know when an external interrupt may occur⁹. Actually, carrying the PC value with each instruction is a necessity in a pipelined processor since any instruction may cause an exception/trap. We need the PC value of the trapping instruction to enable program resumption at the point of trap/exception. Upon an interrupt, the hardware communicates the PC value of the first incomplete instruction (i.e., the instruction from which the program has to be resumed) to the INT macro state for saving. For example, if the interrupt is caught by the EX stage of the pipe, then the current instructions in the MEM and WB stages are allowed to complete, and the PC value corresponding to the instruction in the EX stage of the pipeline is the point of program resumption.

⁹ Recall that in the discussion of the pipeline registers earlier, we mentioned that only the passage of the BEQ instruction needs the PC value to be carried with it.

Example 18:

A pipelined implementation of LC-2200, allows interrupts to be caught in the EX stage of the 5-stage pipeline draining the instructions preceding it, and flushing the ones after it. Assume that there is register forwarding to address RAW hazards.

Consider the following program:

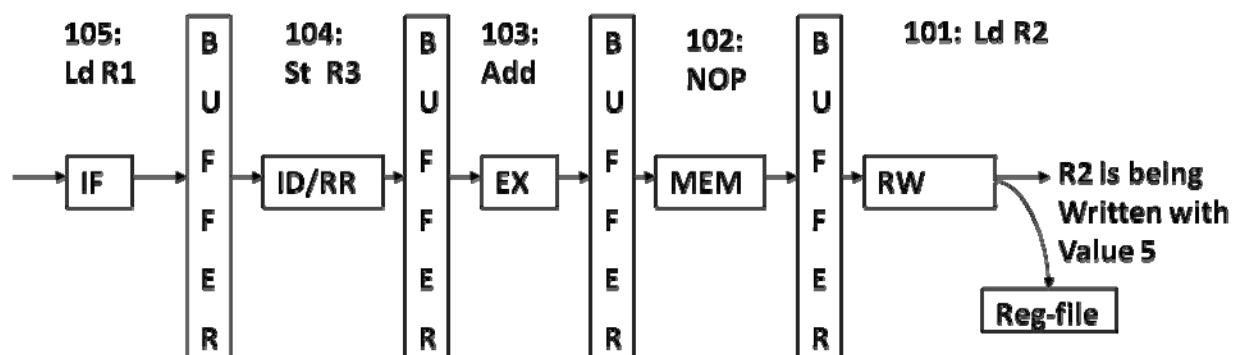
Address	100 Ld R1, MEM[2000];	/* assume MEM[2000] contains value 2 */
	101 Ld R2, MEM[2002];	/* assume MEM[2002] contains value 5 */
	102 NOP;	/* NOP for Ld R2 */
	103 Add R3, R1, R2;	/* addition R3 <- R1+R2 */
	104 St R3, MEM[2004];	/* Store R3 into MEM[2004]
	105 Ld R1, MEM[2006];	/* assume MEM[2006] contains value 3 */

An interrupt occurs when the NOP (at address 102) is in the MEM stage of the pipeline. Answer the following.

- What are the values in R1, R2, R3, MEM[2000], MEM[2002], MEM[2004], and MEM[2006], when the INT state is entered?
- What is the PC value passed to the INT state?

Answer:

When the interrupt occurs the state of the pipeline is as shown below:



Therefore, instructions ahead of the EX stage (namely, MEM and RW) will complete, and the ones in the ID/RR and IF stages will be flushed. The program will restart execution at address 103: Add R3, R1, R2

(a)

$$R1 = 2; R2 = 5;$$

R3 = unknown (since Add instruction has been aborted due to the interrupt)

MEM[2000] = 2; MEM[2002] = 5;
MEM[2004] = unknown; (since Add instruction has been aborted due to the interrupt)
MEM[2006] = 3

(b)

The PC value passed to the INT state is the address of the Add instruction, namely, 103, which is where the program will resume after the interrupt service.

The above discussion is with respect to the program discontinuity being an external interrupt. With traps and exceptions, we have no choice but to complete (i.e., drain) the preceding instructions, flush the succeeding instructions, and go to the INT state passing the PC value of the instruction that caused the trap/exception.

5.15 Advanced topics in processor design

Pipelined processor design had its beginnings in the era of high-performance mainframes and vector processors of the 60's and 70's. Many of the concepts invented in that era are still relevant in modern processor design. In this section, we will review some advanced concepts in processor design including the state-of-the-art in pipelined processor design.

5.15.1 Instruction Level Parallelism

The beauty of pipelined processing is that it does not conceptually change the *sequential programming model*. That is, as far as the programmer is concerned, there is no perceived difference between the simple implementation of the processor presented in Chapter 3 and the pipelined implementation in this chapter. The instructions of the program *appear to execute* in exactly the same order as written by the programmer. The order in which instructions appear in the original program is called *program order*. Pipelined processor shrinks the execution time of the program by recognizing that adjacent instructions of the program are independent of each other and therefore their executions may be overlapped in time with one another. *Instruction Level Parallelism (ILP)* is the name given to this potential overlap that exists among instructions. ILP is a property of the program, and is a type of parallelism that is often referred to as *implicit parallelism*, since the original program is sequential. In Chapter 12, we will discuss techniques for developing explicitly parallel programs, and the architectural and operating systems support for the same. Pipelined processor exploits ILP to achieve performance gains for sequential programs. The reader can immediately see that ILP is limited by hazards. Particularly, control hazards are the bane of ILP exploitation. *Basic block* is a term used to define the string of instructions in a program separated by branches (see Figure 5.19). With reference to Figure 5.19, the amount of ILP available for the first basic block is 4, and that for the second basic block is 3. The actual parallelism exploitable by the pipelined processor is limited further due to other kinds of hazards (data and structural) that we discussed in this chapter.

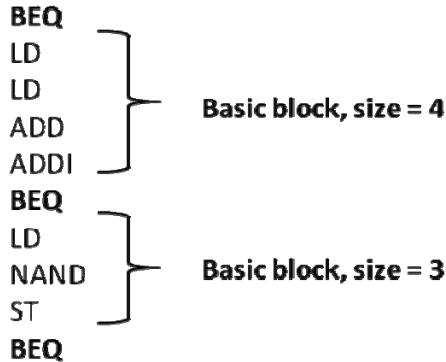


Figure 5.19: Basic Blocks and ILP

We have only scratched the surface of the deep technical issues in processor design and implementation. For example, we presented very basic mechanisms for avoiding conflicts in shared resource requirements (such as register file and ALU) across stages of the pipeline, so that the available ILP becomes fully exploitable by the processor implementation. Further, we also discussed simple mechanisms for overcoming control hazards so that ILP across multiple basic blocks becomes available for exploitation by the processor. Multiple-issue processors that have become industry-standard pose a number of challenges to the architect.

5.15.2 Deeper pipelines

The depth of pipelines in modern processors is much more than 5. For example, Intel Pentium 4 has in excess of 20 pipeline stages. However, since the frequency of branches in compiled code can be quite high, the size of typical basic blocks may be quite small (in the range of 3 to 7). Therefore, it is imperative to invent clever techniques to exploit ILP across basic blocks to make pipelined implementation worthwhile. The field of microarchitecture, the myriad ways to improve the performance of the processor, is both fascinating and ever evolving. In Chapter 3, we mentioned *multiple issue* processors, with Superscalar and VLIW being specific instances of such processors. *Instruction issue* is the act of sending an instruction for execution through the processor pipeline. In the simple 5-stage pipeline we have considered thus far, exactly one instruction is issued in every clock cycle. As the name suggests, multiple issue processors *issue* multiple instructions in the same clock cycle. As a first order of approximation, let us assume that the hardware and/or the compiler would have ensured that the set of instructions being issued in the same clock cycle do not have any of the hazards we discussed earlier, so that they can execute independently. Correspondingly, the processors have *multiple decode* units, and *multiple functional units* (e.g., integer arithmetic unit, floating point arithmetic unit, load/store unit, etc.) to cater to the different needs of the instructions being issued in the same clock cycle (see Figure 5.20). The fetch unit would bring in multiple instructions from memory to take advantage of the multiple decode units. A decode unit should be able to dispatch the decoded instruction to any of the functional units.

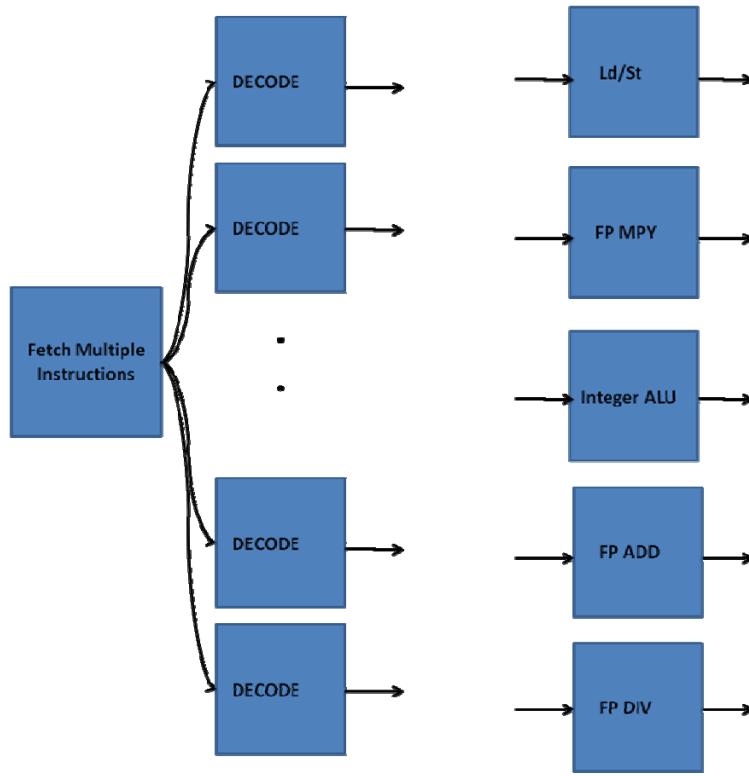


Figure 5.20: Multiple Issue processor pipeline

The effective throughput of such processors approaches the product of the depth of the pipelining and the degree of superscalarity of the processor. In other words, modern processors not only have deep pipelines but also have several such pipelines executing in parallel. Deep pipelining and superscalarity introduce a number of interesting challenges, to both the architect and the developers of system software (particularly compiler writers).

You may be wondering about the necessity for deep pipelining. There are several reasons:

- **Relative increase in time to access storage:** In Chapter 3, we introduced the components of delays in a datapath. As feature sizes of transistors on the chip keep shrinking, wire delays (i.e., the cost of ferrying bits among the datapath elements) rather than logic operations become the limiting factor for deciding the clock cycle time. However, the time to pull values out of registers and caches (which we discuss in detail in Chapter 9) is still much greater than either the logic delays or wire delays. Thus, with faster clock cycle times, it may become necessary to allow multiple cycles for reading values out of caches. This increases the complexity to the unit that expects a value from the cache for performing its task, and increases the overall complexity of the processor. Breaking up the need for multiple cycles to carry out a specific operation into multiple stages is one way of dealing with the complexity.
- **Microcode ROM access:** In Chapter 3, we introduced the concept of microprogrammed implementation of instructions. While this technique has its

pitfalls, pipelined processors may still use it for implementing some selected complex instructions in the ISA. Accessing the microcode ROM could add a stage to the depth of the pipeline.

- **Multiple functional Units:** Modern processors include both integer and floating-point arithmetic operations in their ISA. Typically, the microarchitecture includes specialized functional units for performing floating-point add, multiply, and division that are distinct from the integer ALU. The EX unit of the simple 5-stage pipeline gets replaced by this collection of functional units (see Figure 5.20). There may be an additional stage to help schedule these multiple functional units.
- **Dedicated floating-point pipelines:** Floating-point instructions require more time to execute compared to their integer counter-parts. With a simple 5-stage pipeline, the slowest functional unit will dictate the clock cycle time. Therefore, it is natural to pipeline the functional units themselves so that we end with a structure as shown in Figure 5.21, with different pipeline depth for different functional units. This differential pipelining facilitates supporting multiple outstanding long latency operations (such as floating-point ADD) without stalling the pipeline due to structural hazards.

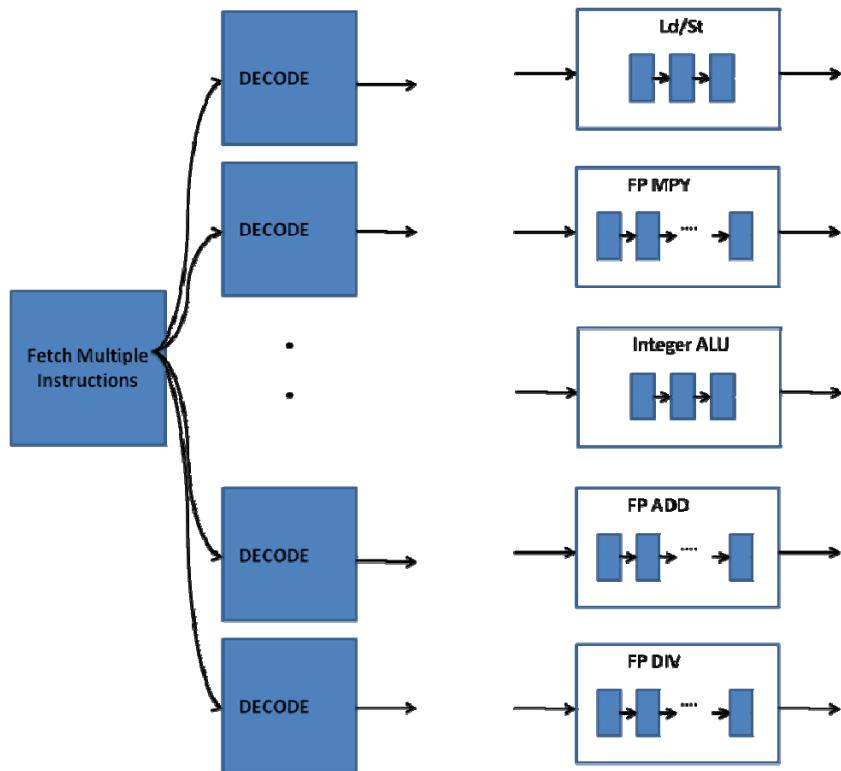


Figure 5.21: Different depths of pipelining for different functional units

- **Out of order execution and Re-Order Buffer:** Recall that the pipelined processor should preserve the appearance of sequential execution of the program (i.e., the program order) despite the exploitation of ILP. With the potential for different instructions taking different paths through the pipeline, there is a complication in maintaining this appearance of sequentiality. For this reason,

modern processors distinguish between *issue order* and *completion order*. The fetch unit issues the instructions *in order*. However, the instructions may execute and complete *out of order*, due to the different depths of the different pipelines and other reasons (such as awaiting operands for executing the instruction). It turns out that this is fine so long as the instructions are *retired* from the processor in program order. In other words, an instruction is *not retired* from the processor even though it has *completed execution* until all the instructions preceding it in program order have also completed execution. To ensure this property, modern processors add additional logic that may be another stage in the pipeline.

Specifically, the pipeline includes a *Re-Order Buffer* (ROB) whose job it is to retire the completed instructions in program order. Essentially, ROB ensures that the effect of an instruction completion (e.g., writing to an architecture-visible register, reading from or writing to memory, etc.) is not *committed* until all its predecessor instructions in program order have been committed. It does this by recording the program order of instructions at issue time and buffering addresses and data that need to be transmitted to the architecture-visible registers and memory. Reading from a register in the presence of a ROB, respects the RAW dependency as already discussed in Section 5.13.2.2.

- **Register renaming:** To overcome data hazards, modern processors have several more physical registers than the architecture-visible general-purpose registers. The processor may use an additional stage to detect resource conflicts and take actions to disambiguate the usage of registers using a technique referred to as *register renaming*¹⁰. Essentially, register renaming is a one-level indirection between the architectural register named in an instruction and the actual physical register to be used by this instruction. We will discuss this in more detail later in this section after introducing Tomasulo algorithm.
- **Hardware-based speculation:** To overcome control hazards and fully exploit the multiple issue capability, many modern processors use *hardware-based speculation*, an idea that extends branch prediction. The idea is to execute instructions in different basic blocks without waiting for the resolution of the branches and have mechanisms to *undo* the effects of the instructions that were executed incorrectly due to the speculation, once the branches are resolved. Both ROB and/or register renaming (in hardware) help hardware-based speculation since the information in the physical registers or ROB used in the speculation can be discarded later when it is determined that the effects of a speculated instruction should not be committed.

5.15.3 Revisiting program discontinuities in the presence of out-of-order processing

In Section 5.14, we discussed simple mechanisms for dealing with interrupts in a pipelined processor. Let's see the ramifications of handling interrupts in the presence of out-of-order processing. Several early processors such as CDC 6600 and IBM 360/91 used out-of-order execution of instructions to overcome pipeline stalls due to data

¹⁰ Note that register renaming could be done by the compiler as well upon detection of data hazards as part of program optimization, in which case such an extra stage would not be needed in the pipeline.

hazards and structural hazards. The basic idea is to issue instructions in order, but let the instructions start their executions as soon as their source operands are available. This out-of-order execution coupled with the fact that different instructions incur different execution time latencies leads to instructions potentially completing execution out of order and retiring from the pipeline. That is, they not only complete execution but also update processor state (architecture-visible registers and/or memory). Is this a problem? It would appear that there should not be since these early pipelined processors did respect the program order at issue time, did honor the data dependencies among instructions, and never executed any instructions speculatively. External interrupts do not pose a problem with out-of-order processing, since we could adopt a very simple solution, namely, stop issuing new instructions and allow all the issued instructions to complete before taking the interrupt.

Exceptions and traps, however, pose a problem since some instructions that are subsequent to the one causing the exception could have completed their execution. This situation is defined as *imprecise exception*, to signify that the processor state, at the time the exception happens, is not the same as would have happened in a purely sequential execution of the program. These early pipelined processors restored precise exception state either by software (i.e., in the exception handling routines), or by hardware techniques for early detection of exceptions in long latency operations (such as floating-point instructions).

As we saw in the previous section, modern processors retire the instructions in program order despite the out-of-order execution. This automatically eliminates the possibility of imprecise exception. Potential exceptions are buffered in the re-order buffer and will manifest in strictly program order.

Detailed discussion of interrupts in a pipelined processor is outside the scope of this book. The interested reader may want to refer to advanced textbooks on computer architecture¹¹.

5.15.4 Managing shared resources

With multiple functional units, managing the shared resources (such as register files) becomes even more challenging. One technique, popularized by CDC 6600, is *scoreboard*, a mechanism for dealing with the different kinds of data hazards we discussed earlier in this chapter. The basic idea is to have a central facility, namely a scoreboard that records the resources in use by an instruction at the time it enters the pipeline. A decision to either progress through the pipeline or stall an instruction depends on the current resource needs of that instruction. For example, if there is a RAW hazard then the second instruction that needs the register value is stalled until the scoreboard indicates that the value is available (generated by a preceding instruction). Thus, the scoreboard keeps track of the resource needs of all the instructions in flight through the pipeline.

¹¹ Hennessy and Patterson, “Computer Architecture: A Quantitative Approach,” Morgan Kaufman publishers.

Robert Tomasulo of IBM came up with a very clever algorithm (named after him), which is a distributed solution to the same resource sharing and allocation problem in a pipelined processor. This solution was first used in IBM 360/91, one of the earliest computers (in the 60's) to incorporate principles of pipelining in its implementation. The basic idea is to associate storage (in the form of local registers) with each functional unit. At the time of instruction issue, the needed register values are transferred to these local registers¹² thus avoiding the WAR hazard. If the register values are unavailable (due to RAW hazard), then the local registers remember the unit from which they should expect to get the value. Upon completing an instruction, a functional unit sends the new register value on a *common data bus (CDB)* to the register file. Other functional units (there could be more than one) that are waiting for this value grab it from the bus and start executing their respective instructions. Since the register file also works like any other functional unit, it remembers the peer unit that is generating a value for a given register thus avoiding the WAW hazard. In this manner, this distributed solution avoids all the potential data hazards that we have discussed in this chapter.

The core of the idea in Tomasulo algorithm is the use of local registers that act as surrogates to the architecture-visible registers. Modern processors use this idea by consolidating all the distributed storage used in Tomasulo algorithm into one large physical register file on the chip. Register renaming technique that we mentioned earlier, creates a dynamic mapping between an architecture-visible register and a physical register at the time of instruction issue. For example, if register R1 is the architecture-visible source register for a store instruction, and the actual physical register assigned is say, P12, then the value of R1 is transferred into P12 in the register renaming stage. Thus, the register renaming stage of the pipeline is responsible for dynamic allocation of physical registers to the instructions. Essentially, register renaming removes the WAR and WAW data hazards. We already discussed in Section 5.13.2.2 how data forwarding solves the RAW hazard in a pipelined processor. Thus, register renaming combined with data forwarding addresses all the data hazards in modern processors. The pipeline stage that is responsible for register renaming keeps track of which physical registers are in use at any point of time and when they get freed (not unlike the scoreboard technique of CDC 6600) upon the retirement of an instruction.

The role of the re-order buffer (ROB) is to ensure that instructions retire in program order. The role of register renaming is to remove data hazards as well as to support hardware-based speculative execution. Some processors eliminate the ROB altogether and incorporate its primary functionality (namely, retiring instructions in program order) into the register renaming mechanism itself.

Let us revisit WAW hazard. We saw that the techniques (such as scoreboard and Tomasulo algorithm) used in the early pipelined machines, and the techniques (such as register renaming and ROB) now being used in modern processors eliminates WAW, despite the out-of-order execution of instructions. Does speculative execution lead to

¹² These local registers in Tomasulo algorithm perform the same function as the large physical register file in modern processors to support register renaming.

WAW hazard? The answer is no, since, despite the speculation, the instructions are retired in program order. Any incorrect write to a register due to a mispredicted branch would have been removed from the ROB without being committed to any architecture-visible register.

Despite multiple issue processors, speculation, and out-of-order processing, the original question that we raised in Section 5.13.2.4, still remains, namely, why would a compiler generate code with a WAW hazard. The answer lies in the fact that a compiler may have generated the first write to fill a delay slot (if the microarchitecture was using delayed branches) and the program took an unexpected branch wherein this first write was irrelevant. However, the out of order nature of the pipeline could mean that the second useful write may finish before the irrelevant first write. More generally, WAW hazard may manifest due to unexpected code sequences. Another example, is due to the interaction between the currently executing program and a trap handler. Let us say, an instruction, which would have written to a particular register (first write) if the execution was normal, traps for some reason. As part of handling the trap, the handler writes to the same register (second write in program order) and resumes the original program at the same instruction. The instruction continues and completes writing to the register (i.e., the first write in program order, which is now an irrelevant one due to the trap handling). If the writes do not occur in program order then this first write would overwrite the second write by the trap handler. It is the hardware's responsibility to detect such hazards and eliminate them.

5.15.5 Power Consumption

Another interesting dimension in processor design is worrying about the power dissipation. Even as it is, current GHz microprocessors dissipate a lot of power leading to significant engineering challenges in keeping such systems cool. As the processing power continues to increase, the energy consumption does too. The challenge for the architect is the design of techniques to keep the power consumption low while aspiring for higher performance.

The ability to pack more transistors on a single piece of silicon is a blessing while posing significant challenges to the architect. First of all, with the increase in density of transistors on the chip, all the delays (recall the width of the clock pulse discussion from Chapter 3) have shrunk. This includes the time to perform logic operations, wire delays, and access times to registers. This poses a challenge to the architect in the following way. In principle, the chip can be clocked at a higher rate since the delays have gone down. However, cranking up the clock increases power consumption. Figure 5.22 shows the growth in power consumption with increasing clock rates for several popular processors. You can see the high correlation between clock cycle time and the power consumption. A 3.6 GHz Pentium 4 'E' 560 processor consumes 98 watts of power. In the chart, you will see some processors with lower clock rating incurring higher power consumption (e.g. A64-4000). The reason for this is due to the fact the power consumption also depends on other on-chip resources such as caches, the design of which we will see in a later chapter.

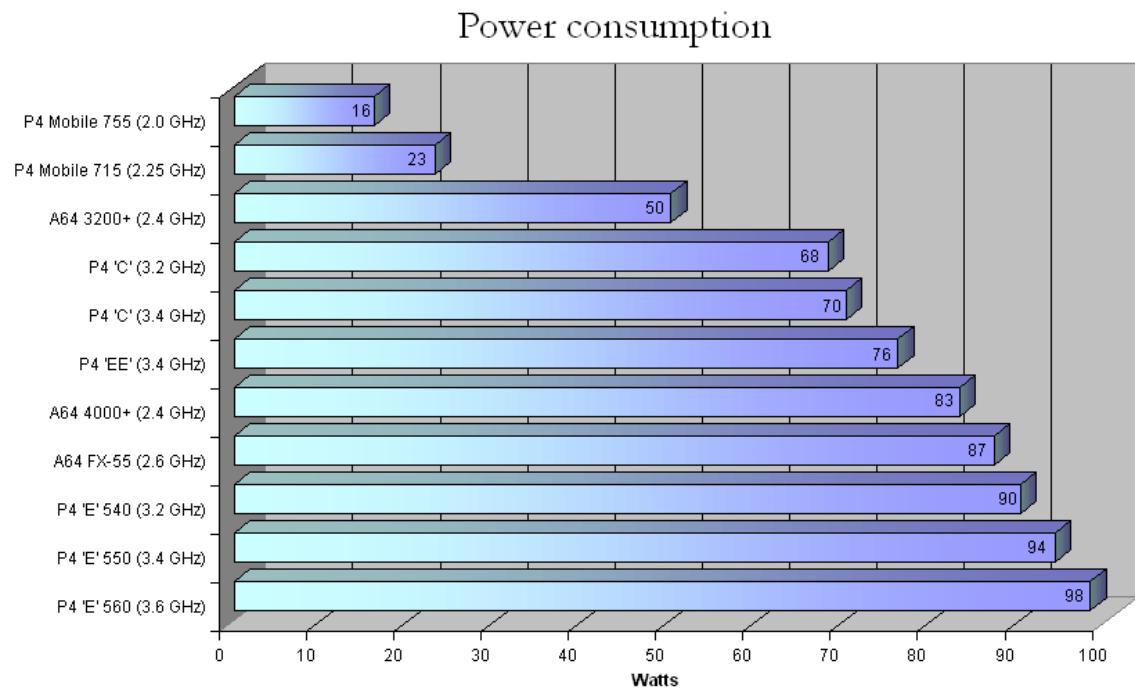


Figure 5.22: CPU Power Consumption¹³

5.15.6 Multi-core Processor Design

The reality is, as technology keeps improving, if the processor is clocked at the rate that is possible for that technology, pretty soon we will end up with a laptop whose power consumption equals that of a nuclear reactor! Of course, the solution is not to stop making chips with higher density and higher clock rates. Architects are turning to another avenue to increase performance of the processor, without increasing the clock rate, namely, *multiprocessing*. This new technology has already hit the market with the name *multi-core* (Intel Core 2 Duo, AMD Opteron quad-core, etc.). Basically, each chip has multiple processors on it and by dividing the processing to be done on to these multiple processors the system throughput, i.e., performance is increased. The architecture and hardware details underlying multi-core processors (or *chip multiprocessors* as they are often referred to) are beyond the scope of this textbook. However, multi-core technology builds on the basic principles of parallel processing that have been around as long as computer science has been around. We will discuss the hardware and software issues surrounding multiprocessors and parallel programming in much more detail in a later chapter.

¹³ Source: http://en.wikipedia.org/wiki/CPU_power_dissipation. Please note that we are not attesting to the veracity of these exact numbers but the charts give a visual ballpark of CPU power dissipation.

5.15.7 Intel Core¹⁴ Microarchitecture: An example pipeline

It is instructive to understand the pipeline structure of modern processors. Intel Pentium 4 Willamette and Galatin use a 20 stage pipeline, while Intel Pentium Prescott and Irwindale use a 31 stage pipeline. One of the chief differentiators between Intel and AMD product lines (though both support the same x86 ISA) is the depth of the pipeline. While Intel has taken the approach of supporting deeper pipelines for larger instruction throughput, AMD has taken the approach of a relatively shallower (14-stage) pipeline.

A family of Intel processors including Intel Core 2 Duo, Intel Core 2 Quad, Intel Xeon, etc., uses a common “core” microarchitecture shown in Figure 5.23. This is a much-simplified diagram to show the basic functionalities of a modern processor pipeline. The interested reader is referred to the Intel architecture manuals available freely on the Intel web site¹⁵. At a high-level the microarchitecture has a *front-end*, an *execution core*, and a *back-end*. The role of the front-end is to fetch instruction streams *in-order* from memory, with four decoders to supply the decoded instructions (also referred to as *microops*) to the execution core. The front-end is comprised of the fetch and pre-decode unit, the instruction queue, the decoders, and microcode ROM. The middle section of the pipeline is an *out-of-order* execution core that can issue up to *six* microops every clock cycle as soon as the sources for the microops are ready (i.e., there are no RAW hazards) and the corresponding execution units are available (i.e., there are no structural hazards). This middle section incorporates register renaming, re-order buffer, reservation station, and an instruction scheduler. The back-end is responsible for retiring the executed instructions in *program order*, and for updating the programmer visible architecture-registers. The functionalities of the different functional units that feature in the pipeline microarchitecture of Intel Core are as follows:

- **Instruction Fetch and PreDecode:** This unit is responsible for two things: fetching the instructions that are most likely to be executed and pre-decoding the instructions recognizing variable length instructions (since x86 architecture supports it). Pre-decode helps the instruction fetcher to recognize a branch instruction way before the outcome of the branch will be decided. An elaborate branch prediction unit (BPU) is part of this stage that helps fetching the most likely instruction stream. BPU has dedicated hardware for predicting the outcome of different types of branch instructions (conditional, direct, indirect, and call and return). The pre-decode unit can write up to six instructions every clock cycle into the instruction queue.
- **Instruction Queue:** The instruction queue takes the place of an instruction register (IR) in a simple 5-stage pipeline. Since it houses many more instructions (has a depth of 18 instructions), the instruction queue can hold a snippet of the original program (e.g., a small loop) to accelerate the execution in the pipelined processor. Further, it can also help in saving power since the rest of the front-end (namely, the instruction fetch unit) can be shut down during the execution of the loop.
- **Decode and Microcode ROM:** This unit contains four decoders, and hence can decode up to 4 instructions in the instruction queue in every clock cycle. Depending

¹⁴ Intel Core is a registered trademark of Intel Corporation.

¹⁵ Intel website: <http://www.intel.com/products/processor/manuals/index.htm>

on the instruction, the decode unit may expand it into several microops with the help of the microcode ROM. The microcode ROM can emit three microops every cycle. The microcode ROM thus facilitates implementing complex instructions without slowing down the pipeline for the simple instructions. The decoders also support *macro-fusion*, fusing together two instructions into a single microop.

- **Register Renaming/Allocation:** This unit is responsible for allocating physical registers to the architectural registers named in the microop. It keeps the mapping thus created between the architectural registers and the actual physical registers in the microarchitecture. It supports hardware-based speculation and removes WAR and WAW hazards.
- **Re-Order Buffer:** This unit has 96 entries and is responsible for registering the original program order of the microops for later reconciliation. It holds the microops in various stages of execution. There can be up to 96 microops in flight (i.e., in various stages of execution) given the size of the ROB.
- **Scheduler:** This unit is responsible for scheduling the microops on the functional units. It includes a **reservation station** that queues all the microops until their respective sources are ready and the corresponding execution unit is available. It can schedule up to six microops every clock cycle subject to their readiness for execution.
- **Functional Units:** As the name suggests these are the units that carry out the microops. They include execution units with single-cycle latency (such as integer add), pipelined execution units for frequently used longer latency microops, pipelined floating-point units, and memory load/store units.
- **Retirement Unit:** This represents the back-end of the microarchitecture and uses the re-order buffer to retire the microops in program order. It also updates the architectural states in program order, and manages the ordering of exceptions and traps that may arise during instruction execution. It also communicates with the reservation station to indicate the availability of sources that microops may be waiting on.

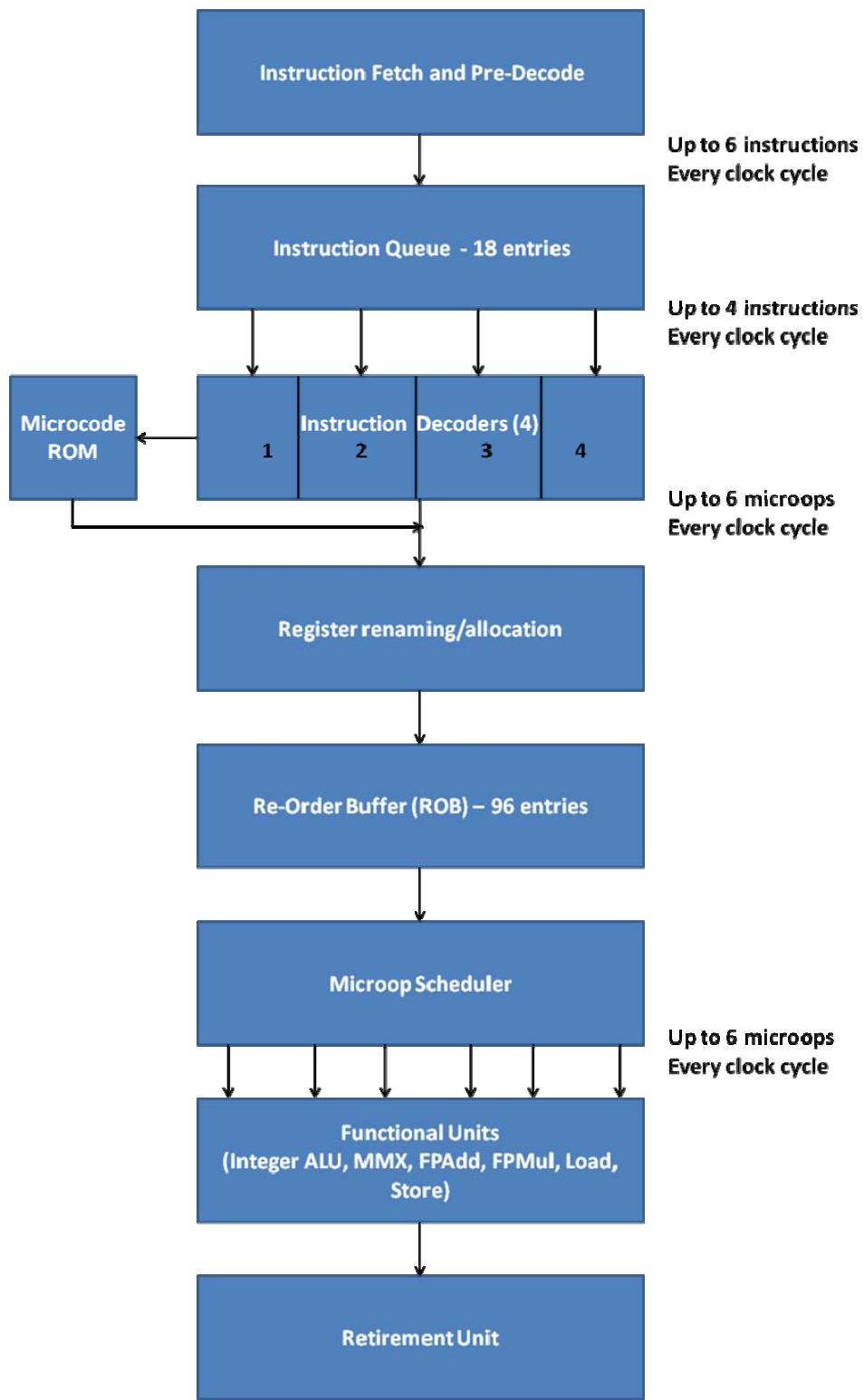


Figure 5.23: Intel Core Microarchitecture Pipeline Functionality

5.16 Summary

In this chapter, we covered a lot of ground. Metrics for evaluating processor performance were covered in Sections 5.1 and 5.2. We introduced Amdahl's law and the notion of speedup in Section 5.5. Discussion of improving processor performance led to the introduction of pipelined processor design in Section 5.7. Datapath elements needed for supporting an instruction pipeline as well as best practices for achieving a pipeline conscious architecture and implementation were discussed in Sections 5.11 and 5.12. The bane of pipelined design is hazards. Different kinds of hazards (structural, data, and control) encountered in a pipelined processor and solutions for overcoming hazards were discussed in Section 5.13. Another thorny issue in pipelined processor implementation is dealing with program discontinuities, which was discussed in Section 5.14. A flavor of advanced topics relating to pipelined processor implementation was presented in Section 5.15. We close this chapter with a discussion of how far we have come in processor implementation from the early days of computing.

5.17 Historical Perspective

Most of us have a tendency to take things for granted depending on where we are (in space and time), be it running water in faucets 24x7, high speed cars, or high-performance computers in the palm of our hand. Of course, many of us realize quickly we have come a long way in a fairly short period of time in computer technology due to the amazing pace of breakthrough development in chip integration.

It is instructive to look back at the road we have traversed. Not so long ago, pipelined processor design was reserved for high-end computers of the day. Researchers at IBM pioneered fundamental work in pipelined processor design in the 60's, which resulted in the design of systems such as the IBM 360 series and later IBM 370 series of high-end computer systems. Such systems dubbed *mainframes*, perhaps due to their packaging in large metal chassis, were targeted mainly towards business applications. Gene Amdahl, who was the chief architect of the IBM 360 series and whose name is immortalized in Amdahl's law, discovered the principles of pipelining in his PhD dissertation at UW-Madison in 1952 in the WISC computer¹⁶. Seymour Cray, a pioneer in high-performance computers, founded Cray Research, which ushered in the age of *vector supercomputers* with the Cray series starting with Cray-1. Prior to founding Cray Research, Seymour Cray was the chief architect at Control Data Corporation, which was the leader in high-performance computing in the 60's with offerings such as the CDC 6600 (widely considered as the first commercial supercomputer) and shortly thereafter, the CDC 7600.

Simultaneous with the development of high-end computers, there was much interest in the development of *minicomputers*, DEC's PDP series leading the way with PDP-8, followed by PDP-11, and later on the VAX series. Such computers were initially targeted towards the scientific and engineering community. Low cost was the primary

¹⁶ Source: http://en.wikipedia.org/wiki/Wisconsin_Integrally_Synchronized_Computer

focus as opposed to high performance, and hence these processors were designed without employing pipelining techniques.

As we observed in Chapter 3, the advent of the killer micros in 80's coupled with pioneering research in compiler technology and RISC architectures, paved the way for instruction pipelines to become the implementation norm for processor design except for very low-end embedded processors. Today, even game consoles in the hands of toddlers use processors that employ principles of pipelining.

Just as a point of clarification of terminologies, supercomputers were targeted towards solving computationally challenging problems that arise in science and engineering (they were called *grand challenge problems*, a research program started by DARPA¹⁷ to stimulate breakthrough computing technologies), while mainframes were targeted towards technical applications (business, finance, etc.). These days, it is normal to call such high-end computers as *servers* that are comprised of a collection of computers interconnected by high-speed network (also known as *clusters*). Such servers cater to both scientific applications (e.g., IBM's BlueGene massively parallel architecture), as well as technical applications (e.g., IBM z series). The processor building block used in such servers are quite similar and adhere to the principles of pipelining that we discussed in this chapter.

5.18 Review Questions

1. Answer True or false with justification: For a given workload and a given ISA, reducing the CPI (clocks per instruction) of all the instructions will always improve the performance of the processor.
2. An architecture has three types of instructions that have the following CPI:

Type	CPI
A	2
B	5
C	3

An architect determines that he can reduce the CPI for B by some clever architectural trick, with no change to the CPIs of the other two instruction types. However, she determines that this change will increase the clock cycle time by 15%. What is the maximum permissible CPI of B (round it up to the nearest integer) that will make this change still worthwhile? Assume that all the workloads that execute on this processor use 40% of A, 10% of B, and 50% of C types of instructions.

3. What would be the execution time for a program containing 2,000,000 instructions if the processor clock was running at 8 MHz and each instruction takes 4 clock cycles?
4. A smart architect re-implements a given instruction-set architecture, halving the CPI for 50% of the instructions, while increasing the clock cycle time of the processor by

¹⁷ DARPA stands for a Federal Government entity called *Defense Advanced Research Projects Agency*

10%. How much faster is the new implementation compared to the original? Assume that the usage of all instructions are equally likely in determining the execution time of any program for the purposes of this problem.

5. A certain change is being considered in the non-pipelined (multi-cycle) MIPS CPU regarding the implementation of the ALU instructions. This change will enable one to perform an arithmetic operation and write the result into the register file all in one clock cycle. However, doing so will increase the clock cycle time of the CPU. Specifically, the original CPU operates on a 500 MHz clock, but the new design will only execute on a 400 MHz clock.

Will this change improve, or degrade performance? How many times faster (or slower) will the new design be compared to the original design? Assume instructions are executed with the following frequency:

Instruction	Frequency
LW	25%
SW	15%
ALU	45%
BEQ	10%
JMP	5%

Compute the CPI of both the original and the new CPU. Show your work in coming up with your answer.

Cycles per Instruction	
Instruction	CPI
LW	5
SW	4
ALU	4
BEQ	3
JMP	3

6. Given the CPI of various instruction classes

Class	CPI
R-type	2
I-type	10
J-type	3
S-type	4

And instruction frequency as shown:

Class	Program 1	Program 2
R	3	10
I	3	1
J	5	2
S	2	3

Which code will execute faster and why?

7. What is the difference between static and dynamic instruction frequency?

8. Given

Instruction	CPI
Add	2
Shift	3
Others	2 (average for all instructions including Add and Shift)
Add/Shift	3

If the sequence ADD followed by SHIFT appears in 20% of the dynamic frequency of a program, what is the percentage improvement in the execution time of the program with all {ADD, SHIFT} replaced by the new instruction?

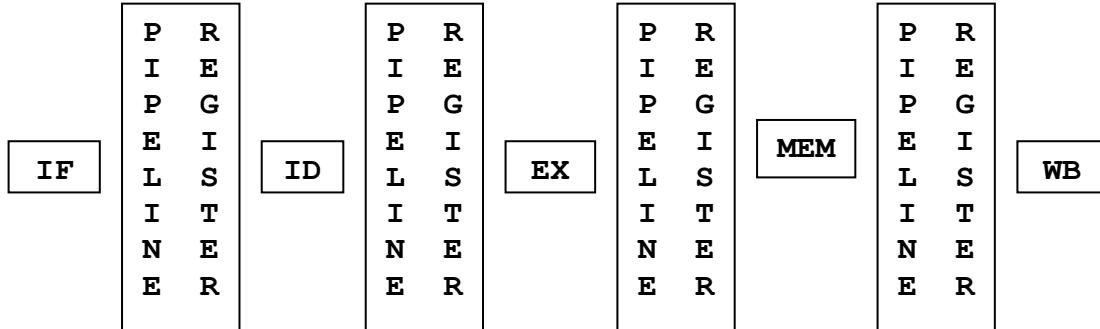
9. Compare and contrast structural, data and control hazards. How are the potential negative effects on pipeline performance mitigated?

10. How can a read after write (RAW) hazard be minimized or eliminated?

11. What is a branch target buffer and how is it used?

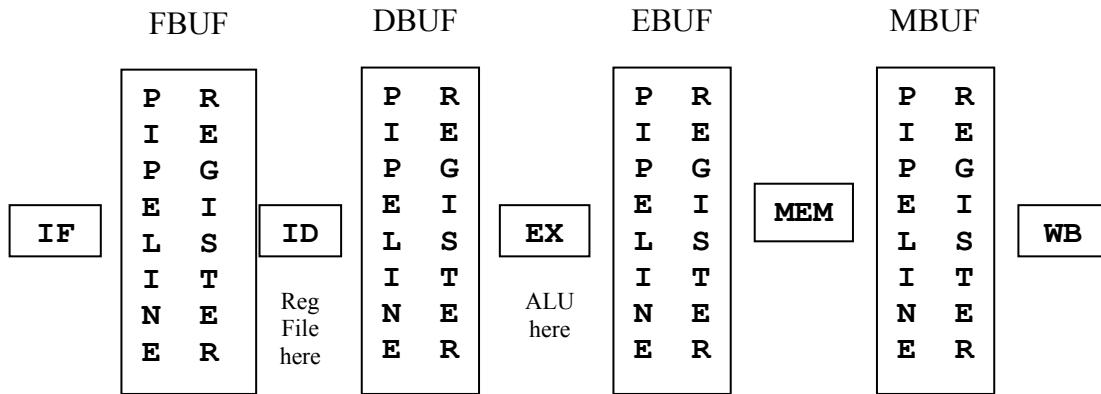
12. Why is a second ALU needed in the Execute stage of the pipeline for a 5-stage pipeline implementation of LC-2200?

13. In a processor with a five stage pipeline as shown in the picture below (with buffers between the stages), explain the problem posed by a branch instruction. Present a solution.



14. Regardless of whether we use a conservative approach or branch prediction (“branch not taken”), explain why there is always a 2-cycle delay if the branch is taken (i.e., 2 NOPs injected into the pipeline) before normal execution can resume in the 5-stage pipeline used in Section 5.13.3.
15. With reference to Figure 5.6a, identify and explain the role of the datapath elements that deal with the BEQ instruction. Explain in detail what exactly happens cycle by cycle with respect to this datapath during the passage of a BEQ instruction. Assume a conservative approach to handling the control hazard. Your answer should include both the cases of branch taken and branch not taken.
16. A smart engineer decides to reduce the 2-cycle “branch taken” penalty in the 5-stage pipeline down to 1. Her idea is to directly use the branch target address computed in the EX cycle to fetch the instruction (note that the approach presented in Section 5.13.3 requires the target address to be saved in PC first).
 - a) Show the modification to the datapath in Figure 5.6a to implement this idea [hint: you have to simultaneously feed the target address to the PC and the Instruction memory if the branch is taken].
 - b) While this reduces the bubbles in the pipeline to 1 for branch taken, it may not be a good idea. Why? [hint: consider cycle time effects.]
17. In a pipelined processor, where each instruction could be broken up into 5 stages and where each stage takes 1 ns what is the best we could hope to do in terms of average time to execute 1,000,000,000 instructions?
18. Using the 5-stage pipeline shown in Figure 5.6b answer the following two questions:
 - a. Show the actions (similar to Section 5.12.1) in each stage of the pipeline for BEQ instruction of LC-2200.

- b. Considering only the BEQ instruction, compute the sizes of the FBUF, DBUF, EBUF, and MBUF.
19. Repeat problem 18 for the SW instruction of LC-2200.
20. Repeat problem 18 for JALR instruction of LC-2200.
21. You are given the pipelined datapath for a processor as shown below.



A load-word instruction has the following 32-bit format:

OPCode	A Reg	B Reg	Offset
8 bits	4 bits	4 bits	16 bits

The semantic of this instruction is

$$B \leftarrow \text{Memory}[A + \text{offset}]$$

Register B gets the data at the memory address given by the effective address generated by adding the contents of Register A to the 16-bit offset in the instruction. All data and addresses are 32-bit quantities.

Show what is needed in the pipeline registers (FBUF, DBUF, EBUF, MBUF) between the stages of the pipeline for the passage of this instruction. Clearly show the layout of each buffer. Show the width of each field in the buffer. You do not have to concern yourself about the format or the requirements of other instructions in the architecture for this problem.

22. Consider



If I_2 is immediately following I_1 in the pipeline with no forwarding, how many bubbles (i.e. NOPs) will result in the above execution? Explain your answer.

23. Consider the following program fragment:

Address	instruction
1000	add
1001	nand
1002	lw
1003	add
1004	nand
1005	add
1006	sw
1007	lw
1008	Add

Assume that there are no hazards in the above set of instructions. Currently the IF stage is about to fetch instruction at 1004.

- (a) Show the state of the 5-stage pipeline
- (b) Assuming we use the “drain” approach to dealing with interrupts, how many cycles will elapse before we enter the INT macro state? What is the value of PC that will be stored in the INT macro state into \$k0?
- (c) Assuming the “flush” approach to dealing with interrupts, how many cycles will elapse before we enter the INT macro state? What is the value of PC that will be stored in the INT macro state into \$k0?

Chapter 6 Processor Scheduling

(Revision number 15)

6.1 Introduction

Processor design and implementation is no longer a mystery to us. From the earlier chapters, we know how to design instruction-sets, how to implement the instruction-set using an FSM, and how to improve upon the performance of a processor using techniques such as pipelining.

Fans of Google Earth would have experienced the cool effect of looking at a country or a continent from a distance and then if needed zoom into see every street name and houses of a city they are interested in exploring further.

We just did that for the processor. After understanding how a processor ISA is designed, we zoomed into the details of how to implement the processor using LC-2200 ISA as a concrete example, and got a good grasp of the details from a hardware perspective. Now it is time to zoom out and look at the processor as a black box, a precious and scarce resource. Several programs may have to run on this resource (Google Earth, e-mail, browser, IM, ...) and the system software has to manage this resource effectively to cater to the needs of the end user.

Therefore, we turn our attention to a complementary topic, namely, how to manage the processor as a resource in a computer system. To do this, we do not need to know the internals of the processor. That is the power of abstraction. Viewing the processor as a black box, we will figure out the software abstractions that would be useful in managing this scarce resource. The portion of the operating system that deals with this functionality is *processor scheduling*, which is the topic of discussion in this chapter. Efficient implementation of this functionality may necessitate revisiting (i.e., zooming into) the processor ISA and adding additional smarts to the ISA. We will re-visit this issue towards the end of the chapter (see Section XX).

Let us consider a simple analogy. You have laundry to do. You have tests to prepare for. You have to get some food ready for dinner. You have to call mom and wish her happy birthday. There is only one of you and you have to get all of this done in a timely manner. You are going to prioritize these activities but you also know that not all of them need your constant attention. For example, once you get the washing machine for the laundry started until it beeps at you at the end of the cycle, you don't have to pay attention to it. Similarly, with our zapping culture, all you need to do to get dinner ready is to stick the "TV dinner" into the microwave and wait until that beeps. So, here is a plausible schedule to get all of this done:

1. Start the wash cycle
2. Stick the food in the microwave and zap it
3. Call Mom
4. Prepare for tests

Notice, that your attention is needed only for a very short time for the first two tasks (relative to tasks 3 and 4). There is a problem, however. You have no idea how long tasks 3 and 4 are going to take. For example, Mom's call could go on and on. The washer may beep at you and/or the microwave might beep at you while mom is still on the line. Well, if the washer beeps at you, you could politely tell your mom to hold for a minute, go transfer the load from the washer to dryer and come back to mom's phone call. Likewise, you could excuse yourself momentarily to take the food out of the microwave and keep it on the dinner table ready to eat. Once you are done with mom's call, you are going to eat dinner peacefully, then start preparing for the tests. You have a total of 8 hours for studying and you have four courses to prepare for. All the tests are equally important for your final course grades. In preparing for the tests, you have a choice to make. You could either spend a little bit of time on each of the courses and cycle through all the courses to ensure that you are making progress on all of them. Alternatively, you could study for the tests in the order in which you have to take them. At this point, you are either scratching your head asking yourself what does any of this have to do with processor scheduling, or more than likely you can already see what is going on here. You are the scarce resource. You are partitioning your time to allocate yourself to these various tasks. You have given higher priority to mom's phone call relative to preparing for the test. You will see later when we discuss processor scheduling algorithms a similar notion of *priority*. You have given starting the wash cycle and the microwave higher priority compared to the other two tasks. This is because you know that these require very little of your time. You will see a similar notion of *shortest job first* scheduling policy later on for processors. When the washer beeps while you are still on the phone, you temporarily put your mom on hold while you attend to the washer. You will see a similar concept, namely *preemption*, later on in the context of processor scheduling. In studying for the tests, the first choice you could make is akin to a processor scheduling policy we will see later on called *round robin*. The second choice is similar to the classic *first come first served* processor-scheduling policy.

6.2 Programs and Processes

Let's start the discussion of processor scheduling with a very basic understanding of what an operating system is. It is just a *program*, whose sole purpose is to supply the resources needed to execute users' programs.

To understand the resource requirements of a user program, let us review how we create programs in the first place. Figure 6.1 shows a plausible layout of the memory footprint of a program written in a high-level language such as C.

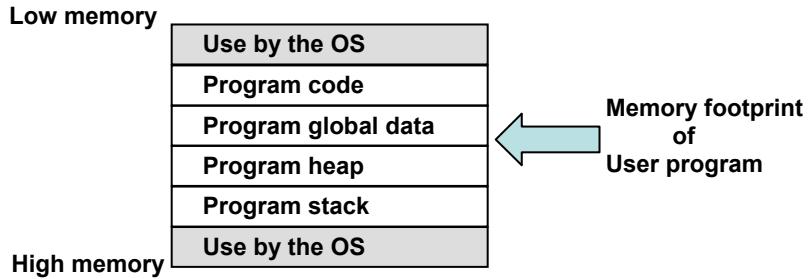


Figure 6.1: Memory Footprint

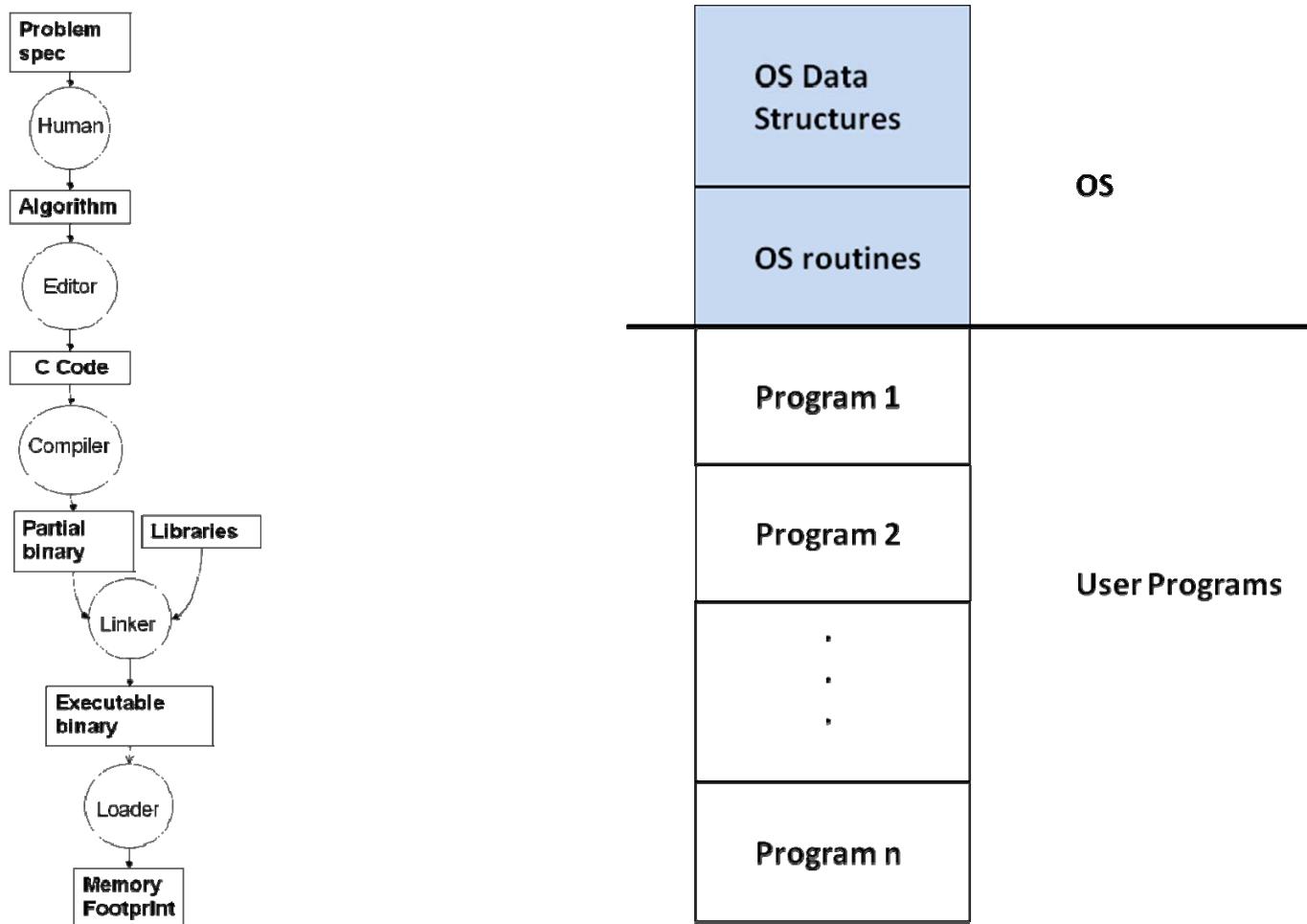


Figure 6.2: Life cycle of program creation

Figure 6.3: OS and user program in memory

We use the term *program* in a variety of connotations. However, generally we use the term to denote a computer solution for a problem. The program may exist in several different forms.

Figure 6.2 shows the life cycle of program creation in a high-level language. First, we have a problem specification from which we develop an algorithm. We code the

algorithm in some programming language (say C) using an editor. Both the algorithm and the C code are different representations of the *program*, a means of codifying our solution to a problem. A compiler compiles the C code to produce a binary representation of the program. This representation is still not “executable” by a processor. This is because the program we write uses a number of facilities that we take for granted and are supplied to us by “someone else”. For example, we make calls to do terminal I/O (such as *scanf* and *printf*), and calls to do mathematical operations (such as *sine* and *cosine*). Therefore, the next step is to *link* together our code with that of the libraries provided by someone else that fill in the facilities we have taken for granted. This is the work of the linker. The output of the linker is once again a binary representation of the program, but now it is in a form that is ready for execution on the processor. These different representations of your program (English text, unlinked binary, and executable binary) ultimately end up on your hard drive usually. *Loader* is another program that takes the disk representation and creates the memory footprint shown in Figure 6.1.

Each of Editor, Compiler, Linker, and Loader are themselves programs. Any program needs resources to execute. The resources are processor, memory, and any input/output devices. Let us say you write a simple “Hello World” program. Let us enumerate the resources needed to run this simple program. You need the processor and memory of course; in addition, the display to send your output. The operating system gives the resources needed by the programs.

As should be evident from the discussion so far, just like any other program the operating system is also memory resident and has a footprint similar to any other program. Figure 6.3 shows the memory contents of user programs and the operating system.

In this chapter, we will focus on the functionality of the operating system that deals with allocating the processor resource to the programs, namely, the *scheduler*.

A scheduler is a set of routines that is part of the operating system; just like any other program, the scheduler also needs the processor to do its work, namely, of selecting a program to run on the processor. The scheduler embodies an algorithm to determine a winner among the set of programs that need cycles on the processor.

You have heard the term *process*. Let us understand what a process is and how it differs from a program. *A process is a program in execution*. With reference to Figure 6.1, we define the *address space* of a process as the space occupied in memory by the program. Once a program starts executing on the processor, the contents of the memory occupied by the program may change due to manipulation of the data structures of the program. In addition, the program may use the processor registers as part of its execution. The current contents of the address space and the register values together constitute the *state* of a program in execution (i.e., the state of a process). We will shortly see how we can concisely represent the state of a process.

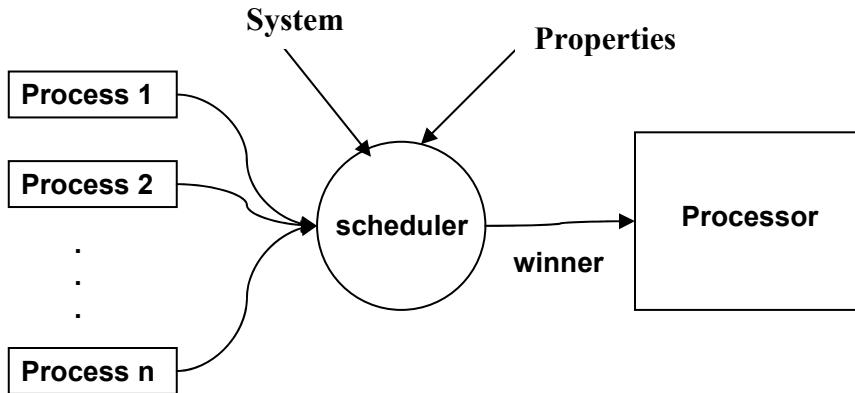


Figure 6.4: Scheduler

Quite often, a process may also be the unit of scheduling on the processor. The inputs to the scheduler are the set of processes that are ready to use the processor, and additional properties that help the scheduler to pick a winner from among the set of ready processes as shown in Figure 6.4. The properties allow the scheduler to establish a sense of *priority* among the set of processes. For example, *expected running time*, *expected memory usage*, and *expected I/O requirements* are *static* properties associated with a program. Similarly, *available system memory*, *arrival time of a program*, and *instantaneous memory requirements of the program* are *dynamic* properties that are available to the scheduler as well. *Urgency* (either expressed as *deadlines*, and/or *importance*) may be other extraneous properties available to the scheduler. Some of these properties (such as urgency) are explicitly specified to the scheduler; while the scheduler may infer others (such as arrival time).

Quite often, terminologies such as *tasks* and *threads* denote units of work and/or units of scheduling. We should warn the reader that while the definition of process is uniform in the literature, the same could not be said for either a task or a thread. In most literature, a task has the same connotation as a process. In this chapter, we will use the term task only to mean a unit of work. We will give a basic definition of threads that is useful from the point of view of understanding the scheduling algorithms.

An analogy will be useful here. You get the morning newspaper. It is lying on the breakfast table (Figure 6.5-(a)). Nobody is reading it yet. That is like a program dormant in memory. You pick it up and start reading (Figure 6.5-(b)).



Figure 6.5: You and your sibling reading the newspaper

Now there is one active entity reading the paper, namely, you. Notice that depending on your interest, you will read different sections of the paper. A process is similar. Depending on the input and the logic of the program, a process may traverse a particular path through the program. Thus, this path defines a *thread of control* for the process. Let us motivate why it makes sense to have multiple threads of control within the process by returning to our paper reading analogy. Imagine now, as you are reading the paper, your sibling joins you at the breakfast table and starts reading the paper as well (Figure 6.5-(c)). Depending on his interests, perhaps he is going to start reading a different section of the paper. Now there are two activities (you and your sibling) or two threads of control navigating the morning paper. Similarly, there could be multiple threads of control within a single process. We will elaborate on why having multiple threads in a process may be a good idea, and the precise differences between a thread and a process later in the context of multiprocessors and multithreaded programs. At this point, it is sufficient to understand a thread as a unit of execution (and perhaps scheduling as well) contained within a process. All the threads within a process execute in the same address space, sharing the code and data structures shown in the program's memory footprint (Figure 6.1). *In other words, a process is a program plus all the threads that are executing in that program.* This is analogous to the newspaper, your sibling, and you, all put together.

Name	Usual Connotation	Use in this chapter
Job	Unit of scheduling	Synonymous with process
Process	Program in execution; unit of scheduling	Synonymous with job
Thread	Unit of scheduling and/or execution; contained within a process	Not used in the scheduling algorithms described in this chapter
Task	Unit of work; unit of scheduling	Not used in the scheduling algorithms described in this chapter, except in describing the scheduling algorithm of Linux

Table 6.1: Jobs, Processes, Threads, and Tasks

There could be multiple threads within a single process but from the point of view of the scheduling algorithms discussed in this chapter, a process has a single thread of control. Scheduling literature uses the term *job* as a unit of scheduling and to be consistent we have chosen to use this term synonymously with process in this chapter. Just as a point of clarification, we summarize these terminologies and the connotations associated with them in Table 6.1.

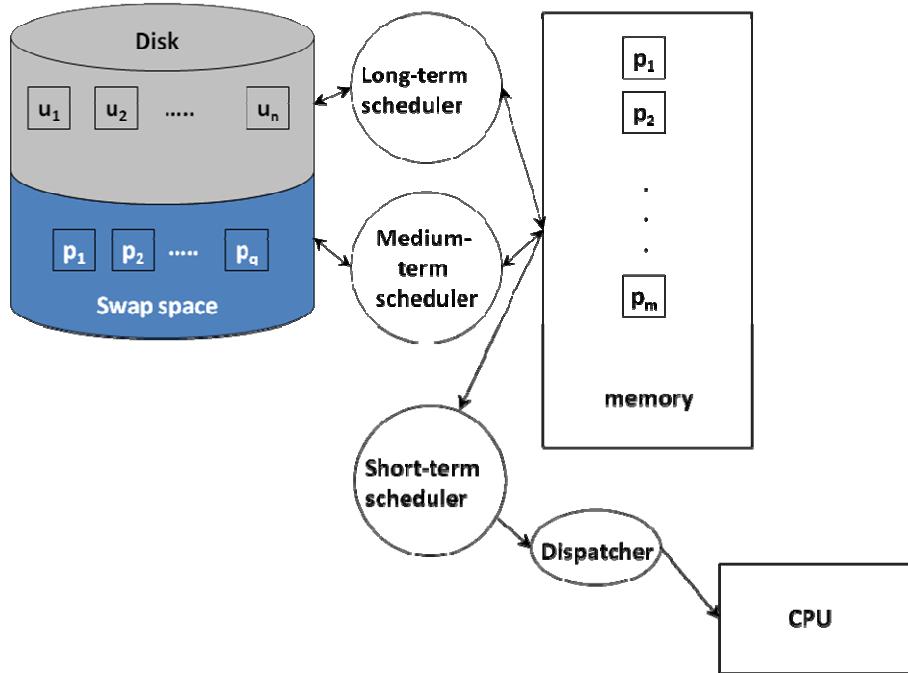


Figure 6.6: Types of schedulers
(u_i represents user programs on the disk;
 p_i represents user processes in memory)

6.3 Scheduling Environments

In general, a processor may be dedicated for a specific set of tasks. Take for example a processor inside an embedded device such as a cell phone. In this case, there may be a task to handle the ringing, one to initiate a call, etc. A scheduler for such a dedicated environment may simply cycle through the tasks to see if any one of them is ready to run.

In the age of large batch-oriented processing (60's, 70's, and early 80's), the scheduling environment was *multiprogrammed*; i.e., there were multiple programs loaded into memory from the disk and the operating system cycled through them based on their relative priorities. These were the days when you handed to a human operator, punched cards containing a description of your program (on a disk) and its execution needs, in a language called *job control language (JCL)*. Typically, you would come back several hours later to collect your output. With the advent of data terminals and mini-computers, *interactive* or *time-sharing* environments became feasible. The processor is time-shared among the set of interactive users sitting at terminals and accessing the computer. *It is important to note that a time-shared environment is necessarily multiprogrammed, whereas a multiprogrammed environment need not necessarily be time-shared.* These different environments gave rise to different kinds of schedulers as well (see Figure 6.6).

A *long-term scheduler*, typically used in batch-oriented multiprogrammed environments, balances the job mix in memory to optimize the use of resources in the system (processor, memory, disk, etc.). With the advent of personal computing and time-shared environments, long-term schedulers are for all practical purposes non-existent in most modern operating systems. Instead, a component of the operating system called *loader* creates a memory footprint when the user (for e.g., by clicking an icon on the desktop or typing a program name from a command line) starts a program that is resident on the disk. The long-term scheduler (or the loader) is responsible for creating the memory resident processes (p_i) out of the disk resident user programs (u_i).

A *medium term scheduler*, used in many environments including modern operating systems, closely monitors the dynamic memory usage of the processes currently executing on the CPU and makes decisions on whether or not to increase or decrease the *degree of multiprogramming*, defined as the number of processes coexisting in memory and competing for the CPU. This scheduler is primarily responsible for controlling a phenomenon called *thrashing*, wherein the current memory requirements of the processes exceed the system capacity, resulting in the processes not making much progress in their respective executions. The medium-term scheduler moves programs back and forth between the disk (shown as *swap* space in Figure 6.6) and memory when the throughput of the system reduces. We will re-visit the thrashing concept in much more detail in Chapter 8.

A *short-term scheduler*, found in most modern operating systems, made its first appearance in time-sharing systems. This scheduler is responsible for selecting a process to run from among the current set of memory resident processes. The focus of this chapter as well as the algorithms presented mostly concern the short-term scheduler. Lastly, a *dispatcher* is an entity that takes the process selected by the short-term scheduler and sets up the processor registers in readiness for executing that process. Long-term scheduler, medium-term scheduler, short-term scheduler, and dispatcher are all components of the operating system and coordinate their activities with one another. Table 6.2 summarizes the different types of schedulers found in different environments and their respective roles.

Name	Environment	Role
Long term scheduler	Batch oriented OS	Control the job mix in memory to balance use of system resources (CPU, memory, I/O)
Loader	In every OS	Load user program from disk into memory
Medium term scheduler	Every modern OS (time-shared, interactive)	Balance the mix of processes in memory to avoid thrashing
Short term scheduler	Every modern OS (time-shared, interactive)	Schedule the memory resident processes on the CPU

Dispatcher	In every OS	Populate the CPU registers with the state of the process selected for running by the short-term scheduler
-------------------	-------------	---

Table 6.2: Types of Schedulers and their roles

6.4 Scheduling Basics

It is useful to understand program behavior before delving deep into the scheduler. Imagine a program that plays music on your computer from a CD player. The program repeatedly reads the tracks of the CD (I/O activity) and then renders the tracks (processor activity) read from the CD player to the speakers. It turns out that this is typical program behavior, cycling between bursts of activity on the processor and I/O devices (see Figure 6.7).

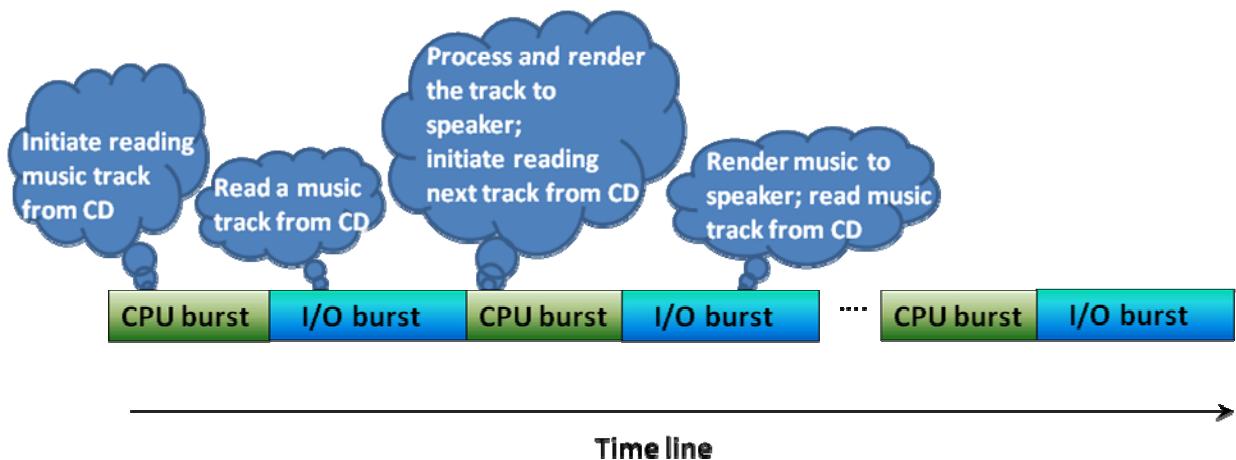


Figure 6.7: Music playing on your CD player

We will use the term *CPU burst* to denote the stretch of time a process would run without making an I/O call. Informally, we define CPU burst of a process as the time interval of continuous CPU activity by the process before making an I/O call. Using the analogy from the beginning of the chapter, CPU burst is similar to stretch of time you would do continuous reading while preparing for a test before going to the fridge for a soda. Similarly, we will use the term *I/O burst* to denote the stretch of time a process would need to complete an I/O operation (such as reading a music file from the CD). It is important to note that during an I/O burst the process does not need to use the processor. In Chapter 4, we introduced the concept of interrupts and its use to grab the attention of the processor for an external event such as I/O completion. Later in Chapter 10, we will discuss the actual mechanics of data transfer to/from the I/O device from/to the processor and memory. For now, to keep the discussion of processor scheduling simple, assume that upon an I/O request by a process, it is no longer in contention for the CPU resource until its I/O is complete.

Processor schedulers are divided into two broad categories: *non-preemptive* and *preemptive*. A process either executes to completion or gives up the processor on its own accord, i.e., voluntarily, (to perform I/O) in a non-preemptive scheduler. On the other hand, the scheduler yanks the processor away from the current process to give it to another process in a preemptive scheduler. In either case, the steps involved in scheduling are the following:

1. Grab the attention of the processor.
2. Save the state of the currently running process.
3. Select a new process to run.
4. Dispatch the newly selected process to run on the processor.

The last step *dispatch* refers to loading the processor registers with the saved state of the selected process.

Let us understand the state of the running program or process. It includes where we are currently executing in the program (PC value); what the contents of the processor registers are (assuming that one of these registers is also the stack pointer); and where in memory is the program's footprint. Besides, the process itself may be newly loaded into memory, ready to run on the processor, or waiting for I/O, currently running, halted for some reason, etc.

```
enum state_type {new, ready, running, waiting, halted};

typedef struct control_block_type {
    enum state_type state;           /* current state */
    address PC;                     /* where to resume */
    int reg_file[NUMREGS];          /* contents of GPRs */
    struct control_block *next_pcb;  /* list ptr */
    int priority;                  /* extrinsic property */
    address address_space;          /* where in memory */
    ...
    ...
} control_block;
```

Figure 6.8: Process Control Block (PCB)

Additionally, properties (intrinsic or extrinsic) that the scheduler may know about a process such as process priority, arrival time of the program, and expected execution time. All this state information is aggregated into a data structure, *process control block (PCB)*, shown in Figure 6.8. There is one PCB for each process and the scheduler maintains all the PCBs in a linked list, the *ready queue*, as shown in Figure 6.9.



Figure 6.9: Ready queue of PCBs

The PCB has all the information necessary to describe the process. It is a key data structure in the operating system. As we will see in later chapters that discuss memory systems and networking, the PCB is the aggregation of all the state information associated with a process (such as memory space occupied, open files, and network connections). Ready queue is the most important data structure in the scheduler. Efficient representation and manipulation of this data structure is a key to the performance of the scheduler. The role of the scheduler is to quickly make its scheduling decision and get out of the way, thus facilitating the use of the CPU for running user programs. Therefore, a key question is identifying the right metrics for evaluating the *efficiency* of a scheduling algorithm. Intuitively, we would like the time spent in the scheduler to be a small percentage of the total CPU time. We will see in a case study (Section 6.12), how Linux scheduler organizes its data structures to ensure high efficiency.

Note that we do not have the internal registers of the CPU datapath (Chapters 3 and 5) as part of the process state. The reason for this will become clear towards the end of this subsection.

Similar to the ready queue of process control blocks, the operating system maintains queues of PCBs of processes that are waiting on I/O (see Figure 6.10).

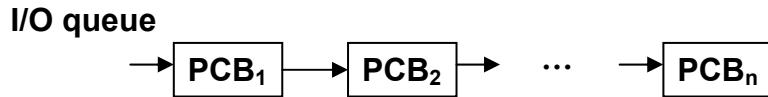


Figure 6.10: I/O queue of PCBs

For the purposes of this discussion, the PCBs go back and forth between the ready queue and the I/O queue depending on whether a process needs the processor or I/O service.

The CPU scheduler uses the ready queue for scheduling processes on the processor. Every one of the scheduling algorithms we discuss in this chapter assumes such a ready queue. The organization of the PCBs in the ready queue depends on the specifics of the scheduling algorithm. The PCB data structure simplifies the steps involved in scheduling which we identified earlier in this section. The scheduler knows exactly which PCB corresponds to the currently running process. The state saving step simply becomes a matter of copying the relevant information (identified in Figure 6.8) into the PCB of the currently running process. Similarly, once the scheduler chooses a process as the next candidate to run on the processor, dispatching it on the processor is simply a matter of populating the processor registers with the information contained in the PCB of the chosen process.

From Chapter 4, we know that system calls (such as an I/O operation) and interrupts are all different flavors of program discontinuities. The hardware treats all of these program discontinuities similarly, waiting for a clean state of the processor before dealing with the discontinuity. Completion of an instruction execution is such a clean state of the processor. Once the processor has reached such a clean state, the registers that are

internal to the processor (i.e., not visible to the programmer) do not contain any information of relevance to the currently running program. Since, the scheduler switches from one process to another triggered by such well-defined program discontinuities, there is no need to save the internal registers of the processor (discussed in Chapter 3 and 5) that are not visible to the programmer through the instruction-set architecture.

Table 6.3 summarizes the terminologies that are central to scheduling algorithms.

Name	Description
CPU burst	Continuous CPU activity by a process before requiring an I/O operation
I/O burst	Activity initiated by the CPU on an I/O device
PCB	Process context block that holds the state of a process (i.e., program in execution)
Ready queue	Queue of PCBs that represent the set of memory resident processes that are ready to run on the CPU
I/O queue	Queue of PCBs that represent the set of memory resident processes that are waiting for some I/O operation either to be initiated or completed
Non-Preemptive algorithm	Algorithm that allows the currently scheduled process on the CPU to voluntarily relinquish the processor (either by terminating or making an I/O system call)
Preemptive algorithm	Algorithm that forcibly takes the processor away from the currently scheduled process in response to an external event (e.g. I/O completion interrupt, timer interrupt)
Thrashing	A phenomenon wherein the dynamic memory usage of the processes currently in the ready queue exceeds the total memory capacity of the system

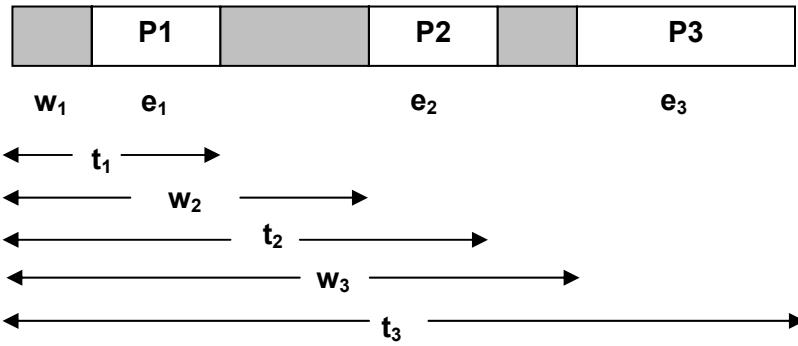
Table 6.3: Scheduling Terminologies

6.5 Performance Metrics

In discussing scheduling algorithms, we use the term *jobs* and *processes* synonymously. Scheduler, a part of the operating system, is also a program and needs to run on the processor. Ultimately, the goal of the scheduler is to run user programs. Therefore, a useful way to think about putting the processor to good use is by using it for running user programs and not the operating systems itself. This brings us to the question of the right metrics to use for evaluating the efficiency of a scheduling algorithm. *CPU Utilization* refers to the percentage of time the processor is busy. While this is a useful metric, it does not tell us what the processor is doing. So let us try some other metrics. The metrics could be *user centric* or *system centric*. *Throughput* is a system centric metric that measures the number of jobs executed per unit time. *Average turnaround time* is another system centric metric that measures the average elapsed time for jobs entering and leaving the system. *Average waiting time* of jobs is another system centric metric.

Response time of a job is a user centric metric that measures the elapsed time for a given job.

Figure 6.11 shows a time line of scheduling three processes P1, P2, and P3 on the processor. Assume that all the processes start at time '0'. For the sake of this discussion, assume that in the shaded regions the processor is busy doing something else unrelated to running these processes.



w_i , e_i , and t_i , are respectively the wait time, execution time, and the elapsed time for a job j_i

Figure 6.11: Time line of scheduling three processes P1, P2, P3

With respect to Figure 6.11,

$$\text{Throughput} = 3/t_3 \text{ jobs/sec}$$

$$\text{Average turnaround time} = (t_1 + t_2 + t_3)/3 \text{ secs}$$

$$\text{Average wait time} = ((t_1 - e_1) + (t_2 - e_2) + (t_3 - e_3))/3 \text{ secs}$$

Generalizing for n jobs,

$$\text{Throughput} = n/T \text{ jobs/sec, where } T \text{ is the total elapsed time for all } n \text{ jobs to complete,}$$

$$\text{Average turnaround time} = (t_1 + t_2 + \dots + t_n)/n \text{ secs}$$

$$\text{Average wait time} = (w_1 + w_2 + \dots + w_n)/n \text{ secs}$$

Response time, is the same as per-process turnaround time,

$$R_{P1} = t_1$$

$$R_{P2} = t_2$$

$$R_{P3} = t_3$$

....

$$R_{Pn} = t_n$$

Variance¹ in response time is a useful metric as well. In addition to these quantitative metrics, we should mention two *qualitative* metrics of importance to scheduling algorithms:

- **Starvation:** In any job mix, the scheduling policy should make sure that all the jobs make forward progress towards completion. We refer to the situation as *starvation*, if

¹ Variance in response time is the average of the squared distances of the possible values for response times from the expected value.

for some reason a job does not make any forward progress. The quantitative manifestation of this situation is an unbounded response time for a particular job.

- **Convoy effect:** In any job mix, the scheduling policy should strive to prevent long-running jobs from dominating the CPU usage. We refer to the situation as *convoy effect*, if for some reason the scheduling of jobs follows a fixed pattern (similar to a military convoy). The quantitative manifestation of this phenomenon is a high variance in the response times of jobs.

Name	Notation	Units	Description
CPU Utilization	-	%	Percentage of time the CPU is busy
Throughput	n/T	Jobs/sec	System-centric metric quantifying the number of jobs n executed in time interval T
Avg. Turnaround time (t_{avg})	$(t_1+t_2+\dots+t_n)/n$	Seconds	System-centric metric quantifying the average time it takes for a job to complete
Avg. Waiting time (w_{avg})	$((t_1-e_1) + (t_2-e_2) + \dots + (t_n-e_n))/n$ or $(w_1+w_2+\dots+w_n)/n$	Seconds	System-centric metric quantifying the average waiting time that a job experiences
Response time/turnaround time	t_i	Seconds	User-centric metric quantifying the turnaround time for a specific job i
Variance in Response time	$E[(t_i - e_i)^2]$	Seconds ²	User-centric metric that quantifies the statistical variance of the actual response time (t_i) experienced by a process (P_i) from the expected value (t_{avg})
Starvation	-	-	User-centric qualitative metric that signifies denial of service to a particular process or a set of processes due to some intrinsic property of the scheduler
Convoy effect	-	-	User-centric qualitative metric that results in a detrimental effect to some set of processes due to some intrinsic

			property of the scheduler
--	--	--	---------------------------

Table 6.4: Summary of Performance Metrics

We will discuss several scheduling algorithms in the next few sections. Before we do that, a few caveats:

- In all the scheduling algorithms, we assume the time to switch from one process to another is negligible to make the timing diagrams for the schedules simple.
- We mentioned that a process might go back and forth between CPU and I/O requests during its lifetime. Naturally, the I/O requests may be to different devices at different times (output to the screen, read from the disk, input from the mouse, etc.). However, since the focus in this chapter is on CPU scheduling, for simplicity we just show one I/O queue.
- Once again, to keep the focus on CPU scheduling, we assume a simple model (first-come-first-served) for scheduling I/O requests. In other words, intrinsic or extrinsic properties of a process that the CPU scheduler uses, do not apply to I/O scheduling. The I/O requests are serviced in the order in which they are made by the processes.

Table 6.4 summarizes the performance metrics of interest from the point of view of scheduling.

6.6 Non-preemptive Scheduling Algorithms

As we mentioned earlier, a non-preemptive algorithm is one in which the scheduler has no control over the currently executing process once it has been scheduled on the CPU. The only way the scheduler can get back control is if the currently running process voluntarily relinquishes the CPU either by terminating or by making a blocking system call (such as a file I/O request). In this section, we will consider three different algorithms that belong to this class: FCFS, SJF, and priority.

6.6.1 First-Come First-Served (FCFS)

The intrinsic property used in this algorithm is the *arrival time* of a process. Arrival time is the time when you launch an application program. For example, if you launched winamp at time t_0 and realplayer at a later time t_1 , then winamp has an earlier arrival time as far as the scheduler is concerned. Thus, during the entire lifetime of the two programs, whenever both programs are ready to run, winamp will always get picked by the scheduler since it has an earlier arrival time. Remember that this “priority” enjoyed by winamp will continue even when it comes back into the ready queue after an I/O completion. Example 1 shows this advantage enjoyed by the early bird process compared to other ready processes in the ready queue.

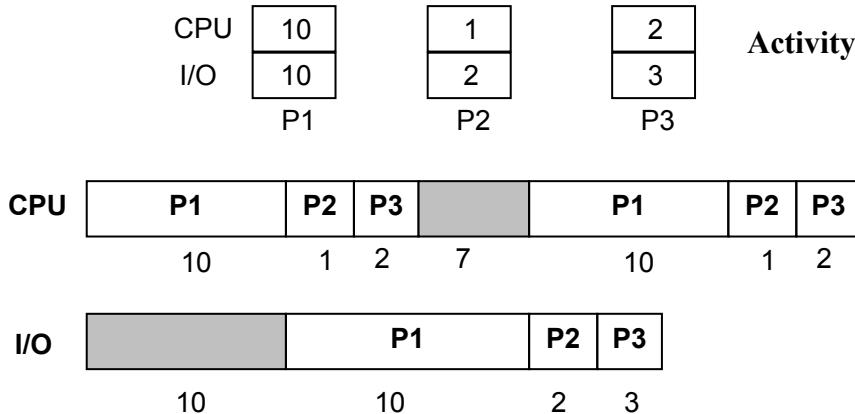


Figure 6.12: FCFS convoy effect

Figure 6.12 shows a set of processes and their activities at the top of the figure. Each process's activity alternates between CPU burst and I/O burst. For example, P2 does 1 unit of processing and 2 units of I/O, and repeats this behavior for its lifetime. The bottom of Figure 6.12 presents a timeline showing the FCFS scheduling of these processes on the CPU and I/O. There can be exactly one process executing on the CPU or carrying out an I/O activity at a time. Assume that each process has to do 2 bursts of CPU activity interspersed with one burst of I/O. All three processes are in the ready queue of the scheduler at the beginning of the time line; however, P1 is the first to arrive, followed by P2 and P3. Thus, given a choice the scheduler will always pick P1 over P2 or P3 for scheduling due to the FCFS policy. The waiting times for the three jobs P1, P2, and P3 are 0, 27, and 26, respectively.

The algorithm has the nice property that there will be no *starvation* for any process, i.e., the algorithm has no inherent bias that results in denial of service for any process. We will see shortly that not all algorithms have this property. However, due to its very nature there can be a huge variation in the response time. For instance if a short job arrives just after a long job then the response time for the short job will be very bad. The algorithm also results in poor processor utilization due to the *convoy effect* depicted in Figure 6.12. The term comes from the primarily military use of the word "convoy", which signifies a group of vehicles traveling together. Of course, in the military sense a convoy is a good thing since the vehicles act as support for one another in times of emergency. You may inadvertently become part of a convoy in a highway when you are stuck behind a slow moving vehicle in single lane traffic. The short jobs (P2 and P3) are stuck behind the long job (P1) for CPU. The convoy effect is unique to FCFS scheduling and is inherent in the nature of this scheduling discipline. Many of us have experienced being stuck behind a customer with a cartload of stuff at the checkout counter, when we ourselves have just a couple of items to check out. Unfortunately, the convoy effect is intrinsic to the FCFS scheduling discipline since by its very nature it does not give any preferential treatment for short jobs.

Example 1:

Consider a non-preemptive First Come First Served (FCFS) process scheduler. There are three processes in the scheduling queue and the arrival order is P1, P2, and P3. The

arrival order is always respected when picking the next process to run on the processor. Scheduling starts at time $t = 0$, with the following CPU and I/O burst times:

	CPU burst time	I/O burst time
P1	8	2
P2	5	5
P3	1	5

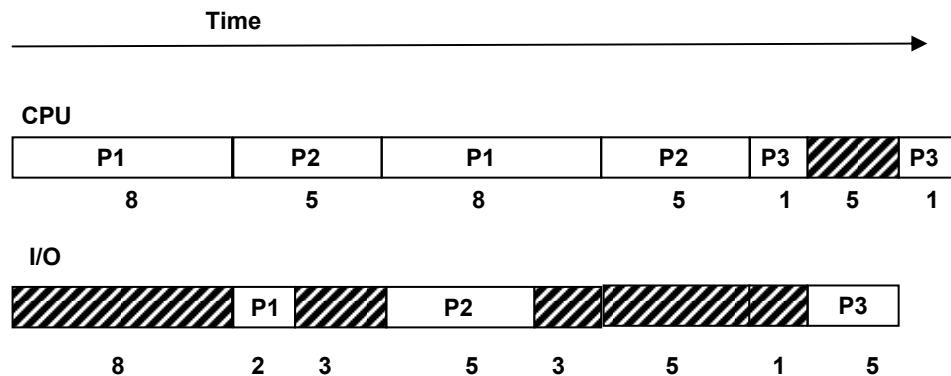
Each process terminates after completing the following sequence of three actions:

1. CPU burst
2. I/O burst
3. CPU burst

- a) Show the CPU and I/O timelines that result with FCFS scheduling from $t = 0$ until all three processes complete.
- b) What is the response time for each process?
- c) What is the waiting time for each process?

Answer:

a)



Notice that at time $t=8$, P2 and P3 are in the ready queue. P1 makes an I/O request and relinquishes the processor. The scheduler picks P2 to run on the processor due to its earlier arrival time compared to P3. At time $t=10$, P1 completes its I/O and rejoins the ready queue. P3 is already in the ready queue at this time. However, by virtue of its earlier arrival time, P1 gets ahead of P3 in the ready queue, which is why P1 gets picked by the scheduler at time $t=13$.

b)

We compute the response time for each process as the total time spent by the process in the system from the time it enters to the time it departs.

$$\begin{aligned} \text{Response time (P1)} &= 21 \\ \text{Response time (P2)} &= 26 \\ \text{Response time (P3)} &= 33 \end{aligned}$$

c)

Each process does useful work when it is either executing on the CPU or performing its I/O operation. Thus, we compute the wait time for each process by subtracting useful work done by a process from its total turnaround time (or response time).

For example, the useful work done by P1

$$\begin{aligned} &= \text{First CPU burst} + \text{I/O burst} + \text{Second CPU burst} \\ &= 8 + 2 + 8 \\ &= 18 \end{aligned}$$

Thus, the wait time for P1

$$\text{Wait-time (P1)} = (21 - 18) = 3$$

Similarly,

$$\text{Wait-time (P2)} = (26 - 15) = 11$$

$$\text{Wait-time (P3)} = (33 - 7) = 26$$

As we mentioned earlier, the scheduler always respects the arrival time of a process for its lifetime, when picking a winner to schedule on the CPU. Further, this intrinsic property is meaningful only for CPU scheduling and not I/O. These two points are illustrated in the following example.

Example 2:

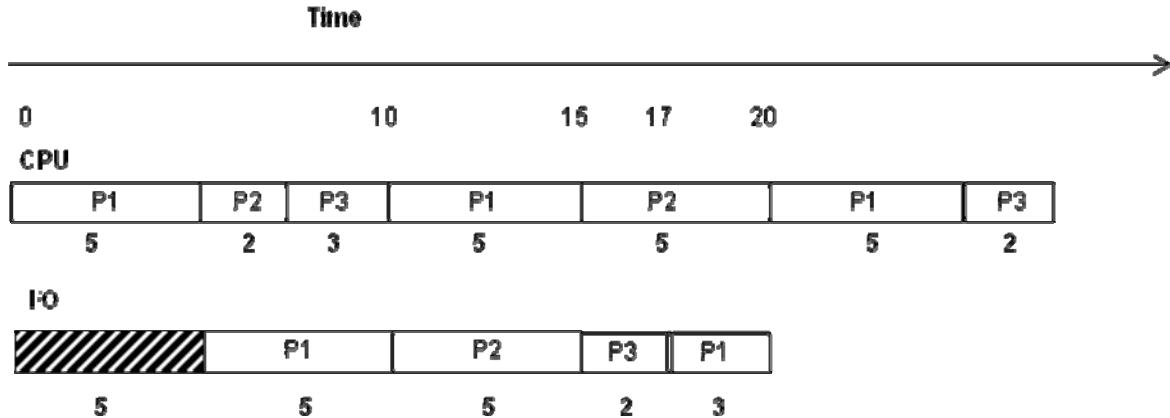
Consider a FCFS process scheduler. There are three processes in the scheduling queue and assume that all three of them are ready to run. As the scheduling discipline suggests, the scheduler always respects the arrival time in selecting a winner. Assume that P1, P2, and P3 arrive in that order into the system. Scheduling starts at time $t = 0$.

The CPU and I/O burst patterns of the three processes are as shown below:

	CPU	I/O	CPU	I/O	CPU	
P1	5	5	5	3	5	P1 is done
P2	2	5	5			P2 is done
P3	3	2	2			P3 is done

Show the CPU and I/O timelines that result with FCFS scheduling from $t = 0$ until all three processes complete.

Answer:



Notes:

- 1) At time $t=15$, both P3 and P1 need to perform I/O. However, P3 made the I/O request first (at time $t=10$), while P1 made its request second (at time $t=15$). As we mentioned earlier, the I/O requests are serviced in the order received. Thus P3 gets serviced first and then P1.
- 2) At time $t=20$, both P3 and P1 are needing to get on the CPU. In fact, P3 finished its I/O at time $t=17$, while P1 just finished its I/O at time $t=20$. Yet, P1 wins this race to get on the CPU since the CPU scheduler always respects the arrival time of processes in making its scheduling decision.

6.6.2 Shortest Job First (SJF)

The term “shortest job” comes from scheduling decisions taken in shop floors, for example in a car mechanic shop. Unfortunately, this can sometimes lead to the wrong connotation when literally applied to CPU scheduling. Since the CPU scheduler does not know the exact behavior of a program, it works only with partial knowledge. In the case of SJF scheduling algorithm, this knowledge is the *CPU burst time* needed, an intrinsic property of each process. As we know, each process goes through bursts of CPU and I/O activities. The SJF scheduler looks at the CPU burst times needed by the current set of processes that are ready to run and picks the one that needs the shortest CPU burst. Recalling the analogy at the beginning of the chapter, you started the washer and the microwave before you made your phone call to your mom. SJF scheduler uses the same principle of favoring short jobs. In reality, a scheduler does not know the CPU burst time for a program at program start up. However, it infers the expected burst time of a process from past CPU bursts.

This algorithm results in a better response time for short jobs. Further, it does not suffer from the convoy effect of FCFS since it gives preferential treatment to short jobs. It turns out that this algorithm is also provably optimal for yielding the best average waiting time. Figure 6.13 shows the time line for executing the same process activities as in Figure 6.12 for FCFS. The waiting times for the three jobs P1, P2, and P3 are 4, 0, and 9, respectively.

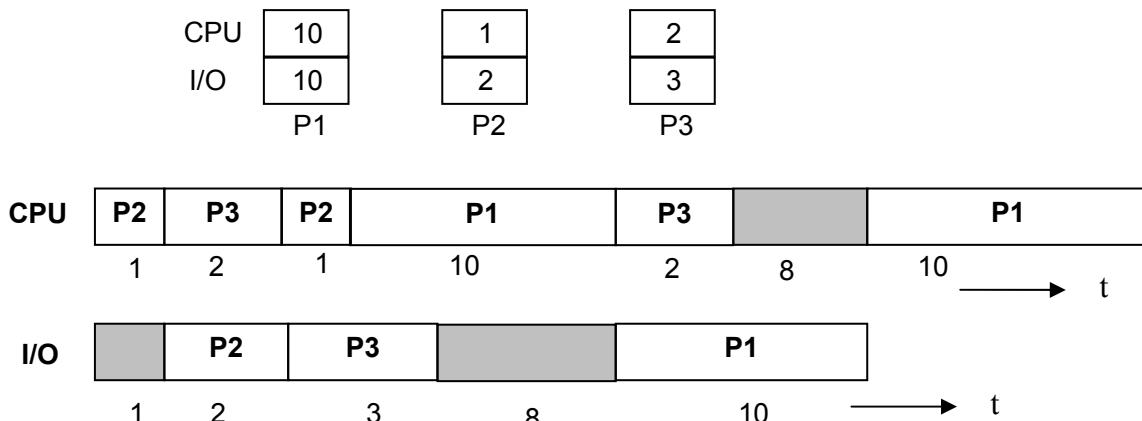


Figure 6.13: SJF schedule

Notice that the shortest job, P2, gets the best service with this schedule. At time $t = 3$, P2 has just completed its I/O burst; fortunately, P3 has also just finished its CPU burst. At this time, though P1 and P2 are ready to be scheduled on the CPU, the scheduler gives preference to P2 over P1 due to P2's shorter CPU burst requirement. At time $t = 4$, P2 finishes its CPU burst. Since there is no other shorter job to schedule, P1 gets a chance to run on the CPU. At time $t = 6$, P3 has just completed its I/O burst and is ready to be scheduled on the CPU. However, P1 is currently executing on the processor. Since the scheduler is non-preemptive, P3 has to wait for P1 to give up the processor on its own accord (which it does at $t = 14$).

There is a potential for starvation for long jobs in the SJF schedule. In the above example, before P1 gets its turn to run, new shorter jobs may enter the system. Thus, P1 could end up waiting a long time, perhaps even forever. To overcome this problem, a technique, *aging*, gives preference to a job that has been waiting in the ready queue for a long time over other short jobs. Basically, the idea is for the scheduler to add another predicate to each job, namely, the time it entered the scheduling mix. When the “age” of a job exceeds a threshold, the scheduler will override the SJF principle and give such a job preference for scheduling.

Example 3:

Consider a non-preemptive Shortest Job First (SJF) process scheduler. There are three processes in the scheduling queue and assume that all three of them are ready to run. As the scheduling discipline suggests, always the shortest job that is ready to run is given priority. Scheduling starts at time $t = 0$. The CPU and I/O burst patterns of the three processes are as shown below:

	CPU	I/O	CPU	I/O	CPU	
P1		4	2	4	2	4 P1 is done
P2		5	2	5		P2 is done
P3		2	2	2	2	P3 is done

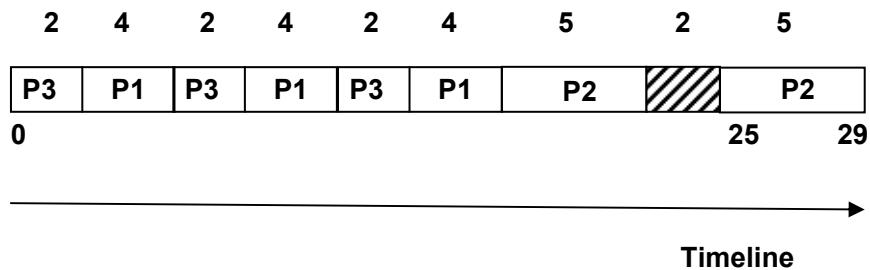
Each process exits the system once its CPU and I/O bursts as shown above are complete.

- Show the CPU and I/O timelines that result with SJF scheduling from $t = 0$ until all three processes exit the system.
- What is the waiting time for each process?
- What is the average throughput of the system?

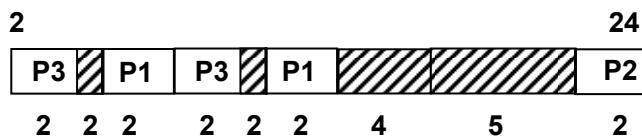
Answer:

(a)

CPU Schedule (SJF)



I/O Schedule



b)

We compute the wait times for each process as we did in Example 1.

$$\text{Wait-time (P1)} = (18 - 16) = 2$$

$$\text{Wait-time (P2)} = (30 - 12) = 18$$

$$\text{Wait-time (P3)} = (14 - 10) = 4$$

c)

$$\text{Total time} = 30$$

$$\text{Throughput} = \text{number of processes completed} / \text{total time}$$

$$= 3/30$$

$$= 1/10 \text{ processes per unit time}$$

6.6.3 Priority

Most operating systems associate an extrinsic property, *priority*, a small integer value, with each process to denote its relative importance compared to other processes for scheduling purposes. For example, in the Unix operating system every user level process

starts with a certain default priority. The ready queue consists of several sub-queues, one for each priority level as shown in Figure 6.14.

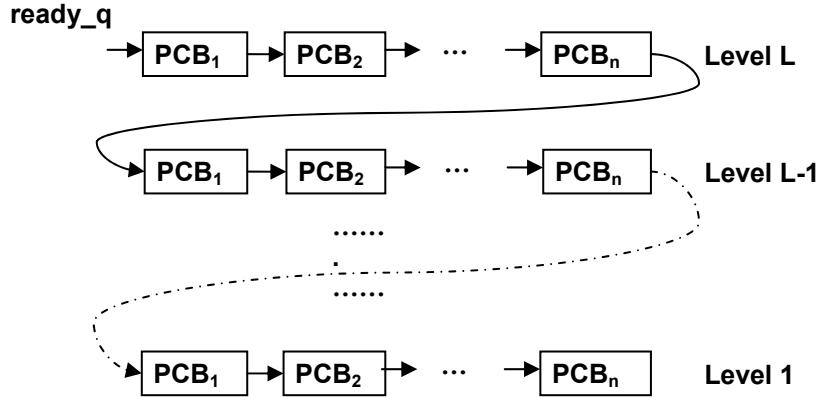


Figure 6.14: Multi-level Ready Queue for Priority Scheduler

The scheduling within each level is FCFS. New processes are placed in the queue commensurate with their priority levels. Since priority is an extrinsic property, the scheduler has complete control on the assignment of priorities to different processes. In this sense, this scheduling discipline is highly flexible compared to SJF or FCFS. For example, the scheduler may assign priorities based on the class of users. This would be particularly attractive from the point of view of running data centers² wherein different users may be willing to pay different rates for their respective jobs.

Priority is a very natural way of providing differential service guarantees for different users, not just in processor scheduling, but also in almost every walk of life. As an example, call centers operated by businesses that deal with customer service requests place different callers in different queues based on their profiles. “High maintenance” callers go into a slower queue with longer service times as opposed to preferred customers. Similarly, airlines use first class, business class, and economy class as a way of prioritizing service to its customers.

With a little bit of reflection, one could figure out that SJF is really a special case of priority scheduling wherein the priority level,

$$L = 1/\text{CPU_burst_time}$$

Thus, similar to SJF, priority scheduler has the problem of starvation of processes with low priority. Countering this problem with explicit priority levels assigned to each process is straightforward. Similar to the solution we suggested for SJF, when the age of a process exceeds a preset threshold, the scheduler will artificially bump up the priority of the process.

² Data centers are becoming popular as a way of providing access to high-performance computing resources to users without the need for individual users owning such resources. Companies such as Amazon, Microsoft, HP, and IBM are at the forefront of providing such services. Cloud computing is the industry buzzword for providing such services.

If you think about it, you will realize that FCFS is also a priority-based algorithm. It is just that the scheduler uses arrival time of the process as its priority. Due to this reason, no newly arriving process can have a priority higher than the current set of processes in FCFS, which is also the reason FCFS does not have the problem of starvation.

Due to the similarity with FCFS, a priority-based algorithm could also exhibit the convoy effect. Let us see how this could happen. If a process with a higher priority also turns out to be a long running process then we could end up with a situation similar to what we observed in the FCFS schedule. However, in FCFS, once assigned the process priority never changes. This need not be the case in a priority-based scheduler; the same mechanism (bumping up the priority based on age) used to overcome the starvation problem would also help to break the convoy effect.

6.7 Preemptive Scheduling Algorithms

This class of scheduling algorithms implies two things simultaneously. First, the scheduler is able to assume control of the processor anytime unbeknown to the currently running process. Second, the scheduler is able to save the state of the currently running process for proper resumption from the point of preemption. Returning to our analogy from the beginning of the chapter, when the washer beeps while you are on the phone with mom, you excuse yourself; and make a mental note of where to resume the conversation when you get back on the phone again.

In principle, any of the algorithms discussed in the previous section could be made preemptive. To accomplish that, in the case of FCFS, whenever a process rejoins the ready queue after I/O completion, the scheduler can decide to preempt the currently running process (if its arrival time is later than the former). Similarly, for SJF and priority, the scheduler re-evaluates and takes a decision to preempt the currently running process whenever a new process joins the ready queue or an existing one rejoins the queue upon I/O completion.

Shortest Remaining Time First (SRTF) is a special case of the SJF scheduler with preemption added in. The scheduler has an estimate of the running time of each process. When a process rejoins the ready queue, the scheduler computes the remaining processing time of the job. Based on this calculation, the scheduler places this process at the right spot in the ready queue. If the remaining time of this process is lower than that of the currently running process, then the scheduler preempts the latter in favor of the former.

Example 4:

Consider the following four processes vying for the CPU. The scheduler uses **SRTF**. The table shows the arrival time of each process.

Process	Arrival Time	Execution Time
P1	T_0	4ms
P2	$T_0 + 1\text{ms}$	2ms
P3	$T_0 + 2\text{ms}$	2ms

P4	$T_0 + 3\text{ms}$	3ms
----	--------------------	-----

- a) Show the schedule starting from time T_0 .

Answer:

- 1) At T_0 , P1 starts running since there is no other process.
- 2) At T_0+1 , P2 arrives. The following table shows the remaining/required running time for P1 and P2:

Process	Remaining time
P1	3ms
P2	2ms

Scheduler switches to P2.

- 3) At T_0+2 , P3 arrives. The following table shows the remaining/required running time for P1, P2, and P3:

Process	Remaining time
P1	3ms
P2	1ms
P3	2ms

Scheduler continues with P2

- 4) At T_0+3 , P4 arrives. P2 completes and leaves. The following table shows the remaining/required running for P1, P3, and P4:

Process	Remaining time
P1	3ms
P3	2ms
P4	3ms

Scheduler picks P3 as the winner and runs it to completion for the next 2 ms.

- 5) At T_0+5 , P1 and P4 are the only two remaining processes with remaining/required times as follows:

Process	Remaining time
P1	3ms
P4	3ms

It is a tie, and the scheduler breaks the tie by scheduling P1 and P4 in the arrival order (P1 first and then P4), which would result in a lower average wait time.

The following table shows the schedule with SRTF.

Interval T_0+	0	1	2	3	4	5	6	7	8	9	10	11	12
Running	P1	P2	P2	P3	P3	P1	P1	P1	P4	P4	P4	P4	

- b) What is the waiting time for each process?

Answer:

Response time = completion time – arrival time

$$R_{p1} = 8 - 0 = 8 \text{ ms}$$

$$R_{p2} = 3 - 1 = 2 \text{ ms}$$

$$R_{p3} = 5 - 2 = 3 \text{ ms}$$

$$R_{p4} = 11-3 = 8\text{ms}$$

Wait time = response time – execution time

$$W_{p1} = R_{p1} - E_{p1} = 8-4 = \mathbf{4 \text{ ms}}$$

$$W_{p2} = R_{p2} - E_{p2} = 2-2 = \mathbf{0 \text{ ms}}$$

$$W_{p3} = R_{p3} - E_{p3} = 3-2 = \mathbf{1 \text{ ms}}$$

$$W_{p4} = R_{p4} - E_{p4} = 8-3 = \mathbf{5 \text{ ms}}$$

- c) What is the average wait time with SRTF?

Answer:

$$\text{Total waiting time} = W_{p1} + W_{p2} + W_{p3} + W_{p4} = \mathbf{10 \text{ ms}}$$

$$\text{Average wait time} = 10/4 = \mathbf{2.5 \text{ ms}}$$

- d) What is the schedule for the same set of processes with FCFS scheduling policy?

Answer:

Interval T ₀₊	0	1	2	3	4	5	6	7	8	9	10	11	12
Running	P1	P1	P1	P1	P2	P2	P3	P3	P4	P4	P4		

- e) What is the average wait time with FCFS?

Answer:

Response time = completion time – arrival time

$$R_{p1} = 4-0 = 4 \text{ ms}$$

$$R_{p2} = 6-1 = 5 \text{ ms}$$

$$R_{p3} = 8-2 = 6 \text{ ms}$$

$$R_{p4} = 11-3 = 8\text{ms}$$

Wait time = response time – execution time

$$W_{p1} = R_{p1} - E_{p1} = 4-4 = \mathbf{0 \text{ ms}}$$

$$W_{p2} = R_{p2} - E_{p2} = 5-2 = \mathbf{3 \text{ ms}}$$

$$W_{p3} = R_{p3} - E_{p3} = 6-2 = \mathbf{4 \text{ ms}}$$

$$W_{p4} = R_{p4} - E_{p4} = 8-3 = \mathbf{5 \text{ ms}}$$

$$\text{Total waiting time} = W_{p1} + W_{p2} + W_{p3} + W_{p4} = \mathbf{12 \text{ ms}}$$

$$\text{Average wait time} = 12/4 = \mathbf{3 \text{ ms}}$$

Note: You can see that SRTF gives a lower average wait time compared to FCFS.

6.7.1 Round Robin Scheduler

Let us discern the characteristics of a scheduler that is appropriate for time-shared environments. As the name implies, in this environment, every process should get its share of the processor. Therefore, a non-preemptive scheduler is inappropriate for such an environment.

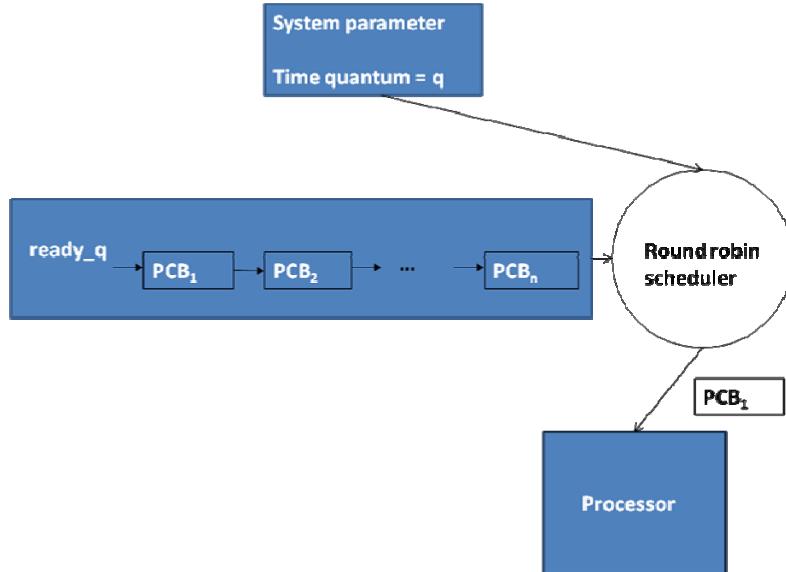


Figure 6.15: Round Robin Scheduler

Time-shared environments are particularly well served by a round robin scheduler. Assume that there are n ready processes. The scheduler assigns the processor in time quantum units, q , usually referred to as *timeslice*, to each process (Figure 6.15). You can see the connection between this type of scheduler and the first strategy we suggested in the analogy at the beginning of the chapter for test preparation. Switching among these ready processes in a round robin scheduler does not come for free. The *Context switching time* has to be taken into account in determining a feasible value for the time quantum q .

Each process in the ready queue gets its share of the processor for timeslice q . Once the timeslice is up, the currently scheduled process is placed at the end of the ready queue, and the next process in the ready queue is scheduled on the processor.

We can draw a connection between the round robin scheduler and the FCFS scheduler. FCFS is a special case of the round robin scheduler wherein the time quantum is *infinity*. *Processor sharing* is another special case of round robin wherein each process gets 1 unit of time on the processor. Thus, each process gets the illusion of running on a dedicated processor with a speed of $1/n$. The processor-sharing concept is useful for proving theoretical results in scheduling.

6.7.1.1 An Example

Let us consider an example of round robin scheduling. Given in Figure 6.16 are three processes with CPU and I/O bursts as noted. Let us assume that the order of arrival of the processes is P1, P2, and P3; and that the scheduler uses a timeslice of 2. Each process exits the system once its CPU and I/O bursts as shown in the figure are complete.

	CPU	I/O	CPU	I/O	
P1	4	2	2		P1 is done
P2	3	2	2		P2 is done
P3	2	2	4	2	P3 is done

Figure 6.16: CPU and I/O bursts for Round robin example

At time $t=0$, the scheduling queue looks as shown in Figure 6.17. The key point to note is that this order at time $t=0$ may not be preserved as the processes go back and forth between the CPU and I/O queues. When a process rejoins the CPU or the I/O queue it is always at the end of the queue since there is no inherent priority associated with any of the processes.

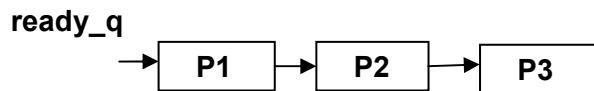


Figure 6.17: Ready Queue for example in Figure 6.16 at time $t = 0$

Figure 6.18 show a snapshot of the schedule for the first 17 time units ($t = 0$ to $t = 16$). Note that P2 uses only one 1 unit of its timeslice the second time it is scheduled (at $t = 6$), since it makes an I/O burst after 3 units of CPU burst. In other words, in a round robin schedule, the timeslice is an upper bound for continuous CPU usage before a new scheduling decision. Also, at time $t = 12$ P2 has just completed its I/O burst, and is ready to rejoin the CPU queue. At this time, P1 is in the middle of its CPU timeslice and P3 is on the ready queue as well. Therefore, P2 joins the ready queue behind P3 as shown in Figure 6.19.

CPU Schedule (Round Robin, timeslice = 2)

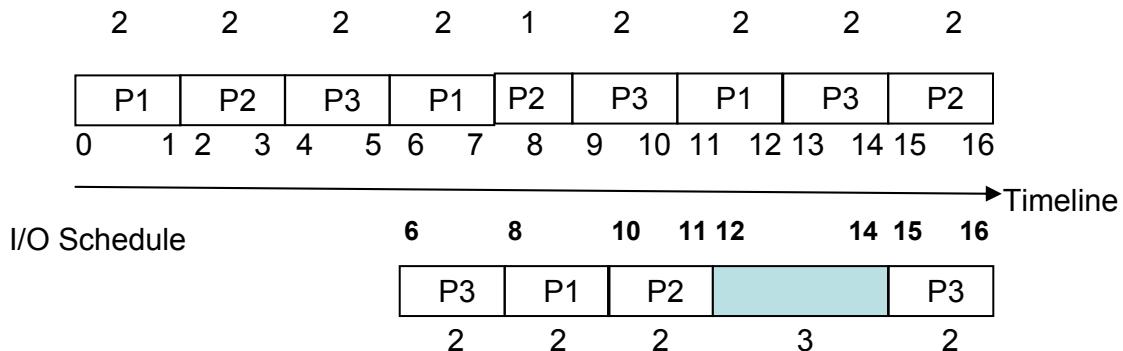


Figure 6.18: Round robin schedule for example in Figure 6.16

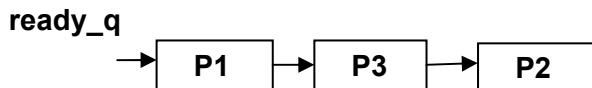


Figure 6.19: Ready Queue for example in Figure 6.16 at time t = 12

Example 5:

What is the wait time for each of the three processes in Figure 6.16 with round robin schedule?

Answer:

As in Examples 1 and 3,

Wait time =

(response time – useful work done either on the CPU or I/O)

Wait time for P1 = $(13 - 8) = 5$

Wait time for P2 = $(17 - 7) = 10$

Wait time for P3 = $(17 - 10) = 7$

6.7.1.2 Details of round robin algorithm

Let us turn our attention to getting control of the processor for the scheduler. A hardware device, *timer*, interrupts the processor when the time quantum q expires. The interrupt handler for the timer interrupt is an integral part of the round robin scheduler. Figure 6.20 shows the organization of the different layers of the system. At any point of time, the processor is hosting either one of the user programs or the scheduler. Consider that a user program is currently running on the processor. Upon an interrupt, the timer interrupt handler takes control of the processor (refer to the sequence of steps taken by an interrupt in Chapter 4). The handler saves the context of the currently running user program in the

associated PCB and hands over control to the scheduler. This handoff, referred to as an *upcall*, allows the scheduler to run its algorithm to decide the next process to be dispatched on the processor. In general, upcall refers to invoking a system function (i.e. a procedure call) from a lower level of the system software to the upper levels.

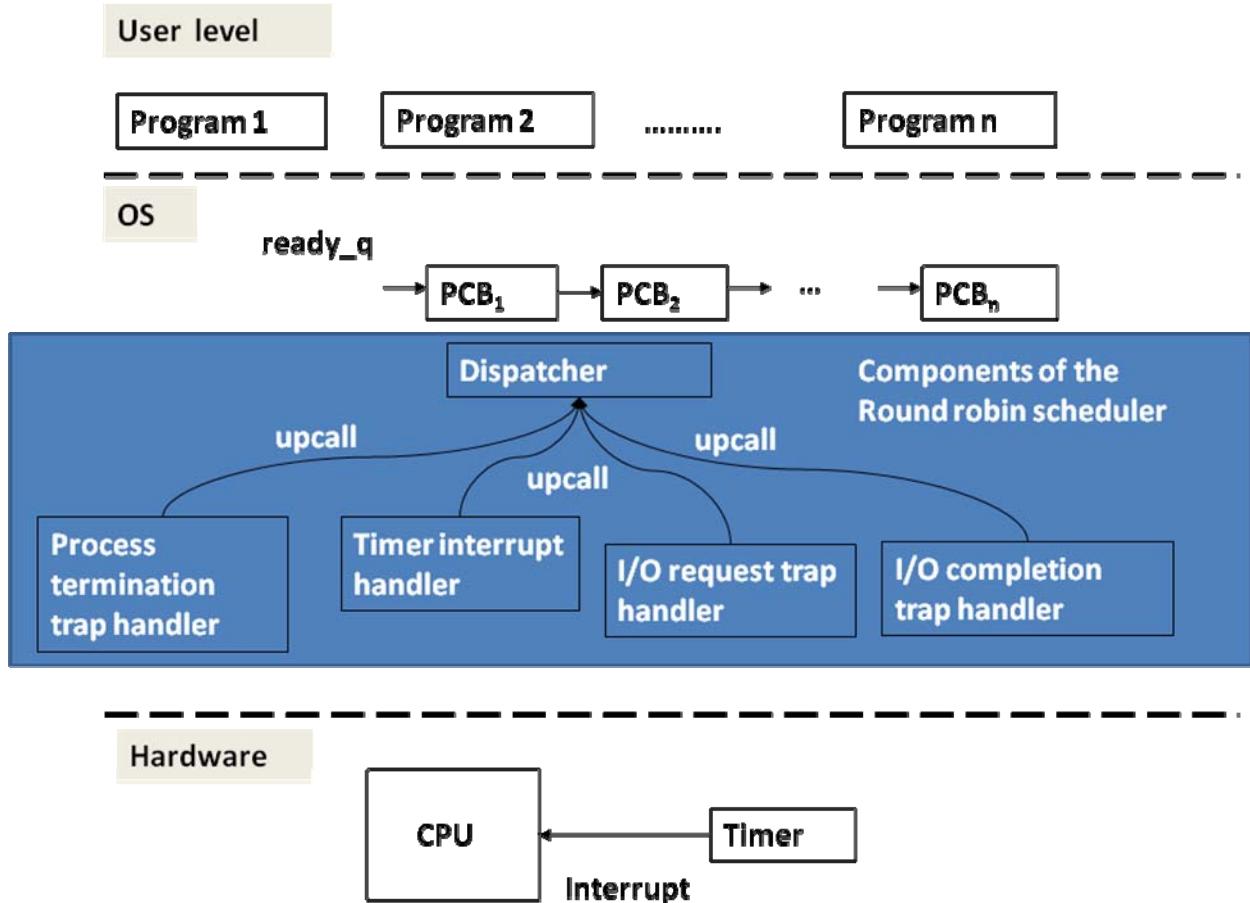


Figure 6.20: Different layers of the system incorporating a round-robin scheduler

Figure 6.21 summarizes the round robin scheduling algorithm. The algorithm comprises of five procedures: *dispatcher*, *timer interrupt handler*, *I/O request trap*, *I/O completion interrupt handler*, and *process termination trap handler*.

```

Dispatcher:
    get head of ready queue;
    set timer;
    dispatch;

Timer interrupt handler:
    save context in PCB;
    move PCB to the end of the ready queue;
    upcall to dispatcher;

I/O request trap:
    save context in PCB;
    move PCB to I/O queue;
    upcall to dispatcher;

I/O completion interrupt handler:
    save context in PCB;
    move PCB of I/O completed process to ready queue;
    upcall to dispatcher;

Process termination trap handler:
    Free PCB;
    upcall to dispatcher;

```

Figure 6.21: Round Robin Scheduling Algorithm

The dispatcher simply dispatches the process at the head of the ready queue on the processor setting the timer to the time quantum q . The currently scheduled process may give up the processor in one of three-ways: it makes an I/O call, or it terminates, or it completes its assigned time quantum on the processor. I/O request trap is the manifestation of the first option, for example, if the currently scheduled process makes a file read request from the disk. In this case, the I/O request trap procedure saves the state of the currently scheduled process in the PCB, moves it to the I/O queue, and upcalls the dispatcher. Timer interrupt handler is the manifestation of the time quantum for the current process expiring. The hardware timer interrupts the processor resulting in the invocation of this handler. As can be seen in Figure 6.21, the timer interrupt handler saves the state in the PCB of the current process, moves the PCB to the end of the ready queue, and makes an upcall to the dispatcher. Upon completion of an I/O request, the processor will get an I/O completion interrupt. The interrupt will result in the invocation of the I/O completion handler. The role of this handler is tricky. Currently some process is executing on the processor. This process's time quantum is not up since it is still executing on the processor. The handler simply saves the state of the currently running process in the corresponding PCB. The I/O completion is on behalf of whichever process requested it in the first place. The handler moves the associated PCB of the process that requested the I/O to the end of the ready queue and upcalls the dispatcher. The dispatcher will do the needful to dispatch the original process that was interrupted by the I/O completion interrupt (since it still has time on the processor). The process termination handler simply frees up the PCB of the process, removing it from the ready queue and upcalls the dispatcher.

6.8 Combining Priority and Preemption

General-purpose operating systems such as Unix and Microsoft Windows XP combine notions of priority and preemption in the CPU scheduling algorithms. Processes have priorities associated with them determined at the time of creation by the operating system. They are organized into a multi-level ready queue similar to Figure 6.14. Additionally, the operating system employs a round robin scheduling with a fixed time quantum for each process at a particular priority level. Lower priority processes get service when there are no higher priority processes to be serviced. To avoid starvation, the operating system may periodically boost the priority of processes that have not received service for a long time (i.e., aging). Further, an operating system may also give larger time quanta for higher priority processes. The software system for such a scheduler has a structure depicted in Figure 6.20.

6.9 Meta Schedulers

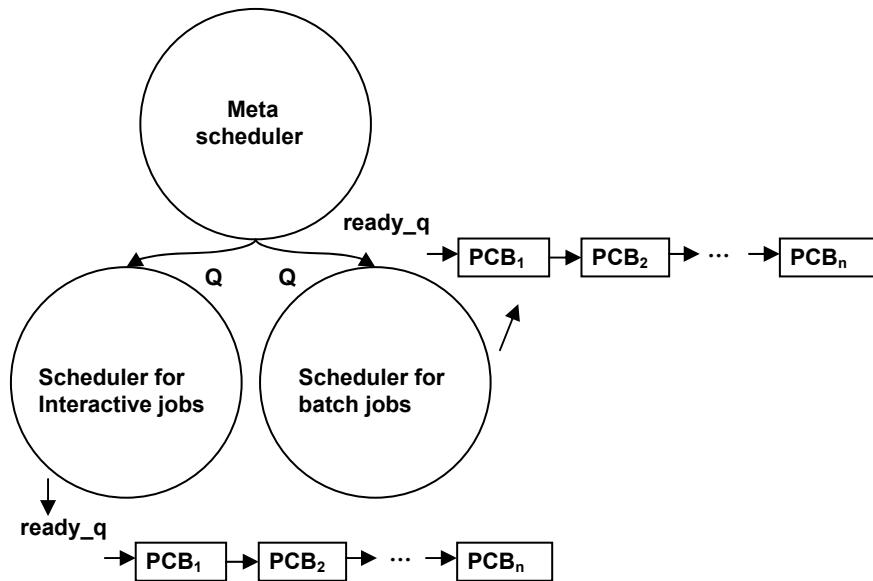


Figure 6.22: Meta Scheduler

In some operating systems, catering to multiple demands, the scheduling environment may consist of several ready queues, each having its own scheduling algorithm. For example, there may be a queue for foreground interactive jobs (with an associated scheduler) and a queue for background batch-oriented jobs (with an associated scheduler). The interactive jobs are scheduled in a round-robin fashion, while the background jobs are scheduled using a priority based FCFS scheduler. A meta-scheduler sits above these schedulers timeslicing among the queues (Figure 6.22). The parameter Q is the time quantum allocated by the meta-scheduler to each of the two schedulers at next level. Often the meta-scheduler provides for movement of jobs between the ready queues of the two lower level schedulers due to the dynamic changes in the application (represented by the different processes) requirements.

A generalization of this meta-scheduler idea is seen in *Grid Computing*, a somewhat nomadic Internet-based computing infrastructure that is gaining traction due to the ability it offers for using *high-performance* computational resources without actually owning them. Grid computing harnesses computational resources that may be geographically distributed and crisscrossing administrative boundaries. The term Grid arises from the fact that the arrangement of these computational resources is analogous to the “Power Grid” distributing electricity. Users of the Grid submit jobs to this environment that are executed using the distributed computational resources. The term high-performance refers to the fact that there may be hundreds or even thousands of computational resources working in concert to cater to the computational demands of the application. For example, an application for such an infrastructure may be global climate modeling. Computing environments designed for grid computing employ such meta-schedulers. The interested reader is referred to advanced books on this topic³.

6.10 Evaluation

Let us understand how we evaluate the efficiency of a scheduler. The key property of any OS entity is to quickly provide the requested resources to the user and get out of the way. Scheduling algorithms are evaluated with respect to the performance metrics we mentioned earlier in the context of the specific environments in which they are to be deployed. We characterized different environments by attributes such as timeshared, batch-oriented, and multiprogrammed. These environments cater to different market forces or application domains for computing. For example, today we can identify the following domains in which computing plays a pivotal role:

- **Desktop:** We are all quite familiar with this personal computing domain.
- **Servers:** This is the domain of mail servers, files servers, and web servers.
- **Business:** This domain encompasses e-commerce and financial applications, and often used synonymously with **enterprise** computing.
- **High-Performance Computing (HPC):** This is the domain requiring high-performance computational resources for solving scientific and engineering problems.
- **Grid:** This domain has all the elements of HPC with the added wrinkle that the computers may be geographically distributed (for example, some machines may be in Tokyo and some in Bangalore) and may overlap administrative boundaries (for example, some machines may belong to Georgia Tech and some to MIT).
- **Embedded:** This is emerging as the most dominant computing domain with the ubiquity of *personal digital assistants* (PDAs) including cellphones, iPODs, iPhones, and the like. This domain also encompasses dedicated computing systems found in automobiles, aircrafts, and space exploration.
- **Pervasive:** This emerging domain combines elements of HPC and embedded computing. Video-based surveillance in airports employing camera networks is an example of this domain.

³ Foster and Kesselman, “The Grid: Blueprint for a New Computing Infrastructure,” Morgan Kaufmann publishers.

We use the term *workload* to signify the nature of the applications that typify a particular domain. We can broadly classify workload as *I/O bound* or *Compute bound*. However, the reader should note that this broad classification might not always be useful, especially in the case of I/O. This is because the nature of I/O can be vastly different depending on the application domain. For example, one could say that both business applications and desktop computing are I/O intensive. Business applications manipulate large databases and hence involve a significant amount of I/O; desktop computing is also I/O bound but differs from the business applications in that it is interactive I/O as opposed to I/O involving mass storage devices (such as disks). Applications in the server, HPC, and grid domains tend to be computationally intensive. Embedded and pervasive computing domains are quite different from the above domains. Such domains employ sensors and actuators (such as cameras, microphones, temperature sensors, and alarms) that need rapid response similar to the interactive workloads; at the same time, analysis of sensor data (for example, camera images) tends to be computationally intensive.

Table 6.5 summarizes the characteristics of these different domains.

Domains	Environment	Workload characteristics	Types of schedulers
Desktop	Timeshared, interactive, multiprogrammed	I/O bound	Medium-term, short-term, dispatcher
Servers	Timeshared, multiprogrammed	Computation bound	Medium-term, short-term, dispatcher
Business	Timeshared, multiprogrammed	I/O bound	Medium-term, short-term, dispatcher
HPC	Timeshared, multiprogrammed	Computation bound	Medium-term, short-term, dispatcher
Grid	Batch-oriented, timeshared, multiprogrammed	Computation bound	Long-term, Medium-term, short-term, dispatcher
Embedded	Timeshared, interactive, multiprogrammed	I/O bounds	Medium-term, short-term, dispatcher
Pervasive	Timeshared, interactive, multiprogrammed	Combination of I/O bound and computation bound	Medium-term, short-term, dispatcher

Table 6.5: Different Application Domains

One can take one of three different approaches to evaluate a scheduler: *modeling*, *simulation*, and *implementation*. Each of these approaches has its *pros* and *cons*. Modeling refers to deriving a mathematical model (using techniques such as queuing theory) of the system. Simulation refers to developing a computer program that simulates the behavior of the system. Lastly, implementation is the actual deployment of the algorithm in the operating system. Modeling, simulation, and implementation represent increasing amounts of effort in that order. Correspondingly, there are payoffs from these approaches (in terms of understanding the performance potential of the system)

commensurate with the effort. At the same time, modeling and simulation offer the ability to ask “what if” questions, while the implementation freezes a particular algorithm making it difficult to experiment with different design choices. In general, early estimates of system performance are obtained using modeling and simulation before committing the design to an actual implementation.

6.11 Impact of Scheduling on Processor Architecture

Thus far, we have treated the processor as a black box. Having reviewed various processor-scheduling algorithms, it is natural to ponder whether this operating system functionality needs any special support from the processor architecture. Once a process is scheduled to run on the processor, the ISA has to cater to the needs of the running program and we already know how that part is handled from Chapter 2. Processor scheduling algorithms themselves are similar to any user level programs, so there is nothing special about their needs insofar as the ISA is concerned. Therefore, it is during the transition, i.e., context switching from one process to another that the processor architecture may offer some special assists to the operating system.

Let us break it down to understand the opportunities for processor architecture to support scheduling. First, since modern operating systems support pre-emption, the processor architecture has to provide a way of *interrupting* the currently executing program so that the operating system can take control of the processor and run the scheduling algorithm instead of the user program. We already are familiar with this processor functionality from Chapter 4. Specifically, the processor architecture needs to provide a *timer* device for the scheduler to make its scheduling decisions and program in time quantum for a process. Further, the ISA offers special instructions to turn on and off interrupts to ensure *atomicity* for a group of instructions that an interrupt handler may have to execute (please see Chapter 4 for more elaboration on this topic).

Executing privileged instructions requires that the processor be in a privileged state, i.e., to ensure that ordinary user programs are not allowed to execute these instructions. We saw in Chapter 4, that the processor offers *user/kernel mode* of operation precisely for this reason.

The scheduling algorithm modifies data structures (e.g., PCBs of processes) that are private to the operating system. There has to be *separation of the memory space* reserved for the operating system from the user programs to ensure the integrity of the operating system and prevent either accidental or malicious corruption of the OS data structures by user programs. This is pictorially represented in Figure 6.20. In later chapters (please see Chapters 7, 8, and 9), we will discuss memory management and memory hierarchies in much more detail that help achieve this isolation of user programs from kernel code. At this point, it is sufficient to say that providing kernel mode of operation is a convenient mechanism for achieving the separation from user programs and kernel code.

Lastly, let us consider context switching, i.e., saving the register values of the currently running process into its PCB, and populating the registers with the values contained in the PCB of the next process to be dispatched on the processor. We know that this has to

be fast to make the OS efficient. Here is an opportunity for the processor ISA to offer some help. Some architectures (e.g., DEC VAX) offer a *single instruction* for loading *all* the processor registers from an area of the memory, and similarly storing all the processor registers into memory. While it is possible to be selective about saving/restoring registers for procedure call/return (please see Chapter 2, Section 2.8), at the point of a context switch, the OS has to assume that all the registers of the currently running process is relevant to that process warranting such new instructions in the ISA. Some architectures (e.g., Sun SPARC) offer *register windows* (see Chapter 2, Section 2.8) that may be used to maintain the context of the processes distinct and thus eliminate the need for saving/restoring all the registers at the point of a context switch. In this case, it is sufficient for the scheduler to switch to the register window associated with the new process upon a context switch.

Name	Property	Scheduling criterion	Pros	Cons
FCFS	Intrinsically non-preemptive; could accommodate preemption at time of I/O completion events	Arrival time (intrinsic property)	Fair; no starvation;	high variance in response time; convoy effect
SJF	Intrinsically non-preemptive; could accommodate preemption at time of new job arrival and/or I/O completion events	Expected execution time of jobs (intrinsic property)	Preference for short jobs; provably optimal for response time; low variance in response times	Potential for starvation; bias against long running computations
Priority	Could be either non-preemptive or preemptive	Priority assigned to jobs (extrinsic property)	Highly flexible since priority is not an intrinsic property, its assignment to jobs could be chosen commensurate with the needs of the scheduling environment	Potential for starvation
SRTF	Similar to SJF but uses preemption	Expected remaining execution time of jobs	Similar to SJF	Similar to SJF
Round robin	Preemptive allowing equal share of the processor for all jobs	Time quantum	Equal opportunity for all jobs;	Overhead for context switching among jobs

Table 6.6: Comparison of Scheduling Algorithms

6.12 Summary and a Look ahead

Table 6.6 summarizes the characteristics of the different scheduling algorithms discussed in this chapter. There are several advanced topics in scheduling for the interested reader. The strategies discussed in this chapter do not guarantee any specific quality of service. Several environments may demand such guarantees. Consider, for example a system that controls a nuclear reactor; or a rocket launcher; or the control system on board an aircraft. Such systems, dubbed *real-time* systems, need deterministic guarantees; advanced topics in scheduling deal with providing such real-time guarantees. For example, *deadline scheduling*, provides hints to the scheduler on the absolute time by which a task has to be completed. The scheduler will use these deadlines as a way of deciding process priorities for scheduling purposes.

As it turns out, many modern applications also demand such real-time guarantees. You play music on your iPOD, play a game on your XBOX, or watch a movie on your laptop. These applications need real-time guarantees as well. However, the impact of missing deadlines may not be as serious as in a nuclear reactor. Such applications, often called *soft real-time* applications, have become part of the applications that run on general-purpose computers. Scheduling for such applications is an interesting topic for the reader to explore as well.

Embedded computing is emerging as a dominant environment that is making quite a bit of inroads, exemplified by cellphones, iPODs, and iPhones. The interested reader should explore scheduling issues in such environments.

Moving beyond uniprocessor environments, scheduling in a multiprocessors environment brings its own set of challenges. We defer discussion of this topic to a later chapter in the book. Lastly, scheduling in distributed systems is another exciting topic, which is outside the scope of this textbook.

6.13 Linux Scheduler – A case study

As we mentioned already, modern general-purpose operating systems use a combination of the techniques presented in this chapter. As a concrete example, let us review how scheduling works in Linux.

Linux is an “open source” operating system project, which means that a community of developers has voluntarily contributed to the overall development of the operating system. New releases of the operating system appear with certain regularity. For example, circa December 2007, the release of the kernel has the number 2.6.13.12. The discussion in this section applies to the scheduling framework found in Linux release 2.6.x.

Linux offers an interesting case study since it is at once trying to please two different environments: (a) desktop computing and (b) servers. Desktop computing suggests an interactive environment wherein response time is important. This suggests that the

scheduler may have to make frequent context switches to meet the interactive needs (mouse clicks, keyboard input, etc.). Servers, on the other hand, handle computationally intensive workload; thus, less the context switching, more the useful work done in a server. Thus, Linux sets out to meet the following goals in its scheduling algorithm:

- High efficiency, meaning spending as little time as possible in the scheduler itself, an important goal for the server environment
- Support for interactivity, which is important for the interactive workload of the desktop environment
- Avoiding starvation, to ensure that computational workload do not suffer as a result of interactive workloads
- Support for soft real-time scheduling, once again to meet the demands of interactive applications with real-time constraints

Linux's connotation for the terms process and threads are a bit non-standard from the traditional meanings associated with these terms. Therefore, we will simply use the term task as the unit of scheduling in describing the Linux scheduling algorithm⁴.

Linux scheduler recognizes three classes of tasks:

- Real-time FCFS
- Real-time round robin
- Timeshared

The scheduler has 140 priority levels. It reserves levels 0-99 for real-time tasks, and the remaining levels for the timeshared tasks. Lower number implies higher priority, thus priority level 0 is the highest priority level in the system. The scheduler uses real-time FCFS and real-time round-robin classes for interactive workloads, and the timeshared class for computational workloads. Real-time FCFS tasks enjoy the highest priority. The scheduler will not preempt a real-time FCFS task currently executing on the processor unless a new real-time FCFS task with a higher priority joins the ready queue. Real-time round robin tasks have lower priority compared to the real-time FCFS tasks. As the name suggests, the scheduler associates a time quantum with each round robin task. All the round robin tasks at a given priority level have the same time quantum associated with them; the higher the priority level of a round robin task the higher the time quantum associated with it. The timeshared tasks are similar to the real-time tasks except that they are at lower priority levels.

The main data structure (Figure 6.23) in the scheduler is a *runqueue*. The runqueue contains two *priority* arrays. One priority array is the active one and the second is the expired one. Each priority array has 140 entries corresponding to the 140 priority levels. Each entry, points to the first task at that level. The tasks at the same level are linked together through a doubly-linked list.

⁴ Please see: http://josh.trancesoftware.com/linux/linux_cpu_scheduler.pdf if you are interested in reading more about the details of Linux CPU scheduler.

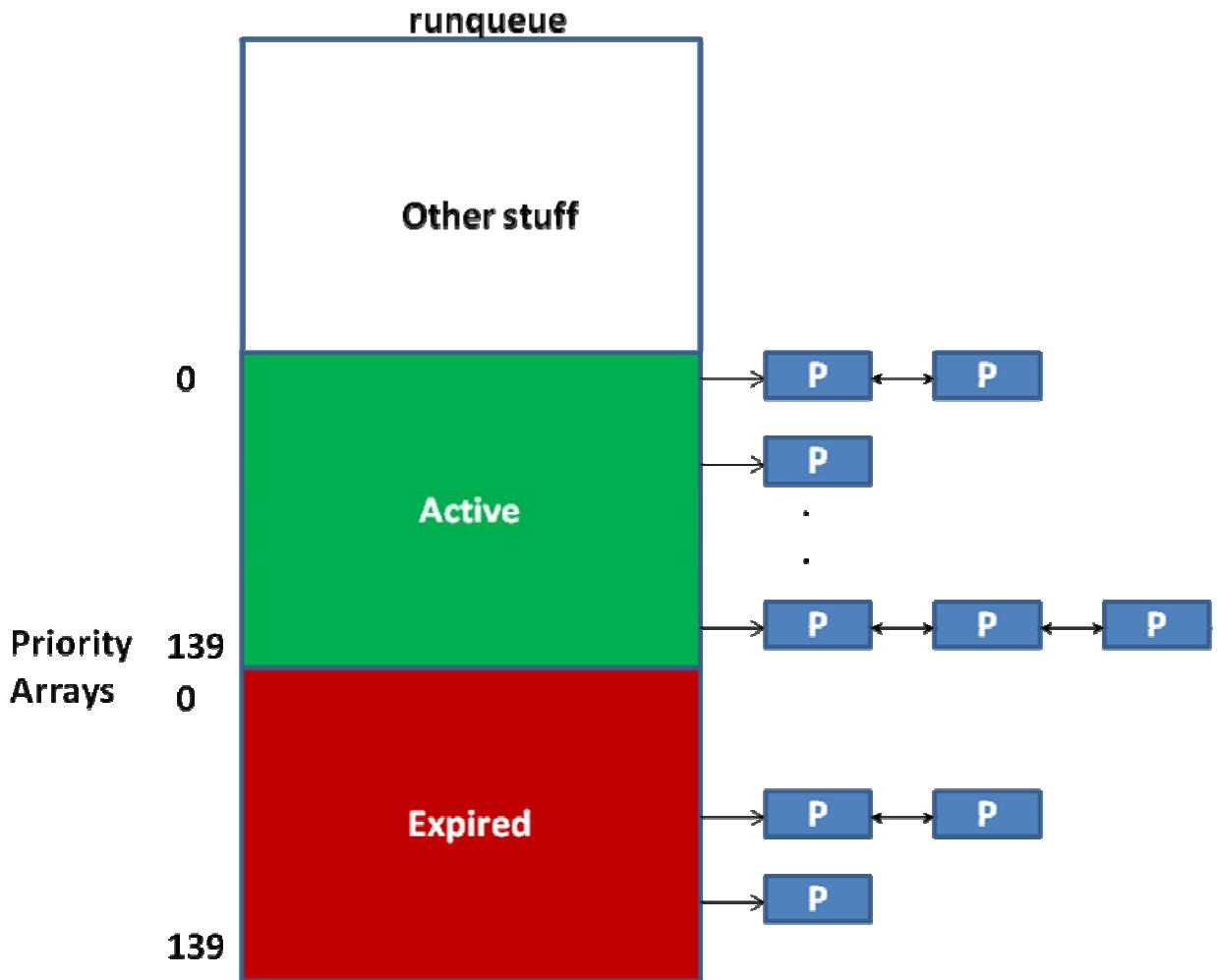


Figure 6.23: Linux Scheduling Data Structures

The scheduling algorithm is straightforward:

- Pick the first task with the highest priority from the active array and run it.
- If the task blocks (due to I/O) put it aside and pick the next highest one to run.
- If the time quantum runs out (does not apply to FCFS tasks) for the currently scheduled task then place it in the expired array.
- If a task completes its I/O then place it in the active array at the right priority level adjusting its remaining time quantum.
- If there are no more tasks to schedule in the active array, simply flip the active and expired array pointers and continue with the scheduling algorithm (i.e., the expired array becomes the active array and vice versa).

The first thing to note with the above algorithm is that the priority arrays enable the scheduler to take a scheduling decision in a *constant* amount of time independent of the number of tasks in the system. This meets the efficiency goal we mentioned earlier. For

this reason, the scheduler is also called O(1) scheduler implying that the scheduling decision is independent of the number of tasks in the system.

The second thing to note is that the scheduler has preferential treatment for meeting soft real-time constraints of interactive tasks through the real-time FCFS and real-time round robin scheduling classes.

As we said earlier, but for the relative priority levels, there is not much difference in terms of scheduling between the real-time round robin and timeshared classes of tasks. In reality, the scheduler does not know which of these tasks are truly interactive. Therefore, it uses a heuristic to determine the nature of the tasks from execution history. The scheduler monitors the pattern of CPU usage of each task. If a task makes frequent blocking I/O calls, it is an interactive task (I/O bound); on the other hand, if a task does not do much I/O, it is a CPU intensive task (CPU bound).

Many us may be familiar with the phrase “carrot and stick.” It is an idiom used to signify rewarding good behavior and punishing bad behavior. The scheduler does the same, rewarding tasks that are interactive by boosting their scheduling priority dynamically; and punishing tasks that are CPU intensive by lowering their scheduling priority dynamically⁵. The scheduler boosts the priority level of an interactive task so that it will get a higher share of the CPU time thus leading to good response times. Similarly, the scheduler lowers the priority of a compute bound task so that it will get a lesser share of the CPU time compared to the interactive tasks.

Lastly, for meeting the starvation goal, the scheduler has a starvation threshold. If a task has been deprived of CPU usage (due to interactive tasks getting higher priority) beyond this threshold then the starved task gets CPU usage ahead of the interactive tasks.

6.14 Historical Perspective

Operating systems have a colorful history just as CPUs. Microsoft Windows, MacOS, and Linux dominate the market place today. Let us take a journey through the history books to see how we got here.

Evolution of the operating system is inexorably tied to the evolution of the processor. Charles Babbage (1792-1871) built the first calculating machine (he called it the “analytical engine”) out of purely mechanical parts: gears, pulleys, and wheels. Not much happened in the evolution of computing until World War II. It is sad but true that wars spur technological innovations. William Mauchley and Presper Eckert at University of Pennsylvania built ENIAC (Electronic Numerical Integrator and Calculator) in 1944 using vacuum tubes, funded by the U.S. Army to carry out the calculations needed to break the German secret codes. ENIAC and other early machines of this era (1944 to 1955) were primarily used in standalone mode and did not need any operating system. In

⁵ A task has a static priority at the time of task creation. Dynamic priority is a temporary deviation from this static level as either a reward or a punishment for good or bad behavior, respectively.

fact, “programs” in these early days of computing were just wired up circuits to perform specific computations, repetitively.

The appearance of mainframes in the late 50’s using solid-state electronics gave birth to the first images of computing machinery, as we know it today. IBM was the main player in the mainframe arena, and introduced the FORTRAN programming language and possibly the first operating system to support it called FMS (Fortran Monitoring System) and later IBSYS, IBM’s operating system for IBM 7094 computer. This is the age of the *batch-oriented* operating system; users submitted their jobs as a deck of punched cards. The punched cards contained the resource needs of the program in a language called the *job control language (JCL)* and the program itself in FORTRAN. The jobs were run one at a time on the computer requiring quite a bit of manual work by the human operator to load the tapes and disks relevant to the resource requirements of each program. Users collected the results of the run of their program on the computer at a later time.

Two distinct user communities were discernible in these early days of computing: scientific and business. In the mid 60’s, IBM introduced the 360 family of computers aimed at integrating the needs of both these communities under one umbrella. It also introduced OS/360 as the operating system for this family of machines. The most important innovation in this operating system is the introduction of *multiprogramming* that ensured that the CPU was being used for some other process when the system was performing I/O for some users. Despite multiprogramming, the external appearance of the system was still a batch-oriented one, since the users submitted their jobs at one time and collected the results later.

The desire to get interactive response to the submitted jobs from the point of view of analyzing the results for further refinement of the programs and/or for simply debugging led to the next evolution in operating systems, namely, *timesharing*, which had its roots in CTSS (Compatible Time Sharing System) developed at MIT in 1962 to run on IBM 7094. A follow on project at MIT to CTSS was MULTICS (MULTIplexed Information and Computing Service), which was perhaps way ahead of its time in terms of concepts for information service and exchange, given the primitive nature of the computing base available at that time. It introduced several seminal operating system ideas related to information organization, sharing, protection, security, and privacy that are relevant to this day.

MULTICS was the seed for the development of the UNIX operating system by Dennis Ritchie and Ken Thompson in Bell Labs in 1974. UNIX⁶ (the name comes from the operating system’s creators to develop a stripped down one-user version of MULTICS) became instantly popular with educational institutions, government labs, and many companies such as DEC and HP. The emergence of Linux obliquely out of UNIX is an interesting story. With the popularity of UNIX and its adoption by widely varying entities, soon there were several incompatible versions of the operating system. In 1987,

⁶ It was originally named UNICS (UNIplexed Information and Computing Service) and the name later changed to UNIX.

Andrew Tannenbaum⁷ of Vrije University developed MINIX as a small clone of UNIX for educational purposes. Linus Torvalds wrote Linux (starting from MINIX) as a free production version of UNIX and soon it took a life of its own due the open software model promoted by the GNU foundation. Despite the interesting history behind the evolution of these different UNIX-based systems, one thing remained the same, namely, the operating systems concepts embodied in these different flavors of UNICES.

All of the above operating systems evolved to support mainframes and minicomputers. In parallel with this evolution, microcomputers were hitting the market and slowly transforming both the computing base as well as the usage model. Computers were no longer necessarily a shared resource, the model suggested by the expensive mainframes and minicomputers. Rather it is a *Personal Computer (PC)* intended for the exclusive use of a single user. There were a number of initial offerings of microcomputers built using Intel 8080/8085 single-chip processors in the 70's. What should the operating system look like for such a personal computer? A simple operating system called CP/M (an acronym for *Control Program Monitor*) was an industry standard for such microcomputers.

IBM developed the IBM PC in the early 1980s. A small company called Microsoft offered to IBM an operating system called MS-DOS for the IBM PC. Early versions of MS-DOS had striking similarities to CP/M, but had a simpler file system and delivered higher performance. With the blessings of an industry giant such as IBM, soon MS-DOS took over the PC market, and sidelined CP/M. Initially, MS-DOS was quite primitive in its functionality given the intended use of the PC, but slowly it started incorporating the ideas of multitasking and timesharing from the UNIX operating system. It is interesting to note that while MS-DOS evolved from a small lean operating system and started incorporating ideas from UNIX operating system, Apple's Mac OS X, also intended for PCs (Apple's Macintosh line), was a direct descendent of UNIX operating system.

When Apple adopted the GUI (Graphical User Interface) in Macintosh personal computer, Microsoft followed suit, offering Windows on top of MS-DOS as a way for users to interact with the PC. Initial offerings of Windows were wrappers on top of MS-DOS until 1995, when Microsoft introduced Windows 95, which integrated the Windows GUI with the operating system. We have gone through several iterations of Windows operating systems since Windows 95 including Windows NT (NT stands for New Technology), Windows NT 4.0, Windows 2000, Windows XP, and Windows Vista (as of January 2007). However, the basic concepts at the level of the core abstractions in the operating system have not changed much in these iterations. If anything, these concepts have matured to the point where they are indistinguishable from those found in the UNIX operating system. Similarly, UNIX-based systems have also absorbed and incorporated the GUI ideas that had their roots in the PC world.

6.15 Review Questions

1. Compare and contrast process and program.

⁷ Author of many famous textbooks in operating systems and architecture.

2. What items are considered to comprise the state of a process?
3. Which metric is the most user centric in a timesharing environment?
4. Consider a pre-emptive priority processor scheduler. There are three processes P1, P2, and P3 in the job mix that have the following characteristics:

Process	Arrival Time	Priority	Activity
P1	0 sec	1	8 sec CPU burst followed by 4 sec I/O burst followed by 6 sec CPU burst and quit
P2	2 sec	3	64 sec CPU burst and quit
P3	4 sec	2	2 sec CPU burst followed by 2 sec I/O burst followed by 2 sec CPU burst followed by 2 sec I/O burst followed by 2 sec CPU burst and quit

What is the turnaround time for each of P1, P2, and P3?

What is the average waiting time for this job mix?

5. What are the deficiencies of FCFS CPU scheduling?
6. Explain the convoy effect in FCFS scheduling.
7. What are the merits of FCFS scheduling?
8. Discuss the different scheduling algorithms with respect to the following criteria: (a) waiting time, (b) starvation, (c) turnaround time, (d) variance in turnaround time.
Which scheduling algorithm was noted as having a high variance in turnaround time?
9. Can any scheduling algorithm be made pre-emptive? What are the characteristics of the algorithm that lends itself to making it pre-emptive? What support is needed from the processor architecture to allow pre-emption?
10. Summarize the enhancements to processor architecture warranted by processor scheduling.
11. Given the following processes that arrived in the order shown

CPU Burst Time IO Burst Time

P1	3	2
P2	4	3
P3	8	4

Show the activity in the processor and the I/O area using the FCFS, SJF and Round Robin algorithms.

12. Redo Example 1 in Section 6.6 using SJF and round robin (timeslice = 2)
13. Redo Example 3 in Section 6.6 using FCFS and round robin (timeslice = 2)

Chapter 7 Memory Management Techniques (Revision number 20)

Let us review what we have seen so far. On the hardware side, we have looked at the instruction-set of the processor, interrupts, and designing a processor. On the software side, we have seen how to use the processor as a resource and schedule it to run different programs. The software entities we have familiarized ourselves with include the *compiler* and *linker* that live above the operating system. We have also familiarized ourselves the *loader* and *process scheduler* that are part of the operating system.

By now, we hope we have de-mystified some of the magic of what is inside “a box.” In this chapter, we continue unraveling the box by looking at another important component of the computer system, namely, the memory.

More than any other subsystem, memory system brings out the strong inter-relationship between hardware and system software. Quite often, it is almost impossible to describe some software aspect of the memory system without mentioning the hardware support. We cover the memory system in three chapters including this one. This chapter focuses on different strategies for memory management by the operating system along with the necessary architectural support. In Chapter 8, we delve into the finer details of page-based memory system, in particular, page replacement policies. Finally, in Chapter 9, we discuss memory hierarchy, in particular, cache memories and main or physical memory.

7.1 Functionalities provided by a memory manager

Let us understand what we mean by memory management. As a point of differentiation, we would like to disambiguate memory management as we mean and discuss in this textbook from the concept of “automatic memory management” that is used by programming languages such as Java and C#. The runtime systems of such languages automatically free up memory not currently used by the program. *Garbage Collection (GC)* is another term used to describe this functionality. Some of the technical issues with GC are similar to those handled by the memory management component of an operating system. Discussion of the similarities and differences between the two are outside the scope of this textbook. The interested reader is referred to other sources to learn more about GC¹.

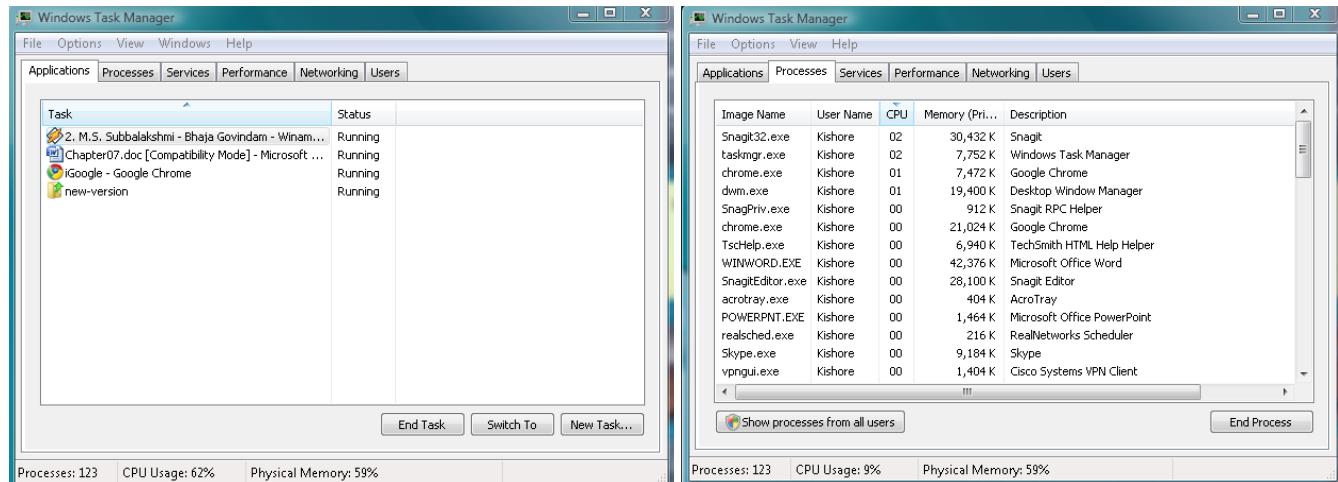
In this textbook, we focus on how the operating system manages memory. Just like the processor, memory is a precious resource and the operating system ensures the best use of this resource. Memory management is an operating system entity that provides the following functionalities.

1. Improved resource utilization

Since the memory is a precious resource, it is best to allocate it on demand. The analogy

¹ Richard Jones, Garbage Collection Algorithms for Automatic Dynamic Memory Management, John Wiley & Sons.

is the use of office space. Each faculty member in the department may ask for a certain amount of space for his/her students and lab. The chair of the department may not give all the space requested but do it incrementally as the faculty member's group grows. Similarly, even though the memory footprint of the program includes the heap (Figure 6.1 in Chapter 6) there is no need to allocate this to a program at startup. It is sufficient to make the allocation if and when the program dynamically requests it. If a faculty member's group shrinks, he/she does not need all the space allocated any more. The chair of the department would reclaim that space and reallocate it to someone else who needs it more. In the same manner, if a process is not actively using memory allocated to it, then perhaps it is best to release it from that process and use it for some of other process that needs it. Both these ideas, incremental allocation and dynamic reallocation, will lead to improved resource utilization of memory. The criticality of using this resource judiciously is best understood by showing a simple visual example. The left half of the following figure shows a screenshot of the task manager listing the actual applications running on a laptop. The right half of the same figure shows a partial listing of the actual processes running on the laptop. There are 4 applications running but there are 123 processes! The purpose for showing this screenshot is to get the point across that in addition to the user processes, the operating system and other utilities spawn a number of background processes (often called *daemon* processes). Thus, the demand placed on the memory system by the collection of processes running at any point of time on the computer is quite intense. The laptop shown in this example has 2 GB of memory out of which 58% is in use despite the fact that only 4 applications are running at this point of time.



2. Independence and Protection

We mentioned that several processes are co-resident in memory at any point of time. A program may have a bug causing it to run amok writing into areas of memory that is not part of its footprint. It would be prudent to protect a buggy process from itself and from other processes. Further, in these days of computer viruses and worms, a malicious program could be intentionally trying to corrupt the memories of other genuine programs. Therefore, the memory manager provides *independence* for each process, and *memory protection* from one another. Once again using the space analogy, you probably can

relate to the need for independence and protection if you grew up in a home with siblings. You either had or wished you had a room for yourself that you can lock when you want to keep the siblings (and the parents) out!

3. Liberation from resource limitations

When a faculty member is hired, the chair of the department would promise as much space as he/she wants, perhaps even larger than the total space available in the department. This gives the faculty member to think “big” in terms of establishing and growing his/her research agenda and the student base. In a similar manner, it is ideal for the programmer not to worry about the amount of physical memory while developing his/her program. Imagine a faculty member has 10 students but has lab space only for five. The students may have to work different shifts sharing the same office space. In the same vein, imagine having to write a multi-player video game program, and your manager says you have a total memory space of 10Kbytes. You, as the programmer, will have to think of identifying data structures you will not need at the same time and use the same memory space for storing them (in essence, overlaying these data structures in memory). The program can get ugly if you have to resort to such methods. To address this problem, the memory manager and the architecture work together to devise mechanisms that gives the programmer an illusion of a large memory. The actual physical memory may be much smaller than what the programmer is given an illusion of.

4. Sharing of memory by concurrent processes

You may want to keep your siblings out most of the time but there are times when you want to be able to let them come into your room, perhaps to play a video game. Similarly, while memory protection among processes is necessary, sometimes processes may want to *share* memory either implicitly or explicitly. For example, you may have multiple browser windows open on your desktop. Each of them may be accessing a different web page, but if all of them are running the same browser application then they could share the code. This is an example of implicit sharing among processes. Copying and pasting an image from a presentation program into a word processor is an example of explicit data sharing. The memory manager facilitates memory sharing when needed among processes.

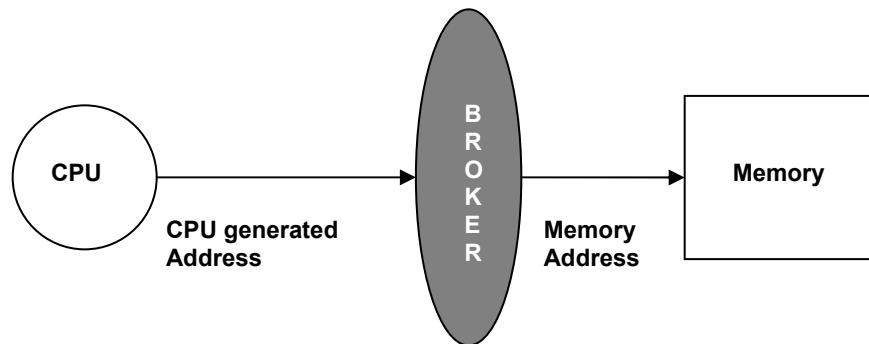


Figure 7.1: A Generic Schematic of a Memory manager

All of the above functionalities come at a price. In particular, we are introducing a *broker* between the CPU and the memory as shown in Figure 7.1. The broker is a piece

of hardware that provides the mechanisms needed for implementing the memory management policies. In principle, it maps a CPU generated (logical) memory address to the *real* memory address for accessing the memory. The sophistication of the broker depends on the functionalities provided by the memory manager.

So far, we have not introduced such functionality in LC-2200. However, as we get ambitious and let LC-2200 take over the world from game players to high-performance computers, such functionalities become indispensable.

The overall goals of a good memory manager are three-fold:

1. Require minimal hardware support
2. Keep the impact on memory accesses low
3. Keep the memory management overhead low (for allocation and de-allocation of memory)

7.2 Simple Schemes for Memory Management

In this section, let us consider some simple schemes for memory management and the corresponding hardware support. This section is written in the spirit of shared discovery to understand how to accomplish the goals enumerated above, while at the same time meeting the functionalities presented in the previous section. The first two schemes and the associated hardware support (fence register and bounds registers) are for illustration purposes only. We do not know of any machine architecture that ever used such schemes. The third scheme (base and limit registers) was widely used in a number of architectures including CDC 6600 (the very first supercomputer) and IBM 360 series. All three have limitations in terms of meeting the functionalities identified in the previous section, and thus set the stage for the sophisticated memory management schemes, namely, paging and segmentation, found in modern architectures.

1. Separation of user and kernel

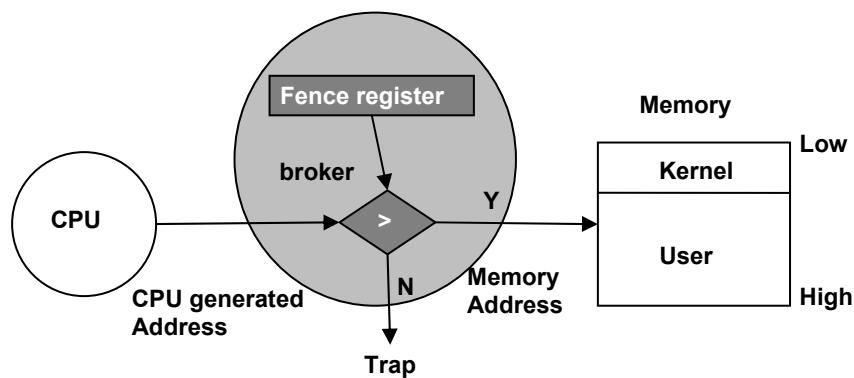


Figure 7.2: Fence Register

Let us consider a very simple memory management scheme. As we have seen before, the operating system and the user programs share the total available memory space. The space used by the operating system is the *kernel space*, and the space occupied by the user programs is the *user space*. As a first order of approximation, we wish to ensure that

there is a boundary between these two spaces, so that a user program does not straddle into the memory space of the operating system. Figure 7.2 shows a simple hardware scheme to accomplish this separation. The shaded area in the figure corresponds to the hardware portion of the work done by the broker in Figure 7.1. The scheme relies on three architectural elements: a *mode* bit that signifies whether the program is in user or kernel mode; a *privileged instruction* that allows flipping this bit; and a *fence* register. As you can imagine, the name for this register comes from the analogy of physical fences one might build for property protection. The memory manager sets the fence register when the user program is scheduled. The hardware validates the processor generated memory address by checking it against this fence register. This simple hardware scheme gives memory protection between the user program and the kernel. For example, let us assume that the fence register is set to 10000. This means that the kernel is occupying the memory region 0 through 10000. In user mode, if the CPU generates any address over 10000 then the hardware considers it a valid user program address. Anything less than or equal to 10000 is a kernel address and generates an access violation trap. The CPU has to be in kernel mode to access the kernel memory region. To understand how the CPU gets into the kernel mode, recall that in Chapter 4 we introduced the notion of *traps*, a synchronous program discontinuity usually caused by program wishing to make a system call (such as reading a file). Such traps result in the processor automatically entering the kernel mode as part of implementation of the trap instruction. Thus, the CPU implicitly gets into kernel mode on system calls and is now able to address the memory region reserved for the kernel. Once the operating system completes the system call, it can explicitly return to the user mode by using the architecture provided privileged instruction. It is a privileged instruction since its use is limited to kernel mode. Any attempt to execute this instruction in user mode will result in an *exception*, another synchronous program discontinuity (which we introduced in Chapter 4 as well) caused by an illegal action by a user program. As you may have already guessed, writing to the fence register is a privileged instruction as well.

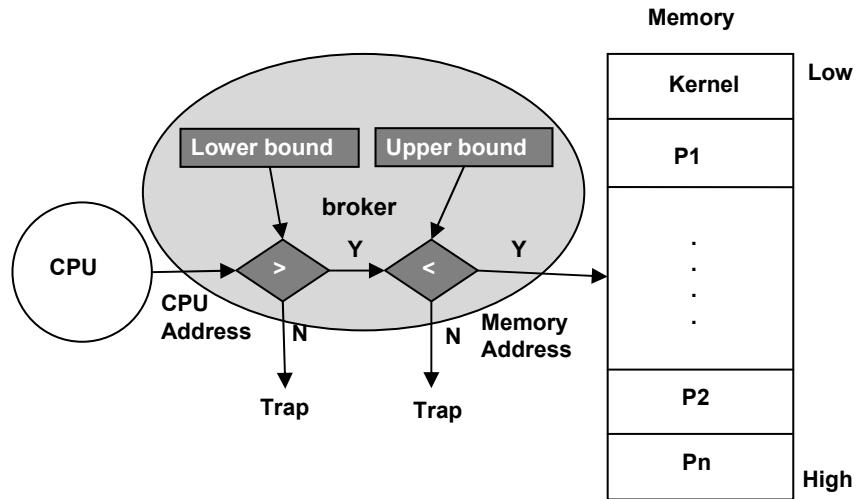


Figure 7.3: Bounds Registers

2. Static relocation

As previously discussed, we would like several user programs to be co-resident in memory at the same time. Therefore, the memory manager should protect the co-resident processes from one another. Figure 7.3 shows a hardware scheme to accomplish this protection. Once again, the shaded area in the figure corresponds to the hardware portion of the work done by the broker in Figure 7.1.

Static relocation refers to the memory bounds for a process being set at the time of linking the program and creating an executable file. Once the executable is created, the memory addresses cannot be changed during the execution of the program. To support memory protection for processes, the architecture provides two registers: *upper bound* and *lower bound*. The *bounds* registers are part of the process control block (PCB, which we introduced in Chapter 6). Writing to the bounds registers is a privileged instruction provided by the architecture. The memory manager sets the values from the PCB into the bounds registers at the time of dispatching the process (please refer to Chapter 6 on scheduling a process on the CPU). At link time, the linker allocates a particular region in memory for a particular process². The loader fixes the lower and upper bound register values for this process at load time and never changes them for the life of the program. Let us assume that the linker has assigned P1 the address range 10001 to 14000. In this case, the scheduler will set the lower bound and upper bound registers to 10000 and 14001, respectively. Thus while P1 is executing, if the CPU generates addresses in the range 10001 and 14000, the hardware permits them as valid memory accesses. Anything outside this range will result in an access violation trap.

A concept that we alluded to in Chapter 6 is *swapping*. “Swapping out” a process refers to the act of moving an inactive process (for example, if it is waiting on I/O completion) from the memory to the disk. The memory manager does this act to put the memory occupied by the inactive process to good use by assigning it to other active processes. Similarly, once a process becomes active again (for example, if its I/O request is complete) the memory manager would “swap in” the process from the disk to the memory (after making sure that the required memory is allocated to the process being swapped in).

Let us consider swapping a process in from the disk with static relocation support. Due to the fixed bounds, the memory manager brings a swapped out process back to memory into exactly the same spot as before. If that memory space is in use by a different process then the swapped out process cannot be brought back into memory just yet, which is the main limitation with static relocation.

In reality, compilers generate code assuming some well-known address where the program will reside in memory³. Thus, if the operating system cannot somehow rectify this assumption a process is essentially *non-relocatable*. We define a process as non-

² Note that modern operating systems use dynamic linking, which allows deferring this decision of binding the addresses to a process until the time of loading the process into memory. Such a dynamic linker could take the current usage of memory into account to assign the bounds for a new process.

³ In most modern compilers, a program starts at address 0 and goes to some system-specified maximum address.

relocatable if the addresses in the program cannot be changed either during loading into memory or during execution. We define static relocation as the technique of locating a process at *load time* in a different region of memory than originally intended at compile time. That is, the addresses used in the program are bound (i.e., fixed) at the time the program is loaded into memory and do not change during execution. IBM used a version of this kind of static relocation in their early models of mainframes (in the early 1960's). At the time of loading a process into memory, the loader will look at unused space in memory and make a decision as to where to locate the new process. It will then "fix" up all the addresses in the executable so that the program will work correctly in its new home. For example, let us say the original program occupied addresses 0 to 1000. The loader decides to locate this program between addresses 15000 and 16000. In this case, the loader will add 15000 to each address it finds in the executable as it loads it into memory. As you can imagine, this is a very cumbersome and arduous process. The loader has an intimate knowledge of the layout of the executable so that it can tell the difference between constant values and addresses to do this kind of fix up.

3. Dynamic relocation

Static relocation constrains memory management and leads to poor memory utilization. This is because once created an executable occupies a fixed spot in memory. Two completely different programs that happen to have the same or overlapping memory bounds cannot co-exist in memory simultaneously even if there are other regions of memory currently unoccupied. This is similar to two kids wanting to play with the same toy though there are plenty of other toys to play with! This is not a desirable situation. *Dynamic relocation* refers to the ability to place an executable into any region of memory that can accommodate the memory needs of the process. Let us understand how this differs from static relocation. With dynamic relocation, the memory address generated by a program can be changed during the *execution* of the program. What this means is that, at the time of loading a program into memory, the operating system can decide where to place the program based on the current usage of memory. You might think that this is what a dynamic linker lets us do in the previous discussion on static relocation. However, the difference is, even if the process is swapped out, when it is later brought in, it need not come to the same spot it occupied previously. Succinctly put, with static relocation, addresses generated by the program are fixed during execution while they can be changed during execution with dynamic relocation.

Now, we need to figure out the architectural support for dynamic relocation. Let us try a slightly different hardware scheme as shown in Figure 7.4. As before, the shaded area in the figure corresponds to the hardware portion of the work done by the broker in Figure 7.1. The architecture provides two registers: *base* and *limit*. A CPU generated address is *always* shifted by the value in the base register. Since this shift necessarily happens during the execution of the program, such an architectural enhancement meets the criterion for dynamic relocation. As in the case of static relocation, these two registers are part of the PCB of every process. Every time a process is brought into memory (at either load time or swap time), the loader assigns the values for the base and bound registers for the process. The memory manager records these loader assigned values into the corresponding fields of the PCB for that process. Similar to the bounds registers for

static relocation, writing to the base and limit registers is a privileged instruction supported by the architecture. When the memory manager dispatches a particular process, it sets the values for the base and limit registers from the PCB for that process. Let us assume that P1's memory footprint is 4000. If the loader assigns the address range 10001 to 14000 for P1, then the memory manager sets the base register to 10001 and the limit register 14001 in the PCB for P1. Thus when P1 is executing, any CPU generated address is automatically shifted up by 10000 by the hardware. So long as the shifted address is less than the value set in the limit register, the hardware permits it as a valid memory access. Anything that is beyond the limit results in an access violation trap. The reader should feel convinced that dynamic relocation will result in better memory utilization than static relocation.

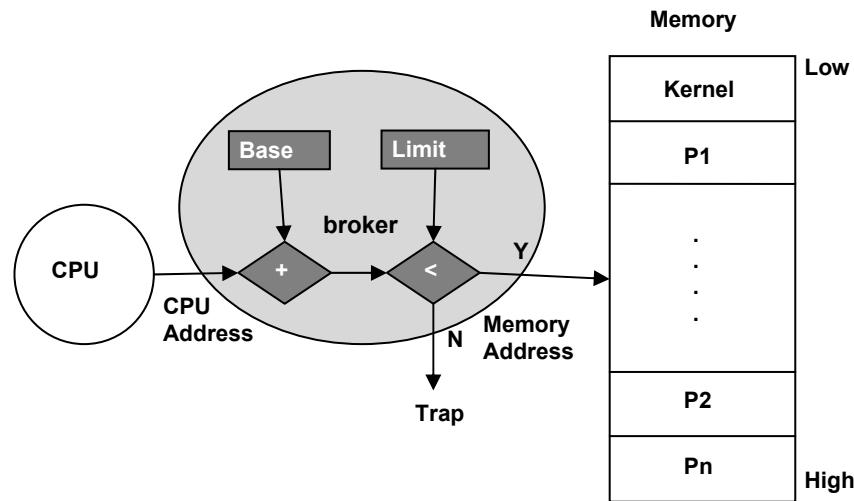


Figure 7.4: Base and Limit Registers

The architectural enhancements we presented for both static and dynamic relocation required two additional registers in the processor. Let us review these two schemes with respect to hardware implementation. Compare the datapath elements that you would need for Figures 7.3 and 7.4. For Figure 7.3, you need two comparison operations for bounds checking. For Figure 7.4, you need an addition followed by a comparison operation. Thus, in both schemes you will need two arithmetic operations to generate the memory address from the CPU address. Therefore, both schemes come out even in terms of hardware complexity and delays. However, the advantage one gets in terms of memory utilization is enormous with the base and limit register scheme. This is an example of how a little bit of human ingenuity can lead to enormous gains with little or no added cost.

7.3 Memory Allocation Schemes

We will assume using the *base plus limit register* scheme as the hardware support available to the memory manager and discuss some policies for memory allocation. In each case, we will identify the data structures needed to carry out the memory management.

7.3.1 Fixed Size Partitions

In this policy, the memory manager would divide the memory into fixed size partitions. Let us understand the data structure needed by the memory manager. Figure 7.5 shows a plausible data structure in the form of an allocation table kept in kernel space. In effect, the memory manager manages the portion of the memory that is available for use by user programs using this data structure. For this allocation policy, the table contains three fields as shown in Figure 7.5. The *occupied bit* signifies whether the partition is in use or not. The bit is 1 if the partition has been allocated; 0 otherwise. When a process requests memory (either at load time or during execution), it is given one of the fixed partitions that is equal to or greater than the current request. For example, if the memory manager has partitions of 1KB, 5KB and 8KB sizes, and if a process P1 requests a 6KB memory chunk, then it is given the 8KB partition. The memory manager sets the corresponding bit in the table to 1; and resets the bit when the process returns the chunk. Upon allocating P1's request for 6KB the allocation table looks as shown in Figure 7.5a. Note that there is wasted space of 2KB within this 8KB partition. Unfortunately, it is not possible to grant a request for a 2KB memory chunk from another process using this wasted space. This is because the allocation table maintains summary information based on fixed size partitions. This phenomenon, called *internal fragmentation*, refers to the wasted space internal to fixed size partitions that leads to poor memory utilization. In general, internal fragmentation is the difference between the granularity of memory allocation and the actual request for memory.

$$\text{Internal fragmentation} = \text{Size of Fixed partition} - \text{Actual memory request} \quad (1)$$

Example 1:

A memory manager allocates memory in fixed chunks of 4 K bytes. What is the maximum internal fragmentation possible?

Answer:

The smallest request for memory from a process is 1 byte. The memory manager allocates 4 K bytes to satisfy this request.

So the maximum internal fragmentation possible

$$\begin{aligned} &= 4\text{K bytes} - 1 \\ &= 4096 - 1 = 4095 \text{ bytes.} \end{aligned}$$

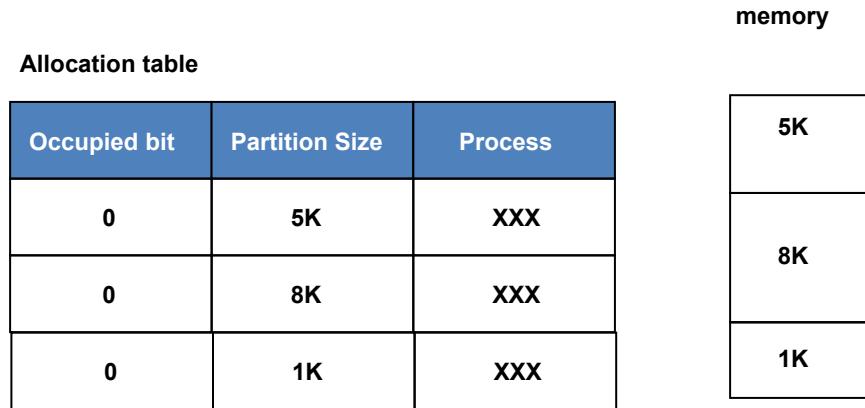


Figure 7.5: Allocation Table for fixed size partitions

Suppose there is another memory allocation request for 6KB while P1 has the 8KB partition. Once again, this request also cannot be satisfied. Even though cumulatively (between the 1KB and 5KB partitions) 6KB memory space is available, it is not possible to satisfy this new request since the two partitions are not contiguous (and a process's request is for a contiguous region of memory). This phenomenon, called *external fragmentation*, also results in poor memory utilization. In general, external fragmentation is the sum of all the non-contiguous memory chunks available to the memory system.

$$\text{External Fragmentation} = \sum \text{All non-contiguous memory partitions} \quad (2)$$

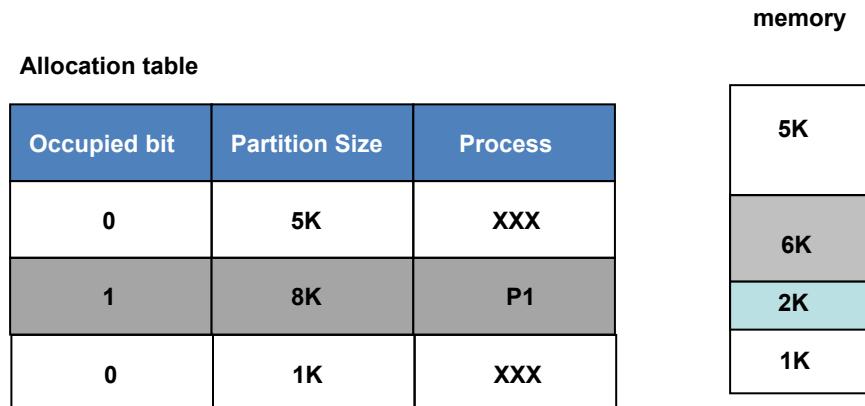


Figure 7.5a: Allocation Table after P1's request satisfied

7.3.2 Variable Size Partitions

To overcome the problem of internal fragmentation, we will discuss a memory manager that allocates variable sized partitions commensurate with the memory requests. Let us assume that 13KB is the total amount of memory that is available to the memory manager. Instead of having a static allocation table as in the previous scheme, the memory manager dynamically builds the table on the fly. Figure 7.6 shows the initial state of the allocation table before any allocation.



Figure 7.6: Allocation Table for Variable Size Partitions

Figure 7.6a shows the table after the memory manager grants a series of memory requests. Note that the manager has 2KB of free space left after satisfying the requests of P1, P2, and P3.

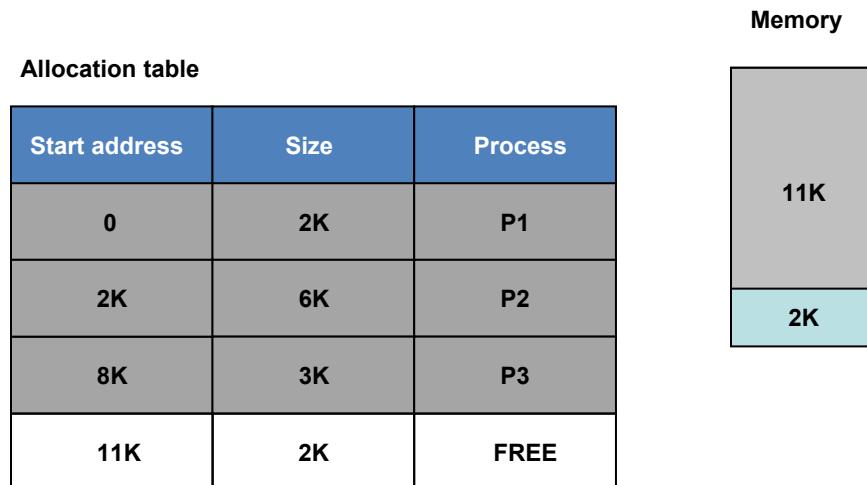


Figure 7.6a: State of the Allocation Table after a series of memory requests from P1 (2KB); P2 (6KB); and P3 (3KB)

Figure 7.6b shows the state upon P1's completion. The 2KB partition occupied by P1 is marked FREE as well.

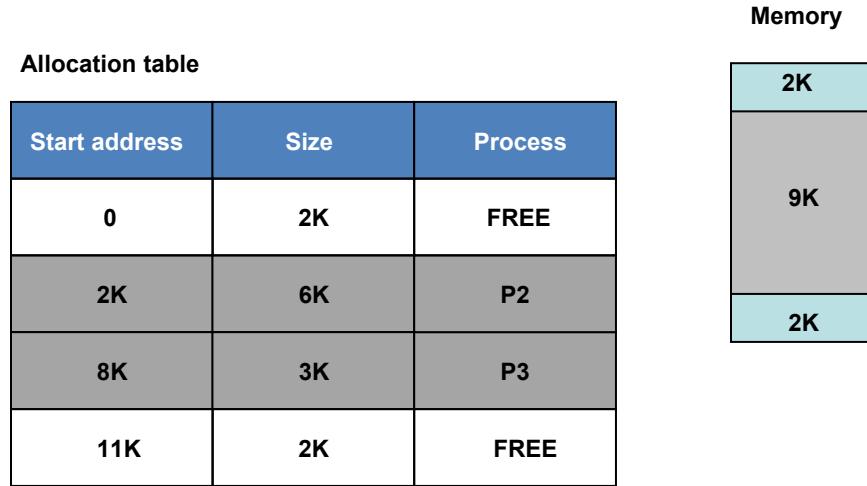


Figure 7.6b: State of allocation table after P1's completion

Suppose there is a new request for a 4KB chunk of memory from a new process P4. Unfortunately, this request cannot be satisfied since the space requested by P4 is contiguous in nature but the available space is fragmented as shown in Figure 7.6b. Therefore, variable size partition, while solving the internal fragmentation problem does not solve the external fragmentation problem.

As processes complete, there will be *holes* of available space in memory created. The allocation table records these available spaces. If adjacent entries in the allocation table free up, the manager will be able to coalesce them into a larger chunk as shown in the before-after Figures 7.6c and 7.6d, respectively.

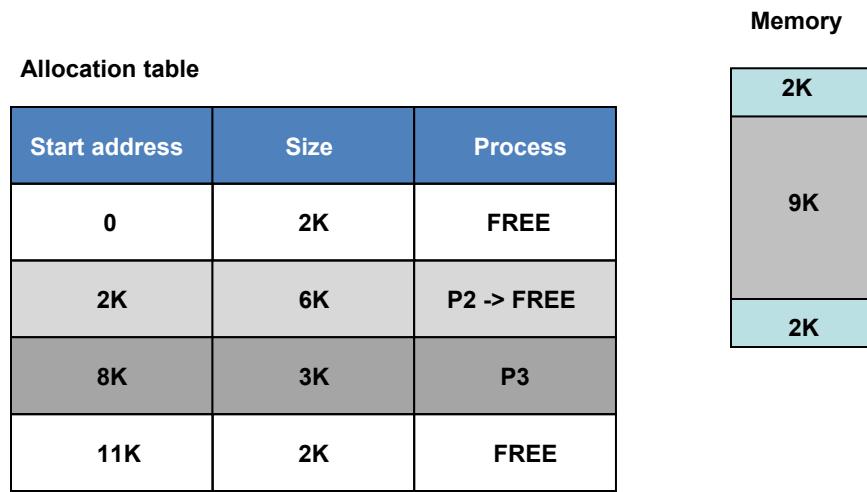


Figure 7.6c: Allocation Table before P2 releases memory



Figure 7.6d: Allocation Table after P2 releases memory (coalesced)

Example 2:

What is the maximum external fragmentation represented by each of the Figures 7.6b, and d?

Answer:

In Figure 7.6b, 2 chunks of 2KB each are available but non contiguous.

$$\text{External fragmentation} = 4\text{K bytes}$$

In Figure 7.6d, two chunks of 8KB and 2KB are available but not contiguous.

$$\text{External fragmentation} = 10\text{K bytes}$$

Upon a new request for space, the memory manager has several options on how to make the allocation. Here are two possibilities:

1. Best fit

The manager looks through the allocation table to find the best fit for the new request. For example, with reference to Figure 7.6d, if the request is for 1KB then the manager will make the allocation by splitting the 2KB free space rather than the 8KB free space.

2. First fit

The manager will allocate the request by finding the first free slot available in the allocation table that can satisfy the new request. For example, with reference to Figure 7.6d, the memory manager will satisfy the same 1KB request by splitting the 8KB free space.

The choice of the allocation algorithm has tradeoffs. The time complexity of the best-fit algorithm is high when the table size is large. However, the best-fit algorithm will lead to better memory utilization since there will be less external fragmentation.

7.3.3 Compaction

The memory manager resorts to a technique called *compaction* when the level of external fragmentation goes beyond tolerable limits. For example, referring to Figure 7.6d, the memory manager may relocate the memory for P3 to start from address 0, thus creating a

contiguous space of 10KB as shown in Figure 7.6e. Compaction is an expensive operation since all the embedded addresses in P3's allocation have to be adjusted to preserve the semantics. This is not only expensive but also virtually impossible in most architectures. Remember that these early schemes for memory management date back to the 60's. It is precisely for allowing dynamic relocation that IBM 360 introduced the concept of a base register⁴ (not unlike the scheme shown in [Figure 7.4](#)). OS/360 would dynamically relocate a program at load time. However, even with this scheme, once a process is loaded in memory, compaction would require quite a bit of gyrations such as stopping the execution of the processes to do the relocation. Further, the cost of compaction goes up with the number of processes requiring such relocation. For this reason, even in architectures where it is feasible to do memory compaction, memory managers perform compaction rarely. It is usual to combine compaction with swapping; i.e., the memory manager will relocate the process as it is swapped back into memory.

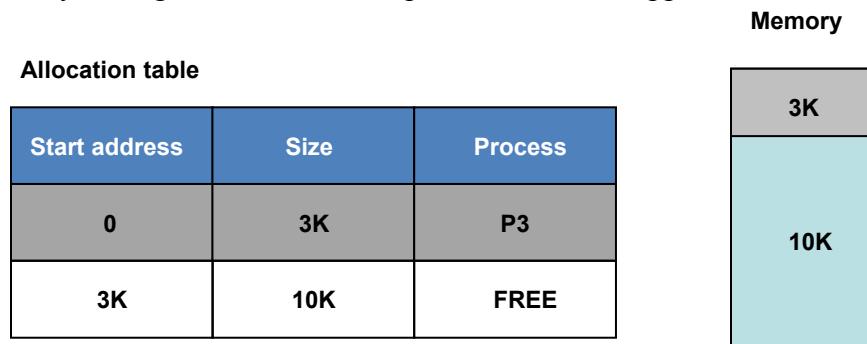


Figure 7.6e: Memory compacted to create a larger contiguous space

7.4 Paged Virtual Memory

As the size of the memory keeps growing, external fragmentation becomes a very acute problem. We need to solve this problem.

Let us go to basics. The user's view of the program is a contiguous footprint in memory. The hardware support for memory management that we have discussed until now, at best relocates the program to a different range of addresses from that originally present in the user's view. Basically, we need to get around this inherent assumption of contiguous memory present in the user's view. The concept of *virtual memory* helps get around this assumption. *Paging* is a vehicle for implementing this concept.

The basic idea is to let the user preserve his view of contiguous memory for his program, which makes program development easy. The broker (in Figure 7.1) breaks up this contiguous view into equal *logical* entities called *pages*. Similarly, the physical memory consists of *page frames*, which we will refer to simply as physical frames. Both the logical page and the physical frame are of the same fixed size, called *pagesize*. A physical frame *houses* a logical page.

⁴ Source: Please see <http://www.research.ibm.com/journal/rd/441/amdahl.pdf> for the original paper by Gene Amdahl on IBM 360.

Let us consider an analogy. The professor likes to get familiar with all the students in his class. He teaches a large class. To help him in his quest to get familiar with all the students, he uses the following ruse. He collects the photos of his students in the class. He has an empty picture frame in his office (Figure 7.7-(a)). When a student comes to visit him during his office hours, he puts up the picture of the student in the frame (Figure 7.7-(b)). When the next student comes to visit, he puts up the picture of that student in the frame (Figure 7.7-(c)). The professor does not have a unique picture frame for each student, but simply re-uses the same frame for the different students. He does not need a unique frame for each student either since he sees him or her one at a time during his office hours anyhow.

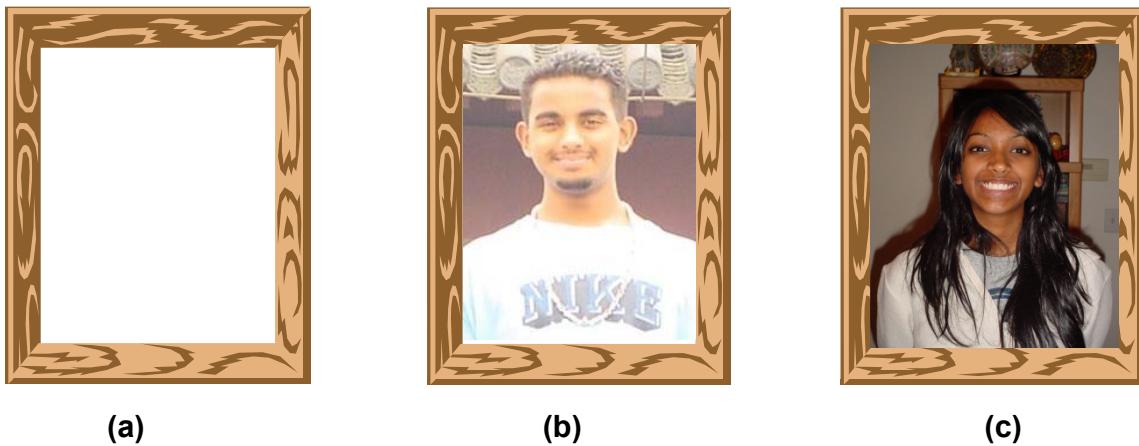


Figure 7.7: Picture Frame Analogy

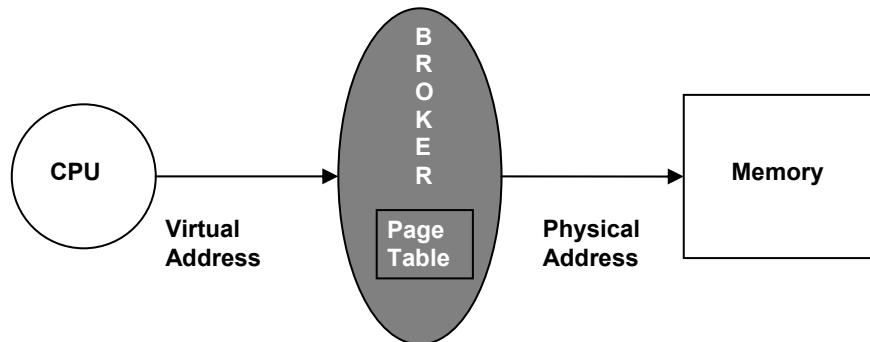


Figure 7.8: Page Table

Allocation of physical memory divided into page frames bears a lot of similarity to this simple analogy. The picture frame can house any picture. Similarly, a given physical frame can be used to house any logical page. The broker maintains a *mapping* between a user's *logical page* and the physical memory's *physical frame*. As one would expect it is the responsibility of the memory manager to create the mapping for each program. An entity called the *page table* holds the mapping of logical pages to physical frames. The page table effectively decouples the user's view of memory from the physical

organization. For this reason, we refer to the user's view as *virtual memory* and to the logical pages as *virtual pages*, respectively. The CPU generates *virtual addresses* corresponding to the user's view. The broker *translates* this virtual address to a physical address by looking up the page table (shown in Figure 7.8). Since we have decoupled the user's view from the physical organization, the relative sizes of the virtual memory and physical memory do not matter. For example, it is perfectly reasonable to have a user's view of the virtual memory to be much larger than the actual physical memory. In fact, this is the common situation in most memory systems today. Essentially, the larger virtual memory removes any resource restriction arising out of limited physical memory, and gives the illusion of a much larger memory to user programs.

The broker is required to maintain the contiguous memory assumption of the user only to addresses within a page. The distinct pages need not be contiguous in physical memory. Figure 7.9 shows a program with four virtual pages mapped using the paging technique to four physical frames. Note that the paging technique circumvents external fragmentation. However, there can be internal fragmentation. Since the frame size is fixed, any request for memory that only partially fills a frame will lead to internal fragmentation.

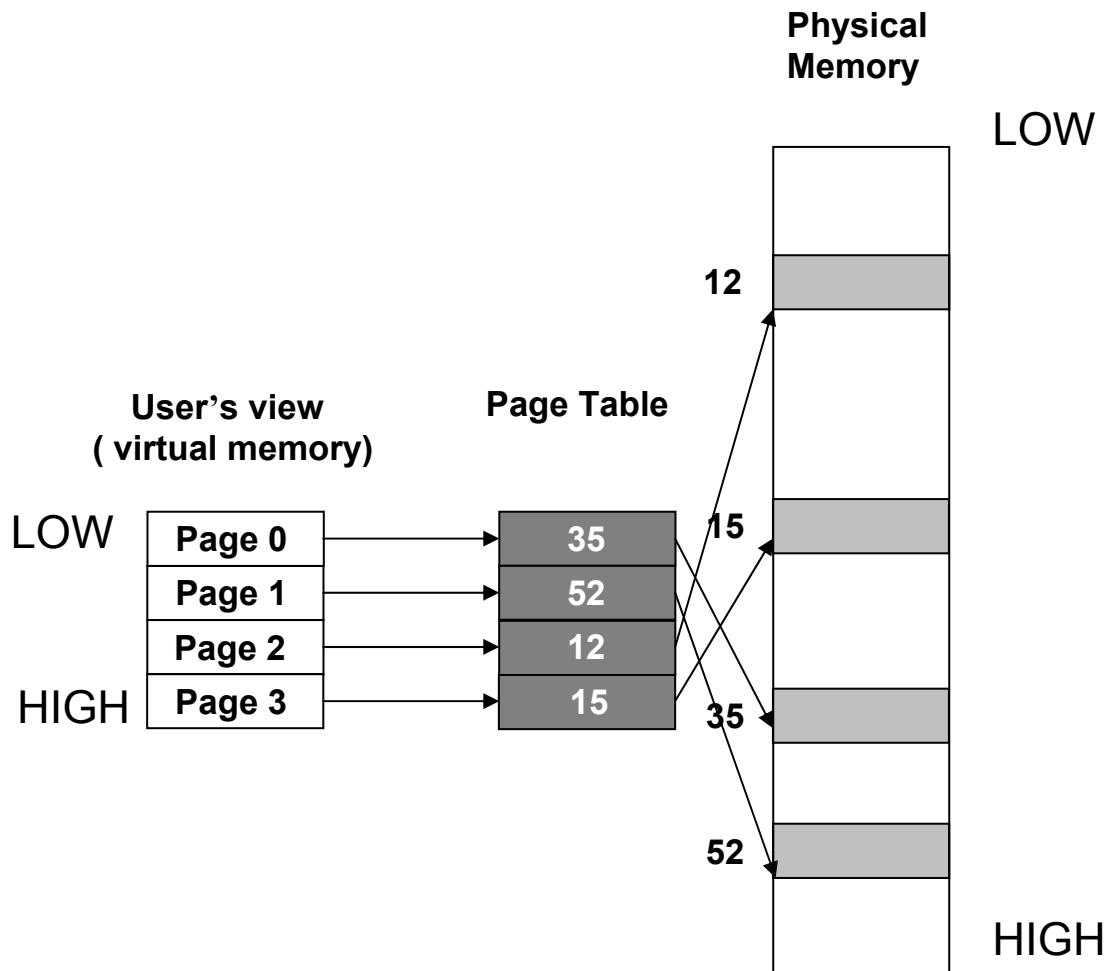


Figure 7.9: Breaking the user's contiguous view of virtual memory

7.4.1 Page Table

Let us drill a little deeper into the paging concept. Given that a virtual page (or a physical frame) is of fixed size, and that addresses within a page are contiguous, we can view the virtual address generated by the CPU as consisting of two things: *virtual page number (VPN)* and *offset* within the page. For the sake of exposition, we will assume that the page size is an integral power of two. The hardware first breaks up the virtual address into these two pieces. This is a straightforward process. Remember that all the locations within a given page are contiguous. Therefore, the offset portion of the virtual address must come from the low order bits. The number of bits needed for the offset is directly discernible from the page size. For example, if the page size is 8 Kbytes then there are 2^{13} distinct bytes in that page; so, we need 13 bits to address each byte uniquely, which will be the size of the offset. The remaining high order bits of the virtual address form the virtual page number. In general, if the page size is N then $\log_2 N$ low order bits of the virtual address form the page offset.

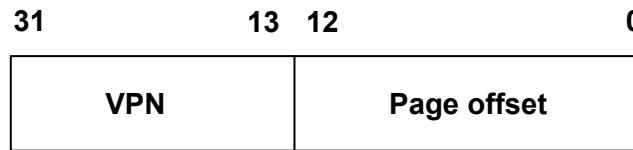
Example 3:

Consider a memory system with a 32-bit virtual address. Assume that the pagesize is 8K Bytes. Show the layout of the virtual address into VPN and page offset.

Answer:

Each page has 8K bytes. We need 13 bits to address each byte within this page uniquely. Since the bytes are contiguous within a page, these 13 bits make up the least significant bits of the virtual address (i.e., bit positions 0 through 12).

The remaining 19 high order bits of the virtual address (i.e., bit positions 13 through 31) signify the virtual page number (VPN).



The translation of a virtual address to a physical address essentially consists of looking up the page table to find the *physical frame number (PFN)* corresponding to the virtual page number (VPN) in the virtual address. Figure 7.10 shows this translation process. The shaded area in Figure 7.10 is the hardware portion of the work done by the broker. The hardware looks up the page table to translate the virtual to physical address. Let us investigate where to place the page table. Since the hardware has to look it up on every memory access to perform the translation, it seems fair to think that it should be part of the CPU datapath. Let us explore the feasibility of this idea. We need an entry for every VPN. In Example 3 above, we need 2^{19} entries in the page table. Therefore, it is infeasible to implement the page table as a hardware device in the datapath of the processor. Moreover, there is not just one page table in the system. To provide memory protection every process needs its own page table.

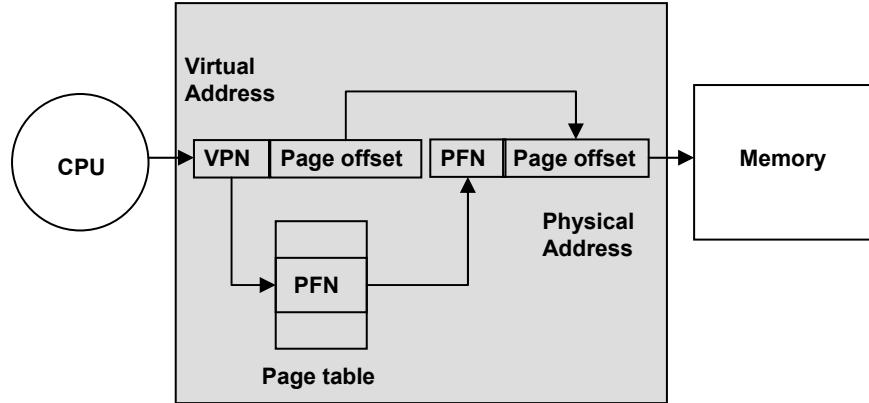


Figure 7.10: Address Translation

Therefore, the page table resides in memory, one per process as shown in Figure 7.11. The CPU needs to know the location of the page table in memory to carry out address translation. For this purpose, we add a new register to the CPU datapath, *Page Table Base Register (PTBR)*, which contains the base address of the page table for the currently running process. The PTBR is part of the process control block. At context switch time, this register value is loaded from the PCB of the newly dispatched process.

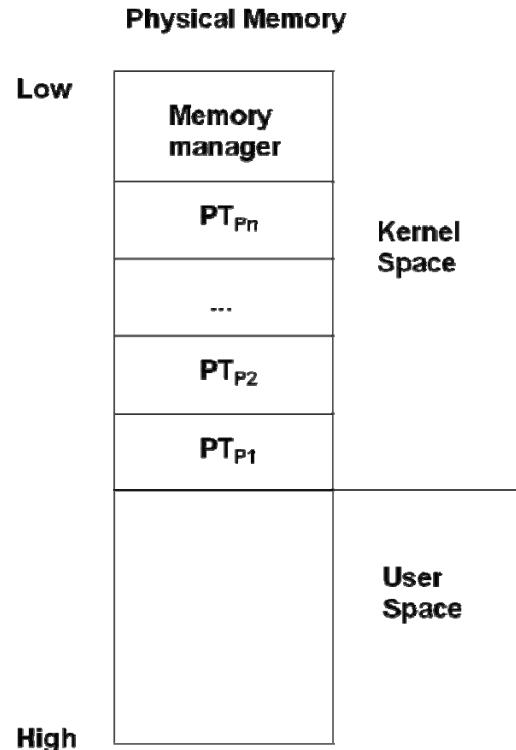


Figure 7.11: Page Tables in Physical Memory

Example 4:

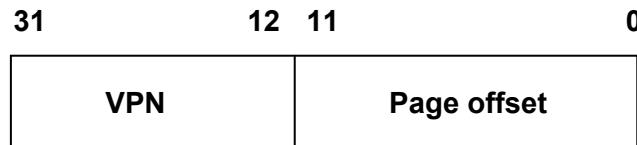
Consider a memory system with 32-bit virtual addresses and 24-bit physical memory addresses. Assume that the pagesize is 4K Bytes. (a) Show the layout of the virtual and physical addresses. (b) How big is the page table? How many page frames are there in this memory system?

Answer:

a)

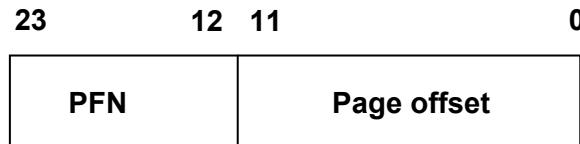
For a pagesize of 4Kbytes, the lower 12-bits of the 32-bit virtual address signify the page offset, and the remaining high order bits (20-bits) signify the VPN.

Layout of the virtual address:



Since the physical address is 24 bits, the high order bits 12 bits (24-12) of the physical address forms the PFN.

Layout of the physical address:



(b)

$$\text{Number of page table entries} = 2^{(\text{Number of bits in VPN})} = 2^{20}$$

Assuming each entry is 1 word of 32 bits (4 bytes),

$$\text{The size of the page table} = 4 \times 2^{20} \text{ bytes} = 4 \text{ Mbytes}$$

$$\text{Number of page frames} = 2^{(\text{Number of bits in PFN})} = 2^{12} = 4096$$

7.4.2 Hardware for Paging

The hardware for paging is straightforward. We add a new register, PTBR, to the datapath. On every memory access, the CPU computes the address of the *page table entry* (PTE) that corresponds to the VPN in the virtual address using the contents of the PTBR. The PFN fetched from this entry concatenated with the page offset gives the

physical address. This is the translation process for fetching either the instruction or the data in the FETCH and MEM stages of the pipelined processor, respectively. The new hardware added to the processor to handle paging is surprisingly minimal for the enormous flexibility achieved in memory management. Let us review the overhead for memory accesses with paging. In essence, the hardware makes two trips to the memory for each access: first to retrieve the PFN and second to retrieve the memory content (instruction or data). This seems grossly inefficient and untenable from the point of view of sustaining a high performance processor pipeline. Fortunately, it is possible to mitigate this inefficiency significantly to make paging actually viable. The trick lies in remembering the recent address translations (i.e., VPN to PFN mappings) in a small hardware table inside the processor for future use, since we are most likely to access many memory locations in the same physical page. The processor first consults this table, called a *translation lookaside buffer (TLB)*. Only on not finding this mapping does it retrieve the PFN from physical memory. More details on TLB will be forthcoming in the next chapter (please see Section 8.5).

7.4.3 Page Table Set up

The memory manager sets up the page table on a process startup. In this sense, the page table does double duty. The hardware uses it for doing the address translation. It is also a data structure under the control of the memory manager. Upon setting up the page table, the memory manager records the PTBR value for this process in the associated PCB. Figure 7.12 shows the process control block with a new field for the PTBR.

```
typedef struct control_block_type {
    enum state_type state;
    address PC;
    int reg_file[NUMREGS];
    struct control_block *next_pcb;
    int priority;
    address PTBR;
    ....
    ....
} control_block;
```

Figure 7.12: PCB with PTBR field

7.4.4 Relative sizes of virtual and physical memories

As motivated in this section, the whole purpose of virtual memory is to liberate the programmer from the limitations of available physical memory. Naturally, this leads us to think that virtual memory should always be larger than physical memory. This is a perfectly logical way to think about virtual memory. Having said that, let us investigate if it makes sense to have physical memory that is larger than the virtual memory. We can immediately see that it would not help an individual program since the address space available to the program is constrained by the size of the virtual memory. For example,

with a 32-bit virtual address space, a given program has access to 4 Gbytes of memory. Even if the system has more physical memory than 4 Gbytes, an individual program cannot have a memory footprint larger than 4 Gbytes. However, a larger physical memory can come in handy for the operating system to have more resident processes that are huge memory hogs. This is the reason for Intel architecture's *Physical Address Extension (PAE)* feature, which extends the physical address from 32-bits to 36-bits. As a result, this feature allows the system to have up to 64 Gbytes of physical memory, and the operating system can map (using the page table) a given process's 4 Gbytes virtual address space to live in different portions of the 64 Gbytes physical memory⁵.

One could argue that with the technological advances that allow us to have larger than 32-bit physical addresses, the processor architecture should support a larger virtual address space as well. One would be right, and in fact, vendors including Intel have already come out with 64-bit architectures. The reason why Intel offers the PAE feature is simply to allow larger physical memories on 32-bit platforms that are still in use for supporting legacy applications.

7.5 Segmented Virtual Memory

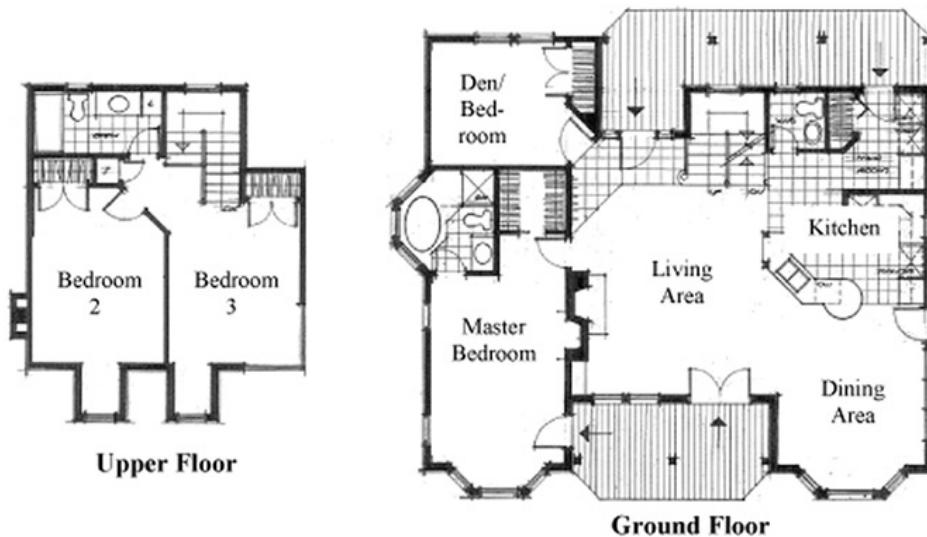


Figure 7.13: Floor plan for a house⁶

Let us consider an analogy. Figure 7.13 shows a floor plan for a house. You may have a living room, a family room, a dining room, perhaps a study room, and one or more bedrooms. In other words, we have first logically organized the space in the house into functional units. We may then decide the amount of actual physical space we would like to allocate given the total space budget for the house. Thus, we may decide to have a

⁵ The interested reader is referred to, "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1."

⁶ Source: <http://www.edenlanehomes.com/images/design/Typical-Floorplan.jpg>

guest bedroom that is slightly larger than say a kid's bedroom. We may have breakfast area that is cozy compared to the formal dining room, and so on. There are several advantages to this functional organization of the space. If you have visitors, you don't have to rearrange anything in your house. You can simply give them the guest bedroom. If you decide to bring a friend for sleepover, you may simply share your bedroom with that friend without disturbing the other members of your family. Let us apply this analogy to program development.

In the previous chapter (see Section 6.2), we defined the address space of a process as the space occupied by the memory footprint of the program. In the previous section, we stressed the need to preserve the contiguous view of the memory footprint of a user's program. We will go a little further and underscore the importance of having the address space always start at 0 to some maximum value, so that it is convenient from the point of view of code generation by the compiler. Virtual memory helps with that view.

Let us investigate if one address space is enough for a program. The analogy of the space planning in the house is useful here. We logically partitioned the space in the house into the different rooms based on their intended use. These spaces are independent of each other and well protected from one another (doors, locks, etc.). This ensures that no one can invade the privacy of the other occupants without prior warning. At some level, constructing a program is similar to designing a house. Although we end up with a single program (`a.out` as it is called in Unix terminology), the source code has a logical structure. There are distinct data structures and procedures organized to provide specific functionality. If it is a group project, you may even develop your program as a team with different team members contributing different functionalities of the total program. One can visualize the number of software engineers who would have participated in the development of a complex program such as MS Word at Microsoft. Therefore, the availability of multiple address spaces would help better organize the program logically. Especially, with object-oriented programming the utility of having multiple address spaces cannot be over-emphasized.

Let us drill a little deeper and understand how multiple address spaces would help the developer. Even for the basic memory footprint (please see Figure 6.1 in Chapter 6), we could have each of the distinct portions of the memory footprint, namely, code, global data, heap, and stack in distinct address spaces. From a software engineering perspective, this organization gives us the ability to associate properties with these address spaces (for example, the code section is read only, etc.). Further, the ability to associate such properties with individual address spaces will be a great boon for program debugging.

Use of multiple address spaces becomes even more compelling in large-scale program development. Let us say, we are writing an application program for video-based surveillance. It may have several components as shown in Figure 7.14. One can see the similarity between a program composed of multiple components each in its dedicated address space and the floor plan of Figure 7.13.

This is a complex enough application that there may be several of us working on this project together. The team members using well-defined interfaces may independently develop each of the boxes shown in Figure 7.14. It would greatly help both the development as well as debugging to have each of these components in separate address spaces with the appropriate levels of protection and sharing among the components (similar to having doors and locks in the house floor plan). Further, it would be easy to maintain such an application. You don't move into a hotel if you wanted to retile just your kitchen. Similarly, you could rewrite specific functional modules of this application without affecting others.

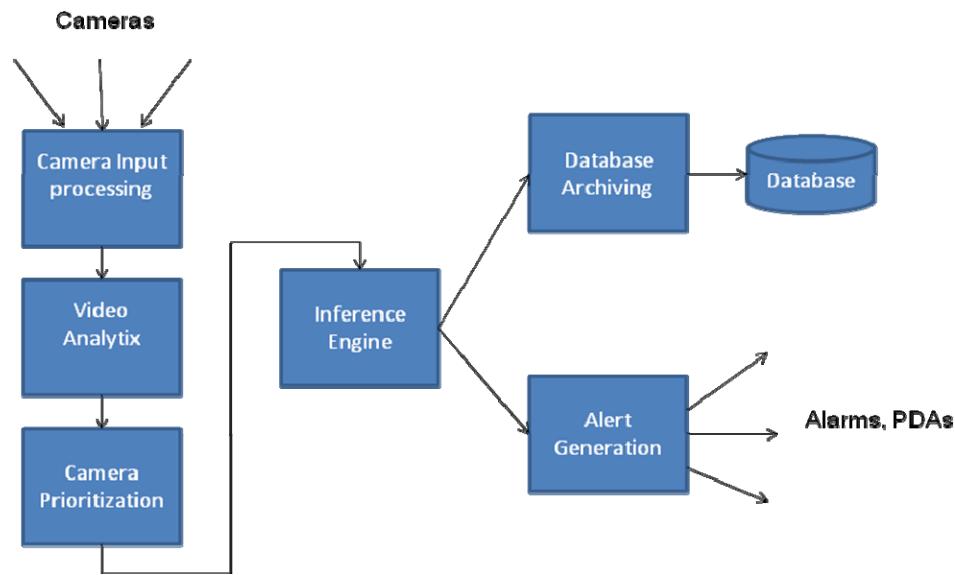


Figure 7.14: An Example Application

Segmentation is a technique for meeting the above vision. As is always the case with any system level mechanism, this technique is also a partnership between the operating system and the architecture.

The user's view of memory is not a single linear address space; but it is composed of several distinct address spaces. Each such address space is called a *segment*. A segment has two attributes:

- Unique segment number
- Size of the segment

Each segment starts at address 0 and goes up to (Segment size – 1). CPU generates addresses that have two parts as shown below in Figure 7.15.

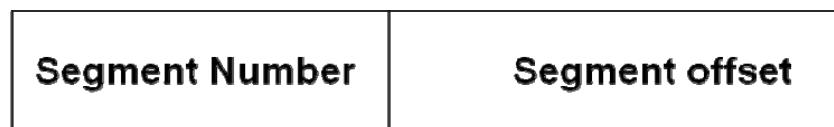


Figure 7.15: A segmented address

As with paging, the broker that sits in the middle between the CPU and memory converts this address into a physical address by looking up the segment table (Figure 7.16). As in the case of paging, it is the operating system that sets up the segment table for the currently running process.

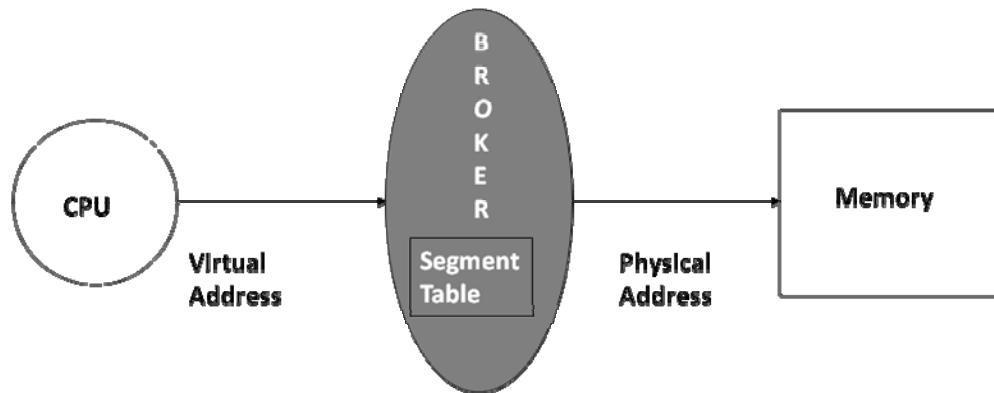


Figure 7.16: Segment Table

Now, you are probably wondering that but for the name change from “Page” to “Segment” there appears to be not much difference between paging and segmentation. Before, we delve into the differences between the two, let us return to the example application in Figure 7.14. With segmentation, we could arrange the application as shown in Figure 7.17. Notice that each functional component is in its own segment, and the segments are individually sized depending on the functionality of that component. Figure 7.18 shows these segments housed in the physical memory.

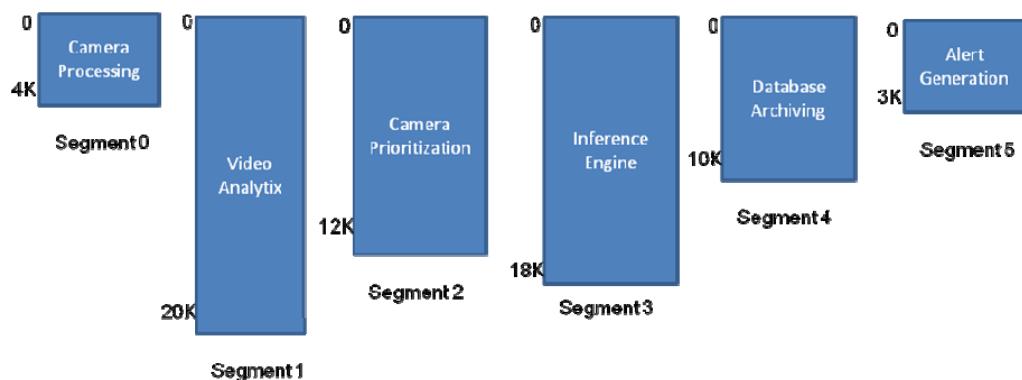


Figure 7.17: Example Application Organized into Segments

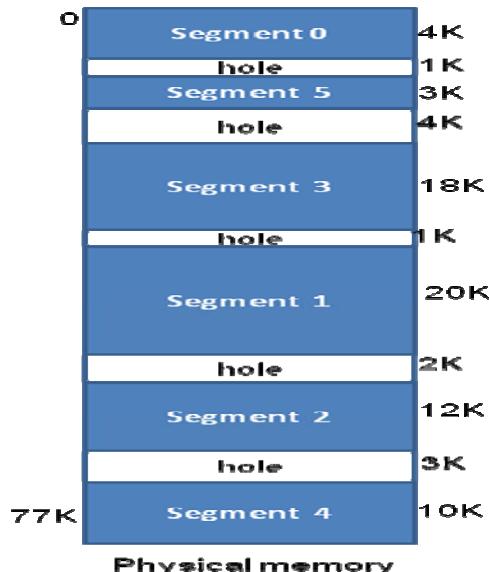


Figure 7.18: Segments of example application in Figure 7.17 mapped into physical memory

Example:

A program has 10KB code space and 3KB global data space. It needs 5KB of heap space and 3KB of stack space. The compiler allocates individual segments for each of the above components of the program. The allocation of space in physical memory is as follows:

Code start address 1000
 Global data start address 14000
 Heap space start address 20000
 Stack space start address 30000

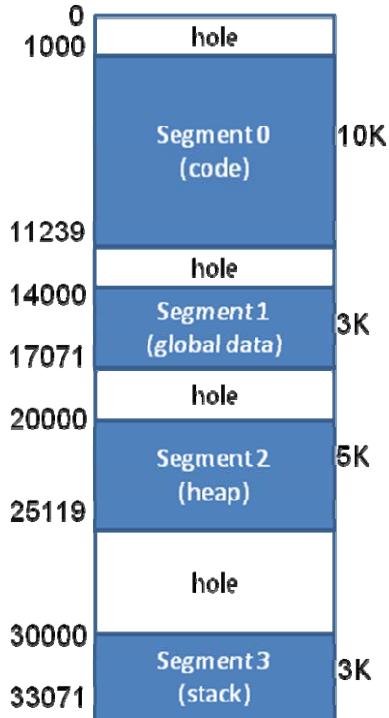
(a) Show the segment table for this program.

Answer:

Segment Number	Start address	Size
0	1000	10KB
1	14000	3KB
2	20000	5KB
3	30000	3KB

(b) Assuming byte-addressed memory, show the memory layout pictorially.

Answer:



(c) What is the physical memory address corresponding to the virtual address:

0	299
---	-----

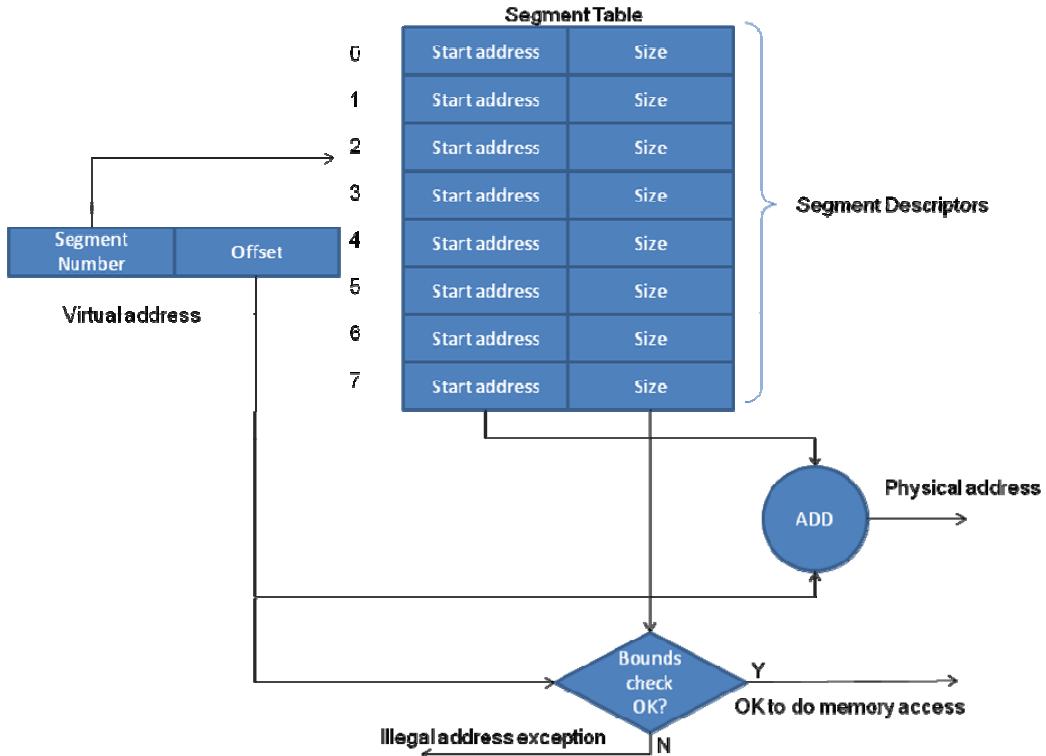
Answer:

- 1) The offset 299 is within the size of segment 0 (10K).
- 2) The physical memory address
$$\begin{aligned} &= \text{Start address of segment 0} + \text{offset} \\ &= 1000 + 299 \\ &= \mathbf{1299} \end{aligned}$$

7.5.1 Hardware for Segmentation

The hardware needed for supporting segmentation is fairly simple. The segment table is a data structure similar to the page table. Each entry in this table called a *segment descriptor*. The segment descriptor gives the start address for a segment and the size of the segment. Each process has its own segment table allocated by the operating system at the time of process creation. Similar to paging, this scheme also requires a special register in the CPU called *Segment Table Base Register (STBR)*. The hardware uses this

register and the segment table to do address translation during the execution of the process (see Figure 7.19). The hardware performs a bounds check first to ensure that the



offset provided is within the size limit of the segment before allowing the memory access to proceed.

Figure 7.19: Address translation with segmentation

Segmentation should remind the reader of the memory allocation scheme from Section 7.3.2, variable-size partition. It suffers from the same problem as in variable size partition, namely, external fragmentation. This can be seen in Figure 7.18.

7.6 Paging versus Segmentation

Now, we are ready to understand the difference between paging and segmentation. Both are techniques for implementing virtual memory but differ greatly in details. We summarize the similarities and differences between the two approaches in Table 7.1.

Attribute	Paging	Segmentation
User shielded from size limitation of physical memory	Yes	Yes
Relationship to physical memory	Physical memory may be less than or greater than virtual memory	Physical memory may be less than or greater than virtual memory
Address spaces per process	One	Several
Visibility to the user	User unaware of paging; user is given an illusion of a single linear address space	User aware of multiple address spaces each starting at address 0
Software engineering	No obvious benefit	Allows organization of the program components into individual segments at user discretion; enables modular design; increases maintainability
Program debugging	No obvious benefit	Aided by the modular design
Sharing and protection	User has no direct control; operating system can facilitate sharing and protection of pages across address spaces but this has no meaning from the user's perspective	User has direct control of orchestrating the sharing and protection of individual segments; especially useful for object-oriented programming and development of large software
Size of page/segment	Fixed by the architecture	Variable chosen by the user for each individual segment
Internal fragmentation	Internal fragmentation possible for the portion of a page that is not used by the address space	None
External fragmentation	None	External fragmentation possible since the variable sized segments have to be allocated in the available physical memory thus creating holes (see Figure 7.18)

Table 7.1: Comparison of Paging and Segmentation

When you look at the table able, you would conclude that segmentation has a lot going for it. Therefore, it is tempting to conclude that architectures should be using segmentation as the vehicle for implementing virtual memory. Unfortunately, the last row of the table is the real killer, namely, external fragmentation. There are other considerations as well. For example, with paging the virtual address that the CPU generates occupies one memory word. However, with segmentation, it may be necessary to have two memory words to specify a virtual address. This is because, we may want a segment to be as big as the total available address space to allow for maximum flexibility. This would mean we would need one memory word for identifying the segment number and the second to identify the offset within the segment. Another serious system level consideration has to do with balancing the system as a whole. Let us elaborate what we mean by this statement. It turns out that the hunger for memory of applications and system software keeps growing incessantly. The increase in memory footprint of desktop publishing applications and web browsers from release to release is an attestation to this growing appetite. The reason of course is the desire to offer increased functionality to the end user. The reality is that we will never be able to afford the physical memory to satisfy our appetite for memory. Therefore, necessarily virtual memory will be far larger than physical memory. Pages or segments have to be brought in “on demand” from the hard drive into the physical memory. Virtual memory extends the memory system from the physical memory to the disk. Thus, the transfer of data from the disk to the memory system on demand has to be efficient for the whole system to work efficiently. This is what we mean by balancing the system as a whole. Since the page-size is a system attribute it is easier to optimize the system as a whole with paging. Since the user has control over defining the size of segments it is more difficult to optimize the system as a whole.

For these reasons, true segmentation as described in this section is not a viable way to implement virtual memory. One way to solve the external fragmentation problem is as we described in Section 7.3.3, by using memory compaction. However, we observed the difficulties in practically implementing compaction in that section as well. Therefore, a better approach is to have a combined technique, namely, *paged-segmentation*. The user is presented with a segmented view as described in this section. Under the covers, the operating system and the hardware use paging as described in the previous section to eliminate the ill effects of external fragmentation.

Description of such paged-segmentation techniques is beyond the scope of this textbook. We do present a historical perspective of paging and segmentation as well as a commercial example of paged-segmentation approach using Intel Pentium architecture as a case study in Section 7.8.

7.6.1 Interpreting the CPU generated address

The processor generates a simple linear address to address the memory.

CPU generated address:



The number of bits in the CPU generated address depends on the memory addressing capability of the processor (usually tied to the word-width of the processor and the smallest granularity of access to memory operands). For example, for a 64-bit processor with byte-addressed memory $n_{CPU} = 64$.

The number of bits in the physical address depends on the actual size of the physical memory.



With byte-addressed memory,

$$n_{phy} = \log_2(\text{size of physical memory in bytes}) \quad (3)$$

For example, for a physical memory size of 1 Gbyte, $n_{phy} = 30$ bits.

The exact interpretation of the CPU generated linear address as a virtual address depends on the memory system architecture (i.e., paging versus segmentation). Correspondingly, the computation of the physical address also changes. Table 7.2 summarizes the salient equations pertaining to paged and segmented memory systems.

Memory System	Virtual Address Computation	Physical Address Computation	Size of Tables
Segmentation	<p>Segment Number Segment Offset</p> $n_{off} = \log_2(\text{segment-size})$ $n_{seg} = n_{CPU} - n_{off}$	<p>Segment Start address = Segment-Table [Segment-Number]</p> <p>Physical address = Segment Start Address + Segment Offset</p>	<p>Segment table size = $2^{n_{seg}}$ entries</p>
Paging	<p>VPN Page Offset</p> $n_{off} = \log_2(\text{page-size})$ $n_{VPN} = n_{CPU} - n_{off}$	<p>PFN = Page-Table[VPN]</p> <p>Physical address:</p> <p>PFN Page Offset</p> $n_{off} = \log_2(\text{page-size})$ $n_{PFN} = n_{phy} - n_{off}$	<p>Page table size = $2^{n_{VPN}}$ entries</p>

Table 7.2: Address Computations in Paged and Segmented Memory Systems

7.7 Summary

The importance of the memory system cannot be over-emphasized. The performance of the system as a whole crucially depends on the efficiency of the memory system. The

interplay between the hardware and the software of a memory system makes the study of memory systems fascinating. Thus far, we have reviewed several different memory management schemes and the hardware requirement for those schemes. In the beginning of the chapter, we identified four criteria for memory management: improved resource utilization, independence and protection of process' memory spaces, liberation from memory resource limitation, and sharing of memory by concurrent processes. These are all important criteria from the point of view of the efficiency of the operating system for managing memory, which is a precious resource. From the discussion on segmentation, we will add another equally important criterion for memory management, namely, *facilitating good software engineering practices*. This criterion determines whether the memory management scheme, in addition to meeting the system level criteria also helps in the development of software that is flexible, easy to maintain, and evolve. Let us summarize these schemes with respect to these memory management criteria. Table 7.3 gives such a qualitative comparison of these memory management schemes.

Memory Management Criterion	User/Kernel Separation	Fixed Partition	Variable-sized Partition	Paged Virtual Memory	Segmented Virtual Memory	Paged-segmented Virtual Memory
Improved resource utilization	No	Internal fragmentation bounded by partition size; External fragmentation	External fragmentation	Internal fragmentation bounded by page size	External fragmentation	Internal fragmentation bounded by page size
Independence and protection	No	Yes	Yes	Yes	Yes	Yes
Liberation from resource limitation	No	No	No	Yes	Yes	Yes
Sharing by concurrent processes	No	No	No	Yes	Yes	Yes
Facilitates good software engineering practice	No	No	No	No	Yes	Yes

Table 7.3: Qualitative Comparison of memory management schemes

Only virtual memory with paged-segmentation meets all the criteria. It is instructive to review which one of these schemes are relevant to this day with respect to the state-of-the-art in memory management. Table 7.4 summarizes the memory management

schemes covered in this chapter along with the required hardware support, and their applicability to modern operating systems.

Scheme	Hardware Support	Still in Use?
User/Kernel Separation	Fence register	No
Fixed Partition	Bounds registers	Not in any production operating system
Variable-sized Partition	Base and limit registers	Not in any production operating system
Paged Virtual Memory	Page table and page table base register	Yes, in most modern operating system
Segmented Virtual Memory	Segment table, and segment table base register	Segmentation in this pure form not supported in any commercially popular processors
Paged-segmented Virtual Memory	Combination of the hardware for paging and segmentation	Yes, most modern operating systems based on Intel x86 use this scheme ⁷

Table 7.4: Summary of Memory management schemes

7.8 Historical Perspective

Circa 1965, IBM introduced System/360 series of mainframes. The architecture provided base and limit registers thus paving the way for memory management that supports dynamic relocation. Any general-purpose register could be used as the base register. The compiler would designate a specific register as the base register so that any program written in high-level language could be dynamically relocated by the operating system OS/360. However, there is a slight problem. Programmers could find out which register was being used as the base register and could use this knowledge to “stash” away the base register value for assembly code that they want to insert into the high-level language program⁸. The upshot of doing this is that the program can no longer be relocated once it starts executing, since programmers may have hard coded addresses in the program. You may wonder why they would want to do such a thing. Well, it is not unusual for programmers who are after getting every ounce of performance from a system to lace in assembly code into a high-level language program. Some of us are guilty of that even in this day and age! Due to these reasons, dynamic relocation never really worked as well as IBM would have wanted it to work in OS/360. The main reason was the fact that the address shifting in the architecture used a general-purpose register that was visible to the programmer.

Circa 1970, IBM introduced virtual memory in their System/370 series of mainframes⁹. The fundamental way in which System/370 differed from System/360 was the

⁷ It should be noted that Intel’s segmentation is quite different from the pure form of segmentation presented in this chapter. Please Section 7.8.2 for a discussion of Intel’s paged-segmentation scheme.

⁸ Personal communication with James R. Goodman, University of Wisconsin-Madison.

⁹ For an authoritative paper that describes System/360 and System/370 Architectures, please refer to:

architectural support for dynamic address translation, which eliminated the above problem with System/360. System/370 represents IBM's first offering of the virtual memory concept. Successive models of the System/370 series of machines refined the virtual memory model with expanded addressing capabilities. The architecture used paged virtual memory.

In this context, it is worth noting a difference in the use of the terminologies “static” and “dynamic”, depending on operating system or architecture orientation. Earlier (see Section 7.2), we defined what we mean by static and dynamic relocation from the operating system standpoint. With this definition, one would say a program is dynamically relocatable if the hardware provides support for changing the virtual to physical mapping of addresses at execution time. By this definition, the base and limit register method of IBM 360 supports dynamic relocation.

Architects look at a much finer grain of individual memory accesses during program execution and use the term address “translation”. An architecture supports dynamic address translation if the mapping of virtual to physical can be altered at any time during the execution of the program. By this definition, the base and limit register method in IBM 360 supports only static address translation, while paging supports dynamic address translation. Operating system definition of “static vs. dynamic” relocation is at the level of the whole program while the architecture definition of “static vs. dynamic” address translation is at the level of individual memory accesses.

Even prior to IBM's entry into the virtual memory scene, Burroughs Corporation¹⁰ introduced segmented virtual memory in their B5000 line of machines. Another company, General Electric, in partnership with the MULTICS project at MIT introduced paged-segmentation in their GE 600 line of machines¹¹ in the mid 60's. IBM quickly sealed the mainframe war of the 60's and 70's with the introduction of VM/370 operating system to support virtual memory and the continuous refinement of paged virtual memory in their System/370 line of machines. This evolution has continued to this day and one could see the connection to these early machines even in the IBM z series of mainframes¹² to support enterprise level applications.

7.8.1 MULTICS

Some academic computing projects have a profound impact on the evolution of the field for a very long time. MULTICS project at MIT was one such. The project had its hey days in the 60's and one could very easily see that much of computer systems as we know it (Unix, Linux, Paging, Segmentation, Security, Protection, etc.) had their birth in this project. In some sense, the operating systems concepts introduced in the MULTICS project were way ahead of their time and the processor architectures of that time were not geared to support the advanced concepts of memory protection advocated by MULTICS.

¹⁰ <http://www.research.ibm.com/journal/rd/255/ibmrd2505D.pdf>

¹¹ Source: http://en.wikipedia.org/wiki/Burroughs_large_systems

¹² Source: http://en.wikipedia.org/wiki/GE_645

¹² Source: <http://www-03.ibm.com/systems/z/>

The MULTICS project is an excellent example of the basic tenet of this textbook, namely, the connectedness of system software and machine architecture.

MULTICS introduced the concept of paged segmentation¹³. Figure 7.20 depicts the scheme implemented in MULTICS.

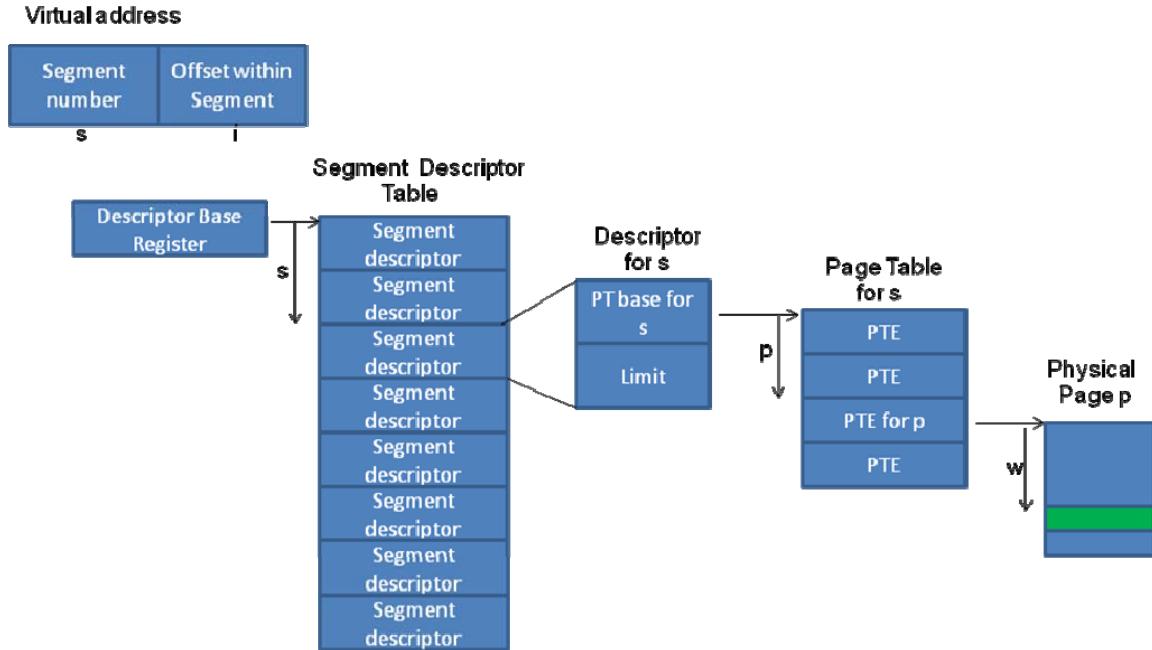


Figure 7.20: Address Translation in MULTICS

The 36-bit virtual address generated by the CPU consists of two parts: an 18-bit segment number (*s*) and an 18-bit offset (*i*) within that segment. Each segment can be arbitrary in size bounded by the maximum segment size of $2^{18} - 1$. To avoid external fragmentation, a segment consists of pages (in MULTICS the page-size is 1024 words, with 36-bits per word). Each segment has its own page table. Dynamic address translation in MULTICS is a two-step process:

- Locate the segment descriptor corresponding to the segment number: Hardware does this lookup by adding the segment number to the base address of the segment table contained in a CPU register called Descriptor Base Register.
- The segment descriptor contains the base address for the page table for that segment. Using the segment offset in the virtual address and the page-size, hardware computes the specific page table entry corresponding to the virtual address. The physical address is obtained by concatenating the physical page number with the page offset (which is nothing but segment offset *mod* page-size).

If *s* is the segment number and *i* is the offset within the segment specified in the virtual address then, the specific memory word we are looking for is at page offset *w* from the beginning of the *p*th page of the segment, where:

¹³ See the original MULTICS paper: <http://www.multicians.org/multics-vm.html>

$$w = i \bmod 1024$$

$$p = (i - w) / 1024$$

Figure 7.20 shows this address translation.

7.8.2 Intel's Memory Architecture

Intel Pentium line of processors also uses paged-segmentation. However, its organization is a lot more involved than the straightforward scheme of MULTICS. One of the realities of academic projects versus an industrial product is the fact that the latter has to worry about backward compatibility with its line of processors. Backward compatibility essentially means that a new processor that is a successor to an earlier one has to be able to run, unchanged, the code that used to run on its predecessors. This is a tall order and constrains the designers of a new processor significantly. The current memory architecture of Intel Pentium evolved from the early editions of Intel's x86 architectures such as the 80286; consequently, it has vestiges of the segmentation scheme found in such older processors coupled with the aesthetic need for providing large numbers of virtual address spaces for software development.

We have intentionally simplified the discussion in this section. As a first order of approximation, a virtual address is a segment selector plus an offset (see Figure 7.21). Intel's architecture divides the total segment space into two halves: *system* and *user*. The system segments are common to all processes, while the user segments are unique to each process. As you may have guessed, the system segments would be used by the operating system since it is common to all the user processes. Correspondingly, there is one descriptor table common to all processes called the *Global Descriptor Table (GDT)* and one unique to each process called the *Local Descriptor Table (LDT)*. The segment selector in Intel is similar to the segment number in MULTICS with one difference. A bit in the segment selector identifies whether the segment being named by the virtual address is a system or a user segment.

As in MULTICS, the segment descriptor for the selected segment contains the details for translating the offset specified in the virtual address to a physical address. The difference is that there is a choice of using simple segmentation without any paging (to be compatible with earlier line of processors) or use paged-segmentation. In the former case, the address translation will proceed as we described in Section 7.5 (i.e., without the need for any page table). This is shown in Figure 7.21 (the choice of GDT versus LDT for the translation is determined by the U/S bit in the selector). The effective address computed is the physical address. In the case of paged-segmentation, the base address contained in the descriptor is the address of the page table base corresponding to this segment number. The rest of the address translation will proceed as we described in Section 7.8.1 for MULTICS, going through the page table for the selected segment (see Figure 7.20). A control bit in a global control register decides the choice of pure segmentation versus paged-segmentation.

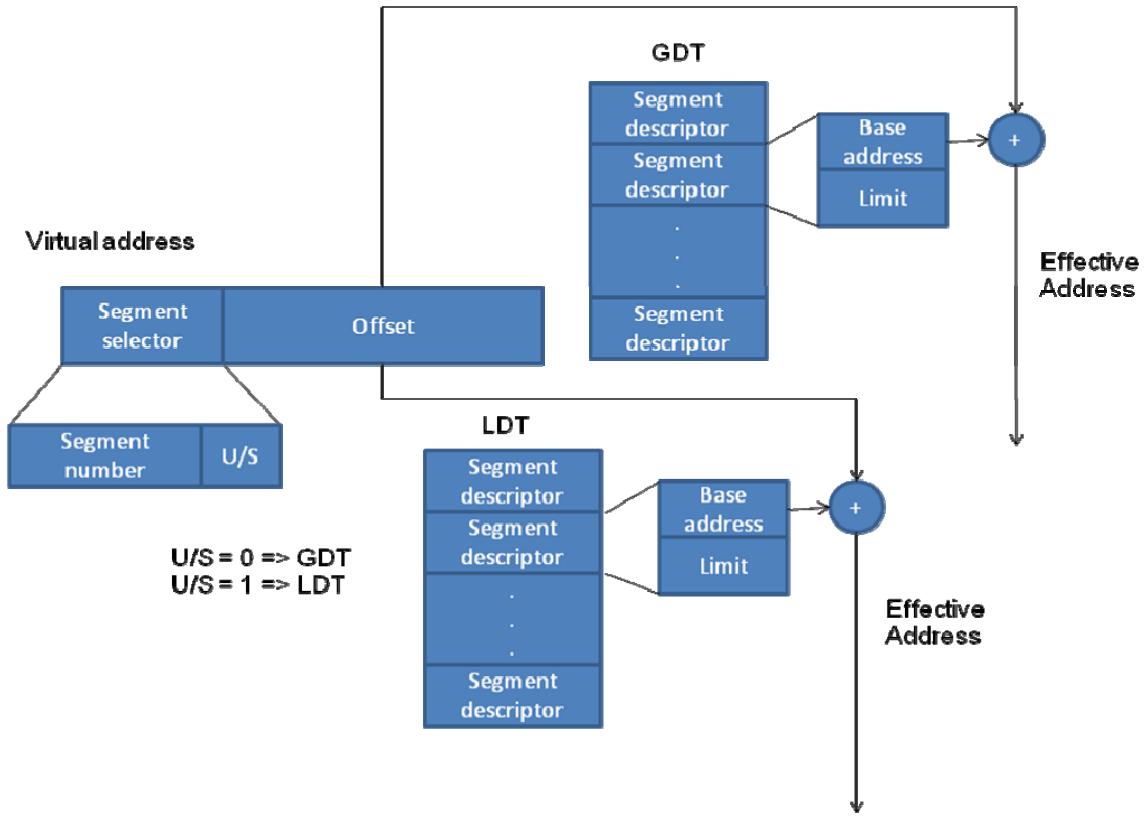


Figure 7.21: Address Translation in Intel Pentium with pure Segmentation

Please refer to Intel's System Programming Guide¹⁴ if you would like to learn more about Intel's virtual memory architecture.

7.9 Review Questions

1. What are the main goals of memory management?
2. Argue for or against the statement: Given that memory is cheap and we can have lots of it, there is no need for memory management anymore.
3. Compare and contrast internal and external fragmentation.
4. A memory manager allocates memory in fixed size chunks of 2048 bytes. The current allocation is as follows:

P1 1200 bytes
 P2 2047 bytes
 P3 1300 bytes
 P4 1 byte

¹⁴ Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide:
<http://www.intel.com/design/processor/manuals/253668.pdf>

What is the total amount memory wasted with the above allocation due to internal fragmentation?

5. Answer True/False with justification: Memory compaction is usually used with fixed size partition memory allocation scheme.
6. Answer True/False with justification: There is no special advantage for the base and limit register solution over the bounds register solution for memory management.
7. Assume an architecture that uses base and limit register for memory management. The memory manager uses variable sized partition allocation. The current memory allocation is as shown below.

Allocation table
Memory

Start address	Size	Process
0	8K	FREE
8K	3K	P3
11K	2K	FREE
13K	2K	P4

The diagram shows a vertical stack of four memory partitions. From top to bottom: a blue 8K partition, a grey 3K partition labeled 'P3', a blue 2K partition, and a grey 2K partition labeled 'P4'.

There is a new memory request for 9 Kbytes. Can this allocation be satisfied? If not why not? What is the amount of external fragmentation represented by the above figure?

8. What is the relationship between page size and frame size in a paged memory system?
9. In terms of hardware resources needed (number of new processor registers and additional circuitry to generate the physical memory address given a CPU generated address) compare base and limit register solution with a paged virtual memory solution.
10. How is a paged virtual memory system able to eliminate external fragmentation?

11. Derive a formula for the maximum internal fragmentation with a page size of p in a paged virtual memory system.
12. Consider a system where the virtual addresses are 20 bits and the page size is 1 KB. How many entries are in the page table?
13. Consider a system where the physical addresses are 24 bits and the page size is 8 KB. What is the maximum number of physical frames possible?
14. Distinguish between paged virtual memory and segmented virtual memory.
15. Given the following segment table:

Segment Number	Start address	Size
0	3000	3KB
1	15000	3KB
2	25000	5KB
3	40000	8KB

What is the physical address that corresponds to the virtual address shown below?

1	399
---	-----

16. Compare the hardware resources needed (number of new processor registers and additional circuitry to generate the physical memory address given a CPU generated address) for paged and segmented virtual memory systems.

Chapter 8 Details of Page-based Memory Management (Revision number 20)

In this chapter, we will focus on page-based memory management. This is fundamental to pretty much all processors and operating systems that support virtual memory. As we mentioned already, even processors that support segmentation (such as Intel Pentium) use paging to combat external fragmentation.

8.1 Demand Paging

As we mentioned in Chapter 7, the memory manager sets up the page table for a process on program startup. Let us first understand what fraction of the entire memory footprint of the program the memory manager should allocate at program startup. Production programs contain the functional or algorithmic logic part of the application as well as the non-functional parts that deal with erroneous situations that may occasionally arise during program execution. Thus, it is a fair expectation that any well-behaved program will need only a significantly smaller portion of its entire memory footprint for proper execution. Therefore, it is prudent for the memory manager not to load the entire program into memory on startup. This requires some careful thinking, and understanding of what it means to execute a program that may not be completely in memory. The basic idea is to load parts of the program that are not in memory *on demand*. This technique, referred to as *demand paging* results in better memory utilization.

Let us first understand what happens in hardware and in software to enable demand paging.

8.1.1 Hardware for demand paging

In Chapter 7 (please see Section 7.4.2), we mentioned that the hardware fetches the PFN from the page table as part of address translation. However, with demand paging the page may not be in memory yet. Therefore, we need additional information in the page table to know if the page is in memory. We add a *valid* bit to each page table entry. If the bit is 1 then the PFN field in this entry is valid; if not, it is invalid implying that the page is not in memory. Figure 8.1 shows the PTE to support demand paging. The role of the hardware is to recognize that a PTE is invalid and help the operating system take corrective action, namely, load the missing page into memory. This condition (a PTE being invalid) is an unintended program interruption since it is no fault of the original program. The operating system deliberately decided not to load this part of the program to conserve memory resources. This kind of program interruption manifests as a *page fault exception or trap*.

Valid	PFN
--------------	------------

Figure 8.1: Page Table Entry

The operating system handles this fault by bringing in the missing page from the disk. Once the page is in memory, the program is ready to resume execution where it left off. Therefore, to support demand paging the processor has to be capable of restarting an instruction whose execution has been suspended in the middle due to a page fault.

Figure 8.2 shows the processor pipeline. The IF and MEM stages are susceptible to a page fault since they involve memory accesses.

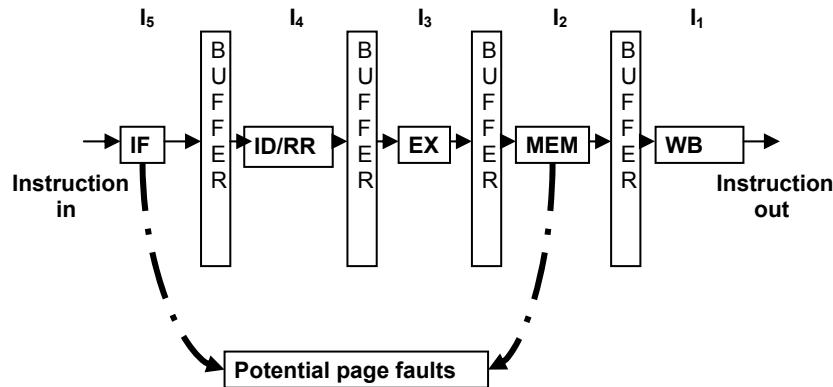


Figure 8.2: Potential page faults in a processor pipeline

Hardware for instruction re-start: Let us first understand what should happen in the hardware. Consider that instruction I_2 in the MEM has a page fault. There are several instructions in partial execution in the pipeline. Before the processor goes to the INT state to handle the interrupt (refer to Chapter 4 for details on how interrupts are handled in hardware in the INT state), it has to take care of the instructions in partial execution already in the pipeline. In Chapter 5, we briefly mentioned the action that a pipelined processor may take to deal with interrupts. The page fault exception that I_2 experiences in the MEM stage is no different. The processor will let I_1 to complete (draining the pipe) and squash (flushing the pipeline) the partial execution of instructions I_3-I_5 before entering the INT state. The INT state needs to save the PC value corresponding to instruction I_2 for re-starting the instruction after servicing the page fault. Note that there is no harm in flushing instructions I_3-I_5 since they have not modified the permanent state of the program (in processor registers and memory) so far. One interesting and important side effect manifesting from page faults (as well as any type of exceptions): The pipeline registers (shown as buffers in Figure 8.2) contain the PC value of the instruction in the event there is an exception (arithmetic in the EX stage or page fault in the MEM stage) while executing this instruction. We discussed the hardware ramifications for dealing with traps and exceptions in a pipelined processor in Chapter 5.

8.1.2 Page fault handler

The page fault handler is just like any other interrupt handler that we have discussed in earlier chapters. We know the basic steps that any handler has to take (saving/restoring state, etc.) that we have seen already in Chapter 4. Here, we will worry about the actual work that the handler does to service the page fault:

1. Find a free page frame

2. Load the faulting virtual page from the disk into the free page frame
3. Update the page table for the faulting process
4. Place the PCB of the process back in the ready queue of the scheduler

We will explore the details of these steps in the next subsection.

8.1.3 Data structures for Demand-paged Memory Management

Now, we will investigate the data structures and algorithms needed for demand paging. First, let us look at the data structures. We already mentioned that the page table is a per-process data structure maintained by the memory manager. In addition to the page tables, the memory manager uses the following data structures for servicing page faults:

1. **Free-list of page frames:** This is a data structure that contains information about the currently unused page frames that a memory manager uses to service a page fault. The free-list does not contain the page frames themselves; each node of the free-list simply contains the page frame number. For example (referring to Figure 8.3), page frames 52, 20, 200, ..., 8 are currently unused. Therefore, the memory manager could use any page frame from this list to satisfy a page fault. To start with when a machine boots up, since there are no user processes, and the free-list contains all the page frames of the user space. The free-list shrinks and grows as the memory manager allocates and releases memory to satisfy the page faults of processes.

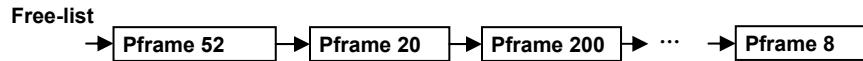


Figure 8.3: Free-list of page frames

2. **Frame table (FT):** This is a data structure that contains the reverse mapping. Given a frame number, it gives the Process ID (PID) and the virtual page number that currently occupies this page frame (Figure 8.4). For example, page frame 1 is currently unallocated while the virtual page 0 of Process 4 currently occupies page frame 6. It will become clear shortly how the memory manager uses this data structure.

0	<P2, 20>
1	free
2	<P5, 15>
3	<P1, 32>
4	free
5	<P3, 0>
6	<P4, 0>
7	free

Figure 8.4: Frame Table

3. **Disk map (DM):** This data structure maps the process virtual space to locations on the disk that contain the contents of the pages (Figure 8.5). This is the disk analog of the page table. There is one such data structure for each process.

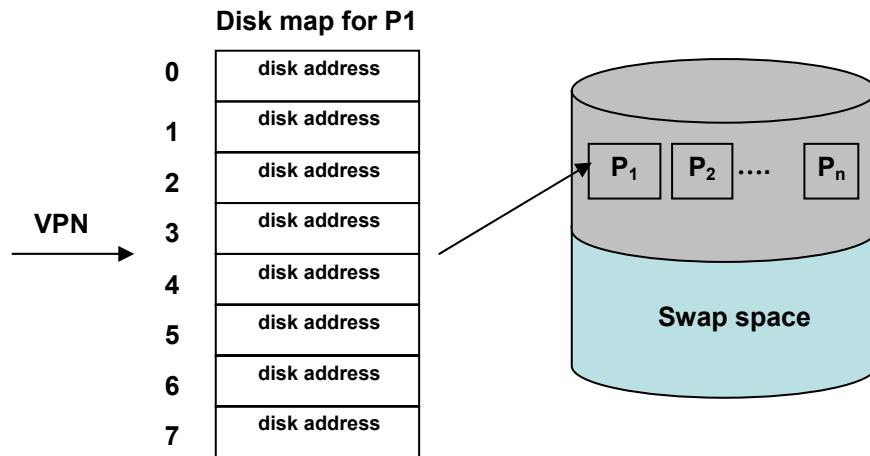


Figure 8.5: Disk map for Process P1

For clarity of discussion, we have presented each of the above data structures as distinct from one another. Quite often, the memory manager may coalesce some of these data structures for efficiency; or have common data structures with multiple views into them to simulate the behavior of the above functionalities. This will become apparent when we discuss some of the page replacement policies (see Section 8.3).

8.1.4 Anatomy of a Page Fault

Let us re-visit the work done by the page fault handler upon a page fault using the data structures above.

1. **Find a free page frame:** The page fault handler (which is part of the memory manager) looks up the *free-list* of page frames. If the list is empty, we have a problem. This means that all the physical frames are in use. However, for the faulting process to continue execution, the memory manager has to bring the faulting page from the disk into physical memory. This implies that we have to make room in the physical memory to bring in the faulting page. Therefore, the manager selects some physical frame as a *victim* to make room for the faulting page. We will describe the policy for selecting a victim in Section 8.3.
2. **Pick the victim page:** Upon selecting a victim page frame, the manager determines the victim process that currently owns it. The *frame table* comes in handy to make this determination. Here, we need to make a distinction between a *clean* page and a *dirty* page. A clean page is one that has not been modified by the program from the time it was brought into memory from the disk. Therefore, the disk copy and the memory copy of a clean page are identical. On the other hand, a dirty page is one that has been modified by the program since bringing from the disk. If the page is clean, then all that the manager needs to do is to set the PTE corresponding to the page as *invalid*, i.e., the contents of this page need not be saved. However, if the page is dirty, then the manager writes the page back to the disk (usually referred to as *flushing* to the disk), determining the disk location from the *disk map* for the victim process.
3. **Load the faulting page:** Using the *disk map* for the faulting process, the manager reads in the page from the disk into the selected page frame.
4. **Update the page table for the faulting process and the frame table:** The manager sets the mapping for the PTE of the faulting process to point to the selected page frame, and makes the entry *valid*. It also updates the *frame table* to indicate the change in the mapping to the selected page frame.
5. **Restart faulting process:** The faulting process is ready to be re-started. The manager places the PCB of the faulting process in the scheduler's ready queue.

Example 5:

Let us say process P1 is currently executing, and experiences a page fault at VPN = 20. The *free-list* is empty. The manager selects page frame PFN = 52 as the victim. This frame currently houses VPN = 33 of process P4. Figures 8.6a and 8.6b show the *before-after* pictures of the *page tables* for P1 and P2 and the *frame table* as a result of handling the page fault.

Answer:

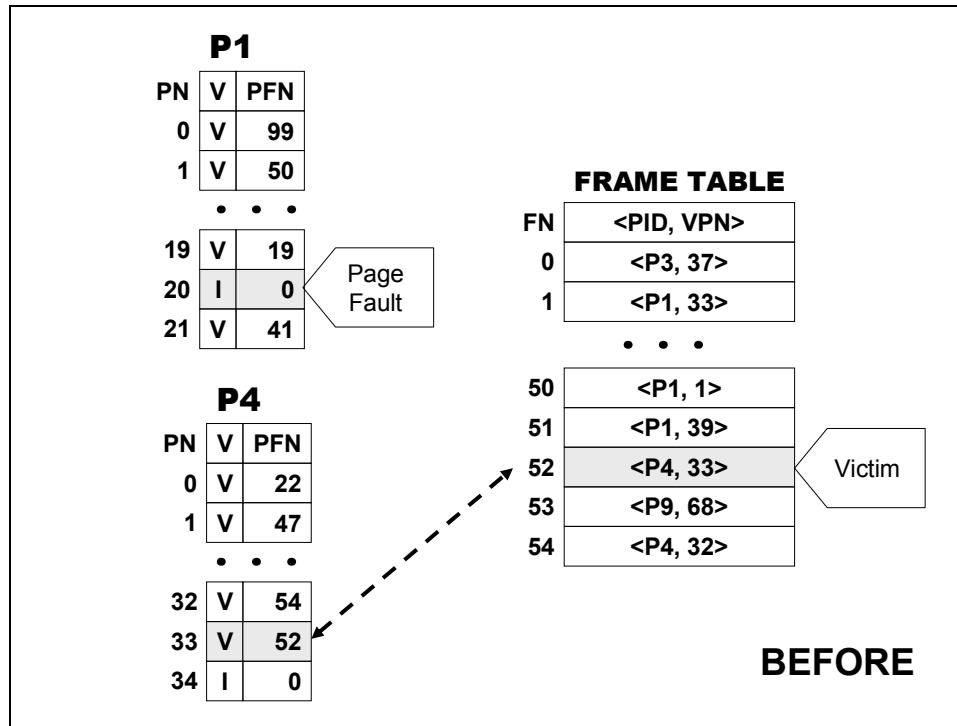


Figure 8.6a: Process P1 is executing and has a page fault on VPN 20

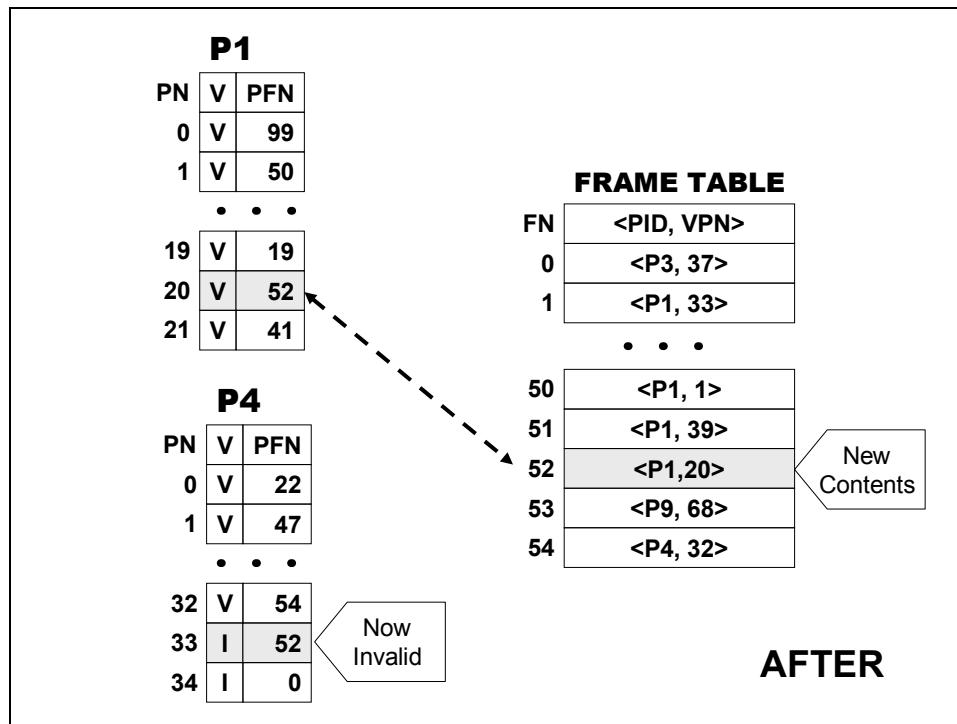


Figure 8.6b: Page fault service for Process P1 complete

Example 6:

Given the before picture of the page manager's data structure as shown below, show the contents of the data structures after P1's page fault at VPN = 2 is serviced. The victim page frame chosen by the page replacement algorithm is PFN = 84. Note that only relevant entries of the Frame Table are shown in the Figures.

BEFORE THE PAGE FAULT

	PFN	V
VPN = 0	50	V
VPN = 1	52	V
VPN = 2	--	I
VPN = 3	60	V

	PFN	V
P1's PT	70	V
80	V	
85	V	
84	V	

	Frame Table
0
50
52	<P1, 0>
60	<P1, 1>
70	<P1, 3>
80	<P2, 0>
84	<P2, 1>
85	<P2, 3>
	<P2, 2>

P2's PT

SHOW THE CONTENTS OF THE DATA STRUCTURES AFTER THE PAGE FAULT FOR P1 at VPN = 2 .

Answer :

	PFN	V
VPN = 0	50	V
VPN = 1	52	V
VPN = 2	84	V
VPN = 3	60	V

	PFN	V
P1's PT	70	V
80	V	
85	V	
--	I	

	Frame Table
0
50
52	<P1, 0>
60	<P1, 1>
70	<P1, 3>
80	<P2, 0>
84	<P1, 2>
85	<P2, 2>

P2's PT

Remember that the page fault handler is also a piece of code just like any other user process. However, we cannot allow the page fault handler itself to page fault. The operating system ensures that certain parts of the operating system such as the page fault handler are always memory resident (i.e., never paged out of physical memory).

Example 7:

Five of the following seven operations take place upon a page fault when there is no free frame in memory. Put the five correct operations in the right temporal order and identify the two incorrect operations.

- a) use the frame table to find the process that owns the faulting page
- b) using the disk map of faulting process, load the faulting page from the disk into the victim frame
- c) select a victim page for replacement (and the associated victim frame)
- d) update the page table of faulting process and frame table to reflect the changed mapping for the victim frame
- e) using the disk map of the victim process, copy the victim page to the disk (if dirty)
- f) look up the frame table to identify the victim process and invalidate the page table entry of the victim page in the victim page table
- g) look up if the faulting page is currently in physical memory

Answer:

Step 1: c

Step 2: f

Step 3: e

Step 4: b

Step 5: d ***

(*** Note: It is OK if this is shown as step 3 or 4 so long as the other steps have the same relative order)

Operations (a) and (g) do not belong to page fault handling.

8.2 Interaction between the Process Scheduler and Memory Manager

Figure 8.7 shows the interaction between the CPU scheduler and the memory manager. At any point of time, the CPU is executing either one of the user processes, or one of the subsystems of the operating system such as the CPU scheduler, or the memory manager. The code for the scheduler, the memory manager, and all the associated data structures are in the kernel memory space (Figure 8.7). The user processes live in the user memory space. Once the CPU scheduler dispatches a process, it runs until one of the following events happen:

1. The hardware timer interrupts the CPU resulting in an upcall (indicated by 1 in the figure) to the CPU scheduler that may result in a process context switch. Recall that we defined an upcall (in Chapter 6) as a function call from the lower levels of the

system software to the higher levels. The CPU scheduler takes the appropriate action to schedule the next process on the CPU.

2. The process incurs a page fault resulting in an upcall (indicated by 2 in the figure) to the memory manager that results in page fault handling as described above.
3. The process makes a system call (such as requesting an I/O operation) resulting in another subsystem (not shown in the figure) getting an upcall to take the necessary action.

While these three events exercise different sections of the operating system, all of them share the PCB data structure, which aggregates the current state of the process.

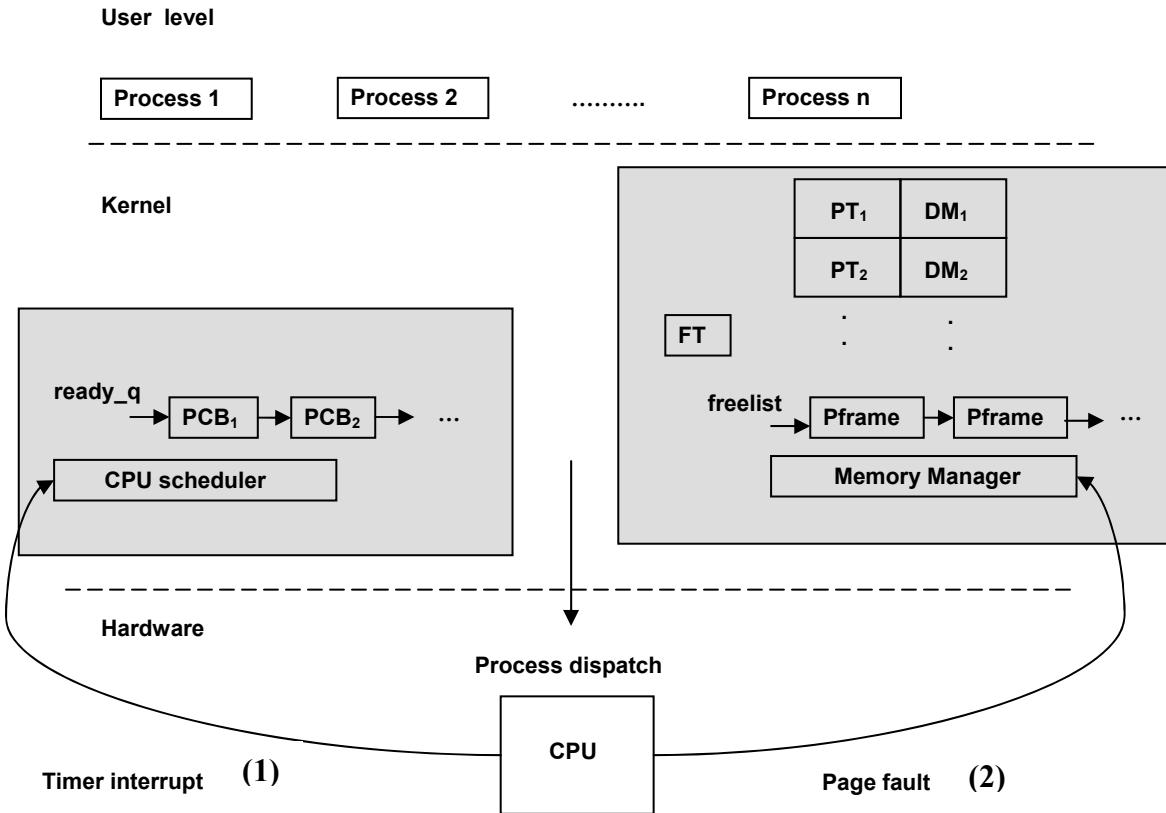


Figure 8.7: Interaction between the CPU scheduler and the memory manager

8.3 Page Replacement Policies

Now we will discuss how to pick a victim page to evict from the physical memory when there is a page fault and the free-list is empty. The process of selecting a victim page to evict from the physical memory is referred to as the *page replacement policy*. Let us understand the attributes of a good page replacement policy.

1. For a given string of page references, the policy should result in the least number of page faults. This attribute ensures that the amount of time spent in the operating system dealing with page faults is minimized.

2. Ideally, once a particular page has been brought into physical memory, the policy should strive not to incur a page fault for the same page again. This attribute ensures that the page fault handler respects the reference pattern of the user programs.

In selecting a victim page, the memory manager has one of two options:

- **Local victim selection:** The idea is to steal a physical frame from the faulting process itself to satisfy the request. Simplicity is the main attraction for this scheme. For example, this scheme eliminates the need for a *frame table*. However, this scheme will lead to poor memory utilization.
- **Global victim selection:** The idea is to steal a physical frame from any process, not necessarily the faulting one. The exact heuristic used to determine the victim process and the victim page depends on the specifics of the algorithm. This scheme will result in good utilization due to the global selection of the victim page.

The norm is to use global victim selection to increase memory utilization. Ideally, if there are no page faults the memory manager never needs to run, and the processor executes user programs most of the time (except for context switching). Therefore, *reducing the page fault rate* is the goal of any memory manager. There are two reasons why a memory manager strives to reduce the page fault rate: (1) the performance of a program incurring a page fault is adversely affected since bringing in a page from secondary storage is slow, and (2) precious processor cycles should not be used too often in overhead activities such as page replacement.

In the rest of the discussion, the presumption is global page replacement policy though the examples will focus on the paging behavior of a single process to keep the discussion simple. The memory manager bases its victim selection on the paging activity of the processes. Therefore, whenever we refer to a page we mean a virtual page. Once the memory manager identifies a virtual page as a victim, then the physical frame that houses the virtual page becomes the candidate for replacement. For each page replacement policy, we will identify the hardware assist (if any) needed, the data structures needed, the details of the algorithm, and the expected performance in terms of number of page-faults.

8.3.1 Belady's Min

Ideally, if we know the entire string of references, we should replace the one that is not referenced for the *longest time in the future*. This replacement policy is not feasible to implement since the memory manager does not know the future memory references of a process. However, in 1966 Laszlo Belady proposed this *optimal replacement algorithm*, called *Belady's Min*, to serve as a gold standard for evaluating the performance of any page replacement algorithm.

8.3.2 Random Replacement

The simplest policy is to replace a page randomly. At first glance, this may not seem like a good policy. The upside to this policy is that the memory manager does not need any hardware support, nor does it have to keep any bookkeeping information about the current set of pages (such as timestamp, and referential order). In the absence of

knowledge about the future, it is worthwhile understanding how bad or good a random policy would perform for any string of references. Just as Belady's Min serves as an upper bound for performance of page replacement policies, a random replacement algorithm may serve as a lower bound for performance. In other words, if a page replacement policy requires hardware support and/or maintenance of bookkeeping information by the memory manager, then it should perform better than a random policy as otherwise it is not worth the trouble. In practice, memory managers may default to random replacement in the absence of sufficient bookkeeping information to make a decision (see Section 8.3.4.1).

8.3.3 First In First Out (FIFO)

This is one of the simplest page replacement policies. FIFO algorithm works as follows:

- Affix a timestamp when a page is brought in to physical memory
- If a page has to be replaced, choose the *longest resident* page as the victim

It is interesting to note that we do not need any hardware assist for this policy. As we will see shortly, the memory manager remembers the order of arrival of pages into physical memory in its data structures.

Let us understand the data structure needed by the memory manager. The memory manager simulates the “timestamp” using a *queue* for recording the order of arrival of pages into physical memory. Let us use a circular queue (Figure 8.8) with a *head* and a *tail* pointer (initialized to the same value, say 0). We insert at the tail of the queue. Therefore, the longest resident page is the one at the head of the queue. In addition to the queue, the memory manager uses a *full* flag (initialized to *false*) to indicate when the queue is full. The occupancy of the queue is the number of physical frames currently in use. Therefore, we will make the queue size equal the number of physical frames. Each index into the circular queue data structure corresponds to a unique physical frame. Initially, the queue is empty (*full* flag initialized to *false*) indicating that none of the physical frames are in use. As the memory manager demand-pages in the faulting pages, it allocates the physical frames to satisfy the page faults (incrementing the tail pointer after each allocation). The queue is full (*full* flag set to *true*) when there are no more page frames left for allocation (*head* and *tail* pointers are equal). This situation calls for page replacement on the next page fault. The memory manager replaces the page at the *head*, which is the longest resident page. The interesting point to note is that this one data structure, *circular queue*, serves the purpose of the *free list* and the *frame table* simultaneously.

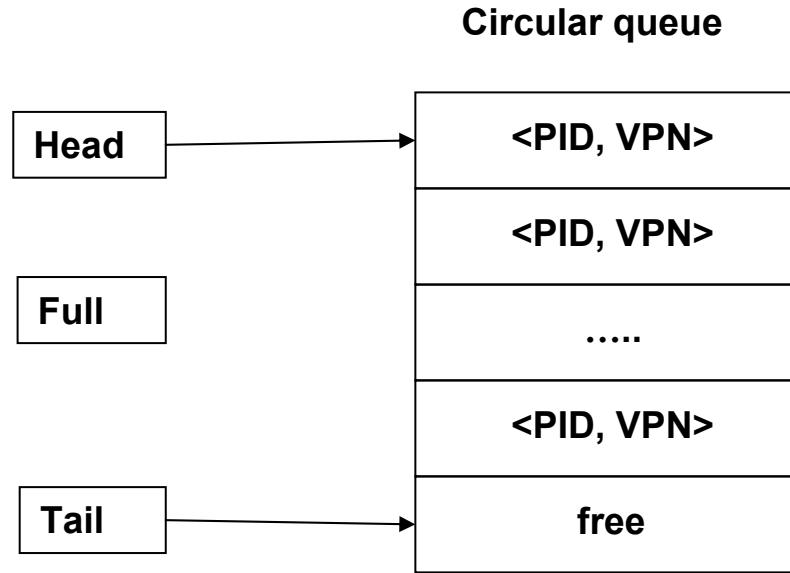


Figure 8.8: Circular queue for FIFO page replacement algorithm.
Tail points to first free physical frame index. Each queue entry corresponds to a physical page frame, and contains the $\langle \text{PID}, \text{VPN} \rangle$ housed in that frame. Head contains the longest resident page.

Example 8:

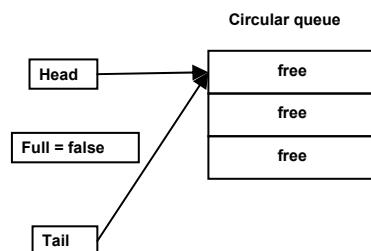
Consider a string of page references by a process:

Reference number:	1	2	3	4	5	6	7	8	9	10	11	12	13
<hr/>													
Virtual page number:	9	0	3	4	0	5	0	6	4	5	0	5	4

Assume there are 3 physical frames. Using a circular queue similar to Figure 8.8, show the state of the queue for the first 6 references.

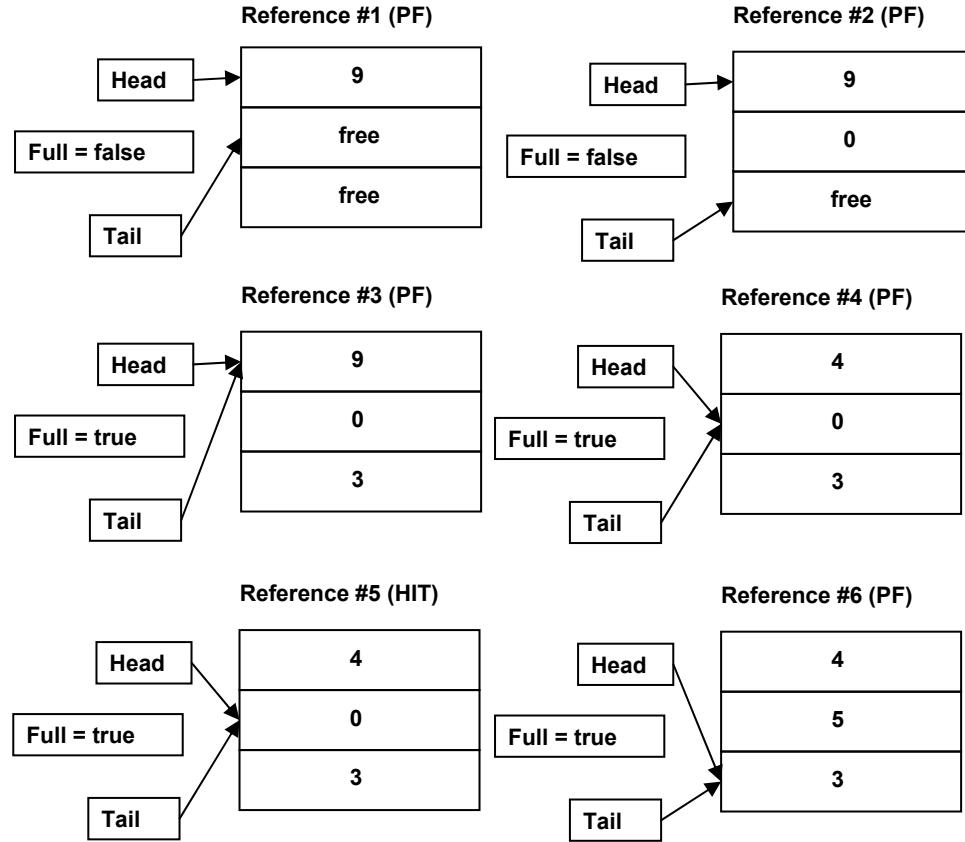
Answer:

Initially the circular queue looks as follows:



Upon a page fault, insertion of a new page happens at the entry pointed to by the tail. Similarly, victim page is one pointed to by the head, since the head always points to the FIFO page. Once selected, the head pointer moves to point to the next FIFO candidate. When the queue is full, both head and pointers move to satisfy a page fault. The

following snapshots of the data structure show the state of the queue after each of the first six references (PF denotes page fault; HIT denotes the reference is a hit, that is, there is no page fault):



The above sequence shows the anomaly in the FIFO scheme. Reference #6 replaces page 0 since that is the longest resident one to make room for page 5. However, page 0 is the immediate next reference (Reference #7).

Looking at the sequence of memory accesses in the above example, one can conclude that page 0 is *hot*. An efficient page replacement policy should try not to replace page 0. Unfortunately, we know this fact post-mortem. Let us see if we can do better than FIFO.

8.3.4 Least Recently Used (LRU)

Quite often, even in real life, we use the past as the predictor of the future. Therefore, even though we do not know the future memory accesses of a process, we do know the accesses made thus far by the process. Let us see how we use this information in page replacement. *Least Recently Used (LRU)* policy makes the assumption that if a page has not been referenced for a long time there is a good chance it will not be referenced in the future as well. Thus, the victim page in the LRU policy is the page that has not been used for the longest time.

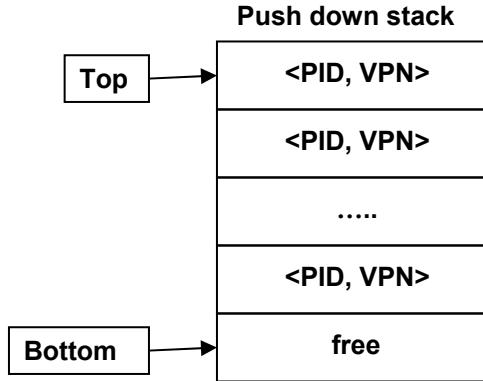


Figure 8.9: Push down stack for LRU replacement. The most recently used page is at the top of the stack. The least recently used page is at the bottom of the stack.

Let us understand the hardware assist needed for the LRU policy. The hardware has to track every memory access from the CPU. Figure 8.9 shows a stack data structure. On every access, the CPU pushes the currently accessed page on the top of the stack; if that page currently exists anywhere else in the stack the CPU removes it as well. Therefore, the bottom of the stack is the least recently used page and the candidate for replacement upon a page fault. If we want to track the references made to every page frame then the size of the stack should be as big as the number of frames.

Next we will investigate the data structure for the LRU policy. The memory manager uses the hardware stack in Figure 8.9 to pick the page at bottom of the stack as the victim. Of course, this requires some support from the instruction-set for the software to read the bottom of the stack. The hardware stack is in addition to the page table that is necessary for the virtual to physical translation.

Notice that the memory manager has to maintain additional data structures such as the *free-list* and *frame table* to service page faults.

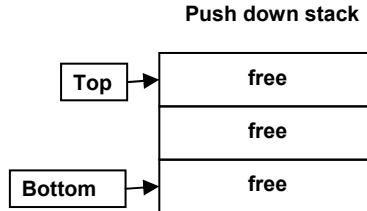
Example 9:

We will consider the same string of page references by a process as in the FIFO example:

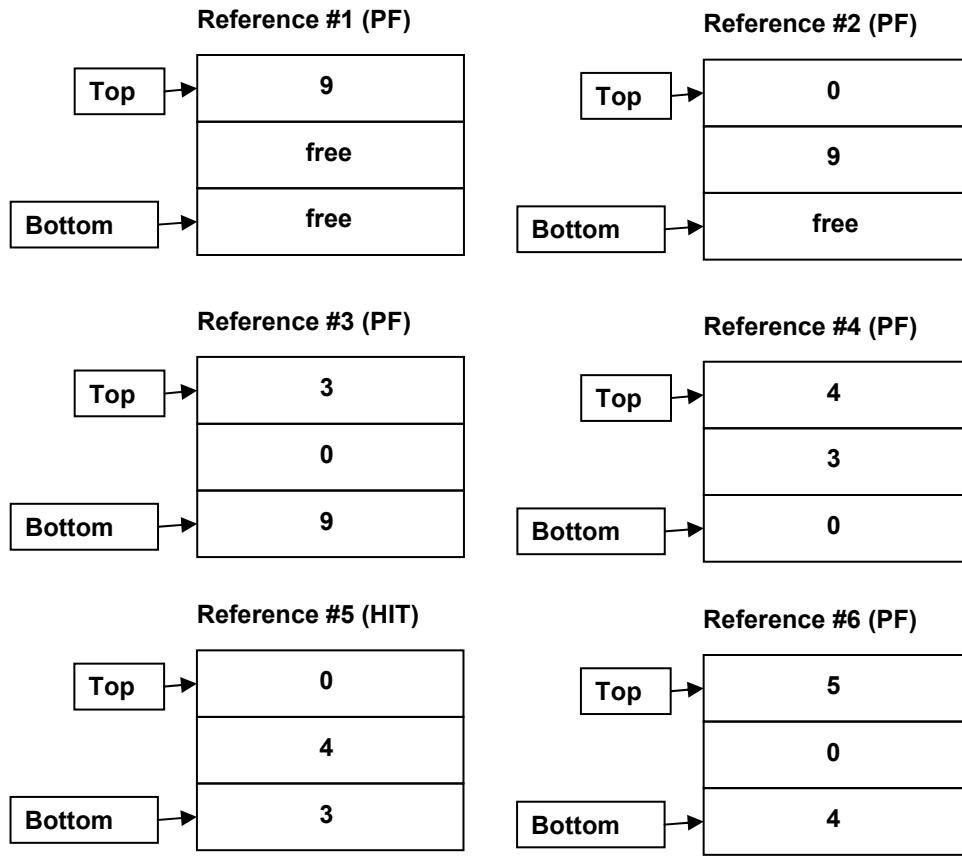
Reference number: 1 2 3 4 5 6 7 8 9 10 11 12 13

Virtual page number: 9 0 3 4 0 5 0 6 4 5 0 5 4

Assume there are 3 physical frames. Initially the stack looks as follows.



The following snapshots show the state of the stack after each of the first six references (PF denotes page fault; HIT denotes the reference does is a hit, that is, there is no page fault):



It is interesting to compare Examples 4 and 5. Both of them experience 5 page faults in the first 6 references. Unfortunately, we cannot do any better than that since these pages (9, 0, 3, 4, and 5) are not in physical memory and therefore these are page faults are unavoidable with any policy. However, notice that Reference #6 replaces page 3 in the LRU scheme (not page 0 as in the FIFO example). Thus, Reference #7 results in a hit for the LRU scheme. In other words, LRU is able to avoid the anomalous situation that we encountered with the FIFO policy.

8.3.4.1 Approximate LRU #1: A Small Hardware Stack

The LRU scheme, while conceptually appealing, is simply not viable from an implementation point of view for a couple of reasons:

1. The stack has as many entries as the number of physical frames. For a physical memory size of 4 GB and a pagesize of 8 KB, the size of the stack is 0.5 MB. Such large hardware structures in the datapath of a pipelined processor add enormous latencies to the clock cycle time of the processor. For this reason, it is not viable to have such a huge hardware stack in the datapath of the processor.
2. On every access, the hardware has to modify the stack to place the current reference on the top of the stack. This expensive operation slows down the processor.

For these reasons, it is not feasible to implement a *true* LRU scheme. A more serious concern arises because true LRU in fact can be detrimental to performance in certain situations. For example, consider a program that loops over $N+1$ pages after accessing them in sequence. If the memory manager has a pool of N page frames to deal with this workload, then there will be a page fault for every access with the LRU scheme. This example, pathological as it sounds, is indeed quite realistic since scientific programs manipulating large arrays are quite common.

It turns out that implementing an approximate LRU scheme is both feasible and less detrimental to performance. Instead of a stack equal in size to the physical memory size (in frames), its size can be some small number (say 16). Thus, the stack maintains the history of the last 16 unique references made by the processor (older references fall off the bottom of the stack). The algorithm would pick a random page that is not in the hardware stack as the victim. Basically, the algorithm protects the most recent N referenced pages from replacement, where N is the size of the hardware stack.

In fact, some simulation studies have found that true LRU may even be worse than approximate LRU. This is because anything other than Belady's Min is simply guessing the page reference behavior of a process, and is therefore susceptible to failure. In this sense, a purely random replacement policy (Section 8.3.2), which requires no sophisticated mechanisms in either hardware or software, has been shown to do well for certain workloads.

8.3.4.2 Approximate LRU #2: Reference bit per page frame

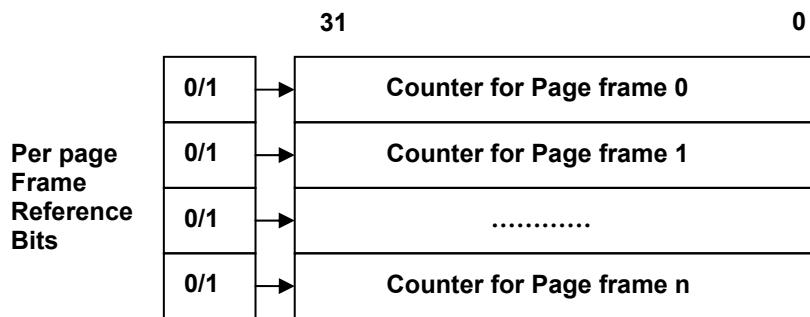
It turns out that tracking every memory access is just not feasible from the point of view of implementing a high-speed pipelined CPU. Therefore, we have to resort to some other means to approximate the LRU scheme.

One possibility is to track references at the page level instead of individual accesses. The idea is to associate a *reference bit* per page frame. Hardware sets this bit when the CPU accesses any location in the page; software reads and resets it. The hardware orchestrates accesses to the physical memory through the page table. Therefore, we can have the reference bits in the page table.

Let us turn our attention to choosing a victim. Here is an algorithm using the reference bits.

1. The memory manager maintains a *bit vector* per page frame, called a *reference counter*.

- Periodically the memory manager reads the reference bits of all the page frames and dumps them in the *most significant bit (MSB)* of the corresponding per frame *reference counters*. The counters are right-shifted to accommodate the reference bits into their respective MSB position. Figure 8.10 shows this step graphically. After reading the reference bits, the memory manager clears them. It repeats this step every time quantum. Thus each counter holds a snapshot of the references (or lack thereof) of the last n time quanta ($n = 32$ in Figure 8.10).



**Figure 8.10: Page frame reference counters;
The reference bit for each page may be 0 or 1**

- The reference counter with the largest absolute value is the most recently referenced page. The reference counter with the smallest absolute value is the least recently referenced page and hence a candidate for selection as a replacement victim.

A *paging daemon* is an entity that is part of the memory manager that wakes up periodically (as determined by the time quantum) to carry out the steps of the above algorithm.

8.3.5 Second chance page replacement algorithm

This is a simple extension to the FIFO algorithm using the *reference bits*. As the name suggests, this algorithm gives a second chance for a page to be **not** picked as a victim. The basic idea is to use the reference bit set by the hardware as an indication that the page deserves a second chance to stay in memory. The algorithm works as follows:

- Initially, **the operating system clears the reference bits of all the pages**. As the program executes, the hardware sets the reference bits for the pages referenced by the program.
- If a page has to be replaced, the memory manager chooses the replacement candidate in FIFO manner.
- If the **chosen victim's reference bit is set**, **then the manager clears the reference bit**, gives it a *new arrival time* and repeats step 1. In other words, this page is moved to the end of the FIFO queue.
- The victim is the first candidate in FIFO order whose reference bit is not set.

Of course, in the pathological case where all the reference bits are set, the algorithm degenerates to a simple FIFO.

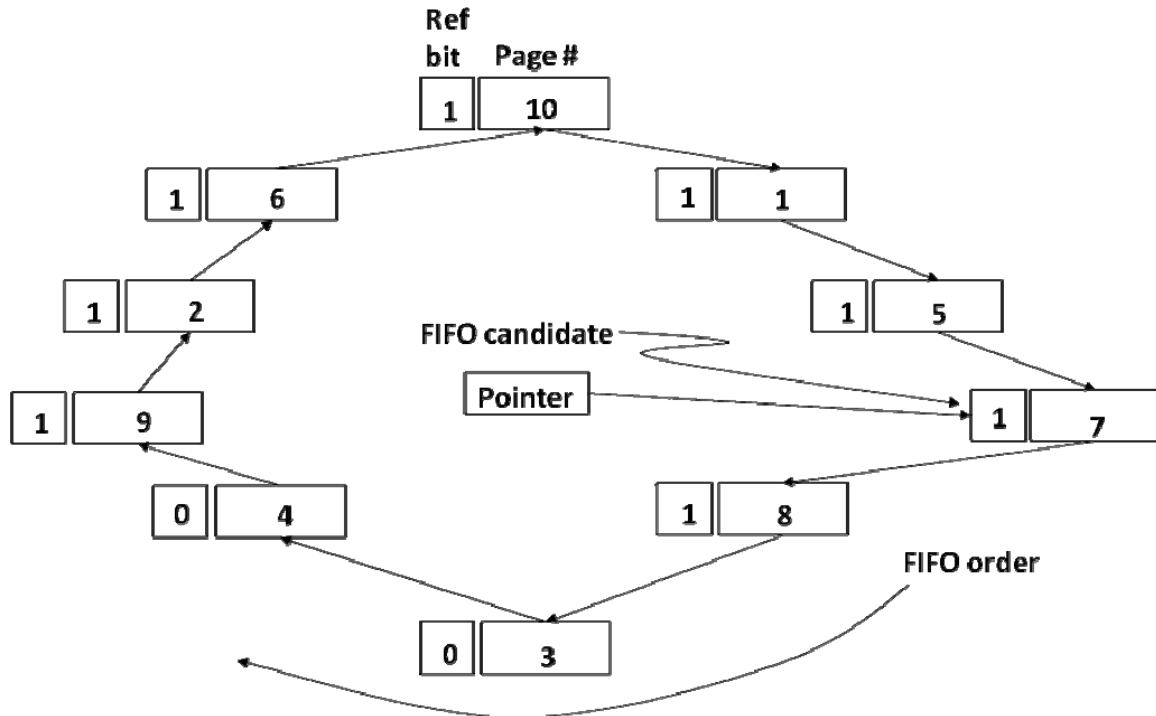


Figure 8.11a: Second chance replacement – the memory manager keeps a software pointer that points to the FIFO candidate at the start of the algorithm. Note the frames that have reference bits set and the ones that have their reference bits cleared. The ones that have their reference bits set to 1 are those that have been accessed by the program since the last sweep by the memory manager.

A simple way to visualize and implement this algorithm is to think of the pages forming a circular queue with a pointer pointing to the FIFO candidate as shown in Figure 8.11a. When called upon to pick a victim, the pointer advances until it finds a page whose reference bit is not set. As it moves towards the eventual victim, the memory manager clears the reference bits of the pages that it encounters. As shown in Figure 8.11a, the FIFO candidate is page 7. However, since its reference bit is set, the pointer moves until it gets to page 3 whose reference bit is not set (first such page in FIFO order) and chooses that page as the victim. The algorithm clears the reference bits for pages 7 and 8 as it sweeps in FIFO order. Note that the algorithm does not change the reference bits for the other pages not encountered during this sweep. It can be seen that the pointer sweep of the algorithm resembles the hand of a clock moving in a circle and for the same reason this algorithm is often referred to as the *clock* algorithm. After choosing page 3 as the victim, the pointer will move to the next FIFO candidate (page number 4 in Figure 8.11b).

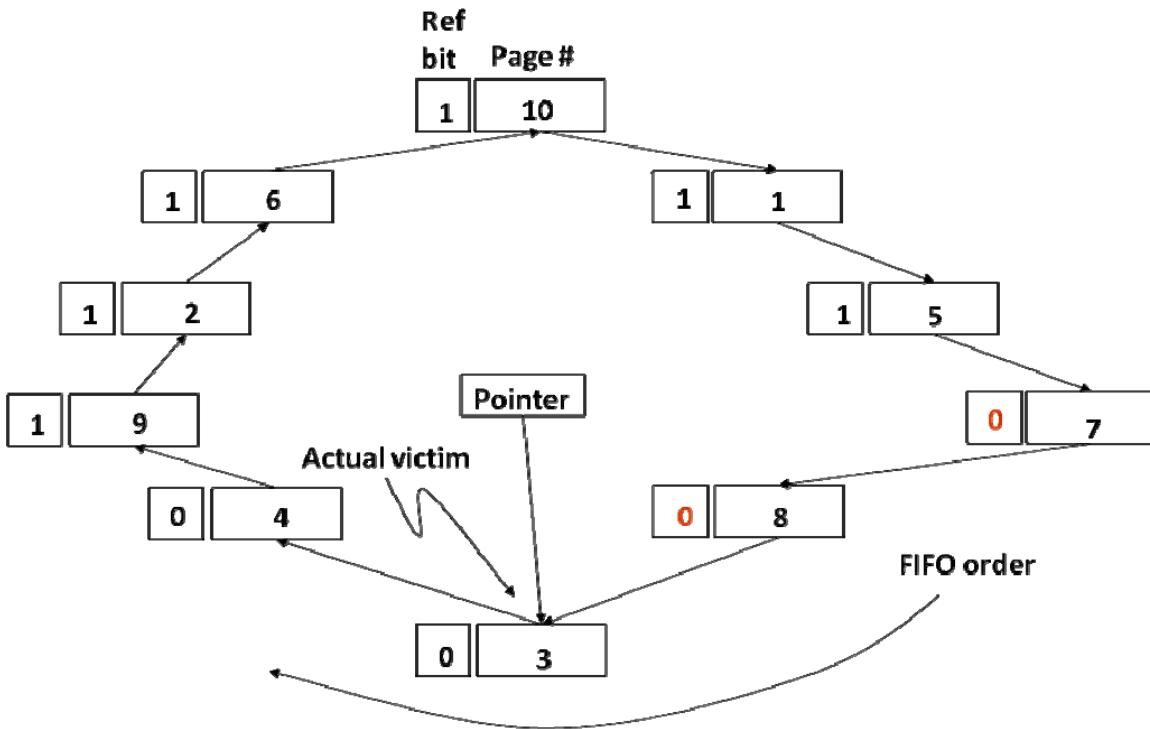


Figure 8.11b: Second chance replacement – algorithm passes over the frames whose reference bits are set to 1 (clearing them during the sweep) until it finds a frame that does not have its reference bit set, the actual victim frame.

Example 10:

Suppose that we only have three physical frames and that we use the second-chance page replacement algorithm. Show the virtual page numbers that occupy these frames for the following sequence of page references.

Reference number:	1	2	3	4	5	6	7	8	9	10
Virtual page number:	0	1	2	3	1	4	5	3	6	4

Answer:

The following sequence of figures show the state of the page frames and the associated reference bits **AFTER** each reference is satisfied. The top entry is always the normal FIFO candidate. Note that the reference bit is set when the page is brought into the page frame.

To understand the choice of the victim in a particular reference, please see the state of the pages shown after the previous reference.

References 1-3: no replacement

Reference 4: Page 0 is the victim (all have ref bits set, so FIFO candidate is the victim)

Reference 5: no replacement (page 1's reference bit set)

Reference 6: Page 2 is the victim (Page 1 the FIFO candidate gets a second chance)

Reference 7: Page 1 is the victim (Page 3 the FIFO candidate gets a second chance)

Reference 8: no replacement (page 3's reference bit set)

Reference 9: Page 4 is the victim (all have ref bits set, so FIFO candidate is the victim)

Reference 10: Page 3 is the victim (FIFO candidate and its ref bit is off)

Page in the frame	ref	Notation																		
Ref 1	Ref 2	Ref 3																		
<table border="1"><tr><td>0</td><td>1</td></tr><tr><td></td><td></td></tr><tr><td></td><td></td></tr></table>	0	1					<table border="1"><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td></tr><tr><td></td><td></td></tr></table>	0	1	1	1			<table border="1"><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td></tr><tr><td>2</td><td>1</td></tr></table>	0	1	1	1	2	1
0	1																			
0	1																			
1	1																			
0	1																			
1	1																			
2	1																			
Ref 4	Ref 5	Ref 6																		
<table border="1"><tr><td>1</td><td>0</td></tr><tr><td>2</td><td>0</td></tr><tr><td>3</td><td>1</td></tr></table>	1	0	2	0	3	1	<table border="1"><tr><td>1</td><td>1</td></tr><tr><td>2</td><td>0</td></tr><tr><td>3</td><td>1</td></tr></table>	1	1	2	0	3	1	<table border="1"><tr><td>3</td><td>1</td></tr><tr><td>1</td><td>0</td></tr><tr><td>4</td><td>1</td></tr></table>	3	1	1	0	4	1
1	0																			
2	0																			
3	1																			
1	1																			
2	0																			
3	1																			
3	1																			
1	0																			
4	1																			
Ref 7	Ref 8	Ref 9																		
<table border="1"><tr><td>4</td><td>1</td></tr><tr><td>3</td><td>0</td></tr><tr><td>5</td><td>1</td></tr></table>	4	1	3	0	5	1	<table border="1"><tr><td>4</td><td>1</td></tr><tr><td>3</td><td>1</td></tr><tr><td>5</td><td>1</td></tr></table>	4	1	3	1	5	1	<table border="1"><tr><td>3</td><td>0</td></tr><tr><td>5</td><td>0</td></tr><tr><td>6</td><td>1</td></tr></table>	3	0	5	0	6	1
4	1																			
3	0																			
5	1																			
4	1																			
3	1																			
5	1																			
3	0																			
5	0																			
6	1																			
Ref 10																				
		<table border="1"><tr><td>5</td><td>0</td></tr><tr><td>6</td><td>1</td></tr><tr><td>4</td><td>1</td></tr></table>	5	0	6	1	4	1												
5	0																			
6	1																			
4	1																			

8.3.6 Review of page replacement algorithms

Table 8.1 summarizes the page replacement algorithms and the corresponding hardware assists needed. It turns out that approximate LRU algorithms using reference bits do quite well in reducing the page fault rate and perform almost as well as true LRU; a case in point for engineering ingenuity that gets you most of the benefits of an exact solution.

PAGE REPLACEMENT ALGORITHM	HARDWARE ASSIST NEEDED	BOOKKEEPING INFORMATION NEEDED	COMMENTS
Belady's MIN	Oracle	None	Provably optimal performance; not realizable in hardware; useful as an upper bound for performance comparison
Random Replacement	None	None	Simplest scheme; useful as a lower bound for performance comparison
FIFO	None	Arrival time of a virtual page into physical memory	Could lead to anomalous behavior; often performance worse than random
True LRU	Push down stack	Pointer to the bottom of the LRU stack	Expected performance close to optimal; infeasible for hardware implementation due to space and time complexity; worst-case performance may be similar or even worse compared to FIFO
Approximate LRU #1	A small hardware stack	Pointer to the bottom of the LRU stack	Expected performance close to optimal; worst-case performance may be similar or even worse compared to FIFO
Approximate LRU #2	Reference bit per page frame	Reference counter per page frame	Expected performance close to optimal; moderate hardware complexity; worst-case performance may be similar or even worse compared to FIFO
Second Chance Replacement	Reference bit per page frame	Arrival time of a virtual page into physical memory	Expected performance better than FIFO; memory manager implementation simplified compared to LRU schemes

Table 8.1: Comparison of page replacement algorithms

8.4 Optimizing Memory Management

In the previous sections, we presented rudimentary techniques for demand-paged virtual memory management. In this section, we will discuss some optimizations that memory managers employ to improve the system performance. It should be noted that the optimizations are not specific to any particular page replacement algorithm. They could be used on top of the basic page replacement techniques discussed in the preceding section.

8.4.1 Pool of free page frames

It is not a good idea to wait until a page fault to select a victim for replacement. Memory managers always keep a *minimum* number of page frames ready for allocation to satisfy page faults. The paging daemon wakes up periodically to check if the pool of free frames on the *free-list* (Figure 8.3) is below this minimum threshold. If so, it will run the page replacement algorithm to free up more frames to meet this minimum threshold. It is possible that occasionally the number of page frames in the free list will be way over the minimum threshold. When a process terminates, all its page frames get on the free-list leading to such a situation.

8.4.1.1 Overlapping I/O with processing

In Section 8.1.4, we pointed out that before the memory manager uses a page frame as a victim, the manager should save the current contents of the page that it houses to the disk if it is dirty. However, since the manager is simply adding the page frame to the *free-list* there is no rush to do the saving right away. As we will see in a later chapter, high-speed I/O (such as to the disk) occurs concurrently with the CPU activity. Therefore, the memory manager simply schedules a write I/O for a dirty victim page frame before adding it to the *free-list*. Before the memory manager uses this dirty physical frame to house another virtual page, the write I/O must be complete. For this reason, the memory manager may skip over a dirty page on the *free-list* to satisfy a page fault (Figure 8.11). This strategy helps to delay the necessity to wait on a write I/O at the time of a page fault.

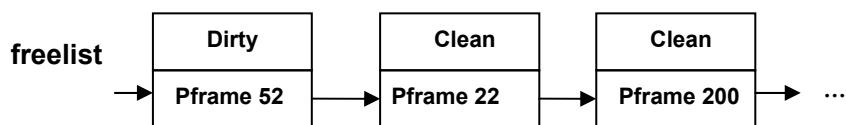


Figure 8.11: Free-list; upon a page fault, the manager may choose pframe 22 over pframe 52 as the victim since the former is clean while the latter is dirty.

8.4.1.2 Reverse mapping to page tables

To meet the minimum threshold requirement, the paging daemon may have to take away page frames currently mapped into running processes. Of course, the process that lost this page frame may fault on the associated page when it is scheduled to run. As it turns out, we can help this process with a little bit of additional machinery in each node of the free-list. If the memory manager has not yet reassigned the page frame to a different process, then we can retrieve it from the *free-list* and give it to the faulting process. To enable this optimization, we augment each node in the *free-list* with the reverse mapping (similar to the *frame table*), showing the virtual page it housed last (See Figure 8.12).

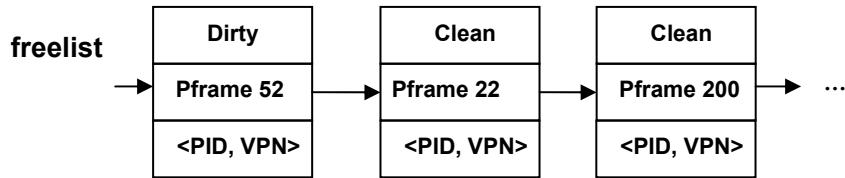


Figure 8.12: Reverse mapping for the page frames on the Free-list

Upon a page fault, the memory manager compares the $\langle \text{PID}, \text{VPN} \rangle$ of the faulting process with entries in the *free-list*. If there is a match, the manager simply re-establishes the original mapping in the page table for the faulting process by using the page frame of the matching entry to satisfy the page fault. This optimization removes the need for a read I/O from the disk to bring in the faulting page.

It is interesting to note that this enhancement when used in tandem with the second change replacement algorithm, essentially gives a *third* chance for a page before it is kicked out of the system.

8.4.2 Thrashing

A term that we often come across with respect to computer system performance is *thrashing*, which is used to denote that the system is not getting useful work done. For example, let us assume that the degree of multiprogramming (which we defined in Chapter 6 as the number of processes coexisting in memory and competing for the CPU) is quite high but still we observe low *processor utilization*. One may be tempted to increase the degree of multiprogramming to keep the processor busy. On the surface, this seems like a good idea, but let us dig a little deeper.

It is possible that all the currently executing programs are I/O bound (meaning they do more I/O than processing on the CPU) in which case the decision to increase the degree of multiprogramming may be a good one. However, it is also possible that the programs are CPU bound. At first glance, it would appear odd that the processor utilization would be low with such CPU bound jobs. The simple answer is too much paging activity. Let us elaborate this point. With a demand-paged memory management, a process needs to have sufficient memory allocated to it to get useful work done. Otherwise, it will page fault frequently. Therefore, if there are too many processes coexisting in memory (i.e., a high degree of multiprogramming), then it is likely that the processes are continuously paging against one another to get physical memory. Thus, none of the processes is making forward progress. In this situation, it is incorrect to increase the degree of multiprogramming. In fact, we should reduce it. Figure 8.13 shows the expected behavior of CPU utilization as a function of the degree of multiprogramming. The CPU utilization drops rapidly after a certain point.

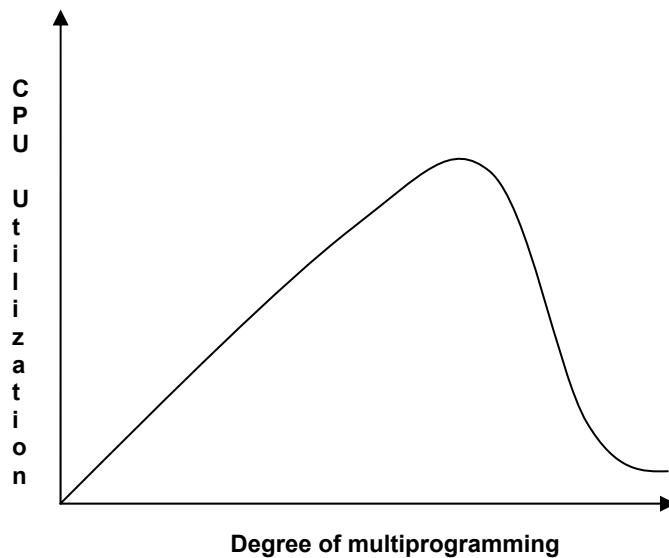


Figure 8.13: CPU Thrashing Phenomenon

This phenomenon, called *thrashing*, occurs when processes spend more time paging than computing. Paging is *implicit I/O* done on behalf of a process by the operating system. Excessive paging could make a process, which is originally compute bound into an I/O bound process. An important lesson to take away from this discussion is that CPU scheduling should take into account memory usage by the processes. It is erroneous to have a CPU scheduling policy based solely on processor utilization. Fortunately, this is a correctable situation by cooperation between the CPU scheduling and memory management parts of the operating system.

Let us investigate how we can control thrashing. Of course, we can load the entire program into memory but that would not be an efficient use of the resources. The trick is to ensure that each process has *sufficient* number of page frames allocated so that it does not page fault frequently. The *principle of locality* comes to our rescue. A process may have a huge memory footprint. However, if you look at a reasonable sized *window of time*, we will observe that it accesses only a small portion of its entire memory footprint. This is the principle of locality. Of course, the locus of program activity may change over time as shown in Figure 8.14. However, the change is gradual not drastic. For example, the set of pages used by the program at time t_1 is $\{p_1, p_2\}$; the set of pages at t_2 is $\{p_2, p_3\}$; and so on.

We do not want the reader to conclude that the locus of activity of a program is always confined to consecutive pages. For example, at time t_7 the set of pages used by the program is $\{p_1, p_4\}$. The point to take away is that the locus of activity is confined to a small subset of pages (see Example 11).

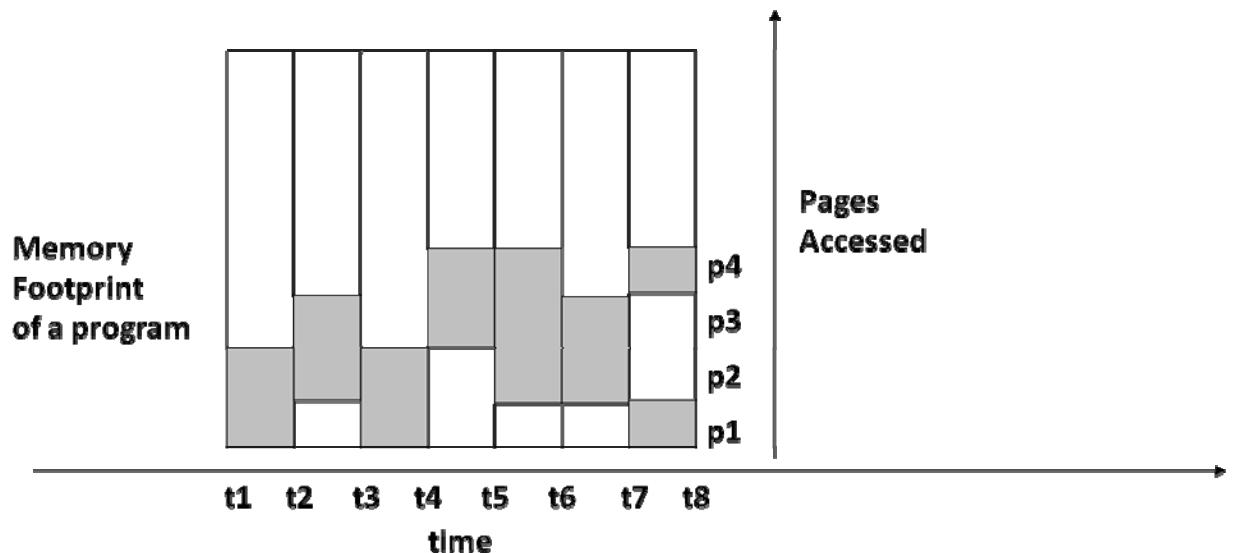


Figure 8.14: Changing loci of program activity as a function of time; the set of pages in use by the program is captured at times t_1 , t_2 , etc.; assume that the locus of program activity remains unchanged in the time intervals such as t_1-t_2 , t_2-t_3 , and so on

It is straightforward to use this principle to reduce page faults. If the current locus of program activity is in memory then the associated process will not page fault until the locus changes. For example, if the set of pages accessed by the program remains unchanged between t_1 and t_2 , and if p_1 and p_2 are in physical memory then the program will not experience any page fault between t_1 and t_2 .

8.4.3 Working set

To determine the locus of program activity, we define and use a concept called *working set*. Working set is the set of pages that defines the locus of activity of a program. Of course, the working set does not remain fixed since the locus of program activity changes over time.

For example, with reference to Figure 8.14,

$$\begin{aligned} \text{Working set}_{t_1-t_2} &= \{p_1, p_2\} \\ \text{Working set}_{t_2-t_3} &= \{p_2, p_3\} \\ \text{Working set}_{t_3-t_4} &= \{p_1, p_2\} \\ \text{Working set}_{t_4-t_5} &= \{p_3, p_4\} \\ \text{Working set}_{t_5-t_6} &= \{p_2, p_3, p_4\} \\ &\dots \end{aligned}$$

The *working set size* (WSS) denotes the number of distinct pages touched by a process in a window of time. For example, the WSS for this process is 2 in the interval t_1-t_2 , and 3 in the interval t_5-t_6 .

The *memory pressure* exerted on the system is the summation of the WSS of all the processes currently competing for resources.

$$\text{Total memory pressure} = \sum_{i=1}^{i=n} WSS_i \quad (1)$$

Example 11:

During the time interval $t_1 - t_2$, the following virtual page accesses are recorded for the three processes P1, P2, and P3, respectively.

P1: 0, 10, 1, 0, 1, 2, 10, 2, 1, 1, 0

P2: 0, 100, 101, 102, 103, 0, 101, 102, 104

P3: 0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5

- a) What is the **working set** for each of the above three processes for this time interval?
- b) What is the **cumulative memory pressure** on the system during this interval?

Answer:

a)

P1's Working set = {0, 1, 2, 10}

P2's Working set = {0, 100, 101, 102, 103, 104}

P3's Working set = {0, 1, 2, 3, 4, 5}

b)

Working set size of P1 = 4

Working set size of P2 = 6

Working set size of P3 = 6

Cumulative memory pressure = sum of the working sets of all processes

$$= 4 + 6 + 6$$

$$= 16 \text{ page frames}$$

8.4.4 Controlling thrashing

1. If the total memory pressure exerted on the system is greater than the total available physical memory then the memory manager decreases the degree of multiprogramming. If the total memory pressure is less than the total available physical memory then the memory manager increases the degree of multiprogramming.

One approximate method for measuring the WSS of a process uses the reference bits associated with the physical frames. Periodically say every Δ time units, a daemon

wakes up and samples the reference bits of the physical frames assigned to a process. The daemon records the page numbers that have their respective reference bits turned on; it then clears the reference bits for these pages. This recoding of the page numbers allows the memory manager to determine the working set and the WSS for a given process for any interval t and $t + \Delta$.

2. Another simple method of controlling thrashing is to use the observed page fault rate as a measure of thrashing. The memory manager sets two markers, a *low water mark*, and a *high water mark* for page faults (See Figure 8.15). An observed page fault rate that exceeds the high water mark implies excessive paging. In this case, the memory manager reduces the degree of multiprogramming. On the other hand, an observed page fault rate that is lower than the low water mark presents an opportunity for the memory manager to increase the degree of multiprogramming.

The shaded region in Figure 8.15 shows the preferred sweet spot of operation for the memory manager. The paging daemon kicks in to increase the pool of free physical frames when the page fault rate goes higher than the high water mark, decreasing the degree of multiprogramming, if necessary.

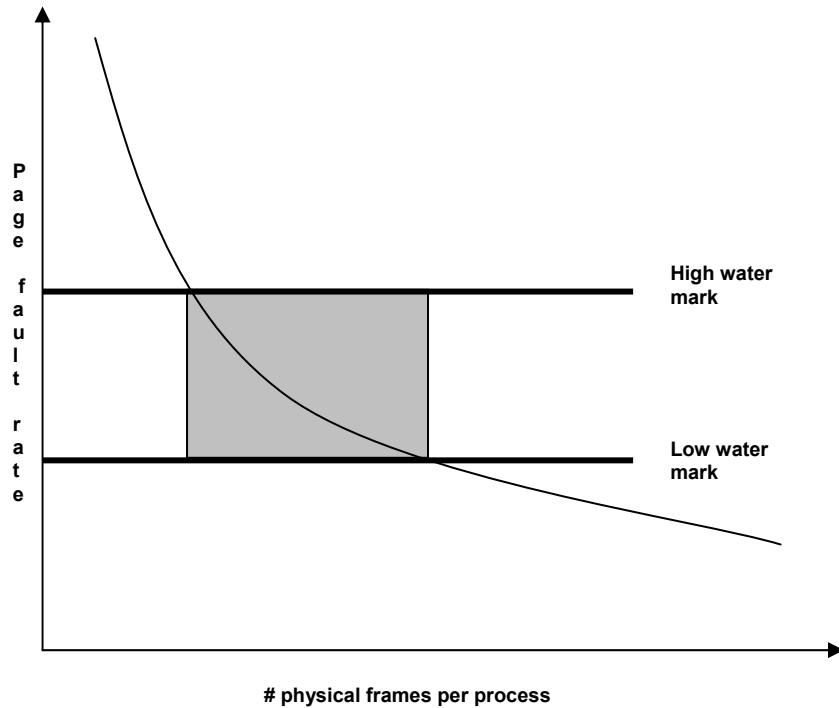


Figure 8.15: Page fault rate

8.5 Other considerations

The operating system takes a number of other measures to reduce page faults. For example, when the memory manager swaps out a process (to decrease the degree of multiprogramming) it remembers the current working set of the process. The memory manager brings in the corresponding working set at the point of swapping in a process. This optimization referred to as *pre-paging* reduces disruption at process start-up.

I/O activity in the system occurs simultaneously and concurrently with the CPU activity. This leads to interesting interaction between the memory and the I/O subsystems of the operating system. For example, the I/O subsystem may be initiating a transfer from a given physical frame to the disk. Simultaneously, the paging daemon may select the same physical frame as a potential victim for servicing a page fault. Just as the CPU scheduler and memory manager work in concert, the I/O and the memory subsystems coordinate their activities. Typically, the I/O subsystem will *pin a page* in physical memory for the duration of the data transfer to prevent the paging daemon from selecting that page as a victim. The page table serves as a convenient communication area between the I/O subsystem and the memory manager for recording information such as the need to pin a page in physical memory. We will discuss I/O in much more detail in a later chapter (see Chapter 10).

8.6 Translation Lookaside Buffer (TLB)

The discussion thus far should make one point very clear. Page faults are disruptive to system performance and the memory manager strives hard to avoid them. To put things in perspective, the context switch time (from one process to another) approximates several tens of instructions; the page fault handling time (without disk I/O) also approximates several tens of instructions. On the other hand, the time spent in disk I/O is in the millisecond range that approximates to perhaps a million instructions on a GHz processor.

However, even if we reduce the number of page faults with a variety of optimizations, it remains that every memory access involves two trips to the memory: one for the address translation and one for the actual instruction or data.

USER/KERNEL	VPN	PFN	VALID/INVALID
U	0	122	V
U	XX	XX	I
U	10	152	V
U	11	170	V
K	0	10	V
K	1	11	V
K	3	15	V
K	XX	XX	I

Figure 8.16: Translation Look-aside Buffer (TLB)

This is undesirable. Fortunately, we can cut this down to one with a little bit engineering ingenuity. The concept of paging removes the user's view of contiguous memory footprint for the program. However, it still is the case that *within* a page the memory locations are contiguous. Therefore, if we have performed the address translation for a page, then the same translation applies to *all* memory locations in that page. This suggests adding some hardware to the CPU to remember the address translations. However, we know that programs may have large memory footprints. The principle of locality that we introduced earlier comes to our rescue again. Referring to Figure 8.14, we may need to remember only a few translations *at a time* for a given program irrespective of its total memory footprint. This is the idea behind *translation look-aside buffer (TLB)*, a small hardware table in which the CPU holds *recent* address translations (Figure 8.16). An address translation for a page needs to be done at least once. Therefore, none of the entries in the table is valid when the processor starts up. This is the reason for the *valid* bit in each entry. The *PFN* field gives the physical frame number corresponding to the *VPN* for that entry.

Notice how the TLB is split into two parts. One part holds the translations that correspond to the user address space. The other part holds the same for the kernel space. On a context switch, the operating system simply invalidates all the user space translations, schedules the new process, and builds up that part of the table. The kernel space translations are valid independent of the user process currently executing on the processor. To facilitate TLB management, the instruction-set provides *purge TLB*, a privileged instruction executable in kernel mode,

8.6.1 Address Translation with TLB

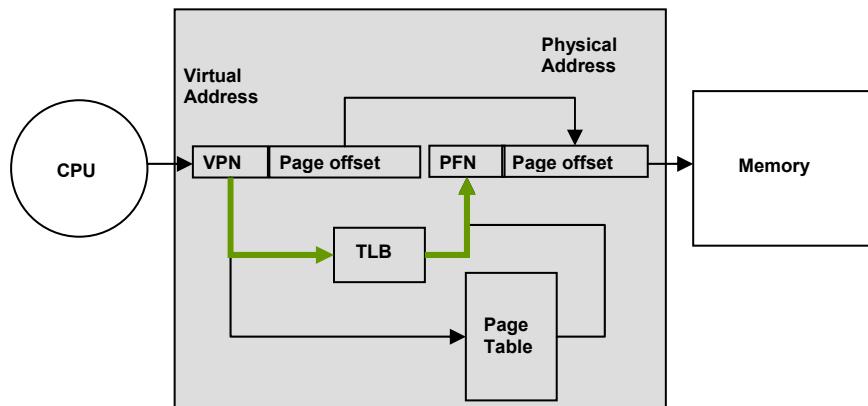


Figure 8.17-a: Address Translation (TLB Hit)

Figure 8.17-a and 8.17-b show the CPU address translation in the presence of the TLB. The hardware first checks if there is a valid translation in the TLB for the CPU generated address. We refer to such a successful lookup of the TLB as a *hit*. It is a *miss* otherwise. On a hit, the PFN from the TLB helps in forming the physical address thus avoiding a trip to the memory for address translation. On a miss, the page table in memory supplies the

PFN. The hardware enters the resulting translation into the TLB for future accesses to the same page.

Of course, it is quite possible that the address translation is done entirely in software. You may be wondering how this would be possible. Basically, the hardware raises a “TLB fault” exception if it does not find the translation it needs in the TLB. At this point, the operating system takes over and handles the fault. The processing that the operating system needs to do to service this fault may escalate to a full-blown page fault if the page itself is not present in the memory. In any event, once it completes the TLB fault handling, the operating system will enter the translation into the TLB so that regular processing could resume. Architectures such as the MIPS and DEC Alpha take this approach of handling the TLB faults entirely in software. Naturally, the ISA for these architectures have special instructions for modifying the TLB entries. It is customary to call such a scheme, *software managed TLB*.

The TLB is a special kind of memory, different from any hardware device we have encountered thus far. Essentially, the TLB is a hash table that contains VPN-PFN matches. The hardware has to look through the entire table to find a match for a given VPN. We refer to this kind of hardware device as *content addressable memory (CAM)* or *associative memory*. The exact details of the hardware implementation of the TLB depend on its organization.

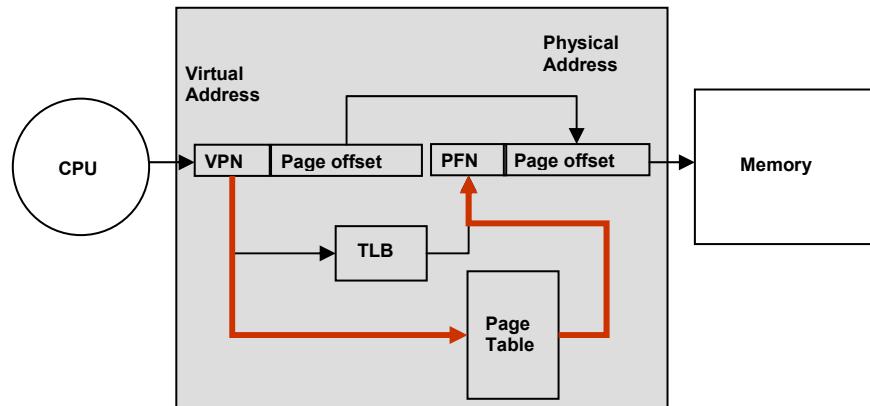


Figure 8.17-b: Address Translation (TLB Miss)

It turns out that TLB is a special case of a general concept, namely, *caching*, which we deal with in much more detail in Chapter 9 in the context of memory hierarchies. Suffice it to say at this point, caching refers to using a small table of items recently referred to by any subsystem. In this sense, this concept is exactly similar to the toolbox/tool-tray analogy we introduced in Chapter 2.

The caching concept surfaces in several contexts. Here are a few concrete examples:

- (a) in processor design for keeping recently accessed memory locations in the processor in a hardware device called *processor cache*,
- (b) in processor design to keep recent address translations in the TLB,
- (c) in disk controller design to keep recently accessed disk blocks in memory (see Chapter 10),
- (d) in file system design to keep an in-memory data structure of recently accessed bookkeeping information regarding the physical location of files on the disk (see Chapter 11),
- (e) in file system design to keep an in-memory software cache of recently accessed files from the disk (see Chapter 11), and
- (f) in web browser design to keep an on-disk cache of recently accessed web pages.

In other words, *caching is a general concept of keeping a smaller stash of items nearer to the point of use than the permanent location of the information*. We will discuss the implementation choices for processor caches in more detail in the next Chapter that deals with memory hierarchy.

8.7 Advanced topics in memory management

We mentioned that the memory manager's data structures include the process page tables. Let us do some math. Assuming a 40-bit byte-addressable virtual address and an 8 KB pagesize, we need 2^{27} page table entries for each process page table. This constitutes a whopping 128 million entries for each process page table. The entire physical memory of the machine may not be significantly more than the size of a single process page table.

We will present some preliminary ideas to get around this problem of managing the size of the page tables in physical memory. A more advance course in operating systems will present a more detailed treatment of this topic.

8.7.1 Multi-level page tables

The basic idea is to break a single page table into a multi-level page table. To make the discussion concrete consider a 32-bit virtual address with a 4 KB page size. The page table has 2^{20} entries. Let us consider a 2-level page table shown in Figure 8.18. The VPN of the virtual address has two parts. VPN1 picks out a unique entry in the first level page table that has 2^{10} entries. There is a second level page table (indexed by VPN2) corresponding to each unique entry in the first level. Thus, there are 2^{10} (i.e., 1024) second level tables, each with 2^{10} entries. The second level tables contain the PFN corresponding to the VPN in the virtual address.

Let us understand the total space requirement for the page tables. With a single-level conventional structure, we will need a page table of size 2^{20} entries (1 M entries). With a two-level structure, we need 2^{10} entries for the first level plus 2^{10} second-level tables,

each with 2^{10} entries. Thus, the total space requirement for a 2-level page table is 1K entries (size of first level table) more than the single-level structure¹.

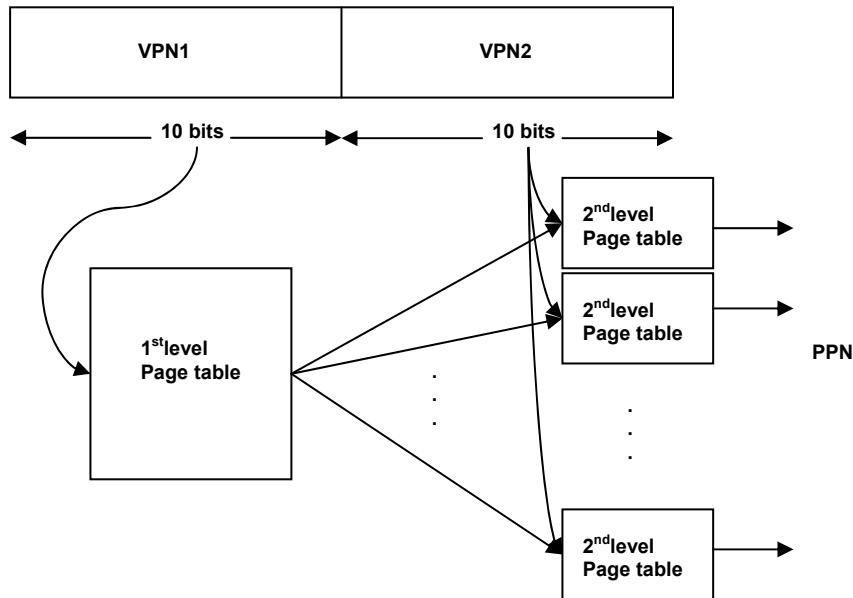


Figure 8.18: A 2-level page table

Let us understand what we have gained by this two-level structure. The 1st level page table has 1K entries per process. This is a reasonable sized data structure to have per process in physical memory. The 2nd level tables need not be in physical memory. The memory manager keeps them in virtual memory and demand-pages in the 2nd level tables based on program locality.

Modern operating systems catering to 64-bit processor architectures implement multi-level page tables (i.e., more than 2 levels). Unfortunately, with multi-level page tables, a memory access potentially has to make multiple trips to the physical memory. While this is true, fortunately, the TLB makes subsequent memory accesses to the same page translation-free.

Example 13:

Consider a memory system with 64-bit virtual addresses and with an 8KB page size. We use a five-level page table. The 1st level page table keeps 2K (2048) entries for each process. The remaining four levels, each keep 1K (1024) entries.

- Show the layout of a virtual address in this system with respect to this multi-level page table structure.
- What is the total page table space requirement for each process (i.e., sum

¹ Note that the size of the page table entries in the 2-level page table is different from that of the single-level structure. However, to keep the discussion simple, we do not get into this level of detail. Please refer to review question 9 for an elaboration of this point.

of all the levels of the page tables)?

Answer:

a)

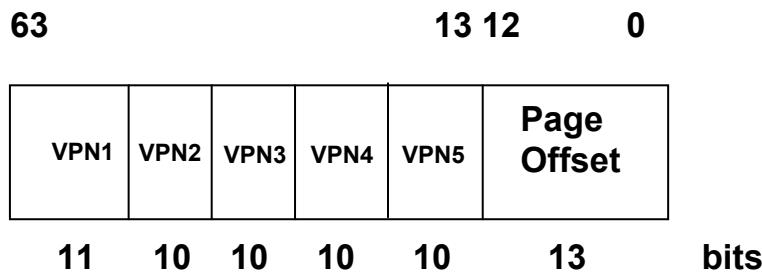
With 8KB pagesize, the number of bits for page offset = 13.

Therefore the VPN = $64 - 13 = 51$ bits

Number of bits for the first level page table (2048 entries) = 11

Number of bits for each of the remaining 4 levels (1024 entries each) = 10

Layout of the virtual address:



b)

Number of entries in the 1st level table = 2^{11}

Number of entries in each 2nd level table = 2^{10}

(Similar to Figure 8.18) There are 2^{11} such 2nd level tables (one for each first level table entry)

So total number of entries at the second level = 2^{21}

Number of entries in each 3rd level table = 2^{10}

There are 2^{21} such 3rd level tables (one for each second level table entry)

So total number of entries at the third level = 2^{31}

Number of entries in each 4th level table = 2^{10}

There are 2^{31} such 4th level tables (one for each third level table entry)

So total number of entries at the fourth level = 2^{41}

Number of entries in each 5th level table = 2^{10}

There are 2^{41} such 5th level tables (one for each fourth level table entry)

So total number of entries at the fifth level = 2^{51}

So total page table size for this virtual memory system

$$= 2^{11} + 2^{21} + 2^{31} + 2^{41} + 2^{51}$$

A single-level page table for this virtual memory system would require a page table with 2^{51} entries.

8.7.2 Access rights as part of the page table entry

A page table entry usually contains information in addition to the PFN and the valid bit. For example, it may contain access rights to the particular page such as *read-only*, *read-write*, etc. This is extremely important from the point of meeting the expectations laid out in Chapter 7 on the functionalities provided by the memory manager (please see Section 7.1). In particular, we mentioned that the memory manager has to provide memory protection and isolation for processes from one another. Further, a process has to be protected against itself from erroneous behavior due to programming bugs (e.g., writing garbage unintentionally across the entire memory footprint due to a bug in the program). Lastly, processes have to be able to share memory with each other when needed. Access rights information in a page table entry is another illustration of the cooperation between the hardware and software. The memory manager sets, in the page table, the access rights for the pages contained in the memory footprint of a program at the time of process creation. For example, it will set the access rights for pages containing code to be read-only and data to be read-write. The hardware checks the access rights as part of the address translation process on every memory access. Upon detection of violation of the access rights, the hardware takes corrective action. For example, if a process attempts to write to a read-only page, it will result in an *access violation trap* giving the control to the operating system for any course correction.

The page table entry is also a convenient place for the operating system to place other information pertinent to each page. For example, the page table entry may contain the information needed to bring in a page from the disk upon a page fault.

8.7.3 Inverted page tables

Since the virtual memory is usually much larger than the physical memory, some architectures (such as the IBM Power processors) use an *inverted page table*, essentially a frame table. The inverted page table alleviates the need for a per-process page table. Further, the size of the table is equal to the size of the physical memory (in frames) rather than virtual memory. Unfortunately, inverted page tables complicate the logical to physical address translation done by the hardware. Therefore, in such processors the hardware handles address translations through the TLB mechanism. Upon a TLB miss, the hardware hands over control (through a trap) to the operating system to resolve the translation in software. The operating system is responsible for updating the TLB as well. The architecture usually provides special instructions for reading, writing, and purging the TLB entries in privileged mode.

8.8 Summary

The memory subsystem in modern operating systems comprises the *paging daemon*, *swapper*, *page fault handler*, and so on. The subsystem works in close concert with the CPU scheduler and the I/O subsystem. Here is a quick summary of the key concepts we learned in this chapter:

- Demand paging basics including hardware support, and data structures in the operating system for demand-paging
- Interaction between the CPU scheduler and the memory manager in dealing with page faults

- Page replacement policies including FIFO, LRU, and second chance replacement
- Techniques for reducing the penalty for page faults including keeping a pool of page frames ready for allocation on page faults, performing any necessary writes of replaced pages to disk lazily, and reverse mapping replaced page frames to the displaced pages
- Thrashing and the use of working set of a process for controlling thrashing
- Translation look-aside buffer for speeding up address translation to keep the pipelined processor humming along
- Advanced topics in memory management including multi-level page tables, and inverted page tables

We observed in Chapter 7, that modern processors support page-based virtual memory. Correspondingly, operating systems on such modern processors such as Linux and Microsoft Windows (NT, XP, and Vista) implement page-based memory management. The second-chance page replacement policy, which we discussed in Section 8.3.5, is a popular one due to its simplicity and relative effectiveness compared to the others.

8.9 Review Questions

1. In a 5-stage pipelined processor, upon a page fault, what needs to happen in hardware for instruction re-start?
2. Describe the role of the frame table and the disk map data structures in a demand-paged memory manager.
3. Enumerate the steps in page fault handling.
4. Describe the interaction between the process scheduler and the memory manager.
5. What is the difference between second chance page replacement algorithm and a simple FIFO algorithm?
6. Consider an architecture wherein for each entry in the TLB there is:
 - 1 reference bit (that is set by hardware when the associated TLB entry is referenced by the CPU for address translation)
 - 1 dirty bit (that is set by hardware when the associated TLB entry is referenced by the CPU for a store access).

These bits are in addition to the other fields of the TLB discussed in Section 8.6.

The architecture provides three special instructions:

- one for sampling the reference bit for a particular TLB entry (`Sample_TLB(entry_num)`);
- one for clearing the reference bit for a particular TLB entry (`Clear_refbit_TLB(entry_num)`); and

- one for clearing the reference bits in all the TLB entries
(Clear_all_refbits_TLB()).

Come up with a scheme to implement page replacement using this additional help from the TLB. You should show data structures and pseudo code that the algorithm maintains to implement page replacement.

7. A process has a memory reference pattern as follows:

1 3 1 2 3 4 2 3 1 2 3 4

The above pattern (which represents the virtual page numbers of the process) repeats throughout the execution of the process. Assume that there are 3 physical frames.

Show the paging activity for a True LRU page replacement policy. Show an LRU stack and the page that is replaced (if any) for the first 12 accesses. Clearly indicate which accesses are hits and which are page faults.

8. A process has a memory reference pattern as follows:

4 3 1 2 3 4 1 4 1 2 3 4

The above pattern (which represents the virtual page numbers of the process) repeats throughout the execution of the process.

What would be the paging activity for an optimal page replacement policy for the first 12 accesses? Indicate which accesses are hits and which are page faults.

9. A processor asks for the contents of virtual memory address 0x30020. The paging scheme in use breaks this into a VPN of 0x30 and an offset of 0x020.

PTB (a CPU register that holds the address of the page table) has a value of 0x100 indicating that this process's page table starts at location 0x100.

The machine uses word addressing and the page table entries are each one word long.

$$\text{PTBR} = 0x100$$

VPN	Offset
0x30	0x020

The contents of selected memory locations are:

Physical Address	Contents

0x00000	0x00000
0x00100	0x00010
0x00110	0x00000
0x00120	0x00045
0x00130	0x00022
0x10000	0x03333
0x10020	0x04444
0x22000	0x01111
0x22020	0x02222
0x45000	0x05555
0x45020	0x06666

What is the physical address calculated?

What are the contents of this address returned to the processor?

How many memory references would be required (worst case)?

10. During the time interval $t_1 - t_2$, the following virtual page accesses are recorded for the three processes P1, P2, and P3, respectively.

P1: 0, 0, 1, 2, 1, 2, 1, 1, 0

P2: 0, 100, 101, 102, 103, 0, 1, 2, 3

P3: 0, 1, 2, 3, 0, 1, 2, 3, 4, 5

What is the working set for each of the above three processes for this time interval?

What is the cumulative memory pressure on the system during this interval?

11. Consider a virtual memory system with 20-bit page frame number. This exercise is to work out the details of the actual difference in page table sizes for a 1-level and 2-level page table arrangement. For a 2-level page table assume that the arrangement is similar to that shown in Figure 8.18. We are only interested in the page table that has to be memory resident always. For a 2-level page table this is only the first level page table. For the single level page table, the entire page table has to be memory resident. You will have to work out the details of the PTE for each organization (1-level and 2-level) to compute the total page table requirement for each organization.

Chapter 9 Memory Hierarchy

(Revision number 22)

Let us first understand what we mean by memory hierarchy. So far, we have treated the physical memory as a black box. In the implementation of LC-2200 (Chapter 3 and Chapter 5), we treated memory as part of the datapath. There is an implicit assumption in that arrangement, namely, accessing the memory takes the same amount of time as performing any other datapath operation. Let us dig a little deeper into that assumption. Today processor clock speeds have reached the GHz range. This means that the CPU clock cycle time is less than a nanosecond. Let us compare that to the state-of-the-art memory speeds (circa 2009). Physical memory, implemented using *dynamic random access memory (DRAM)* technology, has a cycle time in the range of 100 nanoseconds. We know that the slowest member determines the cycle time of the processor in a pipelined processor implementation. Given that IF and MEM stages of the pipeline access memory, we have to find ways to bridge the 100:1 speed disparity that exists between the CPU and the memory.

It is useful to define two terminologies frequently used in memory systems, namely, *access time* and *cycle time*. The delay between submitting a request to the memory and getting the data is called the access time. On the other hand, the time gap needed between two successive requests to the memory system is called the cycle time. A number of factors are responsible for the disparity between the access time and cycle time. For example, DRAM technology uses a single transistor to store a bit. Reading this bit depletes the charge, and therefore requires a replenishment before the same bit is read again. This is why DRAMs have different cycle time and access time. In addition to the specific technology used to realize the memory system, transmission delays on buses used to connect the processor to the memory system add to the disparity between access time and cycle time.

Let us re-visit the processor datapath of LC-2200. It contains a register file, which is also a kind of memory. The access time of a small 16-element register file is at the speed of the other datapath elements. There are two reasons why this is so. The first reason is that the register-file uses a different technology referred to as *static random access memory (SRAM)*. The virtue of this technology is speed. Succinctly put, SRAM gets its speed advantage over DRAM by using six transistors for storing each bit arranged in such a way that eliminates the need for replenishing the charge after a read. For the same reason, there is no disparity between cycle time and access time for SRAMs. As a rule of thumb, SRAM cycle time can be 8 to 16 times faster than DRAMs. As you may have guessed, SRAMs are also bulkier than DRAMs since they use 6 transistors per bit (as opposed to 1 per bit of DRAM), and consume more power for the same reason. Not surprisingly, SRAMs are also considerably more expensive per bit (roughly 8 to 16 times) than DRAMs.

The second and more compelling reason why physical memory is slow compared to register file is the sheer *size*. We usually refer to a register file as a 16-, 32-, or 64-

element entity. On the other hand, we usually specify the size of memory in quantities of Kbytes, Mbytes, or these days in Gbytes. In other words, even if we were to implement physical memory using SRAM technology the larger structure will result in slower access time compared to a register file. A simple state of reality is that independent of the implementation technology (i.e., SRAM or DRAM), you can have high speed or large size but not both. In other words, you cannot have the cake and eat it too!

Pragmatically speaking, SRAMs do not lend themselves to realizing large memory systems due to a variety of reasons, including power consumption, die area, delays that are inevitable with large memories built out of SRAM, and ultimately the sheer cost of this technology for realizing large memories. On the other hand, DRAM technology is much more frugal with respect to power consumption compared to its SRAM counterpart and lends itself to very large scale integration. For example, quoting 2007 numbers, a single DRAM chip may contain up to 256 Mbits with an access time of 70 ns. The virtue of the DRAM technology is the size. Thus, it is economically feasible to realize large memory systems using DRAM technology.

9.1 The Concept of a Cache

It is feasible to have a small amount of fast memory and/or a large amount of slow memory. Ideally, we would like to have the *size* advantage of a *large slow memory* and the *speed* advantage of a *fast small memory*. Given the earlier discussion regarding size and speed, we would choose to implement the small fast memory using SRAM, and the large slow memory using DRAM.

Memory hierarchy comes into play to achieve these twin goals. Figure 9.1 shows the basic idea behind memory hierarchy. *Main Memory* is the physical memory that is visible to the instruction-set of the computer. *Cache*, as the name suggests, is a hidden storage. We already introduced the general idea of caches and its application at various levels of the computer system in Chapter 8 (please see Section 8.6) when we discussed TLBs, which are a special case of caches used for holding address translations. Specifically, in the context of the memory accesses made by the processor, the idea is to stash information brought from the memory in the cache. It is much smaller than the main memory and hence much faster.

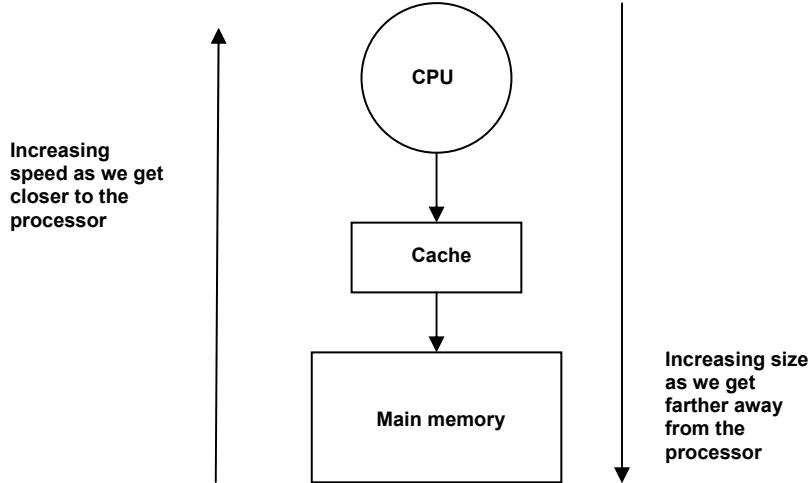


Figure 9.1: A basic memory hierarchy. The figure shows a two-level hierarchy. Modern processors may have a deeper hierarchy (up to 3 levels of caches followed by the main memory).

Our intent is as follows: The CPU looks in the cache for the data it seeks from the main memory. If the data is not there then it retrieves it from the main memory. If the cache is able to service most of the CPU requests then effectively we will be able to get the speed advantage of the cache.

9.2 Principle of Locality

Let us understand why a cache works in the first place. The answer is contained in the principles of locality that we introduced in the previous chapter (see Section 8.4.2).

Stated broadly, a program tends to access a relatively small region of memory irrespective of its actual memory footprint in any given interval of time. While the region of activity may change over time, such changes are gradual. The principle of locality zeroes in on this tendency of programs.

Principle of locality has two dimensions, namely, *spatial* and *temporal*. Spatial locality refers to the high probability of a program accessing *adjacent* memory locations ..., $i-3$, $i-2$, $i-1$, $i+1$, $i+2$, $i+3$, ... if it accesses a location i . This observation is intuitive. A program's instructions occupy contiguous memory locations. Similarly, data structures such as arrays and records occupy contiguous memory locations. Temporal locality refers to the high probability of a program accessing in the near future, the *same* memory location i that it is accessing currently. This observation is intuitive as well considering that a program may be executing a looping construct and thus revisiting the same instructions repeatedly and/or updating the same data structures repeatedly in an iterative algorithm. We will shortly find ways to exploit these locality properties in the cache design.

9.3 Basic terminologies

We will now introduce some intuitive definitions of terms commonly used to describe the performance of memory hierarchies. Before we do that, it would be helpful to remind ourselves of the toolbox and tool tray analogy from Chapter 2. If you needed a tool, you first went and looked in the tool tray. If it is there, it saved you a trip to the toolbox; if not you went to the garage where you keep the toolbox and bring the tool, use it, and put it in the tool tray. Naturally, it is going to be much quicker if you found the tool in the tool tray. Of course, occasionally you may have to return some of the tools back to the toolbox from the tool tray, when the tray starts overflowing. Mathematically speaking, one can associate a probability of finding the tool in the tool tray; one minus this probability gives the odds of going to the toolbox.

Now, we are ready to use this analogy for defining some basic terminologies.

- *Hit*: This term refers to the CPU finding the contents of the memory address in the cache thus saving a trip to the deeper levels of the memory hierarchy, and analogous to finding the tool in the tool tray. *Hit rate (h)* is the probability of such a *successful lookup* of the cache by the CPU.
- *Miss*: This term refers to the CPU *failing* to find what it wants in the cache thus incurring a trip to the deeper levels of the memory hierarchy, and analogous to taking a trip to the toolbox; *Miss rate (m)* is the probability of *missing* in the cache and is equal to $1-h$.
- *Miss penalty*: This is the time penalty associated with servicing a miss at any particular level of the memory hierarchy, and analogous to the time to go to the garage to fetch the missing tool from the tool box.
- *Effective Memory Access Time (EMAT)*: This is the effective access time experienced by the CPU.

This has two components:

- (a) time to lookup the cache to see if the memory location that the CPU is looking for is already there, defined as the *cache access time* or *hit time*, and
- (b) upon a miss in the cache, the time to go to the deeper levels of the memory hierarchy to fetch the missing memory location, defined as the *miss penalty*.

The CPU is always going to incur the first component on every memory access. The second component, namely, miss penalty is governed by the access time to the deeper levels of the memory hierarchy, and is measured in terms of the number of CPU clock cycles that the processor has to be idle waiting for the miss to be serviced. The miss penalty depends on a number of factors including the organization of the cache and the details of the main memory system design.

These factors will become more apparent in later sections of this chapter. Since the CPU incurs this penalty only on a miss, to compute the second component we need to condition the miss penalty with the probability (quantified by the miss rate m) that the memory location that the CPU is looking for is not presently in the cache.

Thus, if m , Tc , and Tm , be the cache miss rate, the cache access time, and the miss penalty, respectively.

$$EMAT = Tc + m * Tm \quad (1)$$

9.4 Multilevel Memory Hierarchy

As it turns out modern processors employ multiple levels of caches. For example, a state-of-the-art processor (circa 2006) has at least 2 levels of caches on-chip, referred to as *first-level (L1), second-level (L2) caches*. You may be wondering if both the first and second level caches are on-chip, why not make it a single big cache? The answer lies in the fact that we are trying to take care of two different concerns simultaneously: fast access time to the cache (i.e., hit time), and lower miss rate. On the one hand, we want the hit time to be as small as possible to keep pace with the clock cycle time of the processor. Since the size of the cache has a direct impact on the access time, this suggests that the cache should be small. On the other hand, the growing gap between the processor cycle time and main memory access time suggests that the cache should be large. Addressing these twin concerns leads to multilevel caches. The first level cache is optimized for speed to keep pace with the clock cycle time of the processor, and hence is small. The speed of the second level cache only affects the miss penalty incurred by the first level cache and does not directly affect the clock cycle time of the processor. Therefore, the second level cache design focuses on reducing the miss rate and hence is large. The processor sits in an integrated circuit board, referred to as *motherboard*, which has the main (physical) memory¹. High-end CPUs intended for enterprise class machines (database and web servers) even have a large off-chip *third-level cache (L3)*. For considerations of speed of access time, these multiple levels of the caches are all usually implemented with SRAM technology.

For reasons outlined above, the sizes of the caches become bigger as we move away from the processor. If S_i is the size of the cache at level i , then,

$$S_{i+n} > S_{i+n-1} > \dots > S_2 > S_1$$

Correspondingly, the access times also increase as we move from the processor. If T_i is the access time at level i , then,

$$T_{i+n} > T_{i+n-1} > \dots > T_2 > T_1$$

Generalizing the EMAT terminology to the memory hierarchy as a whole, let T_i and m_i be the access time and miss rate for any level of the memory hierarchy, respectively. The effective memory access time for any level i of the memory hierarchy is given by the recursive formula:

$$EMAT_i = T_i + m_i * EMAT_{i+1} \quad (2)$$

¹ The typical access times and sizes for these different levels of the cache hierarchy change with advances in technology.

We are now ready to generalize the concept of memory hierarchy:

Memory hierarchy is defined as all the storage containing either instructions and/or data that a processor accesses either directly or indirectly.

By direct access we mean that the storage is visible to the ISA. By indirect access we mean that it is not visible to the ISA. Figure 9.2 illustrates this definition. In Chapter 2, we introduced registers which are the fastest and closest data storage available to the processor for direct access from the ISA. Typically, load/store instructions, and arithmetic/logic instructions in the ISA access the registers. L1, L2, and L3 are different levels of caches that a processor (usually) implicitly accesses every time it accesses main memory for bringing in an instruction or data. Usually L1 and L2 are on-chip, while L3 is off-chip. The main memory serves as the storage for instructions and data of the program. The processor explicitly accesses the main memory for instructions (via the program counter), and for data (via load/store instructions, and other instructions that use memory operands). It should be noted that some architectures also allow direct access to the cache from the ISA by the processor (e.g. for flushing the contents of the cache). The secondary storage serves as the home for the entire memory footprint of the program, a part of which is resident in the main memory consistent with the working set principle that we discussed in Chapter 8. In other words, the secondary storage serves as the home for the virtual memory. The processor accesses the virtual memory implicitly upon a page fault to bring in the faulting virtual page into the main (i.e., physical) memory.

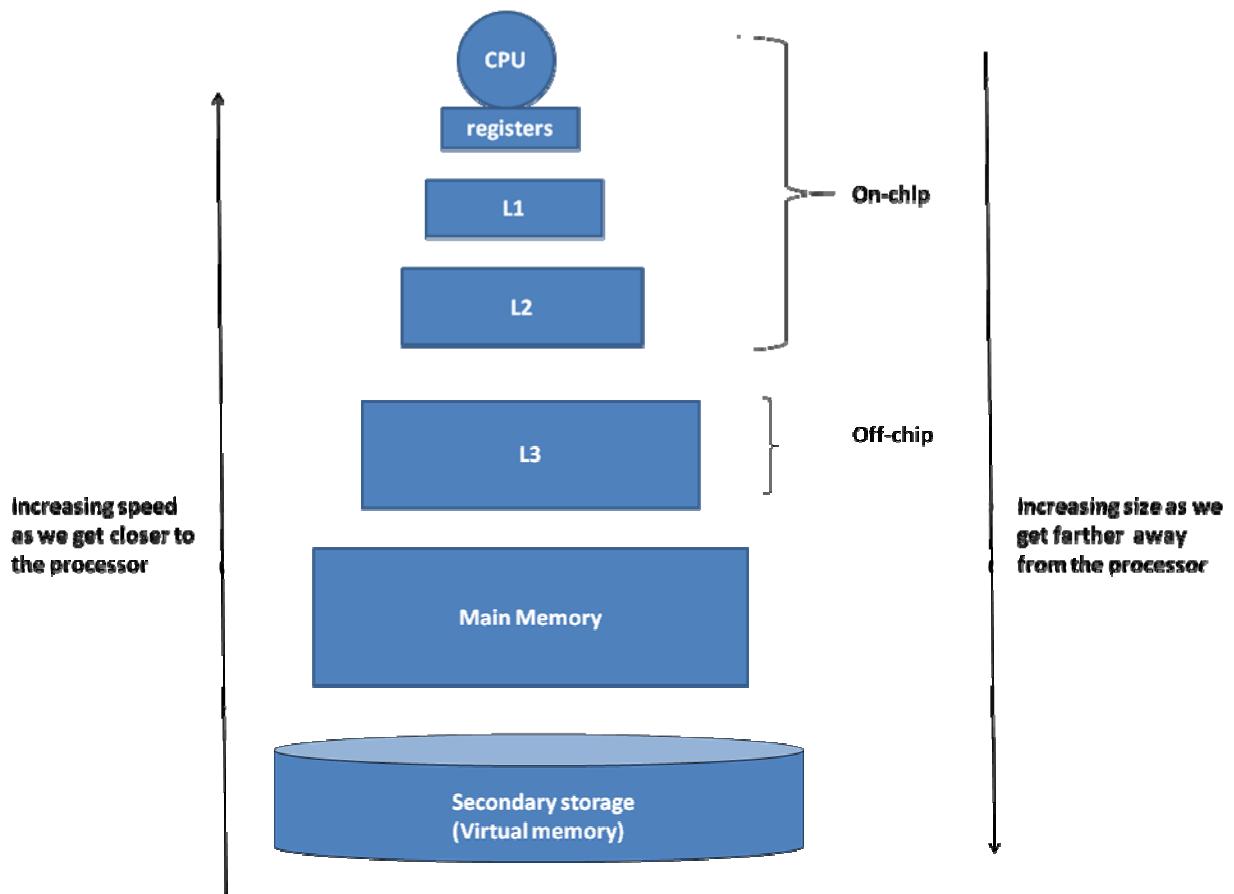


Figure 9.2: The entire memory hierarchy stretching from processor registers to the virtual memory.

This chapter focuses only on the portion of the memory hierarchy that includes the caches and main memory. We already have a pretty good understanding of the registers from earlier chapters (Chapters 2, 3, and 5). Similarly, we have a good understanding of the virtual memory from Chapters 7 and 8. In this chapter, we focus only on the caches and main memory. Consequently, we use the term *cache hierarchy* and *memory hierarchy* interchangeably to mean the same thing in the rest of the chapter.

Example 1:

Consider a three-level memory hierarchy as shown in Figure 9.3. Compute the effective memory access time.

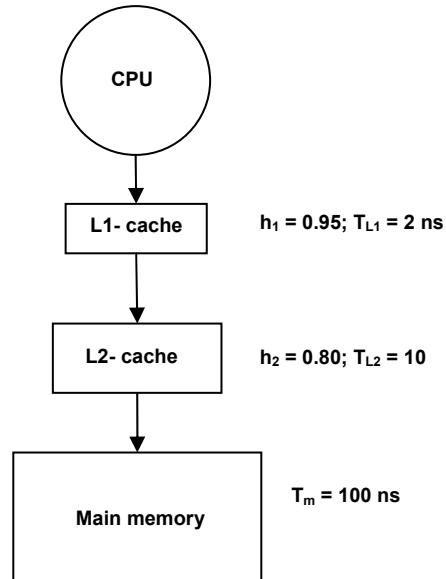


Figure 9.3: Three-level memory hierarchy

Answer:

$$\begin{aligned} \text{EMAT}_{L2} &= T_{L2} + (1 - h_2) * T_m \\ &= 10 + (1 - 0.8) * 100 \\ &= 30 \text{ ns} \end{aligned}$$

$$\begin{aligned} \text{EMAT}_{L1} &= T_{L1} + (1 - h_1) * \text{EMAT}_{L2} \\ &= 2 + (1 - 0.95) * 30 \\ &= 2 + 1,5 \\ &= 3.5 \text{ ns} \end{aligned}$$

EMAT = EMAT_{L1} = 3.5 ns

9.5 Cache organization

There are three facets to the organization of the cache: *placement*, *algorithm for lookup*, and *validity*.

These three facets deal with the following questions, respectively:

1. Where do we place in the cache the data read from the memory?
2. How do we find something that we have placed in the cache?
3. How do we know if the data in the cache is valid?

A mapping function that takes a given memory address to a cache index is the key to answering the first question. In addition to the mapping function, the second question concerns with the additional meta-data in each cache entry that helps identify the cache contents unambiguously. The third question brings out the necessity of providing a valid bit in each cache entry to assist the lookup algorithm.

We will look at these three facets specifically in the context of a simple cache organization, namely, a *direct-mapped* cache. In Section 9.11, we will look at other cache organizations, namely, *fully associative* and *set-associative*.

9.6 Direct-mapped cache organization

A direct-mapped cache has a one-to-one correspondence between a memory location and a cache location². That is, given a memory address there is exactly one place to put its contents in the cache. To understand how direct-mapping works, let us consider a very simple example, a memory with sixteen words and a cache with eight words (Figure 9.4). The shading in the figure shows the mapping of memory locations 0 to 7 to locations 0 to 7 of the cache, respectively; and similarly, locations 8 to 16 of the memory map to locations 0 to 7 of the cache, respectively.

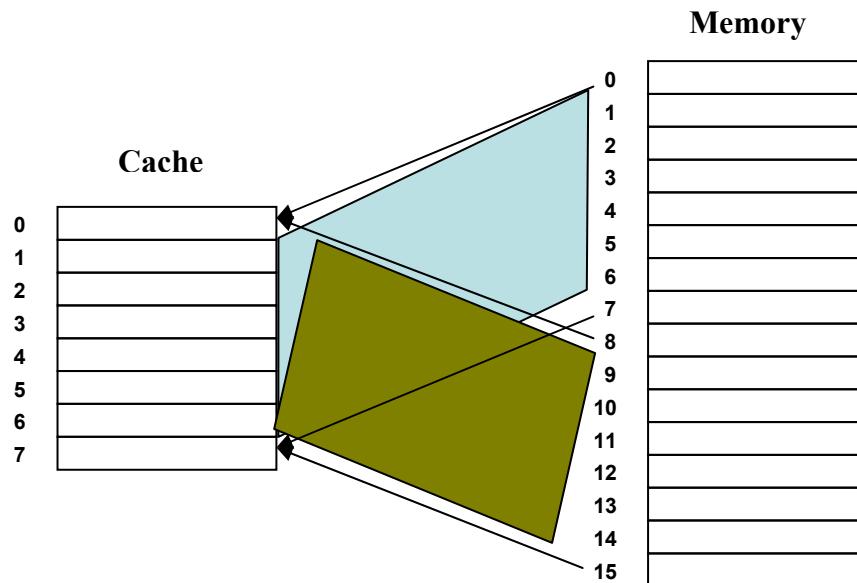


Figure 9.4: Direct-mapped cache

Before we explore the questions of lookup and validity, let us first understand the placement of memory locations in the cache with this direct mapping. To make the discussion concrete, let us assume the cache is empty and consider the following sequence of memory references:

0, 1, 2, 3, 1, 3, 0, 8, 0, 9, 10 (all decimal addresses)

Since the cache is initially empty, the first four references (addresses 0, 1, 2, 3) *miss* in the cache and the CPU retrieves the data from the memory and stashes them in the cache. Figure 9.5 shows the cache after servicing the first four memory references. These are inevitable misses, referred to as *compulsory misses*, since the cache is initially empty.

² Of course, since cache is necessarily smaller than memory, there is a many-to-one relationship between a set of memory locations and a given cache location.

Cache	
0	mem loc 0
1	mem loc 1
2	mem loc 2
3	mem loc 3
4	empty
5	empty
6	empty
7	empty

Figure 9.5: Content of Cache after the first 4 references

The next three CPU references (addresses 1, 3, 0) *hit* in the cache thus avoiding trips to the memory. Let us see what happens on the next CPU reference (address 8). This reference will *miss* in the cache and the CPU retrieves the data from the memory. Now we have to figure out where the system will stash the data in the cache from this memory location. Cache has space in locations 4-7. However, with direct-mapping cache location 0 is the only spot for storing memory location 8. Therefore, the cache has to evict memory location 0 to make room for memory location 8. Figure 9.6 shows this state of the cache. This is also a *compulsory miss* since memory location 8 is not in the cache to start with.

Cache	
0	mem loc 0 / 8
1	mem loc 1
2	mem loc 2
3	mem loc 3
4	empty
5	empty
6	empty
7	empty

Figure 9.6: Memory location 0 replaced by 8

Consider the next reference (address 0). This reference also *misses* in the cache and the CPU has to retrieve it from the memory and stash it in cache location 0, which is the only spot for memory location 0. Figure 9.7 shows the new contents of the cache. This miss occurs due to the *conflict* between memory location 0 and 8 for a spot in the cache, and hence referred to as a *conflict miss*. A conflict miss occurs due to direct mapping despite the fact that there are unused locations available in the cache. Note that the previous miss (location 8 in Figure 9.6) also caused a conflict since location 0 was already present in that cache entry. Regardless of this situation, first access to a memory location will always result in a miss which is why we categorized the miss in Figure 9.6 as a compulsory miss. We will revisit the different types of misses in some more detail in a later section (Section 9.12).

Cache	
0	mem loc 0 8 0
1	mem loc 1
2	mem loc 2
3	mem loc 3
4	empty
5	empty
6	empty
7	empty

Figure 9.7: Memory location 8 replaced by 0 (conflict miss)

9.6.1 Cache Lookup

By now, we know the information exchange or the *handshake* between the CPU and the memory: the CPU supplies the address and the command (e.g. read) and gets back the data from the memory. The introduction of the cache (Figure 9.1) changes this simple set up. The CPU looks up the cache first to see if the data it is looking for is in the cache. Only on a miss does the CPU resort to its normal CPU-memory handshake.

Let us understand how the CPU *looks up* the cache for the memory location of interest. Specifically, we need to figure out what *index* the CPU has to present to the cache with a direct-mapped organization. Looking back at Figure 9.4, one can see how the CPU addresses map to their corresponding cache *indices*.

One can compute the numerical value of the cache index as:

$$\text{Memory-address} \bmod \text{cache-size}$$

For example, given the memory address 15 the cache index is

$$15 \bmod 8 = 7,$$

Similarly the cache index for memory address 7 is

$$7 \bmod 8 = 7.$$

Essentially, to construct the cache index we simply take the least significant bits of the memory address commensurate with the cache size. In our previous example, since the cache has 8 entries, we would need 3 bits for the cache index (the least significant three bits of the memory address).

Suppose the CPU needs to get data from memory address 8; 1000 is the memory address in binary; the cache index is 000. The CPU looks up the cache location at index 000. We need some way of knowing if the contents of this cache entry are from memory location 0 or 8. Therefore, in addition to the data itself, we need information in each entry of the cache to distinguish between multiple memory addresses that may map to the same cache entry. The bits of the memory address that were “dropped” to generate the cache index are exactly the information needed for this purpose. We refer to this additional information (which will be stored in the cache) as the *tag*. For example, the most

significant bit of the memory address 0000 and 1000 are 0 and 1 respectively. Therefore, our simple cache needs a 1-bit tag with each cache entry (Figure 9.8). If the CPU wants to access memory location 11, it looks up location $11 \bmod 8$ in the cache, i.e., location 3 in the cache. This location contains a tag value of 0. Therefore, the data contained in this cache entry corresponds to memory location 3 (binary address 0011) and not that of 11 (binary address 1011).

	tag	data
0	1	mem loc 0 8
1	0	mem loc 1
2	0	mem loc 2
3	0	mem loc 3
4		empty
5		empty
6		empty
7		empty

Figure 9.8: Direct-mapped cache with a tag field and a data field in each entry

Suppose the CPU generates memory address 0110 (memory address 6). Let us assume this is the first time memory address 6 is being referenced by the CPU. So, we know it cannot be in the cache. Let us see the sequence of events here as the CPU tries to read memory location 6. The CPU will first look up location $6 \bmod 8$ in the cache. If the tag happens to be 0, then the CPU will assume that the data corresponds to memory location 6. Well, the CPU did not ever fetch that location from memory so far in our example; it is by chance that the tag is 0. Therefore, the data contained in this cache entry does not correspond to the actual memory location 6. One can see that this is erroneous, and points to the need for additional information in each cache entry to avoid this error. The tag is useful for disambiguation of the memory location currently in the cache, but it does not help in knowing if the entry is *valid*. To fix this problem, we add a *valid* field to each entry in the cache (Figure 9.9).

	valid	tag	data
0	1	1	loc 8
1	1	0	loc 1
2	1	0	loc 2
3	1	0	loc 3
4	0	X	empty
5	0	X	empty
6	0	X	empty
7	0	X	empty

Figure 9.9: Direct-mapped cache with valid field, tag field, and data field in each entry. A value of “X” in the tag field indicates a “don’t care” condition.

9.6.2 Fields of a Cache Entry

To summarize, each cache entry contains three fields (Figure 9.10)



Figure 9.10: Fields of each cache entry

Thus, the memory address generated by the CPU has two parts from the point of view of looking up the cache: *tag* and *index*. Index is the specific cache location that could contain the memory address generated by the CPU; tag is the portion of the address to disambiguate contents of the specific cache entry (Figure 9.11).



Figure 9.11: Interpreting the memory address generated by the CPU for Cache Lookup

We use the least significant (i.e., the right most) bits of the memory address as cache index to take advantage of the principle of spatial locality. For example, in our simple cache, if we use the most significant 3 bits of the memory address to index the cache, then memory locations 0 and 1 will be competing for the same spot in the cache.

For example, consider the access sequence for memory addresses 0, 1, 0, 1, with the 8-entry direct-mapped cache as shown in Figure 9.4. Assume that the most significant three bits of the memory address are used as the cache index and the least significant bit (since the memory address is only 4-bits for this example) is used as the tag. Figure 9.12 shows the contents of the cache after each access. Notice how the same cache entry (first row of the cache) is reused for the sequence of memory accesses though the rest of the cache is empty. Every access results in a miss, replacing what was in the cache previously due to the access pattern.

	valid	tag	data		valid	tag	data		valid	tag	data		valid	tag	data
0	1	0	loc 0	0	1	1	loc.0 1	0	1	0	loc.0 1 0	0	1	1	loc.0 1 0
1	0	X	empty	1	0	X	empty	1	0	X	empty	1	0	X	empty
2	0	X	empty	2	0	X	empty	2	0	X	empty	2	0	X	empty
3	0	X	empty	3	0	X	empty	3	0	X	empty	3	0	X	empty
4	0	X	empty	4	0	X	empty	4	0	X	empty	4	0	X	empty
5	0	X	empty	5	0	X	empty	5	0	X	empty	5	0	X	empty
6	0	X	empty	6	0	X	empty	6	0	X	empty	6	0	X	empty
7	0	X	empty	7	0	X	empty	7	0	X	empty	7	0	X	empty
Access location 0				Access location 1				Access location 0				Access location 1			

Figure 9.12: Sequence of memory accesses if the cache index is chosen as the most significant bits of the memory address

The situation shown in Figure 9.12 is undesirable. Recall the locality properties (spatial and temporal), which we mentioned in Section 9.2. It is imperative that sequential access to memory fall into different cache locations. This is the reason for choosing to interpret the memory address as shown in Figure 9.11.

9.6.3 Hardware for direct mapped cache

Let us put together the ideas we have discussed thus far. Figure 9.13 shows the hardware organization for a direct-mapped cache.

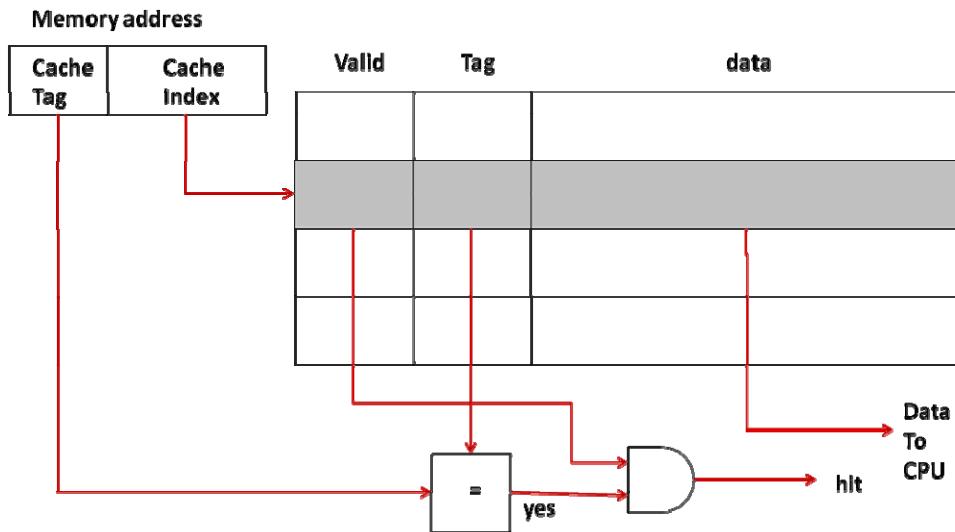


Figure 9.13: Hardware for direct-mapped cache. The shaded entry is the cache location picked out by the cache index.

The index part of the memory address picks out a unique entry in the cache (the textured cache entry in Figure 9.13). The comparator shown in Figure 9.13 compares the tag field of this entry against the tag part of the memory address. If there is a *match* and *if* the entry is *valid* then it signals a *hit*. The cache supplies the data field of the selected entry (also referred to as *cache line*) to the CPU upon a hit. *Cache block* is another term used synonymously with cache line. We have used three terms synonymously thus far: cache entry, cache line, and cache block. While this is unfortunate, it is important that the reader develop this vocabulary since computer architecture textbooks tend to use these terms interchangeably.

Note that the actual amount of storage space needed in the cache is more than that needed simply for the *data* part of the cache. The *valid* bits, and *tag* fields, called *meta-data*, are for managing the actual data contained in the cache, and represent a *space overhead*.

Thus far, we have treated the data field of the cache to hold one memory location. The size of the memory location depends on the granularity of memory access allowed by the instruction-set. For example, if the architecture is byte-addressable then a byte is the smallest possible size of a memory operand. Usually in such an architecture, the word-width is some integral number of bytes. We could place each byte in a separate cache

line, but from Section 9.2, we know that the principle of spatial locality suggests that if we access a byte of a word then there is a good chance that we would access other bytes of the same word. Therefore, it would make sense to design the cache to contain a complete word in each cache line even if the architecture is byte addressable. Thus, the memory address generated by the CPU would be interpreted as consisting of three fields as shown below: cache tag, cache index, and byte offset.



Figure 9.14: Interpreting the memory address generated by the CPU when a single cache block contains multiple bytes

Byte offset is defined as the bits of the address that specify the byte within the word. For example, if the word-width is 32-bits and the architecture is byte addressable then the bottom 2-bits of the address form the byte offset.

Example 2:

Let us consider the design of a direct-mapped cache for a realistic memory system. Assume that the CPU generates a 32-bit byte-addressable memory address. Each memory word contains 4 bytes. A memory access brings in a full word into the cache. The direct-mapped cache is 64K Bytes in size (this is the amount of data that can be stored in the cache), with each cache entry containing one word of data. Compute the additional storage space needed for the valid bits and the tag fields of the cache.

Answer:

Assuming little-endian notation, 0 is the least significant bit of the address. With this notation, the least significant two bits of the address, namely, bits 1 and 0 specify the byte within a word address. A cache entry holds a full word of 4 bytes. Therefore, the least significant two bits of the address, while necessary for uniquely identifying a byte within a word, are not needed to uniquely identify the particular cache entry. Therefore, these bits do not form part of the index for cache lookup.

The ratio of the cache size to the data stored in each entry gives the number of cache entries:

$$64\text{K Bytes} / (4 \text{ Bytes/word}) = 16 \text{ K entries}$$

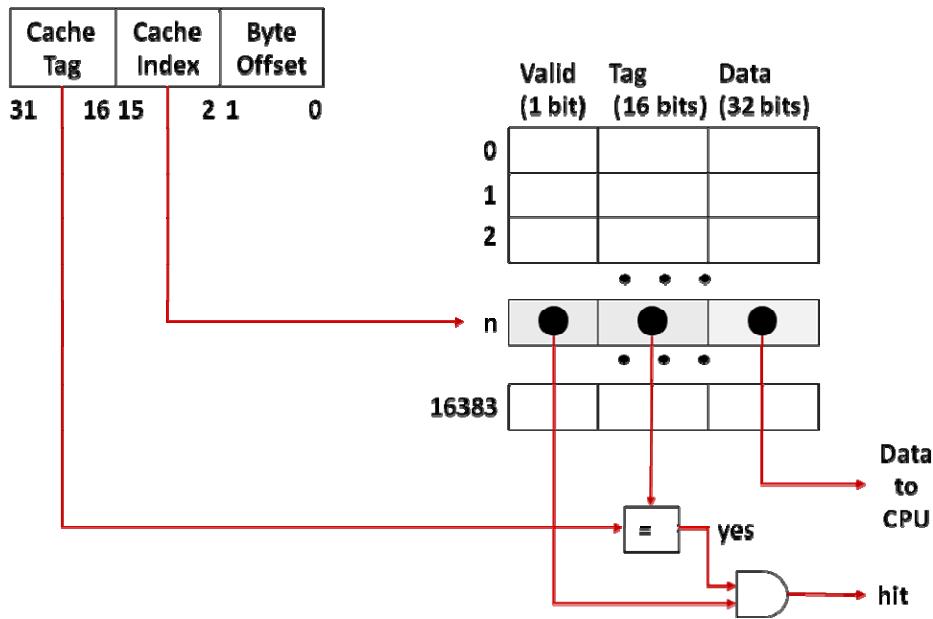
16 K entries require 14 bits to enumerate thus bits 2-15 form the cache index, leaving bits 16-31 to form the tag. Thus, each cache entry has a 16-bit tag.

The meta-data per entry totals 16 bits for the tag + 1 bit for valid bit = 17 bits

Thus, the additional storage space needed for the meta-data:

$$17 \text{ bits} \times 16\text{K entries} = 17 \times 16,384 = \mathbf{278,528 \text{ bits.}}$$

The following figure shows the layout of the cache for this problem.



Total space needed for the cache (actual data + meta-data)

$$= 64\text{K bytes} + 278,528$$

$$= 524,288 + 278,528$$

$$= 802,816$$

The space overhead = meta-data/total space = $278,528/802,816 = 35\%$

Let us see how we can reduce the space overhead. In Example 2, each cache line holds one memory word. One way of reducing the space overhead is to modify the design such that each cache line holds multiple contiguous memory words. For example, consider each cache line holding four contiguous memory words in Example 2. This would reduce the number of cache lines to 4K. *Block size* is the term used to refer to the amount of contiguous data in one cache line. Block size in Example 2 is 4 bytes. If each cache line were to contain 4 words, then the block size would be 16 bytes. Why would we want to have a larger block size? Moreover, how does it help in reducing the space overhead? Will it help in improving the performance of the memory system as a whole? We will let the reader ponder these questions for a while. We will revisit them in much more detail in Section 9.10 where we discuss the impact of block size on cache design.

9.7 Repercussion on pipelined processor design

With the cache memory introduced between the processor and the memory, we can return to our pipelined processor design and re-examine instruction execution in the presence of caches. Figure 9.15 is a reproduction of the pipelined processor from Chapter 6 (Figure 6.6)

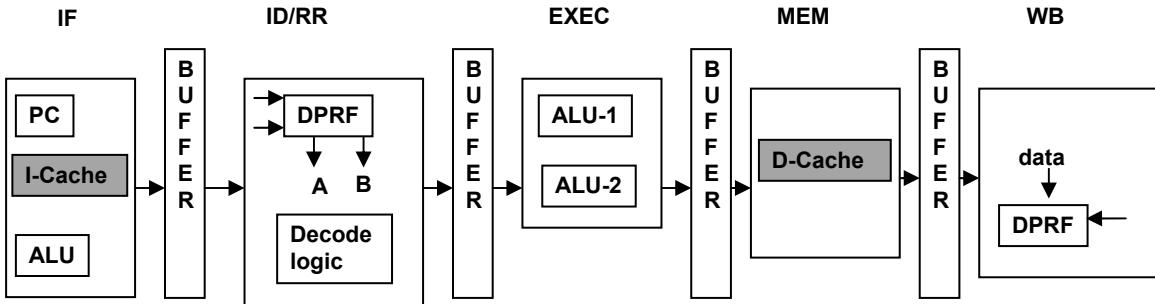


Figure 9.15: Pipelined processor with caches

Notice that we have replaced the memories, I-MEM and D-MEM in the IF and MEM stages, by caches I-Cache and D-Cache, respectively. The caches make it possible for the IF and MEM stages to have comparable cycle times to the other stages of the pipeline, assuming the references result in hits in the respective caches. Let us see what would happen if the references miss in the caches.

- **Miss in the IF stage:** Upon a miss in the I-Cache, the IF stage sends the reference to the memory to retrieve the instruction. As we know, the memory access time may be several 10's of CPU cycles. Until the instruction arrives from the memory, the IF stage sends NOPs (bubbles) to the next stage.
- **Miss in the MEM stage:** Of course, misses in the D-Cache are relevant only for memory reference instructions (load/store). Similar to the IF stage, a miss in the MEM stage results in NOPs to the WB stage until the memory reference completes. It also *freezes* the preceding stages from advancing past the instructions they are currently working on.

We define *memory stall* as the processor cycles wasted due to waiting for a memory operation to complete. Memory stalls come in two flavors: *read stall* may be incurred due to read access from the processor to the cache; *write stall* may be incurred during write access from the processor to the cache. We will define and elaborate on these stalls in the next section together with mechanisms to avoid them since they are detrimental to processor pipeline performance.

9.8 Cache read/write algorithms

In this section, we discuss policies and mechanisms for reading and writing the caches. Different levels of the cache hierarchy may choose different policies.

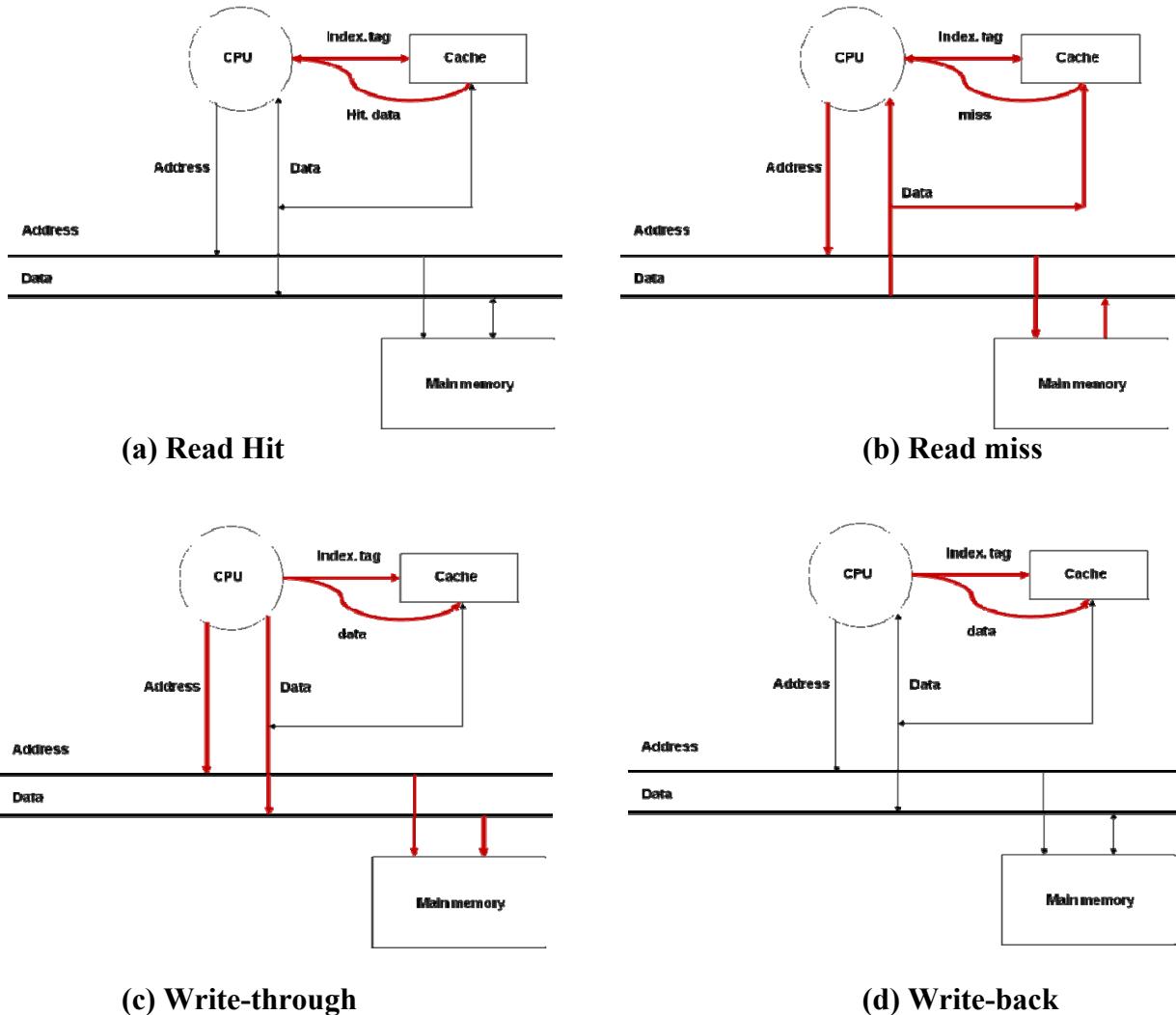


Figure 9.16: CPU, Cache, Memory interactions for reads and writes

9.8.1 Read access to the cache from the CPU

A processor needs to access the cache to read a memory location for either instructions or data. In our 5-stage pipeline for implementing the LC-2200 ISA, this could happen either for instruction fetch in the IF stage, or for operand fetch in response to a load instruction in the MEM stage. The basic actions taken by the processor and the cache are as follows:

- **Step 1:** CPU sends the index part of the memory address (Figure 9.16) to the cache. The cache does a lookup, and if successful (a cache *hit*), it supplies the data to the CPU. If the cache signals a miss, then the CPU sends the address on the memory bus to the main memory. In principle, all of these actions happen in the same cycle (either the IF or the MEM stage of the pipeline).
- **Step 2:** Upon sending the address to the memory, the CPU sends NOPs down to the subsequent stage until it receives the data from the memory. *Read stall* is defined as the number of processor clock cycles wasted to service a read-miss. As we observed earlier, this could take several CPU cycles depending on the memory speed. The

cache allocates a cache block to receive the memory block. Eventually, the main memory delivers the data to the CPU and simultaneously updates the allocated cache block with the data. The cache modifies the tag field of this cache entry appropriately and sets the valid bit.

9.8.2 Write access to the cache from the CPU

Write requests to the cache from the processor are the result of an instruction that wishes to write to a memory location. In our 5-stage pipeline for implementing the LC-2200 ISA, this could happen for storing a memory operand in the MEM stage. There are a couple of choices for handling processor write access to the cache: *write through* and *write back*.

9.8.2.1 Write through policy

The idea is to update the cache and the main memory on each CPU write operation. The basic actions taken by the processor and the cache are as follows:

- **Step 1:** On every write (store instruction in LC-2200), the CPU simply writes to the cache. There is no need to check the valid bit or the cache tag. The cache updates the tag field of the corresponding entry and sets the valid bit. These actions happen in the MEM stage of the pipeline.
- **Step 2:** Simultaneously, the CPU sends the address and data to the main memory. This of course is problematic in terms of performance since memory access takes several CPU cycles to complete. To alleviate this performance bottleneck, it is customary to include a *write-buffer* in the datapath between the CPU and the memory bus (shown in Figure 9.17). This is a small hardware store (similar to a register file) to smoothen out the speed disparity between the CPU and memory. As far as the CPU is concerned, the write operation is complete as soon as it places the address and data in the write buffer. Thus, this action also happens in the MEM stage of the pipeline without stalling the pipeline.
- **Step 3:** The write-buffer completes the write to the main memory independent of the CPU. Note that, if the write buffer is full at the time the processor attempts to write to it, then the pipeline will stall until one of the entries from the write buffer has been sent to the memory. *Write stall* is defined as the number of processor clock cycles wasted due to a write operation (regardless of hit or a miss in the cache).

With the write-through policy, write stall is a function of the cache block allocation strategy upon a write miss. There are two choices:

- **Write allocate:** This is the usual way to handle write misses. The intuition is that the data being written to will be needed by the program in the near future. Therefore, it is judicious to put it in the cache as well. However, since the block is missing from the cache, we have to allocate a cache block and bring in the missing memory block into it. In this sense, a write-miss is treated exactly similar to a read-miss. With a direct-mapped cache, the cache block to receive the memory block is pre-determined. However, as we will see later with flexible placement (please see Section 9.11), the allocation depends on the other design considerations.

- **No-write allocate:** This is an unusual way of handling a write miss. The argument for this strategy is that the write access can complete quickly since the processor does not need the data. The processor simply places the write in the write buffer to complete the operation needed in the MEM stage of the pipeline. Thus, there are write stalls incurred since the missing memory block need not be brought from the memory.

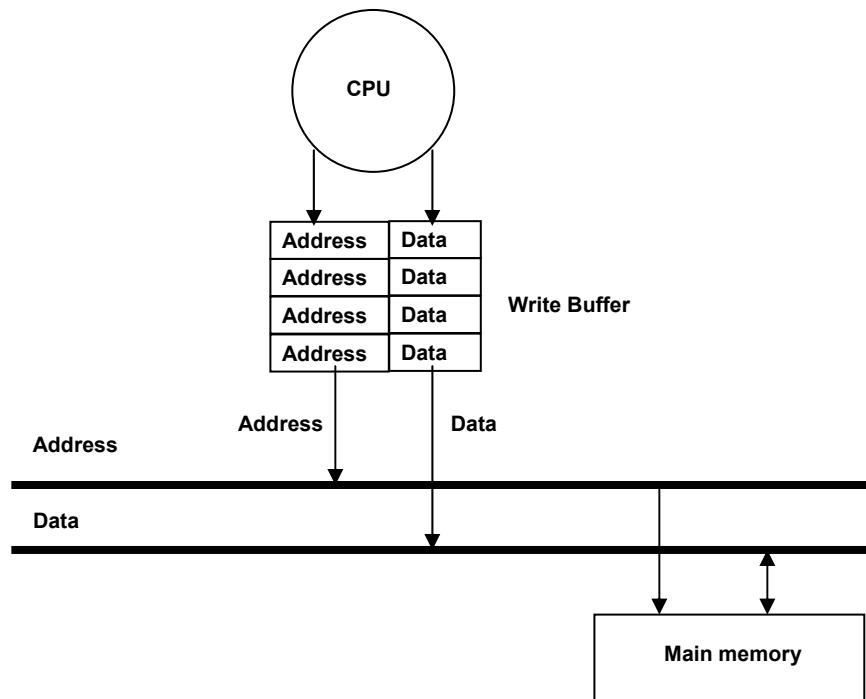


Figure 9.17: A 4-element write-buffer to bridge the CPU and main memory speeds for a write through policy. The processor write is complete and the pipeline can resume as soon as the write has been placed in the write buffer. It will be sent out to the memory in the background in parallel with the pipeline operation. The pipeline will freeze if the write buffer is full.

9.8.2.2 Write back policy

The idea here is to update only the cache upon a CPU write operation. The basic actions taken by the processor and the cache are as follows:

- **Step 1:** The CPU-cache interaction is exactly similar to the write-through policy. Let us assume that the memory location is already present in the cache (i. e., a write hit). The CPU writes to the cache. The cache updates the tag field of the corresponding entry and sets the valid bit. These actions happen in the MEM stage of the pipeline.
- **Step 2:** The contents of this chosen cache entry and the corresponding memory location are inconsistent with each other. As far as the CPU is concerned this is not a problem since it first checks the cache before going to memory on a read operation. Thus, the CPU always gets the latest value from the cache.

- **Step 3:** Let us see when we have to update the main memory. By design, the cache is much smaller than the main memory. Thus, at some point it may become necessary to replace an existing cache entry to make room for a memory location not currently present in the cache. We will discuss cache replacement policies in a later section of this chapter (Section 9.14). At the time of replacement, if the CPU had written into the cache entry chosen for replacement, then the corresponding memory location has to be updated with the latest data for this cache entry.

The processor treats a write-miss exactly like a read-miss. After taking the steps necessary to deal with a read-miss, it completes the write action as detailed above.

We need some mechanism by which the cache can determine that the data in a cache entry is more up to date than the corresponding memory location. The *meta-data* in each cache entry (valid bit and tag field) do not provide the necessary information for the cache to make this decision. Therefore, we add a new *meta-data* to each cache entry, a *dirty bit* (see Figure 9.18).

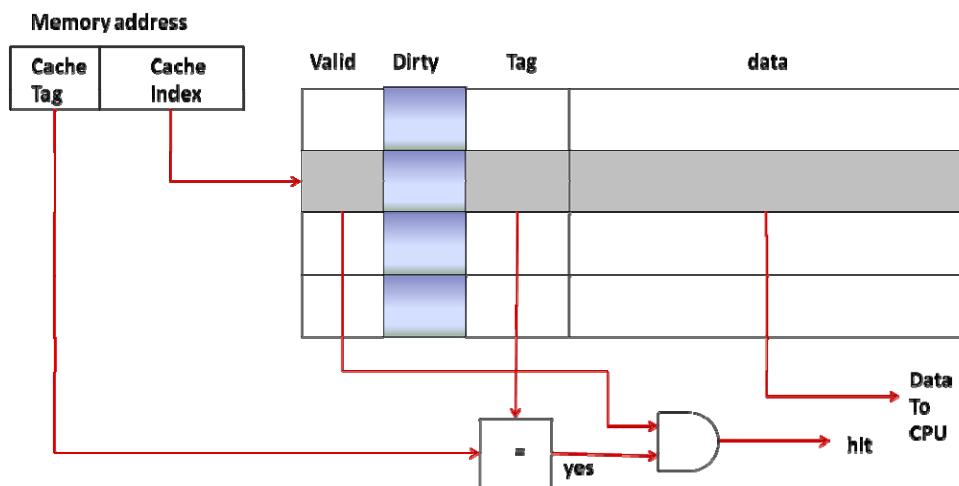


Figure 9.18: A direct-mapped cache organization with write-back policy. Each entry has an additional field (dirty bit) to indicate whether the block is dirty or clean. Horizontally shaded block is the one selected by the cache index.

The cache uses the dirty bit in the following manner:

- The cache *clears* the bit upon processing a miss that brings in a memory location into this cache entry.
 - The cache *sets* the bit upon a CPU write operation.
 - The cache *writes back* the data in the cache into the corresponding memory location upon replacement. Note that this action is similar to writing back a physical page frame to the disk in the virtual memory system (see Chapter 8).

We introduced the concept of a write-buffer in the context of write-through policy. It turns out that the write-buffer is useful for the write-back policy as well. Note that the focus is on keeping the processor happy. This means that the missing memory block should be brought into the cache as quickly as possible. In other words, servicing the

miss is more important than writing back the dirty block that is being replaced. The write-buffer comes in handy to give preferential treatment to reading from the memory as compared to writing to memory. The block that has to be replaced (if dirty) is placed in the write buffer. It will eventually be written back. But the immediate request that is sent to the memory is to service the current miss (be it a read or a write miss). The write requests from the write-buffer can be sent to the memory when there are no read or write misses to be serviced from the CPU. This should be reminiscent of the optimization that the memory manager practices to give preferential treatment for reading in missing pages from the disk over writing out dirty victim pages out to the disk (see Section 8.4.1.1).

9.8.2.3 Comparison of the write policies

Which write policy should a cache use? The answer to this depends on several factors. On the one hand, write through ensures that the main memory is always up to date. However, this comes at the price of sending every write to memory. Some optimizations are possible. For example, if it is a repeated write to the same memory location that is already in the write buffer (i. e., it has not yet been sent to the memory), then it can be replaced by the new write. However, the chance of this occurring in real programs is quite small. Another optimization, referred to as *write merging*, merges independent writes to different parts of the same memory block. Write back has the advantage that it is faster, and more importantly, does not use the memory bus for every write access from the CPU. Thus, repeated writes to the same memory location does not result in creating undue traffic on the memory bus. Only upon a replacement, does the cache update the memory with the latest content for that memory location. In this context, note that the write-buffer is useful for the write-back policy as well, to hold the replacement candidate needing a write-back to the main memory.

Another advantage enjoyed by the write through policy is the fact that the cache is always clean. In other words, upon a cache block replacement to make room for a missing block, the cache never has to write the replaced block to the lower levels of the memory hierarchy. This leads to a simpler design for a write through cache and hence higher speed than a write back cache. Consequently, if a processor has multilevel cache hierarchy, then it is customary to use write through for the closer level to the processor. For example, most modern processors use a write through policy for the L1 D-cache, and a write back policy for the L2 and L3 levels.

As we will see in later chapters (see Chapter 10 and 12), the choice of write policy has an impact on the design of I/O and multiprocessor systems. The design of these systems could benefit from the reduced memory bus traffic of the write back policy; at the same time, they would also benefit from the write through policy keeping the memory always up to date.

9.9 Dealing with cache misses in the processor pipeline

Memory accesses disrupt the smooth operation of the processor pipeline. Therefore, we need to mitigate the ill effect of misses in the processor pipeline. It is not possible to mask a miss in the IF stage of the pipeline. However, it may be possible to *hide* a miss in the MEM stage.

- **Read miss in the MEM stage:** Consider a sequence of instructions shown below:

```

I1: ld      r1, a      ;    r1 <- memory location a
I2: add    r3, r4, r5  ;    r3 <- r4 + r5
I3: and    r6, r7, r8  ;    r6 <- r7 AND r8
I4: add    r2, r4, r5  ;    r2 <- r4 + r5
I5: add    r2, r1, r2  ;    r2 <- r1 + r2

```

Suppose the **ld** instruction results in a miss in the D-Cache; the CPU has to stall this load instruction in the MEM stage (freezing the preceding stages and sending NOPs to the WB stage) until the memory responds with the data. However, I5 is the instruction that uses the value loaded into **r1**. Let us understand how we can use this knowledge to prevent stalling the instructions I2, I3, and I4.

In Chapter 5, we introduced the idea of a *busy* bit with each register in the register file for dealing with hazards. For instructions that modify a register value, the bit is set in the ID/RR stage (see Figure 9.15) and cleared once the write is complete. This idea is extensible to dealing with memory loads as well.

- **Write miss in the MEM stage:** This could be problematic depending on the write policy as well as the cache allocation policy. First, let us consider the situation with write-through policy. If the cache block allocation policy is no-write allocate, then the pipeline will not incur any stalls thanks to the write buffer. The processor simply places the write in the write-buffer to complete the actions needed in the MEM stage. However, if the cache block allocation policy is write-allocate then it has to be handled exactly similar to a read-miss. Therefore, the processor pipeline will incur write stalls in the MEM stage. The missing data block has to be brought into the cache, before the write operation can complete despite the presence of the write buffer. For write-back policy, write stalls in the MEM stage are inevitable since a write-miss has to be treated exactly similar to a read-miss.

9.9.1 Effect of memory stalls due to cache misses on pipeline performance

Let us re-visit the program execution time. In Chapter 5, we defined it to be:

$$\text{Execution time} = \text{Number of instructions executed} * \text{CPI}_{\text{Avg}} * \text{clock cycle time}$$

A pipelined processor attempts to make the CPI_{Avg} equal 1 since it tries to complete an instruction in every cycle. However, structural, data, and control hazards induce bubbles in the pipeline thus inflating the CPI_{Avg} to be higher than 1.

Memory hierarchy exacerbates this problem even more. Every instruction has at least one memory access, namely, to fetch the instruction. In addition, there may be additional accesses for memory reference instructions. If these references result in *misses* then they force bubbles in the pipeline. We refer to these additional bubbles due to the memory hierarchy as the *memory stall cycles*.

Thus, a more accurate expression for the execution time is:

$$\text{Execution time} = (\text{Number of instructions executed} * (\text{CPI}_{\text{Avg}} + \text{Memory-stalls}_{\text{Avg}})) * \text{clock cycle time} \quad (3)$$

We can define an effective CPI of the processor as:

$$\text{Effective CPI} = \text{CPI}_{\text{Avg}} + \text{Memory-stalls}_{\text{Avg}} \quad (4)$$

The total memory stalls experienced by a program is:

$$\text{Total memory stalls} = \text{Number of instructions} * \text{Memory-stalls}_{\text{Avg}} \quad (5)$$

The average number of memory stalls per instruction is:

$$\text{Memory-stalls}_{\text{Avg}} = \text{misses per instruction}_{\text{Avg}} * \text{miss-penalty}_{\text{Avg}} \quad (6)$$

Of course, if reads and writes incur different miss-penalties, we may have to account for them differently (see Example 3).

Example 3:

Consider a pipelined processor that has an average CPI of 1.8 without accounting for memory stalls. I-Cache has a hit rate of 95% and the D-Cache has a hit rate of 98%. Assume that memory reference instructions account for 30% of all the instructions executed. Out of these 80% are loads and 20% are stores. On average, the read-miss penalty is 20 cycles and the write-miss penalty is 5 cycles. Compute the effective CPI of the processor accounting for the memory stalls.

Answer:

The solution to this problem uses the equations (4) and (6) above.

$$\begin{aligned} \text{Cost of instruction misses} &= \text{I-cache miss rate} * \text{read miss penalty} \\ &= (1 - 0.95) * 20 \\ &= 1 \text{ cycle per instruction} \end{aligned}$$

$$\begin{aligned} \text{Cost of data read misses} &= \text{fraction of memory reference instructions in the program} * \\ &\quad \text{fraction of memory reference instructions that are loads} * \\ &\quad \text{D-cache miss rate} * \text{read miss penalty} \\ &= 0.3 * 0.8 * (1 - 0.98) * 20 \\ &= 0.096 \text{ cycles per instruction} \end{aligned}$$

$$\begin{aligned} \text{Cost of data write misses} &= \text{fraction of memory reference instructions in the program} * \\ &\quad \text{fraction of memory reference instructions that are stores} * \\ &\quad \text{D-cache miss rate} * \text{write miss penalty} \\ &= 0.3 * 0.2 * (1 - 0.98) * 5 \\ &= 0.006 \text{ cycles per instruction} \end{aligned}$$

$$\begin{aligned} \text{Effective CPI} &= \text{base CPI} + \text{Effect of I-Cache on CPI} + \text{Effect of D-Cache on CPI} \\ &= 1.8 + 1 + 0.096 + 0.006 = \mathbf{2.902} \end{aligned}$$

Reducing the *miss rate* and reducing the *miss penalty* are the keys to reducing the memory stalls, and thereby increase the efficiency of a pipelined processor.

There are two avenues available for reducing the miss rate and we will cover them in Sections 9.10 and 9.11. In Section 9.12, we will discuss ways to decrease the miss penalty.

9.10 Exploiting spatial locality to improve cache performance

The first avenue to reducing the miss rate exploits the principle of spatial locality. The basic idea is to bring adjacent memory locations into the cache upon a miss for a memory location i . Thus far, each entry in the cache corresponds to the unit of memory access architected in the instruction set. To exploit spatial locality in the cache design, we decouple the *unit of memory access* by an instruction from the *unit of memory transfer* between the memory and the cache. The instruction-set architecture of the processor decides the unit of memory access. For example, if an architecture has instructions that manipulate individual bytes, then the unit of memory access for such a processor would be a byte. On the other hand, the unit of memory transfer is a design parameter of the memory hierarchy. In particular, this parameter is always some integral multiple of the unit of memory access to exploit spatial locality. We refer to the unit of transfer between the cache and the memory as the *block size*. Upon a miss, the cache brings an entire *block* of *block size* bytes that contains the missing memory reference.

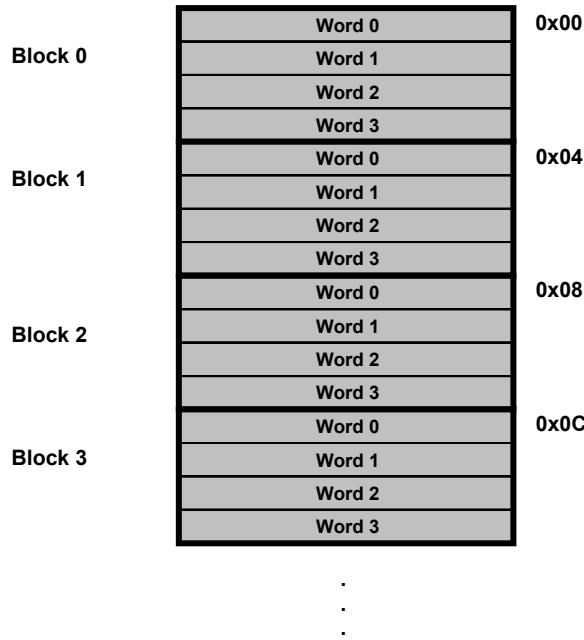


Figure 9.19: Main memory viewed as cache blocks

Figure 9.19 shows an example of such an organization and view of the memory. In this example, the block size is 4 words, where a word is the unit of memory access by a CPU instruction. The memory addresses for a block starts on a block boundary as shown in the Figure. For example, if the CPU misses on a reference to a memory location 0x01,

then the cache brings in 4 memory words that comprises *block 0* (starting at address 0x00), and contains the location 0x01.

With each entry in the cache organized as blocks of contiguous words, the address generated by the CPU has *three* parts: *tag*, *index*, *block-offset* as shown in Figure 9.20.

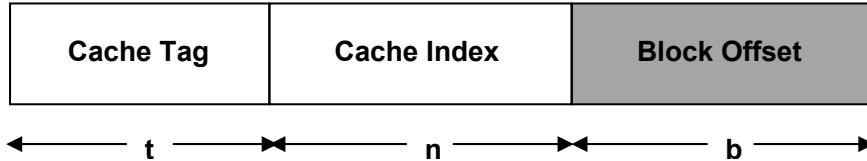


Figure 9.20: Interpreting the memory address generated by the CPU for a cache block containing multiple words

Note that the block offset is the number of bits necessary to enumerate the set of contiguous memory locations contained in a cache block. For example, if the block size is 64 bytes, then the block offset is 6-bits. All the data of a given block are contained in one cache entry. The tag and index fields have the same meaning as before. We will present general expressions for computing the number of bits needed for the different fields shown in Figure 9.20.

Let a be the number of bits in the memory address, S be the total size of the cache in bytes, and B the block size.

We have the following expressions for the fields shown in Figure 9.20:

$$b = \log_2 B \quad (7)$$

$$L = S/B \quad (8)$$

$$n = \log_2 L \quad (9)$$

$$t = a - (b+n) \quad (10)$$

L is the number of lines in a direct-mapped cache of size S and B bytes per block. Field b represents the least significant bits of the memory address; field t represents the most significant bits of the memory address; and field n represents the middle bits as shown in Figure 9.20. Please see Example 4 for a numerical calculation of these fields.

Now let us re-visit the basic cache algorithms (lookup, read, write) in the context of a multi-word block size. Figure 9.21 shows the organization of such a direct-mapped cache.

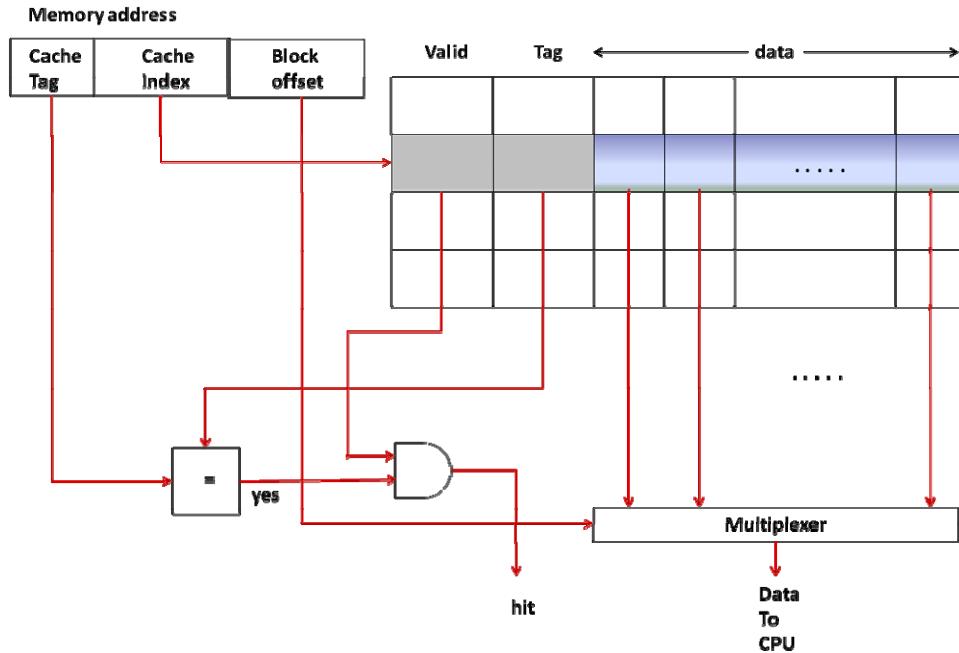


Figure 9.21: A multi-word direct-mapped cache organization. The block offset chooses the particular word to send to the CPU from the selected block using the multiplexer. Horizontally shaded block is the one selected by the cache index.

- 1. Lookup:** The index for cache lookup comes from the middle part of the memory address as shown in Figure 9.20. The entry contains an entire block (if it is a hit as determined by the cache tag in the address and the tag stored in the specific entry). The least significant b bits of the address specify the specific word (or byte) within that block requested by the processor. A multiplexer selects the specific word (or byte) from the block using these b bits and sends it to the CPU (see Figure 9.21).
- 2. Read:** Upon a read, the cache brings out the entire block corresponding to the cache index. If the tag comparison results in a hit, then the multiplexer selects the specific word (or byte) within the block and sends it to the CPU. If it is a miss then the CPU initiates a block transfer from the memory.
- 3. Write:** We have to modify the write algorithm since there is only 1 valid bit for the entire cache line. Similar to the read-miss, the CPU initiates a block transfer from the memory upon a write-miss (see Figure 9.22).

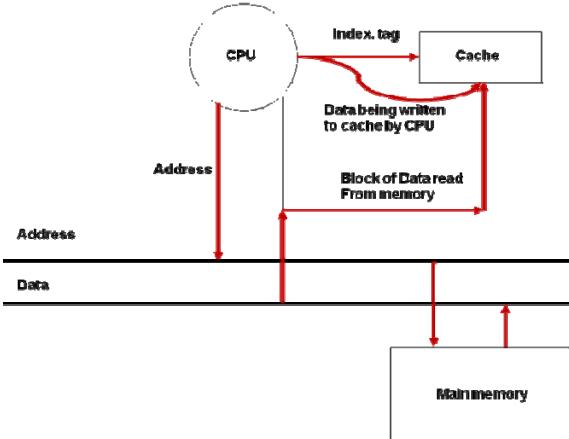


Figure 9.22: CPU, cache, and memory interactions for handling a write-miss

Upon a hit, the CPU writes the specific word (or byte) into the cache. Depending on the write policy, the implementation may require additional meta-data (in the form of dirty bits) and may take additional actions (such as writing the modified word or byte to memory) to complete a write operation (see Example 4).

Example 4:

Consider a multi-word direct-mapped cache with a data size of 64 Kbytes. The CPU generates a 32-bit byte-addressable memory address. Each memory word contains 4 bytes. The block size is 16 bytes. The cache uses a write-back policy with 1 dirty bit per word. The cache has one valid bit per data block.

- (a) How does the CPU interpret the memory address?

Answer:

Referring to formula (7),

Block size

$$B = 16 \text{ bytes}$$

therefore

$$b = \log_2 16 = 4 \text{ bits}$$

We need 4 bits (bits 3-0 of the memory address) for the block offset.

Referring to formula (8), number of cache lines

$$L = 64 \text{ Kbytes}/16 \text{ bytes} = 4096$$

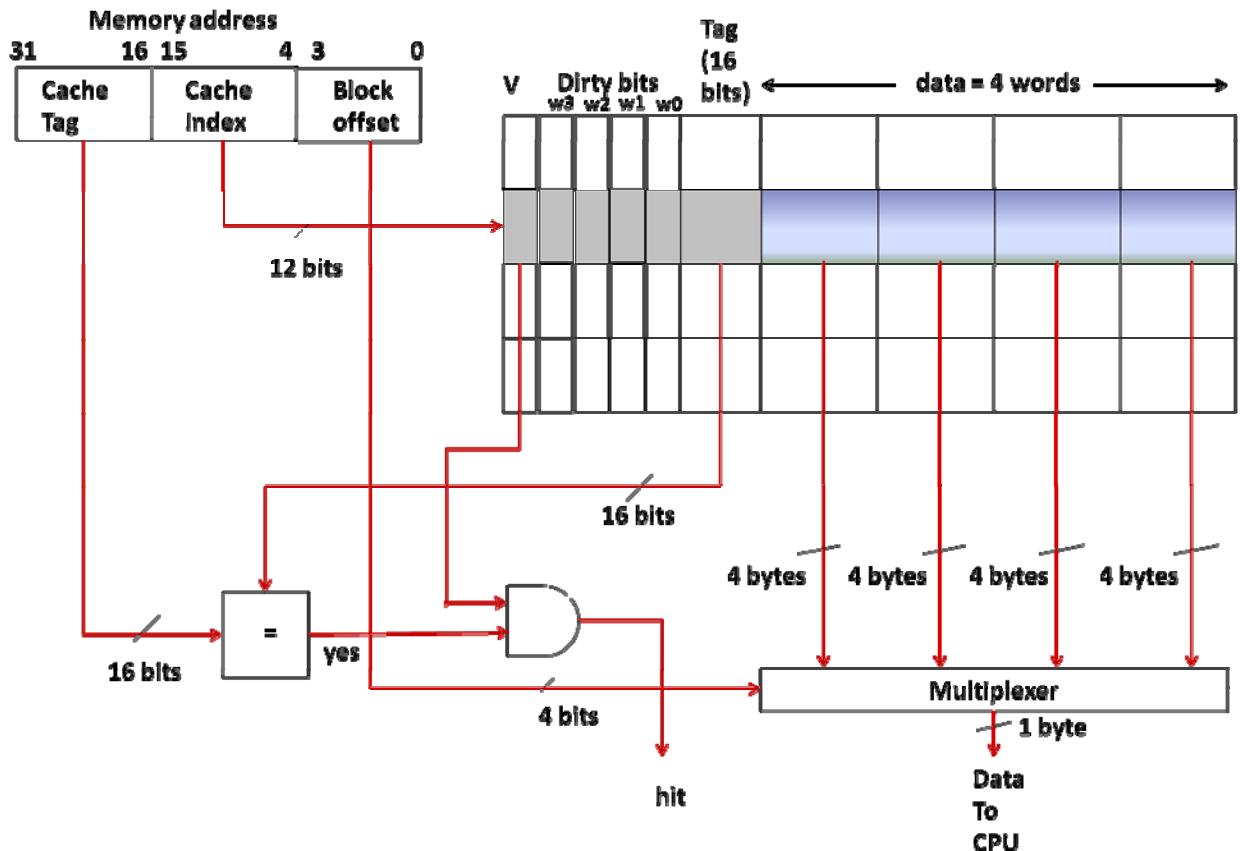
Referring to formula (9), number of index bits

$$n = \log_2 L = \log_2 4096 = 12$$

Referring to formula (10), number of tag bits

$$t = a - (n+b) = 32 - (12+4) = 16$$

Therefore, we need 12 bits (bits 15-4 of the memory address) for the index. The remaining 16 bits (bits 31-16 of the memory address) constitute the tag. The following figure shows how the CPU interprets the memory address.



(b) Compute the total amount of storage for implementing the cache (i.e. actual data plus the meta-data).

Answer:

Each cache line will have:

Data	$16 \text{ bytes} \times 8 \text{ bits/byte} = 128 \text{ bits}$
Valid bit	1 bit
Dirty bits	4 bits (1 for each word)
Tag	16 bits
<hr/>	
149 bits	

Total amount of space for the cache = 149×4096 cache lines = **610,304 bits**

$$\begin{aligned} \text{The space requirement for the meta-data} &= \text{total space} - \text{actual data} = 610,304 - 524,288 \\ &= 86,016 \end{aligned}$$

The space overhead = meta-data/total space = $86,016/610,304 = 14\%$

Recall from Example 2 that the space overhead for the same cache size but with a block size of 4 bytes was 35%. In other words, increased block size decreases the meta-data requirement and hence reduces the space overhead.

9.10.1 Performance implications of increased blocksize

It is instructive to understand the impact of increasing the block size on cache performance. Exploiting spatial locality is the main intent in having an increased block size. Therefore, for a *given total size of the cache*, we would expect the miss-rate to decrease with increasing block size. In the limit, we can simply have one cache block whose size is equal to the total size of the cache. Thus, it is interesting to ask two questions related to the block size:

1. Will the miss-rate keep decreasing forever?
2. Will the overall performance of the processor go up as we increase the block size?

The answer is “No” for the first question. In fact, the miss-rate may decrease up to an inflection point and start increasing thereafter. The *working set* notion that we introduced in Chapter 8 is the reason for this behavior. Recall that the working set of a program changes over time. A cache block contains a contiguous set of memory locations. However, if the working set of the program changes then the large block size results in increasing the miss-rate (see Figure 9.23).

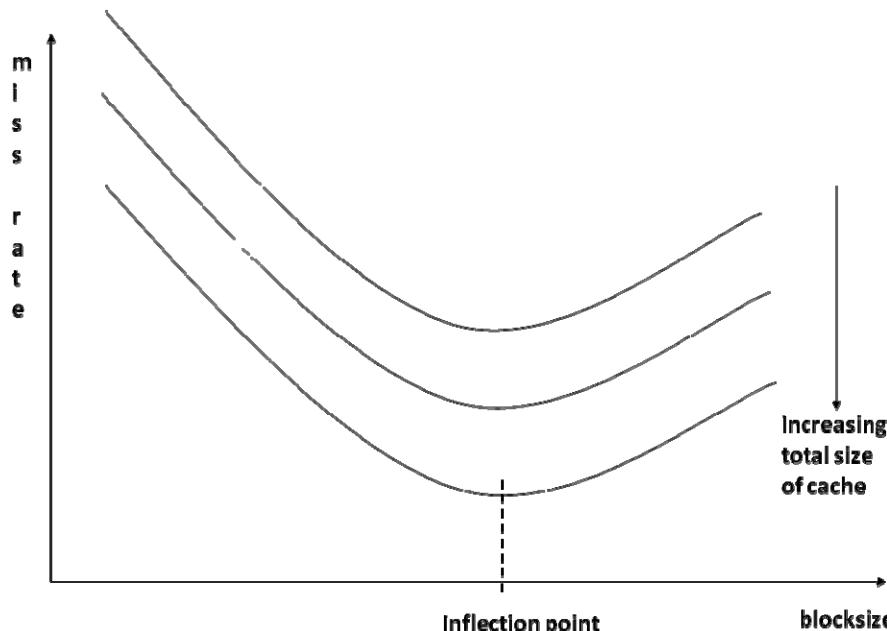


Figure 9.23: Expected behavior of miss-rate as a function of blocksize

The answer to the second question is a little more complicated. As is seen in Figure 9.23, the miss-rate does decrease with increasing block size up to an inflection point for a given cache size. This would translate to the processor incurring fewer memory stalls and thus improve the performance. However, beyond the inflection point, the processor starts incurring more memory stalls thus degrading the performance. However, as it turns

out, the downturn in processor performance may start happening much sooner than the inflection point in Figure 9.23. While the miss-rate decreases with increasing block size up to an inflection point for a given cache size, the increased block size could negatively interact with the miss penalty. Recall that reducing the execution time of a program is the primary objective. This objective translates to reducing the number of stalls introduced in the pipeline by the memory hierarchy. The larger the block size, the larger the time penalty for the transfer from the memory to the cache on misses, thus increasing the memory stalls. We will discuss techniques to reduce the miss penalty shortly. The main point to note is that since the design parameters (block size and miss penalty) are inter-related, optimizing for one may not always result in the best overall performance. Put in another way, just focusing on the miss-rate as the output metric to optimize may lead to erroneous cache design decisions.

In pipelined processor design (Chapter 5), we understood the difference between latency of individual instructions and the throughput of the processor as a whole. Analogously, in cache design, choice of block size affects the balance between latency for a single instruction (that incurred the miss) and throughput for the program as a whole, by reducing the potential misses for other later instructions that might benefit from the exploitation of spatial locality. A real life analogy is income tax. An individual incurs a personal loss of revenue by the taxes (akin to latency), but helps the society as a whole to become better (akin to throughput). Of course, beyond a point the balance tips. Depending on your political orientation the tipping point may be sooner than later.

Modern day processors have a lot more going on that compound these issues. The micro-architecture of the processor, i.e., the implementation details of an ISA is both complex and fascinating. Instructions do not necessarily execute in program order so long as the semantics of the program is preserved. A cache miss does not necessarily block the processor; such caches are referred to as lock-up free caches. These and other micro-level optimizations interact with one another in unpredictable ways, and are of course sensitive to the workload executing on the processor as well. The simple fact is that a doubling of the cache block size requires doubling the amount of data that needs to be moved into the cache. The memory system cannot usually keep up with this requirement. The upshot is that we see a dip in performance well before the inflection point in Figure 9.23.

9.11 Flexible placement

In a direct-mapped cache, there is a one-to-one mapping from a memory address to a cache index. Because of this rigid mapping, the cache is unable to place a new memory location in a currently unoccupied slot in the cache. Due to the nature of program behavior, this rigidity hurts performance. Figure 9.24 illustrates the inefficient use of a direct-mapped cache as a program changes its working set.

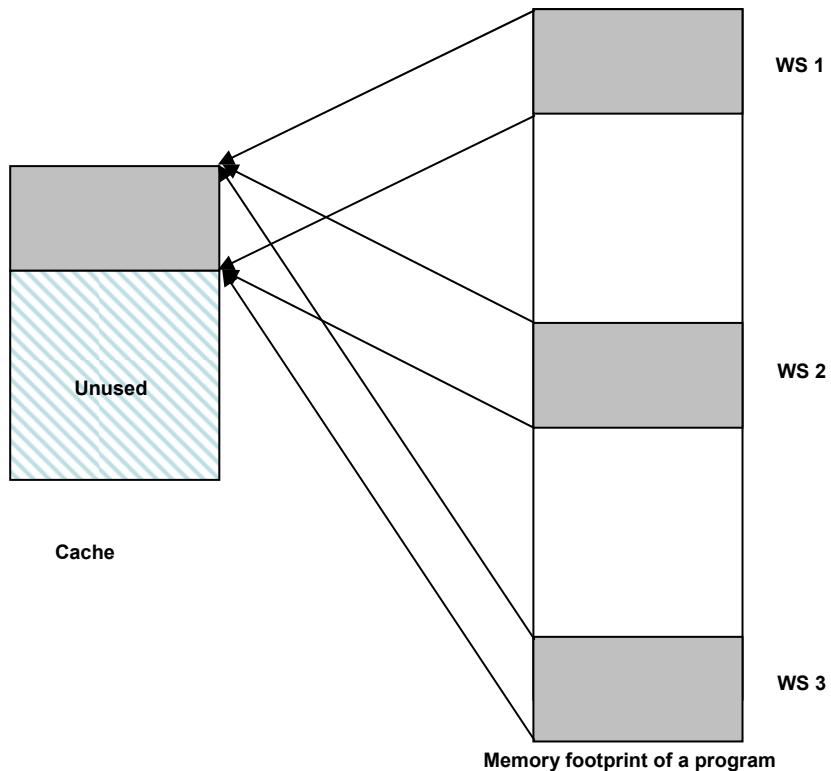


Figure 9.24: Different working sets of a program occupying the same portion of a direct-mapped cache

The program frequently flips among the three working sets (WS1, WS2, and WS3) in the course of its execution that happen to map exactly to the same region of the cache as shown in the Figure. Assume that each working set is only a third of the total cache size. Therefore, in principle there is sufficient space in the cache to hold all three working sets. Yet, due to the rigid mapping, the working sets displace one another from the cache resulting in poor performance. Ideally, we would want all three working sets of the program to reside in the cache so that there will be no more misses beyond the compulsory ones. Let us investigate how we can work towards achieving this ideal. Essentially, the design of the cache should take into account that the locality of a program may change over time. We will first discuss an extremely flexible placement that avoids this problem altogether.

9.11.1 Fully associative cache

In this set up, there is no unique mapping from a memory block to a cache block. Thus, a cache block can host any memory block. Therefore, by design, *compulsory* and *capacity misses* are the only kind of misses encountered with this organization. The cache interprets a memory address presented by the CPU as shown in Figure 9.25. Note that there is no cache index in this interpretation.



Figure 9.25: Memory address interpretation for a fully associative cache

This is because, in the absence of a unique mapping, a memory block could reside in any of the cache blocks. Thus with this organization, to perform a look up, the cache has to search through all the entries to see if there is a match between the cache tag in the memory address and the tags in any of the valid entries. One possibility is to search through each of the cache entries sequentially. This is untenable from the point of view of processor performance. Therefore, the hardware comprises replicated comparators, one for each entry, so that all the tag comparisons happen in parallel to determine a hit (Figure 9.26).

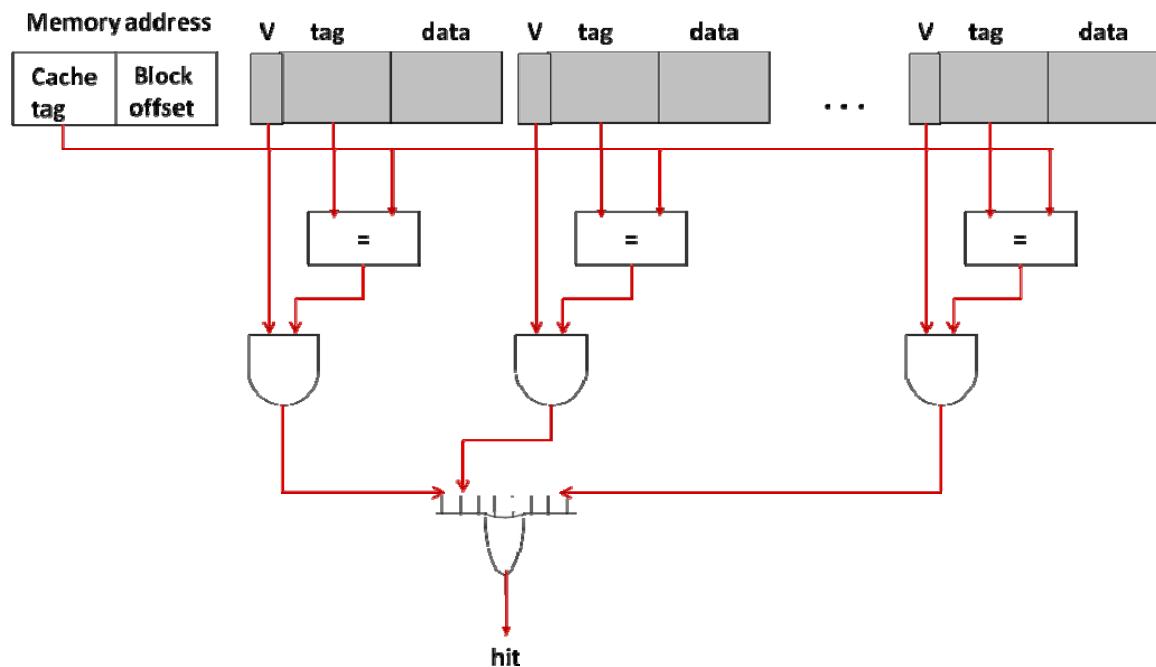


Figure 9.26: Parallel tag matching hardware for a fully associative cache. Each block of the cache represents an independent cache. Tag matching has to happen in parallel for all the caches to determine hit or a miss. Thus, a given memory block could be present in any of the cache blocks (shown by the shading). Upon a hit, the data from the block that successfully matched the memory address is sent to the CPU.

The complexity of the parallel tag matching hardware makes a fully associative cache infeasible for any reasonable sized cache. At first glance, due to its flexibility, it might appear that a fully associative cache could serve as a gold standard for the best possible miss-rate for a given workload and a given cache size. Unfortunately, such is not the case. A cache being a high-speed precious resource will be mostly operating at near full utilization. Thus, misses in the cache will inevitably result in replacement of something that is already present in the cache. The choice of a replacement candidate has a huge

impact on the miss-rates experienced in a cache (since we don't know what memory accesses will be made in the future), and may overshadow the flexibility advantage of a fully associative cache for placement of the missing cache line. We will shortly see more details on cache replacement (see Section 9.14). Suffice it to say at this point that for all of these reasons, a fully associative cache is seldom used in practice except in very special circumstances. Translation look-aside buffer (TLB), introduced in Chapter 8 lends itself to an implementation as a fully associative organization due to its small size. Fully associative derives its name from the fact that a memory block can be *associated* with *any* cache block.

9.11.2 Set associative cache

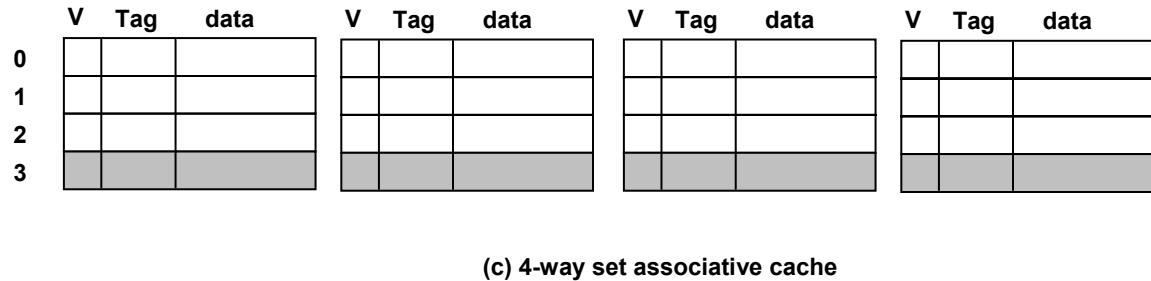
An intermediate organization between direct-mapped and fully associative is a *set associative cache*. This organization derives its name from the fact that a memory block can be *associated* with a *set* of cache blocks. For example, a 2-way set associative cache, gives a memory block two possible homes in the cache. Similarly, a 4-way set associative cache gives a memory block one of four possible homes in the cache. The *degree of associativity* is defined as the number of homes that a given memory block has in the cache. It is 2 for a 2-way set associative cache; and 4 for a 4-way set associative cache; and so on.

	V	Tag	data
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

(a) Direct-mapped cache

	V	Tag	data
0			
1			
2			
3			
4			
5			
6			
7			

(b) 2-way set associative cache



(c) 4-way set associative cache

Figure 9.27: Three different organizations of 16 cache blocks. A given memory block could be present in one of several cache blocks depending on the degree of associativity. The shaded blocks in each of the three figures represent the possible homes for a given memory block.

One simple way to visualize a set associative cache is as multiple direct-mapped caches. To make this discussion concrete, consider a cache with a total size of 16 data blocks. We can organize these 16 blocks as a direct-mapped cache (Figure 9.27-(a)), or as a 2-way set associative cache (Figure 9.27-(b)), or as a 4-way set associative cache (Figure 9.27-(c)). Of course, with each data block there will be associated meta-data.

In the direct-mapped cache, given an index (say 3) there is exactly one spot in the cache that corresponds to this index value. The 2-way set associative cache has two spots in the cache (shaded spots in Figure 9.27-(b)) that correspond to the same index. The 4-way has four spots that correspond to the same index (shaded spots in Figure 9.27-(c)). With 16 data blocks total, the first organization requires a 4-bit index to lookup the cache, while the second organization requires a 3-bit index and the third a 2-bit index. Given a memory address, the cache simultaneously looks up all possible spots, referred to as a *set*, corresponding to that index value in the address for a match. The amount of tag matching hardware required for a set associative cache equals the degree of associativity.

Figure 9.28 shows the complete organization of a 4-way set associative cache, with a block size of 4 bytes.

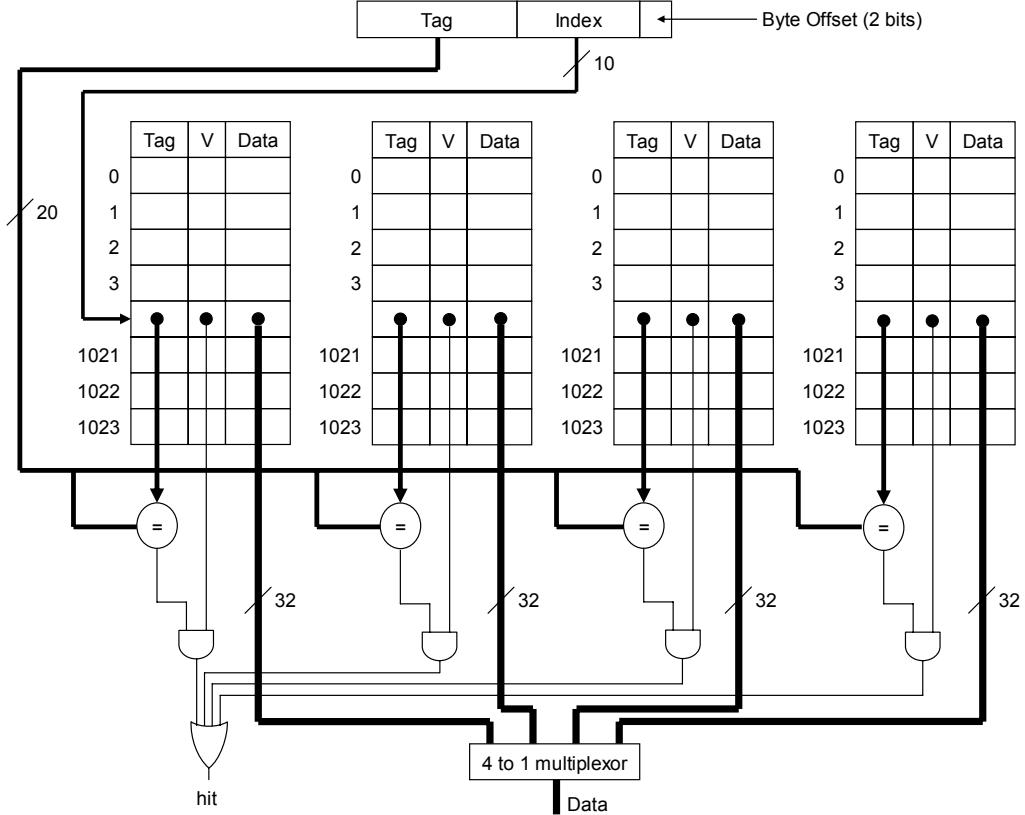


Figure 9.28: 4-way set associative cache organization

The cache interprets the memory address from the CPU as consisting of *tag*, *index*, and *block offset*, similar to a direct-mapped organization.

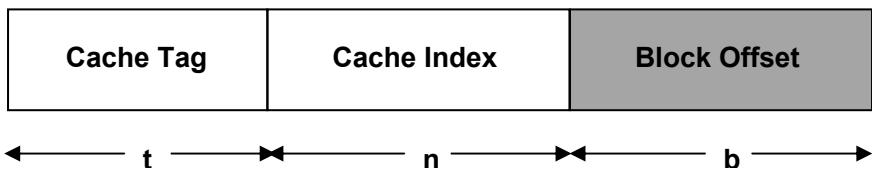


Figure 9.29: Interpreting the memory address generated by the CPU for a set-associative cache is no different from that for a direct-mapped cache

Let us understand how to break up the memory address into index and tag for cache lookup. The total size of the cache determines the number of index bits for a direct-mapped organization ($\log_2(S/B)$, where S is the cache size and B the block size in bytes, respectively). For a set associative cache with the same total cache size, $\log_2(S/pB)$

determines the number of index bits, where p is the degree of associativity. For example, a 4-way set associative cache with a total cache size of 16 data blocks requires $\log_2(16/4) = 2$ bits (Figure 9.27-(c)).

9.11.3 Extremes of set associativity

Both direct-mapped and fully associative caches are special cases of the set-associative cache. Consider a cache with total size S bytes and block size B bytes. N , the total number of data blocks in the cache is given by S/B . Let us organize the data blocks into p parallel caches. We call this an p -way *set associative* cache. The cache has N/p cache lines (or sets³), each cache line having p data blocks. The cache requires p parallel hardware for tag comparisons.

What if $p = 1$? In this case, the organization becomes a direct-mapped cache. The cache has N cache lines with each set having 1 block.

What if $p = N$? In this case, the organization becomes a fully associative cache. The cache has 1 cache line of N blocks.

Given this insight, we will re-visit the formulae (7) through (10).

Total cache size = S bytes
 Block size = B bytes
 Memory address = a bits
 Degree of associativity = p

Total number of data blocks in the cache:

$$N = S/B \quad (11)$$

We replace equation (8) for the number of cache lines by the following general equation:

$$L = S/pB = N/p \quad (12)$$

Equation (8) is a special case of (12) when $p = 1$. The number of index bits n is computed as $\log_2 L$ (given by equation (9)).

As we said earlier, a fully associative cache is primarily used for TLBs. The degree of associativity usually depends on the level of the cache in the memory hierarchy. Either a direct-mapped cache or a 2-way set-associative cache is typically preferred for L1 (close to the CPU). Higher degree of associativity is the norm for deeper levels of the memory hierarchy. We will revisit these design considerations later in this chapter (please see Section 9.19).

³ So now, we have four terminologies to refer to the same thing: cache line, cache block, cache entry, set.

Example 5:

Consider a 4-way set associative cache with a data size of 64 Kbytes. The CPU generates a 32-bit byte-addressable memory address. Each memory word contains 4 bytes. The block size is 16 bytes. The cache uses a write-through policy. The cache has one valid bit per data block.

- How does the CPU interpret the memory address?
- Compute the total amount of storage for implementing the cache (i.e. actual data plus the meta-data).

Answer:

a)

Number of bits in memory address:

$$a = 32 \text{ bits}$$

Since the block size is 16 bytes, using equation (7), the block offset

$$b = 4 \text{ bits (bits 0-3 of the memory address)}$$

Since the cache is 4-way set associative (Figure 9.27-(c)),

$$p = 4$$

Number of cache lines (equation (12)):

$$L = S/pB = 64 \text{ K bytes} / (4 * 16) \text{ bytes} = 1 \text{ K}$$

Number of index bits (equation (9)):

$$n = \log_2 L = \log_2 1024 = 10 \text{ bits}$$

Number of tag bits (equation (10)):

$$t = a - (n+b) = 32 - (10+4) = 18 \text{ bits}$$

Thus the memory address with 31 as msb, and 0 as the lsb is interpreted as:

Tag	18 bits (31 to 114)
Index	10 bits (13 to 4)
Block offset	4 bits (3 to 0)

b)

A block in each of the 4 parallel caches contains (data plus meta-data):

Data: 16 x 8 bits	= 128 bits (16 bytes in one cache block)
Valid bit	= 1 bit (1 valid bit per block)
<u>Tag</u>	= 18 bits
Total	= 147 bits

Each line of the cache contains 4 of these blocks = $147 * 4 = 588$ bits

With 1K such cache lines in the entire cache, **the total size of the cache =**
588 bits/cache line * 1024 cache lines = **602,112 bits**

Example 6:

Consider a 4-way set-associative cache.

- Total data size of cache = 256KB.
- CPU generates 32-bit byte-addressable memory addresses.
- Each memory word consists of 4 bytes.
- The cache block size is 32 bytes.
- The cache has one valid bit per cache line.
- The cache uses write-back policy with one dirty bit per word.

- a) Show how the CPU interprets the memory address (i.e., which bits are used as the cache index, which bits are used as the tag, and which bits are used as the offset into the block?).
- b) Compute the total size of the cache (including data and metadata).

Answer:

- a) Using reasoning similar to example 5,

Tag	16 bits (31 to 16)
Index	11 bits (15 to 5)
Block offset	5 bits (0 to 4)

b)

A block in each of the 4 parallel caches contains:

Data: 32 x 8 bits	= 256 bits (32 bytes in one cache block)
Valid bit	= 1 bit (1 valid bit per block)
Dirty bits: 8 x 1 bit	= 8 bits (1 dirty bit per word of 4 bytes)
<u>Tag</u>	<u>16 bits</u>
Total	= 281 bits

Each cache line contains 4 of these blocks = $281 * 4 = 1124$ bits

With 2K such cache lines in the entire cache, **the total size of the cache =**
1124 bits/cache line * 2048 cache lines = **281 KBytes**

9.12 Instruction and Data caches

In the processor pipeline (please see Figure 9.15), we show two caches, one in the IF stage and one in the MEM stage. Ostensibly, the former is for instructions and the latter is for data. Some programs may benefit from a larger instruction cache, while some other programs may benefit from a larger data cache.

It is tempting to combine the two caches and make one single large *unified* cache. Certainly, that will increase the hit rate for most programs for a given cache size irrespective of their access patterns for instructions and data.

However, there is a down side to combining. We know that the IF stage accesses the I-cache in every cycle. The D-cache comes into play only for memory reference instructions (loads/stores). The contention for a unified cache could result in a structural hazard and reduce the pipeline efficiency. Empirical studies have shown that the detrimental effect of the structural hazard posed by a unified cache reduces the overall pipeline efficiency despite the increase in hit rate.

There are hardware techniques (such as multiple read ports to the caches) to get around the structural hazard posed by a unified I- and D-cache. However, these techniques increase the complexity of the processor, which in turn would affect the clock cycle time of the pipeline. Recall that caches cater to the twin requirements of speed of access and reducing miss rate. As we mentioned right at the outset (see Section 9.4), the primary design consideration for the L1 cache is matching the hit time to the processor clock cycle time. This suggests avoiding unnecessary design complexity for the L1 cache to keep the hit time low. In addition, due to the differences in access patterns for instructions and data, the design considerations such as associativity may be different for I- and D-caches. Further, thanks to the increased chip density, it is now possible to have large enough separate I- and D-caches that the increase in miss rate due to splitting the caches is negligible. Lastly, I-cache does not have to support writes, which makes them simpler and faster. For these reasons, it is usual to have a split I- and D-caches on-chip. However, since the primary design consideration for L2 cache is reducing the miss rate, it is usual to have a unified L2 cache.

9.13 Reducing miss penalty

The miss penalty is the service time for the data transfer from memory to cache on misses. As we observed earlier, the penalty may be different for reads and writes, and the penalty depends on other hardware in the datapath such as *write buffers* that allow overlapping computation with the memory transfer.

Usually, the main memory system design accounts for the cache organization. In particular, the design supports block transfer to and from the CPU to populate the cache. The memory bus that connects the main memory to the CPU plays a crucial role in determining the miss penalty. We refer to *bus cycle time* as the time taken for each data transfer between the processor and memory. The amount of data transfer in each cycle between the processor and memory is referred to as the *memory bandwidth*. Memory bandwidth is a measure of the throughput available for information transfer between processor and memory. The bandwidth depends on the number of data wires connecting the processor and memory. Depending on the width of bus, the memory system may need multiple bus cycles to transfer a cache block. For example, if the block size is four words and the memory bus width is only one word, then it takes four bus cycles to complete the block transfer. As a first order, we may define the miss penalty as the total time (measured in CPU clock cycles) for transferring a cache block from the memory to

the cache. However, the actual latency experienced by the processor for an individual miss may be less than this block transfer time. This is because the memory system may respond to a miss by providing the specific data that the processor missed on before sending the rest of the memory block that contains the missed reference.

Despite, such block transfer support in the memory system, increasing the block size beyond a certain point has other adverse effects. For example, if the processor incurs a read-miss on a location x , the cache subsystem initiates bringing in the whole block containing x , perhaps ensuring that the processor is served with x first. Depending on the bandwidth available between the processor and memory, the memory system may be busy transferring the rest of the block for several subsequent bus cycles. In the meanwhile, the processor may incur a second miss for another memory location y in a different cache block. Now, the memory system cannot service the miss immediately since it is busy completing the block transfer for the earlier miss on x . This is the reason we observed earlier in Section 9.10 that it is not sufficient to focus just on the miss-rate as the output metric to decide on cache block size. This is the classic tension between latency and throughput that surfaces in the design of every computer subsystem. We have seen it in the context of processor in Chapter 5; we see it now in the context of memory system; later (Chapter 13), we will see it in the context of networks.

9.14 Cache replacement policy

In a direct-mapped cache, the placement policy pre-determines the candidate for replacement. Hence, there is no choice.

There is a possibility to exercise a choice in the case of a set associative or a fully associative cache. Recall that temporal locality consideration suggests using an LRU policy. For a fully associative cache, the cache chooses the replacement candidate applying the LRU policy across all the blocks. For a set associative cache, the cache's choice for a replacement candidate is limited to the set that will house the currently missing memory reference.

To keep track of LRU information, the cache needs additional meta-data. Figure 9.30 shows the hardware for keeping track of the LRU information for a 2-way set associative cache. Each set (or cache line) has one LRU bit associated with it.

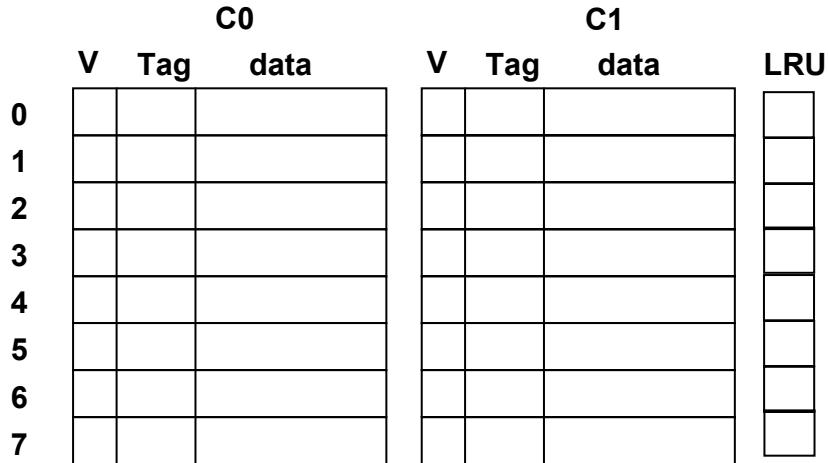


Figure 9.30: 1-bit LRU per set in a 2-way set associative cache

On each reference, the hardware sets the LRU bit for the set containing the currently accessed memory block. Assuming valid entries in both the blocks of a given set, the hardware sets the LRU bit to 0 or 1 depending on whether a reference hits in cache C0 or C1, respectively. The candidate for replacement is the block in C0 if the LRU bit is 1; it is the block in C1 if the LRU bit is 0.

The hardware needed for a 2-way set associative cache is pretty minimal but there is a time penalty since the hardware updates the LRU bit on every memory reference (affects the IF and MEM stages of the pipelined design).

The LRU hardware for higher degrees of associativity quickly becomes more complex. Suppose we label the 4 parallel caches C0, C1, C2, and C3 as shown in Figure 9.31. Building on the 2-way set associative cache, we could have a 2-bit field with each set. The encoding of the 2-bit field gives the block accessed most recently. Unfortunately, while this tells us the most recently used block, it does not tell us which block in a set is the least recently used. What we really need is an ordering vector as shown in Figure 9.31 for each set. For example, the ordering for set S0 shows that C2 was the least recently used and C1 the most recently used. That is, the decreasing time order of access to the blocks in S0 is: C1, followed by C3, followed by C0, followed by C2. Thus, at this time if a block has to be replaced from S0 then the LRU victim is the memory block in C2. On each reference, the CPU updates the ordering for the currently accessed set. Figure 9.32 shows the change in the LRU ordering vector to set S0 on a sequence of references that map to this set. Each row shows how the candidate for replacement changes based on the currently accessed block.

	C0	C1	C2	C3	LRU
	V Tag data	V Tag data	V Tag data	V Tag data	c1 -> c3 -> c0 -> c2
S0					c0 -> c2 -> c1 -> c3
S1					c2 -> c3 -> c0 -> c1
S2					c3 -> c2 -> c1 -> c0
S3					

Figure 9.31: LRU information for a 4-way set associative cache

LRU for set S0	
Access to C1	c1 -> c3 -> c0 -> c2
Access to C2	c2 -> c1 -> c3 -> c0
Access to C2	c2 -> c1 -> c3 -> c0
Access to C3	c3 -> c2 -> c1 -> c0
Access to C0	c0 -> c3 -> c2 -> c1

Figure 9.32: Change in the LRU vector for set S0 on a sequence of references that map to that set

Let us understand what it takes to implement this scheme in hardware. The number of possibilities for the ordering vector with 4 parallel caches is $4! = 24$. Therefore, we need a 5-bit counter to encode the 24 possible states of the ordering vector. An 8-way set associative cache requires a counter big enough to encode 8 factorial states. Each set needs to maintain such a counter with the associated fine state machine that implements the state changes as illustrated in the above example. One can see that the hardware complexity of implementing such a scheme increases rapidly with the degree of associativity and the number of lines in the cache. There are other simpler encodings that are approximations to this true LRU scheme. There is ample empirical evidence that less complex replacement policies may in fact perform better (i.e., result in better miss-rates) than true LRU for many real code sequences.

9.15 Recapping Types of Misses

We identified three categories of misses in the cache: *compulsory*, *capacity*, and *conflict*. As the name suggests, compulsory misses result because the program is accessing a given memory location for the first time during execution. Naturally, the location is not in the cache and a miss is inevitable. Using the analogy of an automobile engine being cold or warm (at the time of starting), it is customary to refer to such misses as *cold* misses.

On the other hand, consider the situation when the CPU incurs a miss on a memory location X that used to be in the cache previously⁴. This can be due to one of two reasons: the cache is full at the time of the miss, so there is no choice except to evict some memory location to make room for X. This is what is referred to as a *capacity*

⁴ We do not know why X was evicted in the first place but it is immaterial from the point of view of characterizing the current miss.

miss. On the other hand, it is conceivable that the cache is not full but the mapping strategy forces X to be brought into a cache line that is currently occupied by some other memory location. This is what is referred to as a *conflict* miss. By definition, we cannot have a conflict miss in a fully associative cache, since a memory location can be placed anywhere. Therefore, in a fully associative cache the only kinds of misses that are possible are compulsory and capacity.

Sometimes it can be confusing how to categorize the misses. Consider a fully associative cache. Let us say, the CPU accesses a memory location X for the first time. Let the cache be full at this time. We have a cache miss on X. Is this a capacity or a compulsory miss? We can safely say it is both. Therefore, it is fair to categorize this miss as either compulsory or capacity or both.

Note that a capacity miss can happen in any of direct-mapped, set-associative, or fully associative caches. For example, consider a direct-mapped cache with 4 cache lines. Let the cache be initially empty and each line hold exactly 1 memory word. Consider the following sequence of memory accesses by the CPU:

0, 4, 0, 1, 2, 3, 5, 4

We will denote the memory word at the above addresses as m0, m1, ..., m5, respectively. The first access (m0) is a compulsory miss. The second access (m4) is also a compulsory miss and it will result in evicting m0 from the cache, due to the direct-mapped organization. The next access is once again to m0, which will be a miss. This is definitely a conflict miss since m0 was evicted earlier by bringing in m4, in spite of the fact that the cache had other empty lines. Continuing with the accesses, m1, m2, and m3 all incur compulsory misses.

Next, we come to memory access for m5. This was never in the cache earlier, so the resulting miss for m5 may be considered a compulsory miss. However, the cache is also full at this time with m0, m1, m2, and m3. Therefore, one could argue that this miss is a capacity miss. Thus, it is fair to categorize this miss as either a compulsory miss or a capacity miss or both.

Finally, we come to the memory access for m4 again. This could be considered a conflict miss since m4 used to be in the cache earlier. However, at this point the cache is full with m0, m5, m2, and m3. Therefore, we could also argue that this miss is a capacity miss. Thus, it is fair to categorize this miss as either a conflict miss or a capacity miss or both.

Compulsory misses are unavoidable. Therefore, this type of a miss dominates the other kinds of misses. In other words, if a miss can be characterized as either compulsory or something else, we will choose compulsory as its characterization. Once the cache is full, independent of the organization, we will incur a miss on a memory location that is not currently in the cache. Therefore, a capacity miss dominates a conflict miss. In other words, if a miss can be characterized as either conflict or capacity, we will choose capacity as its characterization.

Example 7:

Given the following:

- Total number of blocks in a 2-way set associative cache: 8
- Assume LRU replacement policy

	C1	C2
index	0	
	1	
	2	
	3	

The processor makes the following 18 accesses to memory locations in the order shown below:

0, 1, 8, 0, 1, 16, 8, 8, 0, 5, 2, 1, 10, 3, 11, 10, 16, 8

With respect to the 2-way set associative cache shown above, show in a tabular form the cache which will host the memory location and the specific cache index where it will be hosted, and in case of a miss the type of miss (cold/compulsory, capacity, conflict).

Memory location	C1	C2	Hit/miss	Type of miss

Note:

- The caches are initially empty
- Upon a miss, if both the spots (C1 and C2) are free then the missing memory location will be brought into C1
- Capacity miss dominates over conflict miss
- Cold/compulsory miss dominates over capacity miss

Answer:

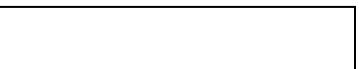
Memory location	C1	C2	Hit/miss	Type of miss
0	Index = 0		Miss	Cold/compulsory
1	Index = 1		Miss	Cold/compulsory
8		Index = 0	Miss	Cold/compulsory
0	Index = 0		Hit	
1	Index = 1		Hit	
16		Index = 0	Miss	Cold/compulsory
8	Index = 0		Miss	Conflict
8	Index = 0		Hit	
0		Index = 0	Miss	Conflict
5		Index = 1	Miss	Cold/compulsory
2	Index = 2		Miss	Cold/compulsory
1	Index = 1		Hit	
10		Index = 2	Miss	Cold/compulsory
3	Index = 3		Miss	Cold/compulsory
11		Index = 3	Miss	Cold/compulsory
10		Index = 2	Hit	
16	Index = 0		Miss	Capacity
8		Index = 0	Miss	Capacity

9.16 Integrating TLB and Caches

In Chapter 8, we introduced TLB, which is nothing but a cache of addresses. Recall that given a virtual page number (VPN) the TLB returns the physical frame number (PFN) if it exists in the TLB. The TLB is usually quite small for considerations of speed of access, and the space of virtual pages is quite large. Similar to the processor caches, there is the need for a mapping function for looking up the TLB given a VPN. The design considerations for a TLB are quite similar to those of processor caches, i.e., the TLB may be organized as a direct-mapped or set-associative structure. Depending on the specifics of the organization, the VPN is broken up into tag and index fields to enable the TLB look up. The following example brings home this point.

Example 8:

Given the following:

Virtual address	64 bits	
		31 0
Physical address	32 bits	
Page size	4 K Bytes	

A direct mapped TLB with 512 entries

- (a) How many tag bits per entry are there in the TLB?
- (b) How many bits are needed to store the page frame number in the TLB?

Answer:

(a)

With a pagesize of 4 KB the number of bits needed for page offset = 12

Therefore, the number of bits for VPN = $64-12 = 52$

Number of index bits for looking up a direct mapped TLB of size 512 = 9

So, the number of tag in the TLB = $52-9 = 43 \text{ bits}$

(b)

The number of bits needed to store the PFN in the TLB equals the size of the PFN.

With a pagesize of 4 KB, the PFN is $32-12 = 20 \text{ bits}$

Now we can put the TLB, and the memory hierarchies together to get a total picture. Figure 9.33 shows the path of a memory access from the CPU (in either the IF or the MEM stages) that proceeds as follows:

- The CPU (IF or MEM stage of the pipeline) generates a virtual address (VA)
- The TLB does a virtual to physical address (PA) translation. If it is a hit in the TLB, then the pipeline continues without a hiccup. If it is a miss, then the pipeline freezes until the completion of the miss service.
- The stage uses the PA to look up the cache (I-Cache or the D-Cache, respectively). If it is a hit in the cache, then the pipeline continues without a hiccup. If it is a miss, then the pipeline freezes until the completion of the miss service.

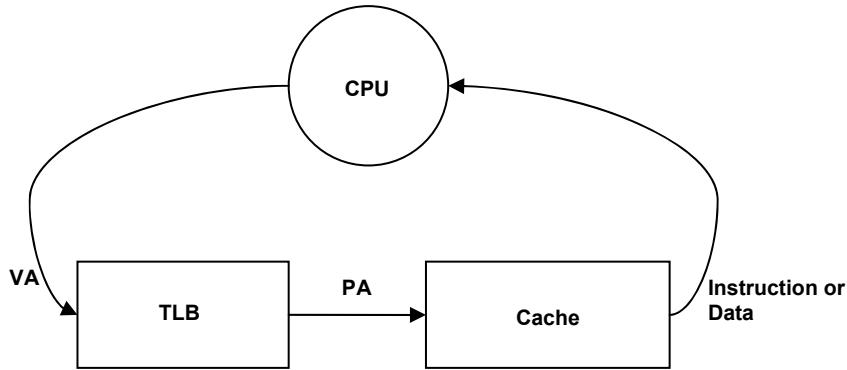


Figure 9.33: Path of a memory access

Note that the IF and MEM stages of the pipeline may access the TLB simultaneously. For this reason, most processors implement split TLB for instructions and data (I-TLB and D-TLB), respectively, so that two address translations can proceed in parallel. As shown in Figure 9.33, the TLB is in the critical path of determining the processor clock cycle time, since every memory access has to go first through the TLB and then the cache. Therefore, the TLB is small in size, typically 64 to 256 entries for each of I- and D-TLBs.

9.17 Cache controller

This hardware interfaces the processor to the cache internals and the rest of the memory system and is responsible for taking care of the following functions:

- Upon a request from the processor, it looks up the cache to determine hit or miss, serving the data up to the processor in case of a hit.
- Upon a miss, it initiates a bus transaction to read the missing block from the deeper levels of the memory hierarchy.
- Depending on the details of the memory bus, the requested data block may arrive asynchronously with respect to the request. In this case, the cache controller receives the block and places it in the appropriate spot in the cache.
- As we will see in the next chapter, the controller provides the ability for the processor to specify certain regions of the memory as “uncachable.” The need for this will become apparent when we deal with interfacing I/O devices to the processor (Chapter 10).

Example 9:

Consider the following memory hierarchy:

- A 128 entry fully associative TLB split into 2 halves; one-half for user processes and the other half for the kernel. The TLB has an access time of 1 cycle. The hit rate for the TLB is 95%. A miss results in a main memory access to complete the address translation.
- An L1 cache with a 1 cycle access time, and 99% hit rate.
- An L2 cache with a 4 cycle access time, and a 90% hit rate.
- An L3 cache with a 10 cycle access time, and a 70% hit rate.

- A physical memory with a 100 cycle access time.

Compute the average memory access time for this memory hierarchy. Note that the page table entry may itself be in the cache.

Answer:

Recall from Section 9.4:

$$\text{EMAT}_i = T_i + m_i * \text{EMAT}_{i+1}$$

$\text{EMAT}_{\text{physical memory}} = 100$ cycles.

$\text{EMAT}_{L3} = 10 + (1 - 0.7) * 100 = 40$ cycles

$\text{EMAT}_{L2} = (4) + (1 - 0.9) * (40) = 8$ cycles

$\text{EMAT}_{L1} = (1) + (1 - 0.99) * (8) = 1.08$ cycles

$\text{EMAT}_{\text{TLB}} = (1) + (1 - 0.95) * (1.08) = 1.054$ cycles

$\text{EMAT}_{\text{Hierarchy}} = \text{EMAT}_{\text{TLB}} + \text{EMAT}_{L1} = 1.054 + 1.08 = 2.134$ cycles.

9.18 Virtually indexed physically tagged cache

Looking at Figure 9.33, every memory access goes through the TLB and then the cache. The TLB helps in avoiding a trip to the main memory for address translation. However, TLB lookup is in the critical path of the CPU. What this means is that the virtual to physical address translation has to be done first before we can look up the cache to see if the memory location is present in the cache. In other words, due to the sequential nature of the TLB lookup followed by the cache lookup, the CPU incurs a significant delay before it can be determined if a memory access is a hit or a miss in the cache. It would be nice if accessing the TLB for address translation, and looking up the cache for the memory word can be done in parallel. In other words, we would like to get the address translation “out of the way” of the CPU access to the cache. That is, we want to take the CPU address and lookup the cache bypassing the TLB. At first, it seems as though this is impossible since we need the physical address to look up the cache.

Let us re-examine the virtual address (Figure 9.34).

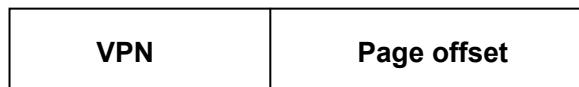


Figure 9.34: Virtual address

The VPN changes to PFN because of the address translation. However, the page offset part of the virtual address remains unchanged.

A direct-mapped or set associative cache derives the index for lookup from the least significant bits of the physical address (see Figure 9.11).

This gives us an idea. If we derive the cache index from the unchanging part of the virtual address (i.e. the page offset part), then we can lookup the cache in parallel with the TLB lookup. We refer to such an organization as a *virtually indexed physically*

tagged cache (Figure 9.35). The cache uses virtual index but derives the tag from the physical address.

With a little bit of thought it is not difficult to figure out the limitation with this scheme. The unchanging part of the virtual address limits the size of the cache. For example, if the page size is 8 Kbytes then the cache index is at most 13 bits, and typically less since part of the least significant bits specify the block offset. Increasing the set associativity allows increasing the size of the cache despite this restriction. However, we know there is a limit to increasing the associativity due to the corresponding increase in hardware complexity.

A partnership between software and hardware helps in eliminating this restriction. Although the hardware does the address translation, the memory manager is the software entity that sets up the VPN to PFN mapping. The memory manager uses a technique called *page coloring* to guarantee that more of the virtual address bits will remain unchanged by the translation process by choosing the VPN to PFN mapping carefully (see Example 7). Page coloring allows the processor to have a larger virtually indexed physically tagged cache independent of the page size.

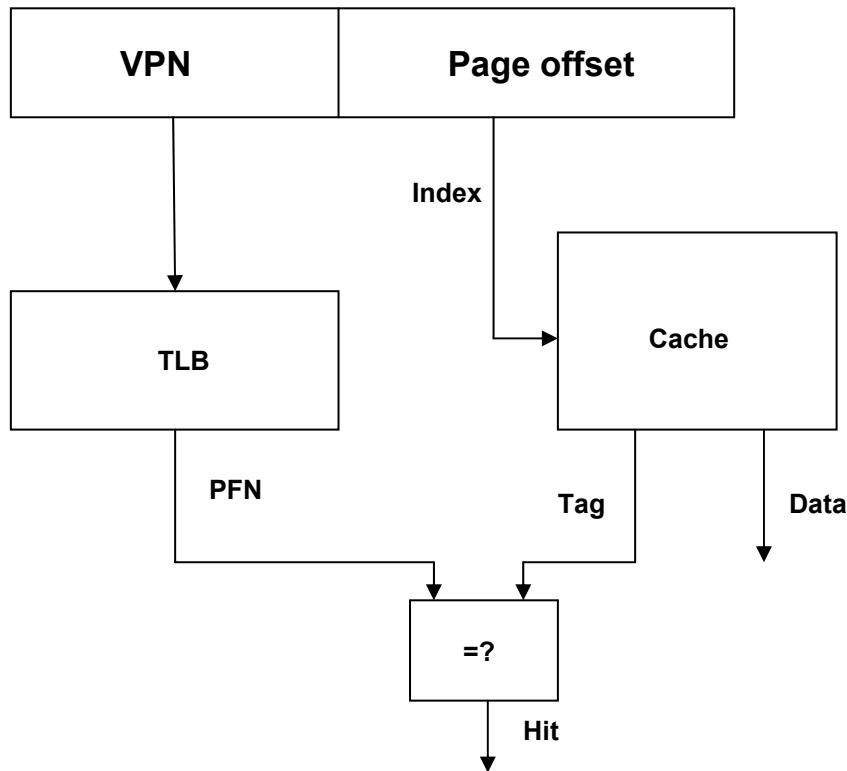


Figure 9.35: A virtually indexed physically tagged cache

Another way to get the translation out of the way is to use *virtually tagged* caches. In this case, the cache uses virtual index and tag. The reader should ponder on the challenges such an organization introduces. Such a discussion is beyond the scope of this book⁵.

Example 10:

Consider a virtually indexed physically tagged cache:

1. Virtual address is 32 bits
2. Page size 8 Kbytes
3. Details of the cache
 - o 4-way set associative
 - o Total Size = 512 Kbytes
 - o Block size = 64 bytes

The memory manager uses page coloring to allow the large cache size.

1. How many bits of the virtual address should remain unchanged for this memory hierarchy to work?
2. Pictorially show the address translation and cache lookup, labeling the parts of the virtual and physical addresses used in them.

Answer:

An 8 Kbytes page size means the page offset will be 13 bits leaving 19 bits for the VPN.

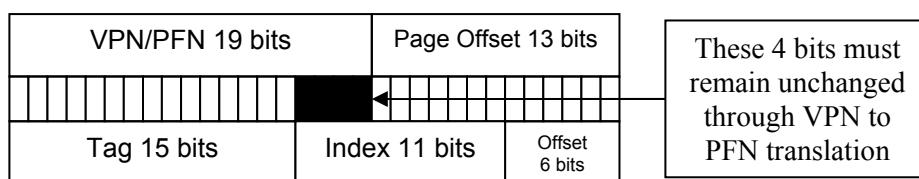
The cache has 4 blocks/set x 64 bytes/block = 256 bytes/set and

512 Kbytes total cache size / 256 bytes/set = 2 K sets which implies 11 bits for the index.

Therefore the cache breakdown of a memory reference is

Tag, 15 bits; Index, 11 bits; Offset 6 bits

Thus, the memory management software must assign frames to pages in such a way that the least significant 4 bits of the VPN must equal to the least significant 4 bits of the PFN.



⁵ Please see advanced computer architecture textbooks for a more complete treatment of this topic.
Hennessy and Patterson, Computer Architecture: A Quantitative Approach, Morgan-Kauffman publishers.

9.19 Recap of Cache Design Considerations

Thus far, we have introduced several concepts and it will be useful to enumerate them before we look at main memory:

1. Principles of spatial and temporal locality (Section 9.2)
2. Hit, miss, hit rate, miss rate, cycle time, hit time, miss penalty (Section 9.3)
3. Multilevel caches and design considerations thereof (Section 9.4)
4. Direct mapped caches (Section 9.6)
5. Cache read/write algorithms (Section 9.8)
6. Spatial locality and blocksize (Section 9.10)
7. Fully- and set-associative caches (Section 9.11)
8. Considerations for I- and D-caches (Section 9.12)
9. Cache replacement policy (Section 9.14)
10. Types of misses (Section 9.15)
11. TLB and caches (Section 9.16)
12. Cache controller (Section 9.17)
13. Virtually indexed physically tagged caches (Section 9.18)

Modern processors have on-chip TLB, L1, and L2 caches. The primary design consideration for TLB and L1 is reducing the hit time. Both of them use a simple design consistent with this design consideration. TLB is quite often a small fully associative cache of address translations with sizes in the range of 64 to 256 entries. It is usually split into a system portion that survives context switches, and a user portion that is flushed upon a context switch. Some processors provide process tags in the TLB entries to avoid flushing at context switch time. L1 is optimized for speed of access. Usually it is split into I- and D-caches employing a small degree of associativity (usually 2), and the size is relatively small compared to the higher levels of the memory hierarchy (typically less than 64KB for each of I- and D-caches in processors circa 2008). L2 is optimized for reducing the miss rate. Usually it is a unified I-D cache, employing a larger associativity (4-way and 8-way are quite common; some are even 16-way). They may even employ a larger block size than L1 to reduce the miss rate. L2 cache size for processors circa 2008 is in the range of several hundreds of Kilobytes to a few Megabytes. Most modern processors also provide a larger off-chip L3 cache, which may have similar design considerations as L2 but even larger in size (several tens of Megabytes in processors circa 2008).

9.20 Main memory design considerations

The design of the processor-memory bus and the organization of the physical memory play a crucial part in the performance of the memory hierarchy. As we mentioned earlier, the implementation of the physical memory uses DRAM technology that has nearly a 100:1 speed differential compared to the CPU. The design may warrant several trips to the physical memory on a cache miss depending on the width of the processor-memory bus and the cache block size.

In the spirit of the textbook in undertaking joint discovery of interesting concepts, we will start out by presenting very simple ideas for organizing the main memory system. At the outset, we want the reader to understand that these ideas are far removed from the

sophisticated memory system that one might see inside a modern box today. We will end this discussion with a look at the state-of-the-art in memory system design.

First, let us consider three different memory bus organizations and the corresponding miss penalties. For the purposes of this discussion, we will assume that the CPU generates 32-bit addresses and data; the cache block size is 4 words of 32 bits each.

9.20.1 Simple main memory

Figure 9.36 shows the organization of a simple memory system. It entertains a block read request to service cache misses. The CPU simply sends the block address to the physical memory. The physical memory internally computes the successive addresses of the block, retrieves the corresponding words from the DRAM, and sends them one after the other back to the CPU.

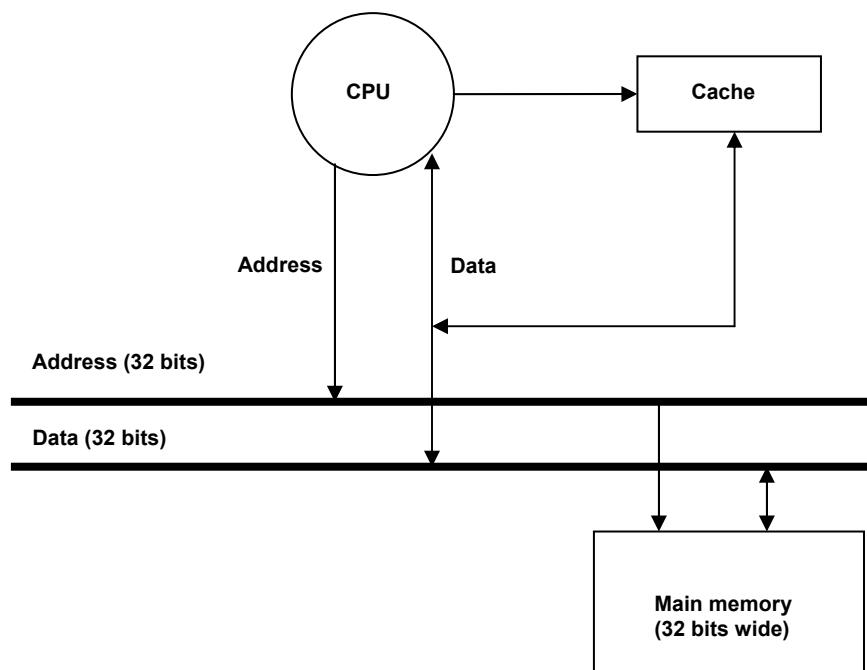


Figure 9.36: A simple memory system

Example 11:

Assume that the DRAM access time is 70 cycles. Assume that the bus cycle time for address or data transfer from/to the CPU/memory is 4 cycles. Compute the block transfer time for a block size of 4 words. Assume all 4 words are first retrieved from the DRAM before the data transfer to the CPU.

Answer:

Address from CPU to memory = 4 cycles

DRAM access time = $70 * 4 = 280$ cycles (4 successive words)

Data transfer from memory to CPU = $4 * 4 = 16$ cycles (4 words)

Total block transfer time = **300 cycles**

9.20.2 Main memory and bus to match cache block size

To cut down on the miss penalty, we could organize the processor-memory bus and the physical memory to match the block size. Figure 9.37 shows such an organization. This organization transfers the block from the memory to CPU in a single bus cycle and a single access to the DRAM. All 4 words of a block form a single row of the DRAM and thus accessed with a single block address. However, this comes at a significant hardware complexity since we need a 128-bit wide data bus.

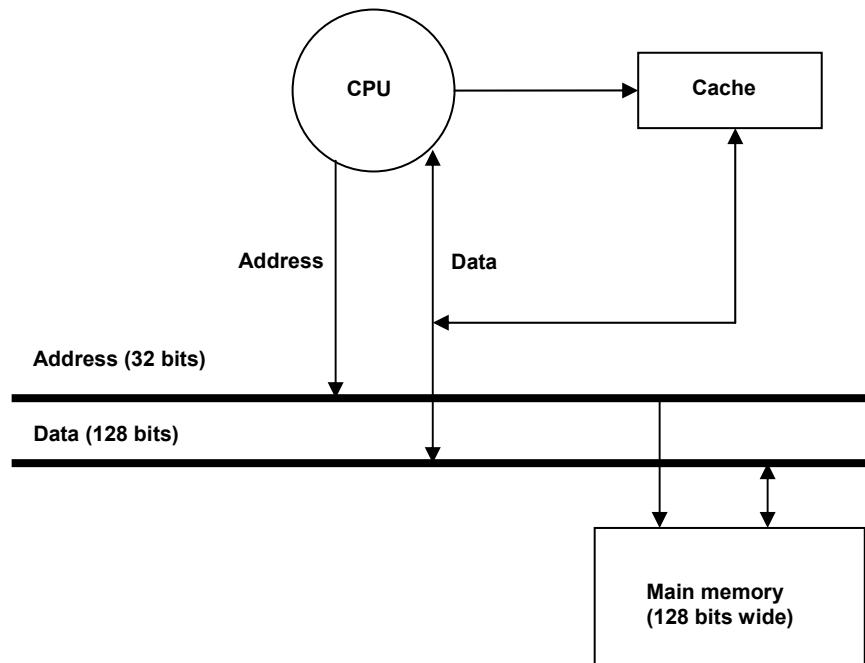


Figure 9.37: Main memory organization matching cache block size

Example 12:

Assume that the DRAM access time is 70 cycles. Assume that the bus cycle time for address or data transfer from/to the CPU/memory is 4 cycles. Compute the block transfer time with the memory system, wherein the bus width and the memory organization match the block size of 4 words.

Answer:

Address from CPU to memory = 4 cycles

DRAM access time = 70 cycles (all 4 words retrieved with a single DRAM access)

Data transfer from memory to CPU = 4 cycles

Total block transfer time = **78 cycles**

9.20.3 Interleaved memory

The increased bus width makes the previous design infeasible from a hardware standpoint. Fortunately, there is a way to achieve the performance advantage of the previous design with a little engineering ingenuity called *memory interleaving*. Figure 9.38 shows the organization of an interleaved memory system. The idea is to have multiple *banks* of memory. Each bank is responsible for providing a specific word of the cache block. For example, with a 4-word cache block size, we will have 4 banks labeled M0, M1, M2, and M3. M0 supplies word 0, M1 supplies word 1, M2 supplies word 3, and M3 supplies word 3. The CPU sends the block address that is simultaneously received by all the 4 banks. The banks work in parallel, each retrieving the word of the block that it is responsible for from their respective DRAM arrays. Once retrieved, the banks take turns sending the data back to the CPU using a standard bus similar to the first simple main memory organization.

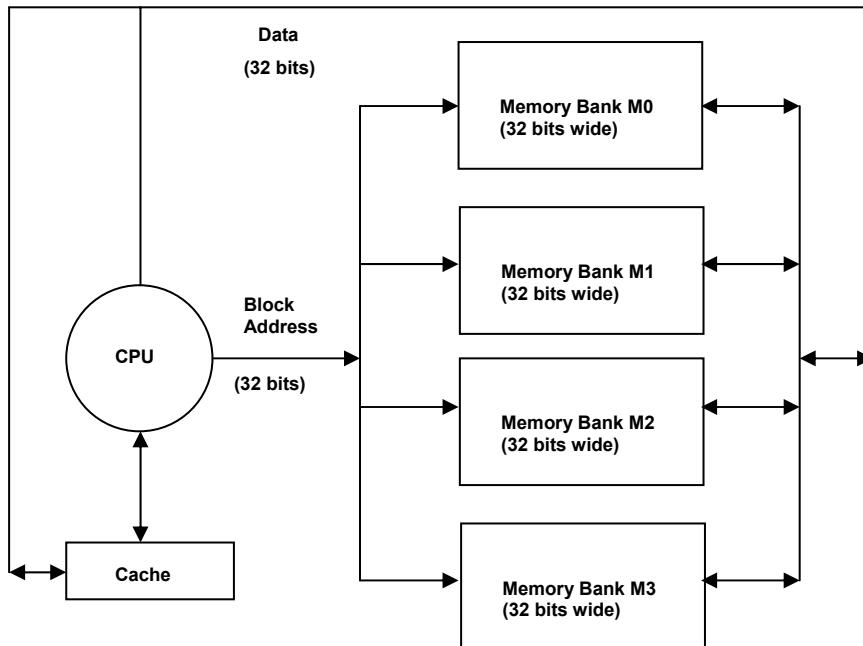


Figure 9.38: An interleaved main memory

The interleaved memory system focuses on the fact that DRAM access is the most time intensive part of the processor-memory interaction. Thus, interleaved memory achieves most of the performance of the wide memory design without its hardware complexity.

Example 13:

Assume that the DRAM access time is 70 cycles. Assume that the bus cycle time for address or data transfer from/to the CPU/memory is 4 cycles. Compute the block transfer time with an interleaved memory system as shown in Figure 9.38.

Answer:

Address from CPU to memory = 4 cycles (all four memory banks receive the address simultaneously).

DRAM access time = 70 cycles (all 4 words retrieved in parallel by the 4 banks)
Data transfer from memory to CPU = $4 * 4$ cycles (the memory banks take turns sending their respective data back to the CPU)

Total block transfer time = **90 cycles**

Keeping the processor busy is by far the most important consideration in the design of memory systems. This translates to getting the data from the memory upon a cache read miss to the processor as quickly as possible. Writing to an interleaved memory system need not be any different from a normal memory system. For the most part, the processor is shielded from the latency for writing to memory, with techniques such as write-buffers that we discussed in Section 9.7. However, many processor-memory buses support block write operations that work quite nicely with an interleaved memory system, especially for write-back of an entire cache block.

9.21 Elements of a modern main memory systems

Modern memory systems are a far cry from the simple ideas presented in the previous section. Interleaved memories are a relic of the past. With advances in technology, the conceptual idea of interleaving has made it inside the DRAM chips themselves. Let us explain how this works. Today, circa 2007, DRAM chips pack 2 Gbits in one chip.

However, for illustration purposes let us consider a DRAM chip that has 64×1 bit capacity. That is, if we supply this chip with a 6-bit address it will emit 1 bit of data. Each bit of the DRAM storage is called a *cell*. In reality, the DRAM cells are arranged in a rectangular array as shown in Figure 9.39. The 6-bit address is split into a row address *i* (3 bits) and a column address *j* (3 bits) as shown in the figure. To access one bit out of this DRAM chip, you would first supply the 3-bit row address *i* (called a *Row Access Strobe – RAS – request*). The DRAM chip pulls the whole row corresponding to *i* as shown in the figure. Then, you would supply the 3-bit column address *j* (called a *Column Access Strobe – CAS – request*). This selects the unique bit (the purple bit in the row buffer) given by the 6-bit address and sends it to the memory controller, which in turn sends it to the CPU.

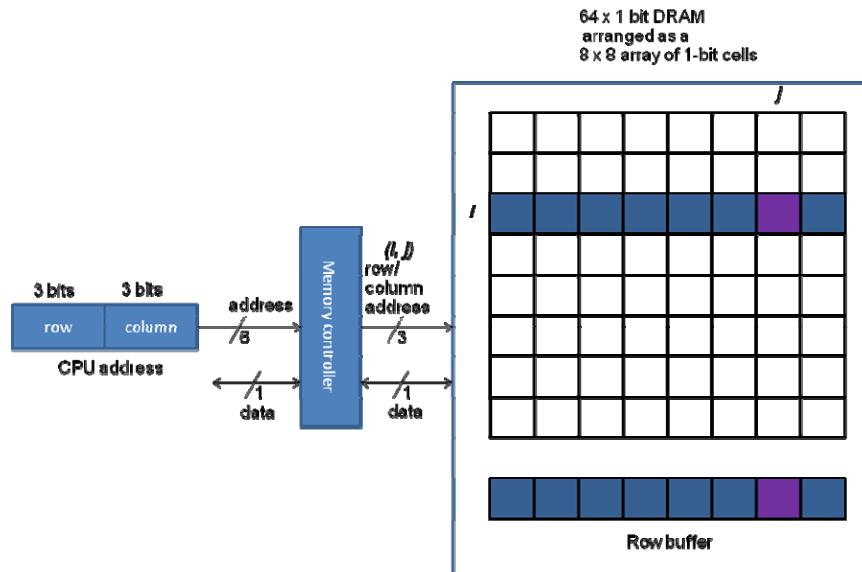


Figure 9.39: Accessing a 64x1 bit DRAM

For a 1 G-bit chip⁶, we will need a 32 K x 32 K array of cells. The size of the row buffer in this case would be 32 K-bits. It is instructive to understand what constitutes the *cycle time* of a DRAM. As we have said before, each cell in the DRAM is a capacitive charge. There is circuitry (not shown in the figure) that senses this capacitive charge for every cell in the chosen row and buffers the values as 0's and 1's in the corresponding bit positions of the row buffer. In this sense, reading a DRAM is destructive operation. Naturally, after reading the chosen row, the DRAM circuitry has to re-charge the row so that it is back to what it originally contained. This destructive read followed by the re-charge combined with the row and column address decode times add up to determine the cycle time of a DRAM. Reading a row out of the array into the buffer is the most time consuming component in the whole operation. You can quickly see that after all this work, only 1 bit, corresponding to the column address, is being used and all the other bits are discarded. We will shortly see (Section 9.21.1) how we may be able to use more of the data in the row buffer without discarding all of them.

We can generalize the DRAM structure so that each cell (i, j) instead of emitting 1 bit, emits some k bits. For example, a 1 M x 8-bit DRAM will have a 1 K x 1 K array of cells, with each cell of the array containing 8 bits. Address and data are communicated to the DRAM chip via *pins* on the chip (see Figure 9.40). One of the key design considerations in chip design is reducing the number of such input/output pins. It turns out the electrical current needed to actuate logic elements within a chip is quite small. However, to communicate logic signals in and out of the chip you need much larger currents. This means having bulky signal driving circuitry at the edges of the chip, which eats into the real estate available on the chip for other uses (such as logic and memory). This is the reason why the cells within a DRAM are arranged as a square array rather than a linear array, so that the same set of pins can be time multiplexed for sending the row and column addresses to the DRAM chip. This is where the RAS (row address

⁶ Recall that 1 G-bit is 2^{30} bits.

strobe) and CAS (column address strobe) terminologies come from in the DRAM chip parlance. The downside to sharing the pins for row and column addresses is that they have to be sent to the DRAM chip in sequence, thus increasing the cycle time of the DRAM chip.

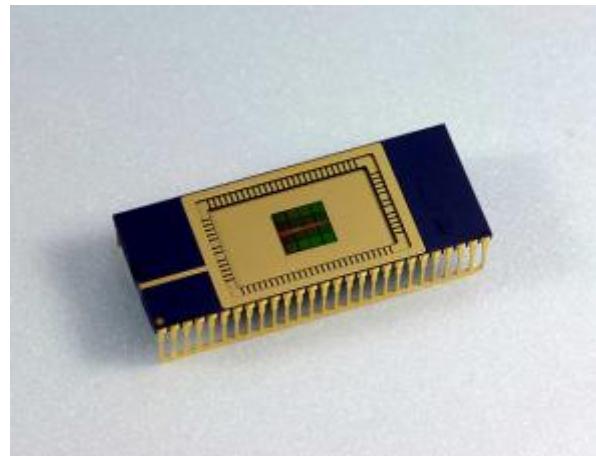


Figure 9.40: Samsung's 1 Gbit DRAM chip⁷

The following example illustrates how to design a main memory system out of these basic DRAM chips.

Example 14:

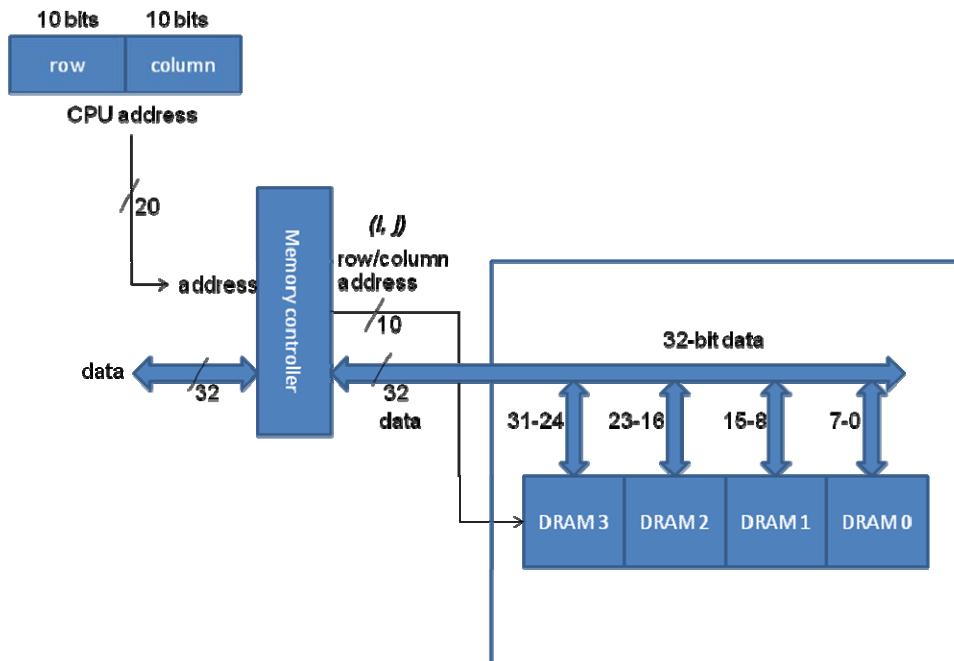


Figure 9.41: 32 Mbits memory system using 1 M x 8-bits DRAM chips

⁷ Source: <http://www.physorg.com/news80838245.html>

Design a main memory system given the following:

- Processor to memory bus
 - Address lines = 20
 - Data lines = 32
- Each processor to memory access returns a 32-bit word specified by the address lines
- DRAM chip details: 1 M x 8-bits

Answer:

$$\begin{aligned}\text{Total size of the main memory system} &= 220 \text{ words} \times 32\text{-bits/word} = 1 \text{ M words} \\ &= 32 \text{ Mbits}\end{aligned}$$

Therefore, we need 4 DRAM chips each of 1 M x 8-bits, arranged as shown in Figure 9.41.

It is straightforward to extend the design to a byte-addressed memory system. The following example shows how to build a 4 GB memory system using 1 Gbit DRAM chips.

Example 15:

Design a 4 GByte main memory system given the following:

- Processor to memory bus
 - Address lines = 32
 - Data lines = 32
- Each CPU word is 32-bits made up of 4 bytes
- CPU supports byte addressing
- The least significant 2-bits of the address lines specify the byte in the 32-bit word
- Each processor to memory access returns a 32-bit word specified by the word address
- DRAM chip details: 1 Gbits

Answer:

In the 32-bit address, the most significant 30 bits specify the word address.

$$\text{Total size of the main memory system} = 2^{30} \text{ words} \times 4 \text{ bytes/word} = 4 \text{ GBytes} = 32 \text{ Gbits}$$

Therefore, we need 32 DRAM chips, each of 1 Gbits arranged for convenience in a 2-dimensional array as shown in Figure 9.42. For writing individual byte in a word, the memory controller (not shown in the figure) will select the appropriate row in this 2-dimensional array and send the 15-bit RAS and CAS requests to that row along with other control signals (not shown in the figure). For reading a 32-bit word, it will send the 15-bit RAS and CAS requests to all the DRAMs so that a full 32-bit word will be returned to the controller.

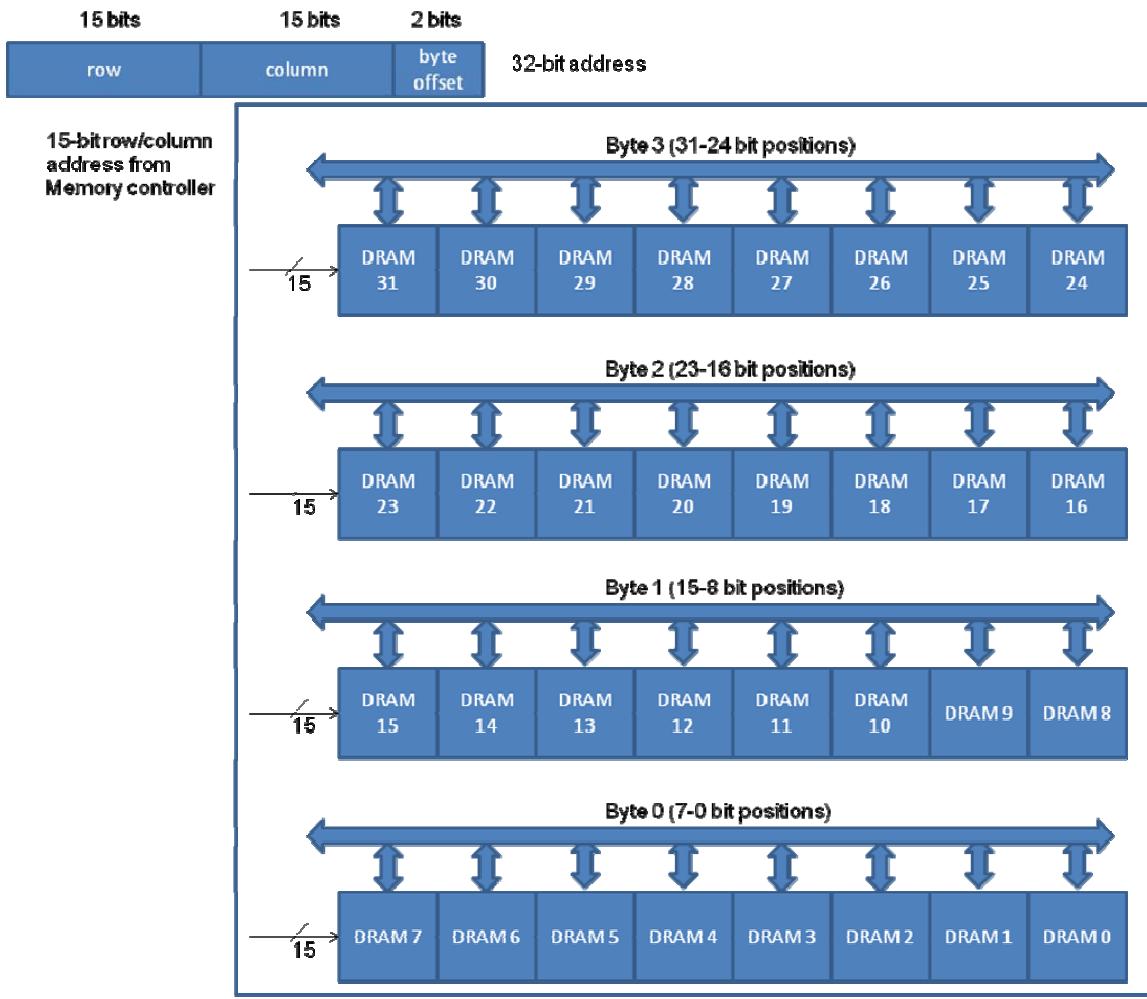


Figure 9.42: A 4 GByte memory system using a 1 Gbit DRAM chip

Manufacturers package DRAM chips in what are called *Dual In Line Memory Modules* (DIMMs). Figure 9.43 shows a picture of a DIMM. Typically, a DIMM is a small printed circuit board and contains 4-16 DRAM chips organized in an 8-byte wide datapath. These days, DIMM is the basic building block for memory systems.



Figure 9.43: Dual in Line Memory Module (DIMM)⁸

⁸ Source: <http://static.howstuffworks.com/gif/motherboard-dimm.jpg>

9.21.1 Page mode DRAM

Recall, what happens within a DRAM to read a cell. The memory controller first supplies it a row address. DRAM reads the entire selected row into a row buffer. Then the memory controller supplies the column address, and the DRAM selects the particular column out of the row buffer and sends the data to the memory controller. These two components constitute the *access time* of the DRAM and represent the bulk of a DRAM cycle time. As we mentioned earlier in this section, the rest of the row buffer is discarded once the selected column data is sent to the controller. As we can see from Figures 9.41 and 9.42, consecutive column addresses within the same row map to contiguous memory addresses generated by the CPU. Therefore, getting a block of data from memory translates to getting successive columns of the *same row* from a DRAM. Recall the trick involved in building interleaved memory (see Section 9.20.3). Basically, each memory bank stored a different word of the same block and sent it on the memory bus to the CPU in successive bus cycles. DRAMs support essentially the same functionality through a technique referred to as *fast page mode (FPM)*. Basically, this technique allows different portions of the row buffer to be accessed in successive CAS cycles without the need for additional RAS requests. This concept is elaborated in the following example.

Example 16:

The memory system in Example 15 is augmented with a processor cache that has a block size of 16 bytes. Explain how the memory controller delivers the requested block to the CPU on a cache miss.

Answer:

In Figure 9.44, the top part shows the address generated by the CPU. The memory controller's interpretation of the CPU address is shown in the bottom part of the figure. Note that the bottom two bits of the column address is the top two bits of the block offset in the CPU address. The block size is 16 bytes or 4 words. The successive words of the requested cache block is given by the 4 successive columns of row i with the column address only changing in the last two bits.

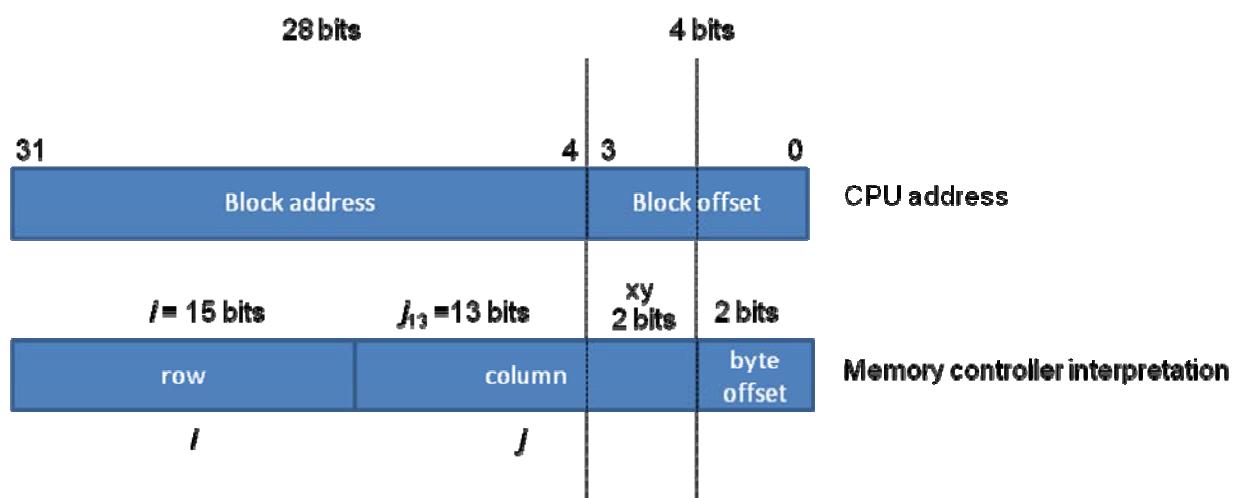


Figure 9.44: Memory controller's interpretation of a 32-bit CPU address

For example, if the CPU address for the block is (i, j) , where $j = 010000011000101$. Let us denote

j by $j_{13}xy$, where j_{13} represents the top 13 bits of the column address j .

To fetch the entire block from the DRAM array, the memory controller does the following:

1. Send RAS/CAS request to DRAM array for address (i, j)
2. Send 3 more CAS requests with addresses $j_{13}xy$, where $xy = 00, 10, 11$

Each of the CAS requests will result in the DRAM array sending 4 words (the first word returned will be the actual address that incurred a miss) successively. The memory controller will pipeline these 4 words in responses back to the CPU in four successive memory bus cycles.

Over the years, there have been numerous enhancements to the DRAM technology. We have given a flavor of this technology in this section. Hopefully, we have stimulated enough interest in the reader to look at more advanced topics in this area.

9.22 Performance implications of memory hierarchy

There is a hierarchy of memory that the CPU interacts with either explicitly or implicitly: processor registers, cache (several levels), main memory (in DRAM), and virtual memory (on the disk). The farther the memory is from the processor the larger the size and slower the speed. The cost per byte decreases as well as we move down the memory hierarchy away from the processor.

Type of Memory	Typical Size	Approximate latency in CPU clock cycles to read one word of 4 bytes
CPU registers	8 to 32	Usually immediate access (0-1 clock cycles)
L1 Cache	32 (Kilobyte) KB to 128 KB	3 clock cycles
L2 Cache	128 KB to 4 Megabyte (MB)	10 clock cycles
Main (Physical) Memory	256 MB to 4 Gigabyte (GB)	100 clock cycles
Virtual Memory (on disk)	1 GB to 1 Terabyte (TB)	1000 to 10,000 clock cycles (not accounting for the software overhead of handling page faults)

Table 9.1: Relative Sizes and Latencies of Memory Hierarchy circa 2006

As we already mentioned the actual sizes and speeds continue to improve yearly, though the relative speeds and sizes remain roughly the same. Just as a concrete example, Table 1 summarizes the relative latencies and sizes of the different levels of the memory hierarchy **circa 2006**. The clock cycle time **circa 2006** for a 2 GHz Pentium processor is 0.5 ns.

Memory hierarchy plays a crucial role in system performance. One can see that the miss penalty affects the pipeline processor performance for the currently executing program. More importantly, the memory system and the CPU scheduler have to be cognizant of the memory hierarchy in their design decisions. For example, page replacement by the memory manager wipes out the contents of the corresponding physical frame from all the levels of the memory hierarchy. Therefore, a process that page faulted may experience a significant loss of performance immediately after the faulting is brought into physical memory from the disk. This performance loss continues until the contents of the page fill up the nearer levels of the memory hierarchy.

CPU scheduling has a similar effect on system performance. There is a *direct* cost of context switch which includes saving and loading the process control blocks (PCBs) of the de-scheduled and newly scheduled processes, respectively. Flushing the TLB of the de-scheduled process forms part of this direct cost. There is an *indirect* cost of context switching due to the memory hierarchy. This costs manifests as misses at the various levels of the memory hierarchy all the way from the caches to the physical memory. Once the working set of the newly scheduled process gets into the nearer levels of the memory hierarchy then the process performance reaches the true potential of the processor. Thus, it is important that the *time quantum* used by the CPU scheduler take into account the true cost of context switching as determined by the effects of the memory hierarchy.

9.23 Summary

The following table provides a glossary of the important terms and concepts introduced in this chapter.

Category	Vocabulary	Details
Principle of locality (Section 9.2)	Spatial	Access to contiguous memory locations
	Temporal	Reuse of memory locations already accessed
Cache organization	Direct-mapped	One-to-one mapping (Section 9.6)
	Fully associative	One-to-any mapping (Section 9.11.1)
	Set associative	One-to-many mapping (Section 9.11.2)
Cache reading/writing (Section 9.8)	Read hit/Write hit	Memory location being accessed by the CPU is present in the cache
	Read miss/Write miss	Memory location being accessed by the CPU is not present in the cache
Cache write policy (Section 9.8)	Write through	CPU writes to cache and memory
	Write back	CPU only writes to cache; memory updated on replacement
Cache parameters	Total cache size (S)	Total data size of cache in bytes
	Block Size (B)	Size of contiguous data in one data block
	Degree of associativity (p)	Number of homes a given memory block can reside in a cache
	Number of cache lines (L)	S/pB
	Cache access time	Time in CPU clock cycles to check hit/miss in cache
	Unit of CPU access	Size of data exchange between CPU and cache
	Unit of memory transfer	Size of data exchange between cache and memory
	Miss penalty	Time in CPU clock cycles to handle a cache miss
Memory address interpretation	Index (n)	$\log_2 L$ bits, used to look up a particular cache line
	Block offset (b)	$\log_2 B$ bits, used to select a specific byte within a block
	Tag (t)	$a - (n+b)$ bits, where a is number of bits in memory address; used for matching with tag stored in the cache

Cache entry/cache block/cache line/set	Valid bit	Signifies data block is valid
	Dirty bits	For write-back, signifies if the data block is more up to date than memory
	Tag	Used for tag matching with memory address for hit/miss
	Data	Actual data block
Performance metrics	Hit rate (h)	Percentage of CPU accesses served from the cache
	Miss rate (m)	$1 - h$
	Avg. Memory stall	Misses-per-instruction _{Avg} * miss-penalty _{Avg}
	Effective memory access time (EMAT _i) at level i	$\text{EMAT}_i = T_i + m_i * \text{EMAT}_{i+1}$
	Effective CPI	$\text{CPI}_{\text{Avg}} + \text{Memory-stalls}_{\text{Avg}}$
Types of misses	Compulsory miss	Memory location accessed for the first time by CPU
	Conflict miss	Miss incurred due to limited associativity even though the cache is not full
	Capacity miss	Miss incurred when the cache is full
Replacement policy	FIFO	First in first out
	LRU	Least recently used
Memory technologies	SRAM	Static RAM with each bit realized using 6 transistors
	DRAM	Dynamic RAM with each bit realized using a single transistor
Main memory	DRAM access time	DRAM read access time
	DRAM cycle time	DRAM read and refresh time
	Bus cycle time	Data transfer time between CPU and memory
	Simulated interleaving using DRAM	Using page mode bits of DRAM

Table 9.2: Summary of Concepts Relating to Memory Hierarchy

9.24 Memory hierarchy of modern processors – An example

Modern processors employ several levels of caches. It is not unusual for the processor to have on-chip an L1 cache (separate for instructions and data), and a combined L2 cache for instructions and data. Outside the processor there may be an L3 cache followed by the main memory. With multi-core technology, the memory system is becoming even

more sophisticated. For example, AMD introduced the Barcelona⁹ chip in 2006¹⁰. This chip which has quad-core, has a per core L1 (split I and D) and L2 cache. This is followed by an L3 cache that is shared by all the cores. Figure 9.45 show the memory hierarchy of the Barcelona chip. The L1 cache is 2-way set-associative (64 KB for instructions and 64 KB for data). The L2 cache is 16-way set-associative (512 KB combined for instructions and data). The L3 cache is 32-way set-associative (2 MB shared among all the cores).

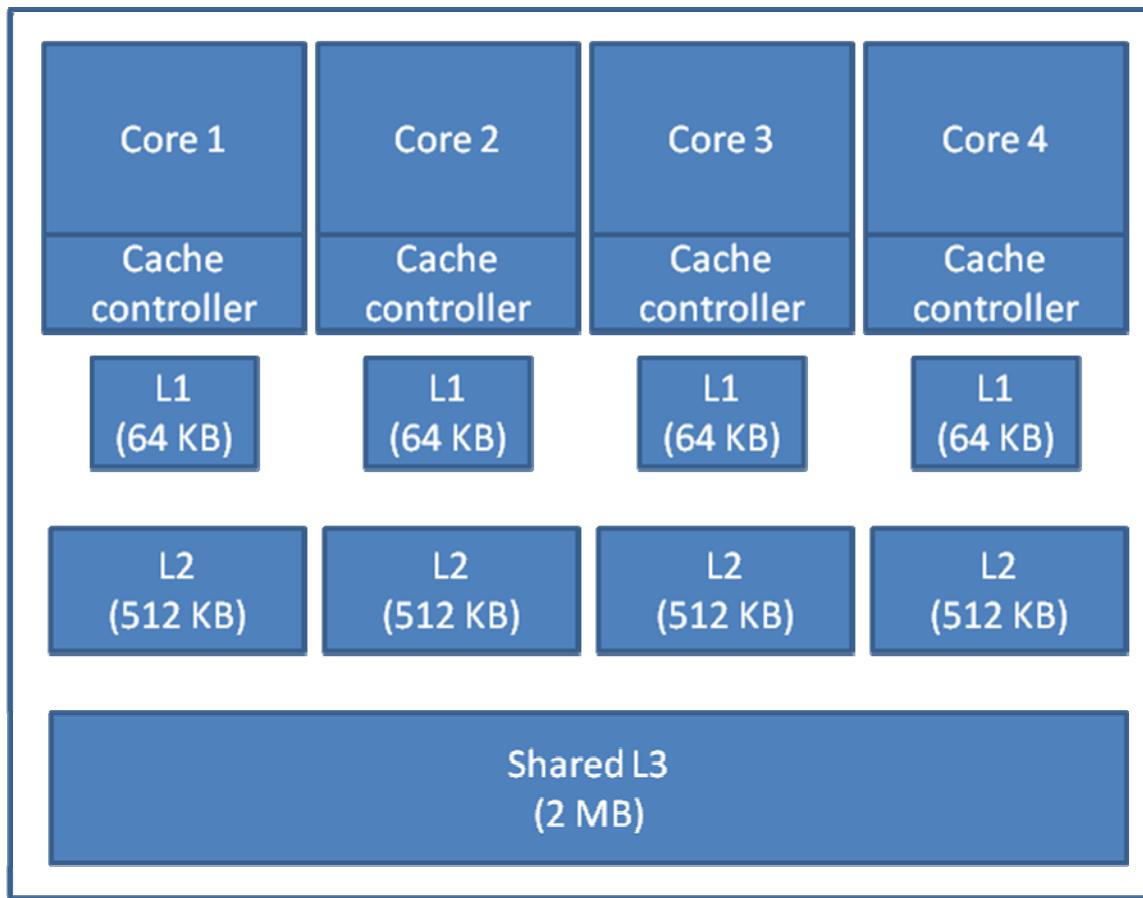


Figure 9.45: AMD's Barcelona chip

9.25 Review Questions

1. Compare and contrast spatial locality with temporal locality.
2. Compare and contrast direct-mapped, set-associative and fully associative cache designs.

⁹ Phenom is the brand name under which AMD markets this chip on desktops.

¹⁰ AMD Phenom processor data sheet:

http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/44109.pdf

3. A fellow student has designed a cache with the most significant bits containing the index and the least significant bits containing the tag. How well do you suppose this cache will perform?
4. In a direct-mapped cache with a t-bit long tag how many tag comparators would you expect to find and how many bits would they compare at a time?
5. Explain the purpose of dirty bits in a cache.
6. Give compelling reasons for a multilevel cache hierarchy.
7. What are the primary design considerations of caches? Discuss how these considerations influence the different levels of the memory hierarchy.
8. What is the motivation for increasing the block size of a cache?
9. What is the motivation for a set-associative cache design?
10. Answer True/False with justification: L1 caches usually have higher associativity compared to L2 caches.
11. Give three reasons for having a split I- and D-cache at the L1 level.
12. Give compelling reasons for a unified I- and D-caches at deeper levels of the cache hierarchy.
13. Answer True/False with justification: As long as there is an L2 cache, the L1 cache design can focus exclusively on matching its speed of access to the processor clock cycle time.
14. Your engineering team tells you that you can have 2 MB of on-chip cache total. You have a choice of making it one big L1 cache, two level L1 and L2 caches, split I- and D-caches at several levels, etc. What would be your choice and why? Consider hit time, miss rate, associativity, and pipeline structure in making your design decisions. For this problem, it is sufficient if you give qualitative explanation for your design decisions.
15. How big a counter do you need to implement a True LRU replacement policy for a 4-way set associative cache?
16. Consider the following memory hierarchy:
 - L1 cache: Access time = 2ns; hit rate = 99%
 - L2 cache: Access time = 5ns; hit rate = 95%
 - L3 cache: Access time = 10ns; hit rate = 80%
 - Main memory: Access time = 100ns

Compute the effective memory access time.

17. A memory hierarchy has the following resources:

L1 cache	2 ns access time	98% hit rate
L2 cache	10 ns access time	??
Memory	60 ns access time	

Assume that for each of L1 and L2 caches, a lookup is necessary to determine whether a reference is a hit or miss. What should be the hit rate of L2 cache to ensure that the effective memory access time is no more than 3 ns.

18. You are designing a cache for a 32-bit processor. Memory is organized into words but byte addressing is used. You have been told to make the cache 2-way set associative with 64 words (256 bytes) per block. You are allowed to use a total of 64K words (256K bytes) for the data (excluding tags, status bits, etc.)

Sketch the layout of the cache.

Show how a CPU generated address is interpreted for cache lookup into block offset, index, and tag.

19. From the following statements regarding cache memory, select the ones that are True.

- It can usually be manipulated just like registers from the instruction-set architecture of a processor
- It cannot usually be directly manipulated from the instruction-set architecture of a processor
- It is usually implemented using the same technology as the main memory
- It is usually much larger than a register file but much smaller than main memory

20. Distinguish between cold miss, capacity miss, and conflict miss.

21. Redo Example 7 with the following changes:

- 4-way set associative cache with 16 blocks total
- LRU replacement policy
- reference string:

0, 1, 8, 0, 1, 16, 24, 32, 8, 8, 0, 5, 6, 2, 1, 7, 10, 0, 1, 3, 11, 10, 0, 1, 16, 8

22. Redo Example 8 with the following changes:

- 32-bit virtual address, 24-bit physical address, pagesize 8 Kbytes, and direct mapped TLB with 64 entries

23. Explain virtually indexed physically tagged cache. What is the advantage of such a design? What are the limitations of such a design?
24. Explain page coloring. What problem is this technique aimed to solve? How does it work?
25. Redo Example 10 with the following changes:
 - virtual address is 64 bits
 - page size is 8 Kbytes
 - cache parameters: 2-way set associative, 512 Kbytes total size, and block size of 32 bytes
26. Redo Example 15 with the following changes:
 - 64-bit address and data lines
 - 64-bit CPU word
 - use a 1 Gbit DRAM chip
27. Explain page mode DRAM.

Chapter 10 Input/Output and Stable Storage

(Revision number 11)

The next step in demystifying what is inside a box is understanding the I/O subsystem. In Chapter 4, we discussed interrupts, a mechanism for I/O devices to grab the attention of the processor. In this chapter, we will present more details on the interaction between processor and I/O devices, as well as different methods of transferring data among the processor, memory, and I/O devices.

We start out by discussing the basic paradigms for the CPU to communicate with I/O devices. We then identify the hardware mechanisms needed to carry out such communication in hardware, as well as the details of the buses that serve as the conduits for transferring data between the CPU and I/O. Complementary to the hardware mechanisms is the operating system entity, called a device driver that carries out the actual dialogue between the CPU and each specific I/O device. Stable storage, often referred to as the *hard drive*, is unquestionably one of the most important and intricate member of the I/O devices found inside a box. We discuss the details of the hard disk including scheduling algorithms for orchestrating the I/O activities.

10.1 Communication between the CPU and the I/O devices

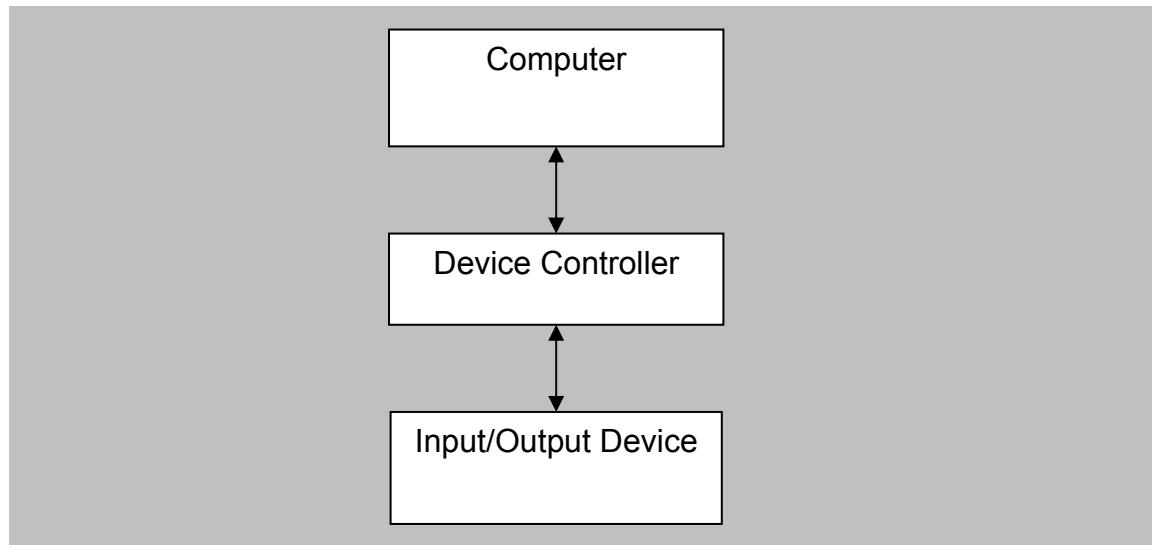


Figure 10.1 Relationship of device controller to other components

Although we started our exploration of the computer system with the processor, most users of computers may not even be aware of the processor that is inside the gadget they are using. Take for example the cellphone or the iPod. It is the functionality provided by iPod that attracts a teenager (or even adults for that matter) to it. An iPod or a cellphone gets its utility from the input/output devices that each provides for interacting with it. Thus, knowing how I/O devices interact with the rest of the computer system is a key component to unraveling the “box.” Although there are a wide variety of I/O devices, their connection to the rest of the system is quite similar. As we have seen so far, the

processor executes instructions in its instruction set repertoire. LC-2200 has no special instruction that would make it communicate with a CD player or a speaker directly. Let's understand how a device such an iPod plays music.

A special piece of hardware known as a *device controller* acts as an intermediary between an I/O device and the computer. This controller knows how to communicate with both the I/O device and with the computer.

10.1.1 Device controller

To make this discussion concrete, let us consider a very simple device, the *keyboard*. The device itself has circuitry inside it to map the mechanical action of tapping on a key to a binary encoding of the character that the key represents. This binary encoding, usually in a format called ASCII (*American Standard Code for Information Interchange*), has to be conveyed to the computer. For this information exchange to happen, two things are necessary. First, we need temporary space to hold the character that was typed. Second, we have to grab the attention of the processor to give it this character. This is where the device controller comes in. Let us consider the minimal smarts needed in the keyboard device controller. It has two registers: *data register*, and *status register*. The data register is the storage space for the character typed on the keyboard. As the name suggests, the status register is an aggregation of the current state of information exchange of the device with the computer.

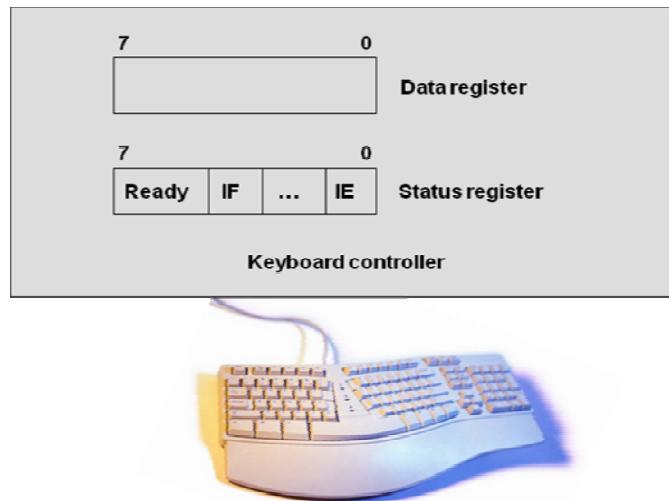


Figure 10.2: Keyboard controller

With respect to the keyboard, the state includes:

- A bit, called *ready* bit that represents the answer to the question, “is the character in the data register new (i.e. not seen by the processor yet)?”
- A bit, called *interrupt enable (IE)* that represents the answer to the question, “is the processor allowing the device to interrupt it?”

- A bit, called *interrupt flag (IF)* that serves to answer the question, “is the controller ready to interrupt the processor?

Figure 10.2 shows such a minimal keyboard controller. Depending on the sophistication of the device, additional status information may need to be included. For example, if the input rate of the device exceeds the rate of information transfer to the processor, a *data overrun flag* may record this status on the controller.

Sometimes it may be a good idea to separate the *status* that comes from the device, and the *command* that comes from the processor. The keyboard is a simple enough device that the only command from the processor is to turn on or off the interrupt enable bit. A more sophisticated device (for example a camera) may require additional commands (take picture, pan, tilt, etc.). In general, a device controller may have a set of registers through which it interacts with the processor.

10.1.2 Memory Mapped I/O

Next, we investigate how to connect the device controller to the computer. Somehow, we have to make the registers in the controller visible to the processor. A simple way to accomplish this is via the processor-memory bus as shown in Figure 10.3. The processor reads and writes to memory using load/store instructions. If the registers in the controller (data and status) appear as memory locations to the CPU then it can simply use the same load/store instructions to manipulate these registers. This technique, referred to as *memory mapped I/O*, allows interaction between the processor and the device controller without any change to the processor.

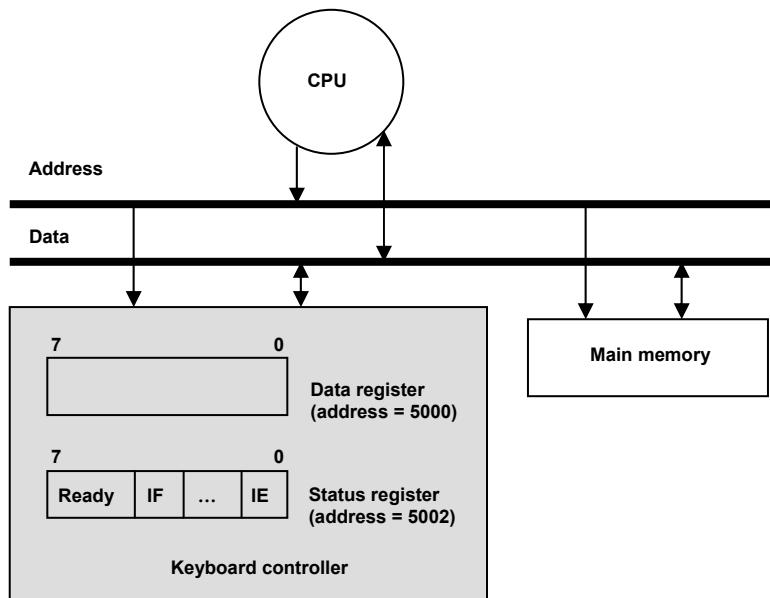


Figure 10.3: Connecting the device controller to the processor-memory bus

It is a real simple trick to make the device registers appear as memory locations. We give the device registers unique memory addresses. For example, let us arbitrarily give the

data register the memory address 5000 and the status register the memory address 5002. The keyboard controller has smarts (i.e. circuitry) in it to react to these two addresses appearing on the address bus. For example, if the processor executes the instruction:

Load r1, Mem[5000]

The controller recognizes that address 5000 corresponds to the data register, and puts the contents of the data register on the data bus. The processor has no idea that it is coming from a special register on the controller. It simply takes the value appearing on the data bus and stores it into register **r1**.

This is the power of memory mapped I/O. Without adding any new instructions to the instruction set, we have now integrated the device controller into the computer system. You may be wondering if this technique causes confusion between the memory that also reacts to addresses appearing on the bus and the device controllers. The basic idea is to reserve a portion of the address space for such device controllers. Assume that we have a 32-bit processor and that we wish to reserve 64Kbytes for I/O device addresses. We could arbitrarily designate addresses in the range 0xFFFF0000 through 0xFFFFFFF for I/O devices, and assign addresses within this range to the devices we wish to connect to the bus. For example, we may assign 0xFFFF0000 as the data register and 0xFFFF0002 as the status register for the keyboard controller. The design of the keyboard device controller considers this assignment and reacts to these addresses. What this means is that every device controller has circuitry to decode an address appearing on the bus. If an address appearing on the bus corresponds to one assigned to this controller, then it behaves like “memory” for the associated command on the bus (read or write). Correspondingly, the design of the memory system will be to ignore addresses in the range designated for I/O. Of course, it is a matter of convention as to the range of addresses reserved for I/O. The usual convention is to reserve high addresses for I/O device registers.

You are perhaps wondering how to reconcile this discussion with the details of the memory hierarchy that we just finished learning about in Chapter 9. If a memory address assigned to a device register were present in the cache, wouldn't the processor get a stale value from the cache, instead of the contents of the device register? This is a genuine concern and it is precisely for this reason that cache controllers provide the ability to treat certain regions of memory as “uncachable”. Essentially, even though the processor reads a device register as if it were a memory location, the cache controller has been *set a priori* to not encache such locations but read them afresh every time the processor accesses them.

The advantage of memory mapped I/O is that no special I/O instructions are required but the disadvantage is that a portion of the memory address space is lost to device registers, and hence not available for users or the operating system for code and data. Some processors provide special I/O instructions and connect I/O devices to a separate I/O bus. Such designs (called *I/O mapped I/O*) are especially popular in the domain of embedded systems where memory space is limited. However, with modern general-purpose processors with a 64-bit address space, reserving a small portion of this large address space for I/O seems a reasonable design choice. Thus, memory mapped I/O is the design

of choice for modern processors, especially since it integrates easily into the processor architecture without the need for new instructions.

10.2 Programmed I/O

Now that we know how to interface a device via a device controller to the processor, let us turn our attention to moving data back and forth between a device and the processor. *Programmed I/O (PIO)* refers to writing a computer program that accomplishes such data transfer. To make the discussion concrete, we will use the keyboard controller we introduced earlier.

Let us summarize the actions of the keyboard controller:

1. It sets the ready bit in the status register when a new character gets into the data register.
2. Upon the CPU reading the data register, the controller clears the ready bit.

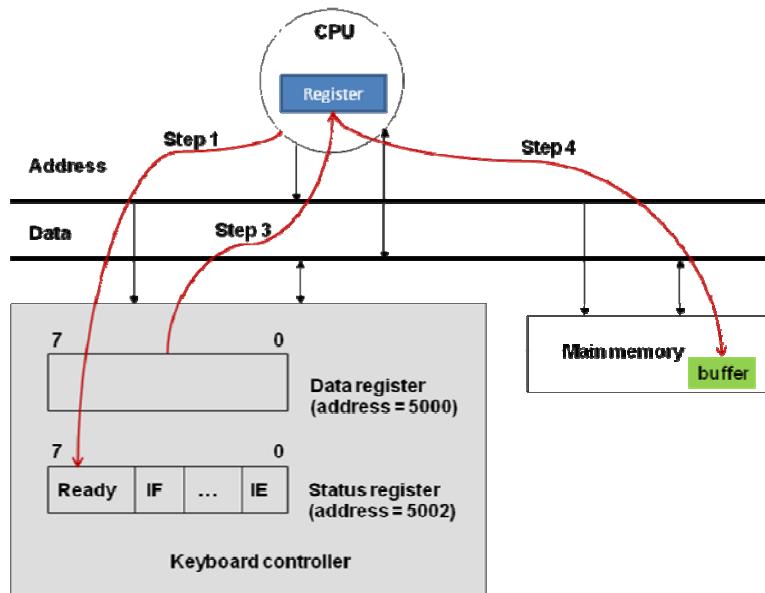


Figure 10.4: An example of PIO Data Transfer

With the above semantics of the keyboard controller, we can write a simple program to move data from the keyboard to the processor (shown pictorially in Figure 10.4):

1. Check the ready bit (step 1 in Figure 10.4).
2. If not set go to step 1.
3. Read the contents of the data register (step 3 in Figure 10.4).
4. Store the character read into memory (step 4 in Figure 10.4).
5. Go to step 1.

Step 1 and 2 constitute the handshake between the processor and the device controller. The processor continuously checks if the device has new data through these two steps. In other words, the processor *polls* the device for new data. Consider the speed differential between the processor and a device such as a keyboard. A 1 GHz processor executes an instruction in roughly 1 ns. Even if one were to type at an amazing speed, say of 300

characters a minute, the controller inputs a character only once every 200 ms. The processor would have been polling the ready bit of the status register several million times before it gets to input a character. This is an inefficient use of the processor resource. Instead of polling, the processor could enable the interrupt bit for a device and upon an interrupt execute the instructions as above to complete the data transfer. The operating system schedules other programs to run on the CPU, and context switches to handle the interrupts as we described in Chapter 4.

Programmed data transfer, accomplished either by polling or interrupt, works for slow speed devices (for example, keyboard and mouse) that typically produce data *asynchronously*. That is the data production is not rhythmic with any clock pulse. However, high-speed devices such as disks produce data *synchronously*. That is the data production follows a rhythmic clock pulse. If the processor fails to pick up the data when the device has it ready, then the device may likely overwrite it with new data, leading to data loss. The memory bus bandwidth of state-of-the-art processors is around 200Mbytes/sec. All the entities on the bus, processor and device controllers, share this bandwidth. A state-of-the-art disk drive produces data at the rate of 150Mbytes/sec. Given the limited margin between the production rate of data and available memory bus bandwidth, it is just not feasible to orchestrate the data transfer between memory and a synchronous high-speed device such as a disk through programmed I/O.

Further, even for slow speed devices using the processor to orchestrate the data transfer through programmed I/O is an inefficient use of the processor resource. In the next subsection, we introduce a different technique for accomplishing this data transfer.

10.3 DMA

Direct Memory Access (DMA), as the name suggests is a technique where the device controller has the capability to transfer data between itself and memory without the intervention of the processor.

The transfer itself is initiated by the processor, but once initiated the device carries out the transfer. Let us try to understand the smarts needed in the DMA controller. We will consider that the controller is interfacing a synchronous high-speed device such as a disk. We refer to such devices as *streaming devices*: once data transfer starts in either direction (“from” or “to” the device), data moves in or out the device continuously until completion of the transfer.

As an analogy, let us say you are cooking. You need to fill a large cooking vessel on the stove with water. You are using a small cup, catching water from the kitchen faucet, pouring it into the vessel, and going back to refilling the cup and repeating this process until the vessel is full. Now, you know that if you keep the faucet open water will continue to stream. Therefore, you use the cup as a *buffer* between the faucet that runs continuously and the vessel, turning the faucet on and off as needed.

A streaming device is like a water faucet. To read from the device, the device controller turns on the device, gets a stream of bits, turns it off, transfers the bits to memory, and

repeats the process until done with the entire data transfer. It does the reverse to write to the device. To make it concrete let us focus on reading from the device into memory. Let us see what is involved in transferring the bits to memory. The controller acquires the bus and sends one byte (or whatever the granularity of the bus transfer is) at a time to memory. Recall that the bus is a shared resource. There are other contenders for the bus including the processor. Therefore, the data transfer between the device controller and the memory is asynchronous; worse yet, the controller may not get the bus for quite a while if there are other higher priority requests for the bus. To smooth out this dichotomy of synchronous device and an asynchronous bus, the controller needs a *hardware buffer* similar to the cup in the above analogy. It is fairly intuitive to reason that the buffer should be as big as the *unit of synchronous transfer* between the device and device controller. For example, if the device is a camcorder, then the size of a single image frame (e.g., 100 K pixels) may be the smallest unit of synchronous data transfer between the device and the controller. Thus, in the case of a device controller for a camcorder, the size of the hardware buffer should be at least as big as one image frame.

To initiate a transfer, the processor needs to convey four pieces of information to the device controller: *command*, *address on the device*, *memory buffer address*, and *amount of data transfer*. Note that the data transfer is between contiguous regions of the device space and memory space. In addition, the controller has a status register to record the device status. As in the case of the keyboard controller, we can assign five memory-mapped registers in the controller: *command*, *status*, *device address*, *memory buffer address*, and *count*. All of these registers have unique memory addresses assigned to them. Figure 10.5 shows a simplified block diagram for a DMA controller for such a streaming device.

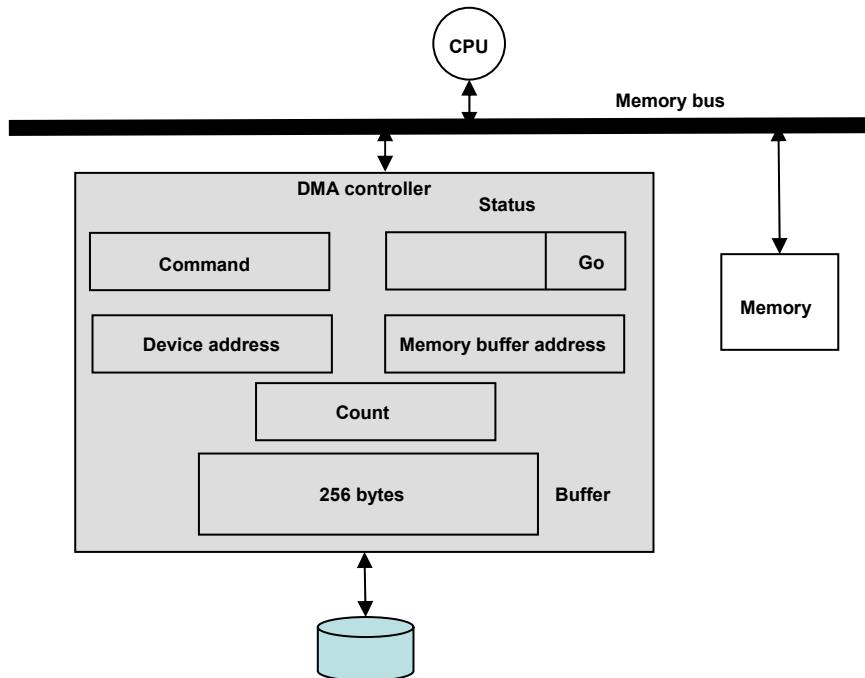


Figure 10.5: DMA controller

For example, to transfer N bytes of data from a memory buffer at address M to the device at address D , the CPU executes the following instructions:

1. Store N into the *Count* register.
2. Store M into the *Memory buffer address* register.
3. Store D into the *Device address* register.
4. Store *write to the device* command into the *Command* register.
5. Set the *Go* bit in the *Status* register.

Note that all of the above are simple *memory store* instructions (so far as the CPU is concerned) since these registers are memory mapped.

The device controller is like a CPU. Internally, it has the data path and control to implement each of a set of instructions it gets from the processor. For instance, to complete the above transfer (see Figure 10.6) the controller will access the memory bus repeatedly to move into its buffer, N contiguous bytes starting from the memory address M . Once the buffer is ready, it will initiate the transfer of the buffer contents into the device at the specified device address. If the processor had enabled interrupt for the controller, then it will interrupt the processor upon completion of the transfer.

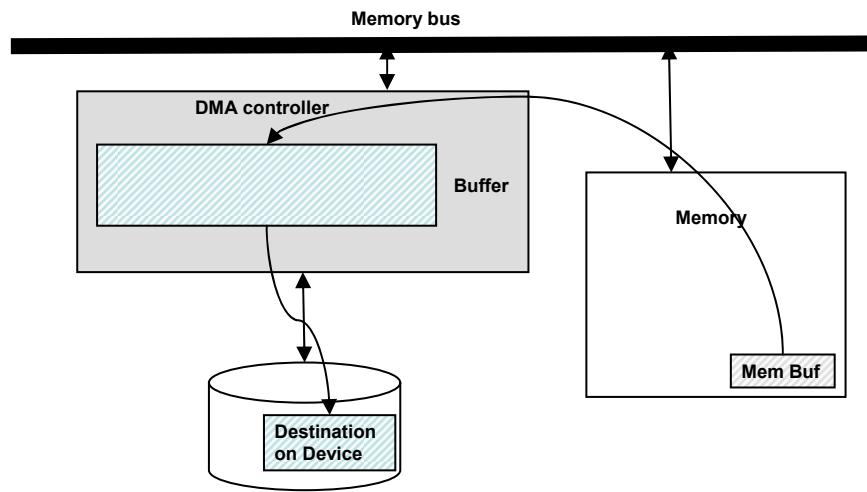


Figure 10.6: An example of DMA Data Transfer

The device controller competes with the processor for memory bus cycles, referred to as *cycle stealing*. This is an archaic term and stems from the fact that the processor is usually the bus master. The devices *steal* bus cycles when the processor does not need them. From the discussion on memory hierarchies in Chapter 9, we know that the processor mostly works out of the cache for both instructions and data. Therefore, devices stealing bus cycles from the processor does not pose a problem with a well-balanced design that accounts for the aggregate bandwidth of the memory bus being greater than the cumulative needs of the devices (including the processor) connected to it.

10.4 Buses

The system bus (or memory bus) is a key resource in the entire design. Functionally, the bus has the following components:

- Address lines
- Data lines
- Command lines
- Interrupt lines
- Interrupt acknowledge lines
- Bus arbitration lines

Electrically, high-performance systems run these wires in parallel to constitute the system bus. For example, a processor with 32-bit addressability will have 32 address lines. The number of data lines on the bus depends on the command set supported on the bus. As we have seen in the Chapter on memory hierarchy (Chapter 9), it is conceivable that the data bus is wider than the word width of the processor to support large cache block size. The command lines encode the specific command to the memory system (read, write, block read, block write, etc.), and hence should be sufficient in number to carry the binary encoding of the command set. We have seen in Chapter 4, details of the interrupt handshake between the processor and I/O devices. The number of interrupt lines (and acknowledgment lines) corresponds to the number of interrupt levels supported. In Chapter 9, we have emphasized the importance of the memory system. In a sense, the performance of the system depends crucially on the performance of the memory system. Therefore, it is essential that the system bus, the porthole for accessing the memory, be used to its fullest potential. Normally, during the current bus cycle, devices compete to acquire the bus for the next cycle. Choosing the winner for the next bus cycle happens before the current cycle completes. A variety of techniques exists for *bus arbitration*, from centralized scheme (at the processor) to more distributed schemes. Detailed discussion of bus arbitration schemes is beyond the scope of this textbook. The interested reader is referred to advanced textbooks in computer architecture for more elaboration on this topic¹.

Over the years, there have been a number of standardization efforts for buses. The purpose of standardization is to allow third party vendors to interface with the buses for developing I/O devices. Standardization poses interesting challenge to “box makers”. On the one hand, such standardization helps a box maker since it increases the range of peripherals available on a given platform from third party sources. On the other hand, to keep the competitive edge, most “box makers” resist such standardization. In the real world, we will see a compromise. System buses tend to be proprietary and not adhering to any published open standards. On the other hand, I/O buses to connect peripherals tend to conform to standards. For example, *Peripheral Component Interchange (PCI)* is an open standard for I/O buses. Most box makers will support PCI buses and provide internal *bridges* to connect the PCI bus to the system bus (see Figure 10.7).

¹ For example, Hennessy and Patterson, Computer Architecture: A Quantitative Approach, Morgan-Kauffman publishers.

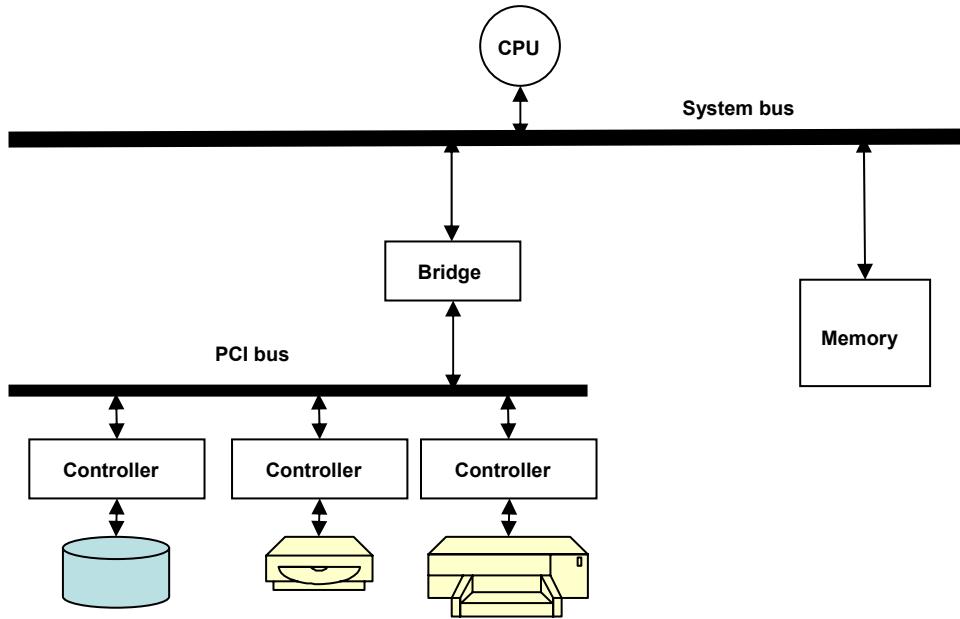


Figure 10.7: Coexistence of Standard buses and system buses

Some bus designs may electrically share the lines for multiple functions. For example, the PCI bus uses the same 32 wires for both addresses and data. The Command lines and the details of the protocol determine whether these wires carry data or addresses at any point of time.

An important consideration in the design of buses is the control regime. Buses may operate in a *synchronous* fashion. In this case, a common *bus clock* line (similar to the CPU clock) orchestrates the protocol action on individual devices. Buses may also operate in an *asynchronous* fashion. In this case, the bus *master* initiates a bus action; the bus action completes when the *slave* responds with a reply. This *master-slave* relationship obviates the need for a bus clock. To increase bus utilization, high-performance computer systems use *split transaction buses*. In this case, several independent conversations can simultaneously proceed. This complicates the design considerably in the interest of increasing the bus throughput. Such advanced topics are the purview of more advanced courses in computer architecture.

10.5 I/O Processor

In very high-performance systems used in enterprise level applications such as web servers and database servers, *I/O processors* decouple the I/O chores from the main processor. An I/O processor takes a *chain* of commands for a set of devices and carries them out without intervention from or to the main processor. The I/O processor reduces the number of interruptions that the main processor has to endure. The main processor sets up an I/O program in shared memory (see Figure 10.8), and starts up the I/O processor. The I/O processor completes the program and then interrupts the main processor.

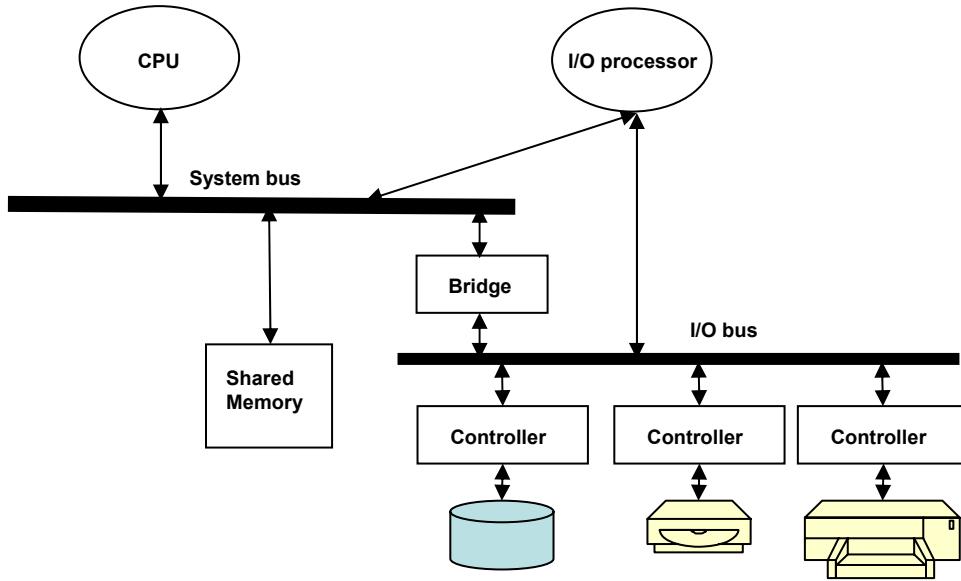


Figure 10.8: I/O processor

IBM popularized I/O processors in the era of mainframes. As it turns out, even today mainframes are popular for large enterprise servers. IBM gave these I/O processors the name *channels*. A *multiplexer channel* controls slow speed character-oriented devices such as a bank of terminals or displays. A *block multiplexer channel* controls several medium speed block-oriented devices (stream-oriented devices as we refer to them in this chapter) such as tape drives. A *selector channel* services a single high-speed stream-oriented device such as a disk.

An I/O processor is functionally similar to a DMA controller. However, it is at a higher level since it can execute a series of commands (via an I/O program) from the CPU. On the other hand, the DMA controller works in *slave mode* handling one command at a time from the CPU.

10.6 Device Driver

Device driver is a part of the operating system, and there is a device driver for controlling each device in the computer system. Figure 10.9 shows the structure of the system software, specifically the relationship between the device drivers and the rest of the operating system such as the CPU scheduler, I/O scheduler and the memory manager.

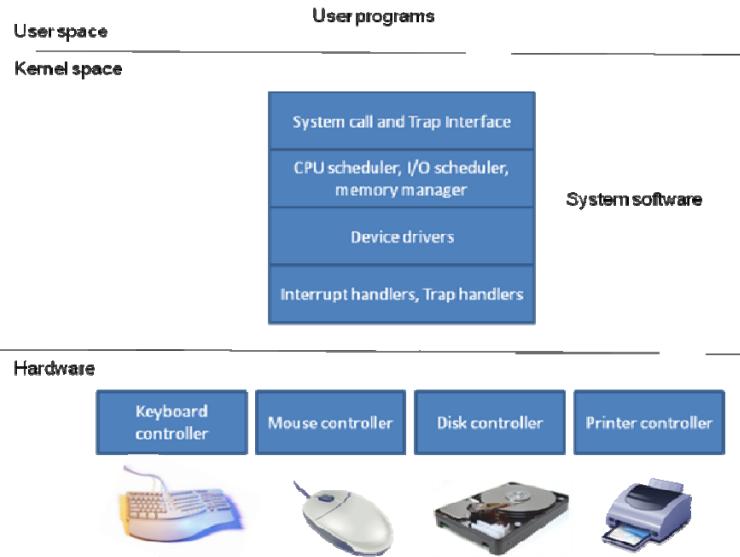


Figure 10.9: Place of the device driver in the system software stack

The specifics of the device driver software depend on the characteristics of the device it controls. For example, a keyboard driver may use interrupt-driven programmed I/O for data transfer between the keyboard controller and the CPU. On the other hand, the device driver for the hard-drive (disk) would set up the descriptors for DMA transfer between the disk controller and the memory and await an interrupt that indicates completion of the data transfer. You can see the power of abstraction at work here. It does not really matter what the details of the device is as far as the device driver is concerned. For example, the device could be some slow speed device such as a keyboard or a mouse. As far as the device driver is concerned it is simply executing code similar to the pseudo code we discussed in Section 10.2 for moving the data from the device register in the controller to the CPU. Similarly, the high-speed device may be a disk, a scanner, or a video camera. As far as the data movement is concerned, the device driver for such high-speed devices does pretty much the same thing as we discussed in the pseudo code in Section 10.3, namely, set up the descriptor for the DMA transfer and let the DMA controller associated with the device take charge. Of course, the device driver needs to take care of control functions specific to each device. As should be evident by now, there is an intimate relationship between the device controller and the device driver.

10.6.1 An Example

As a concrete example, consider a device driver for a pan-tilt-zoom (PTZ) camera. The device controller may provide a memory mapped command register for the CPU to specify the control functions such as the zoom level, tilt level, and x-y coordinates of the space in front of the camera for focusing the camera view. Similarly, the controller may provide commands to start and stop the camera. In addition to these control functions, it may implement a DMA facility for the data transfer. Table 10.1 summarizes the capabilities of the device controller for a PTZ camera.

Command	Controller Action
pan($\pm\theta$)	Pan the camera view by $\pm\theta$
tilt($\pm\theta$)	Tilt camera position by $\pm\theta$
zoom($\pm z$)	Zoom camera focus by $\pm z$
Start	Start camera
Stop	Stop camera
memory buffer(M)	Set memory buffer address for data transfer to M
number of frames (N)	Set number of frames to be captured and transferred to memory to N
enable interrupt	Enable interrupt from the device
disable interrupt	Disable interrupt from the device
start DMA	Start DMA data transfer from camera

Table 10.1: Summary of Commands for a PTZ Camera Controller

Device driver for such a device may consist of the following modules shown as pseudo code.

```
// device driver: camera
// The device driver performs several functions:
//   control_camera_position;
//   convey_DMA_parameters;
//   start/stop data transfer;
//   interrupt_handler;
//   error handling and reporting;
// Control camera position
camera_position_control
    (angle pan_angle; angle tilt_angle; int z)
{
    pan(pan_angle);
    tilt(tilt_angle);
    zoom(z);
}

// Set up DMA parameters for data transfer
camera_DMA_parameters(address mem_buffer;int num_frames)
{
    memory_buffer(mem_buffer);
    capture_frames(num_frames);
}
```

```

// Start DMA transfer
camera_start_data_transfer()
{
    start_camera();
    start_DMA();
}

// Stop DMA transfer
camera_stop_data_transfer();
{
    // automatically aborts data transfer
    // if camera is stopped;
    stop_camera();
}

// Enable interrupts from the device
camera_enable_interrupt()
{
    enable_interrupt();
}

// Disable interrupts from the device
camera_disable_interrupt()
{
    disable_interrupt();
}

// Device interrupt handler
camera_interrupt_handler()
{
    // This will be coded similar to any
    // interrupt handler we have seen in
    // chapter 4.
    //
    // The upshot of interrupt handling may
    // to deliver "events" to the upper layers
    // of the system software (see Figure 10.9)
    // which may be one of the following:
    //      - normal I/O request completion
    //      - device errors for the I/O request
    //
}

}

```

This simplistic treatment of a device driver is meant to give you the confidence that writing such a piece of software is a straightforward exercise similar to any programming assignment. We should point out that modern devices might be much more

sophisticated. For example, a modern PTZ camera may incorporate the device controller in the camera itself so that the level of interface presented to the computer is much higher. Similarly, the camera may plug directly into the local area network (we cover more on networking in Chapter 13), so communicating with the device may be similar to communicating with a peer computer on a local area network using the network protocol stack.

The main point to take away from this pseudo-code is that the code for a device driver is straightforward. From your programming experience, you already know that writing any program required taking care of corner cases (such as checking array bounds) and dealing with exceptions (such as checking the return codes on system calls). Device driver code is no different. What makes device driver code more interesting or more challenging depending on your point of view is that a number of things could happen that may have nothing to do with the logic of the device driver code. It is possible to plan for some of these situations in the device driver code. Examples include:

1. The parameters specified in the command to the controller are illegal (such as illegal values for pan, tilt, zoom, and illegal memory address for data transfer).
2. The device is already in use by some other program.
3. The device is not responding due to some reason (e.g., device is not powered on; device is malfunctioning; etc.).

Some of the situations may be totally unplanned for and could occur simply because of “human in the loop.” Examples include:

1. The device is unplugged from the computer while data transfer is going on.
2. The power chord for the device is unplugged while the data transfer is going on.
3. The device starts malfunctioning during data transfer (e.g., someone accidentally knocked the camera off its moorings; etc.)

10.7 Peripheral Devices

Historically, I/O devices have been grouped into *character-oriented*² and *block-oriented* devices. Dot matrix printers, cathode ray terminals (CRT), teletypewriters are examples of the former. The input/output from these devices happen a character at a time. Due to the relative slowness of these devices, programmed I/O (PIO) is a viable method of data transfer to/from such devices from/to the computer system. Hard drive (disk), CD-RW, and tape player are examples of block-oriented devices. As the name suggests, such devices move a block of data to/from the device from/to the computer system. For example, with a laser printer, once it starts printing a page, it continually needs the data for that page, since there is no way to pause the laser printer in the middle of printing a page. The same is true of a magnetic tape drive. It reads or writes a block of data at a time from the magnetic tape. Data transfers from such devices are subject to the data overrun problem that we mentioned earlier in Section 10.1.1. Therefore, DMA is the only viable way of effecting data transfers between such devices and the computer system.

² We introduced the terms character-oriented and block-oriented earlier in Section 10.5 without formally defining these terms.

Device	Input/output	Human in the loop	Data rate (circa 2008)	PIO	DMA
Keyboard	Input	Yes	5-10 bytes/sec	X	
Mouse	Input	Yes	80-200 bytes/sec	X	
Graphics display	Output	No	200-350 MB/sec		X
Disk (hard drive)	Input/Output	No	100-200 MB/sec		X
Network (LAN)	Input/Output	No	1 Gbit/sec		X
Modem³	Input/Output	No	1-8 Mbit/sec		X
Inkjet printer⁴	Output	No	20-40 KB/sec	X ⁵	X
Laser printer⁶	Output	No	200-400 KB/sec		X
Voice (microphone/speaker)⁷	Input/Output	Yes	10 bytes/sec	X	
Audio (music)	Output	No	4-500 KB/sec		X
Flash memory	Input/Output	No	10-50 MB/sec		X
CD-RW	Input/Output	No	10-20 MB/sec		X
DVD-R	Input	No	10-20 MB/sec		X

Table 10.2: A Snapshot of Data Rates of Computer Peripherals⁸

Table 10.2 summarizes the devices typically found in a modern computer system, their data rates (circa 2008), and the efficacy of programmed I/O versus DMA. The second column signifies whether a “human in the loop” influences the data rate from the device. Devices such as a keyboard and a mouse work at human speeds. For example, a typical typewriting speed of 300-600 characters/minute translates to a keyboard input rate of 5-10 bytes/second. Similarly, moving a mouse generating 10-20 events/second translates to an input rate of 80 to 200 bytes/second. A processor could easily handle such data rates without data loss using programmed I/O (with either polling or interrupt). A graphics display found in most modern computer systems has a frame buffer in the device controller that is updated by the device driver using DMA transfer. For a graphics display, a screen resolution of 1600 x 1200, a screen refresh rate of 60 Hz, and 24 bits per pixel, yields a data rate of over 300 MB/sec. A music CD holds over 600 MB of data with a playtime of 1 hour. A movie DVD holds over 4GB of data with a playtime of 2 hours. Both these technologies allow faster than real time reading of data of the media. For example, a CD allows reading at 50x real time, while a DVD allows reading at 20x real time resulting in the data rates shown. It should be noted that technology changes continuously. Therefore, this table should be taken as a snapshot of technology circa

³ Slow speed modems of yester years with data rates of 2400 bits/s may have been amenable to PIO with interrupts. However, modern cable modems that support upstream data rates in excess of 1 Mbits/s, and downstream bandwidth in excess of 8 Mbits/s require DMA transfer to avoid data loss.

⁴ This assumes an Inkjet print speed of 20 pages per min (ppm) for text and 2-4 ppm for graphics.

⁵ Inkjet technology allows pausing printing awaiting data from the computer. Since the data rate is slow enough, it is conceivable to use PIO for this device.

⁶ This assumes a Laser print speed of 40 ppm for text and about 4-8 ppm for graphics.

⁷ Typically, speakers are in the range of outputting 120 words/minute.

⁸ Many thanks to Yale Patt, UT-Austin, and his colleague for shedding light on the speed of peripheral devices.

2008, just as a way of understanding how computer peripherals may be interfaced with the CPU.

10.8 Disk Storage

We will study disk as a concrete example of an important peripheral device. Disk drives are the result of a progression of technology that started with magnetically recording analog data onto wire. This led to recording data on to a magnetic coating applied to Mylar tape. Tapes only allowed sequential access to the recorded data. To increase the data transfer rate as well as to allow random access, magnetic recording transitioned to a rotating drum. The next step in the evolution of magnetic recording was the disk.

Modern disk drives typically consist of some number of *platters* of a lightweight non-ferromagnetic metal coated with a ferromagnetic material on both the top and bottom *surfaces* (see Figure 10.10), thus allowing both surfaces for storing data. A central spindle gangs these platters together and rotates them at very high speed (~15,000 RPM in state-of-the-art high-capacity drives). There is an array of magnetic *read/write heads*, one per surface. The heads do not touch the surfaces. There is a microscopic air gap (measured in nanometers and tinier than a smoke or dust particle) between the head and the surface to allow the movement of the head over the surface without physical contact with the surface. An *arm* connects each head to a common *shaft* as shown in the figure. The shaft, the arms, and the attached heads form the *head assembly*. The arms are mechanically fused together to the shaft to form a single aligned structure such that all the heads can be moved in unison (like a swing door) in and out of the disk. Thus, all the heads simultaneously line up on their respective surfaces at the **same** radial position.

Depending on the granularity of the *actuator* that controls the motion of the head assembly, and the *recording density* allowed by the technology, this arrangement leads to configuring each recording surface into *tracks*, where each track is at a certain pre-defined radial distance from the center of the disk. As the name suggest, a track is a circular band of magnetic recording material on the platter. Further, each track consists of *sectors*. A sector is a contiguous recording of bytes of information, fixed in size, and forms the basic *unit of recording* on the disk. In other words, a sector is the smallest unit of information that can be read or written to the disk. Sensors around the periphery of the disk platters demarcate the sectors. The set of corresponding tracks on all the surfaces form a *logical cylinder* (see Figure 10.11). Cylinder is an aggregation of the corresponding tracks on all the surfaces. The reason for identifying cylinder as a logical entity will become evident shortly when we discuss the latencies involved in accessing the disk and performing I/O operations. The entire disk (platters, head assembly, and sensors) is vacuum-sealed since even the tiniest of dust particles on the surface of the platters will cause the disk to fail.

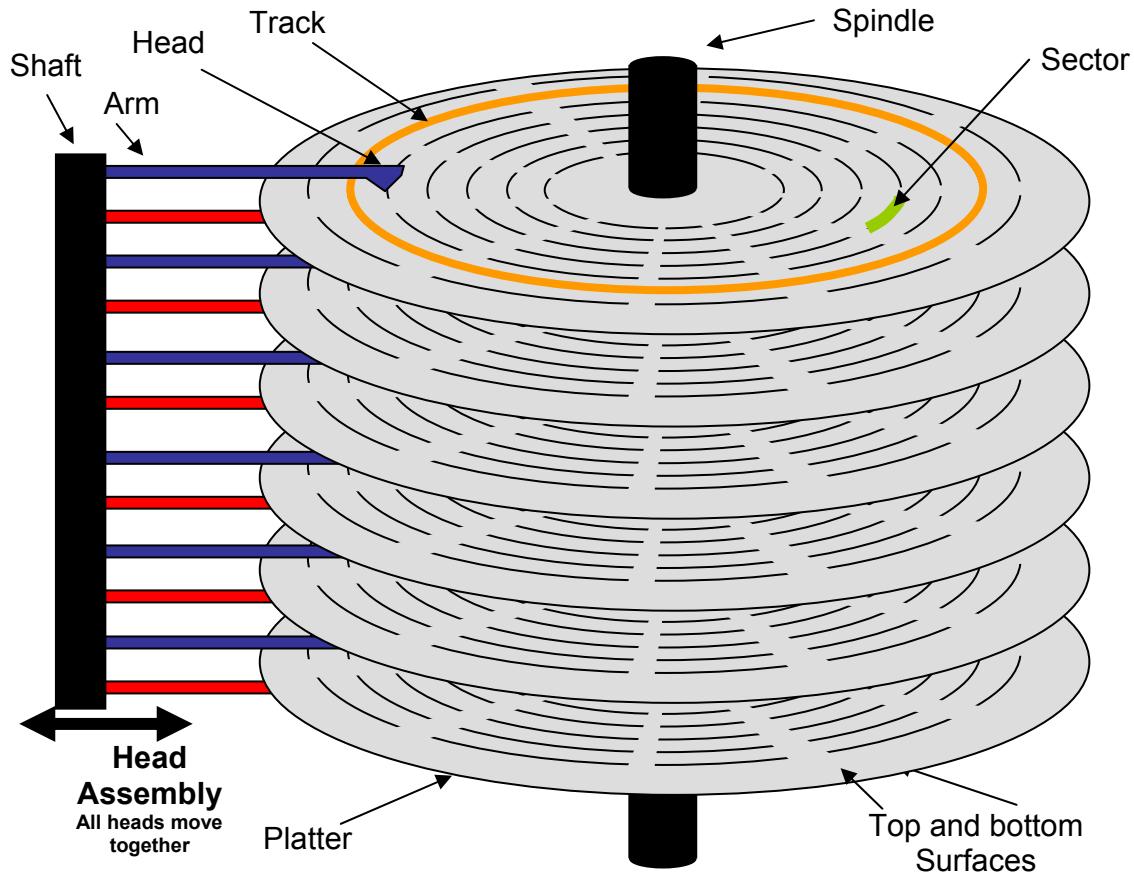


Figure 10.10: Magnetic Disk

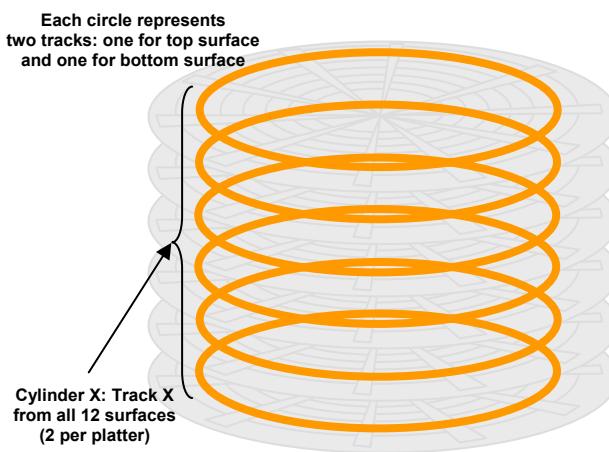


Figure 10.11: A Logical Cylinder for a disk with 6 platters

Tracks are concentric bands on the surface of the disk platters. Naturally, the outer tracks have larger circumference compared to the inner tracks. Therefore, the outer sectors have a larger footprint compared to the inner ones (see Figure 10.12a). As we mentioned, a

sector has a fixed size in terms of recorded information. To reconcile the larger sector footprint and the fixed sector size in the outer tracks, earlier disk technology took the approach of reducing the recording density in the outer tracks. This results in under-utilizing the available space on the disk.

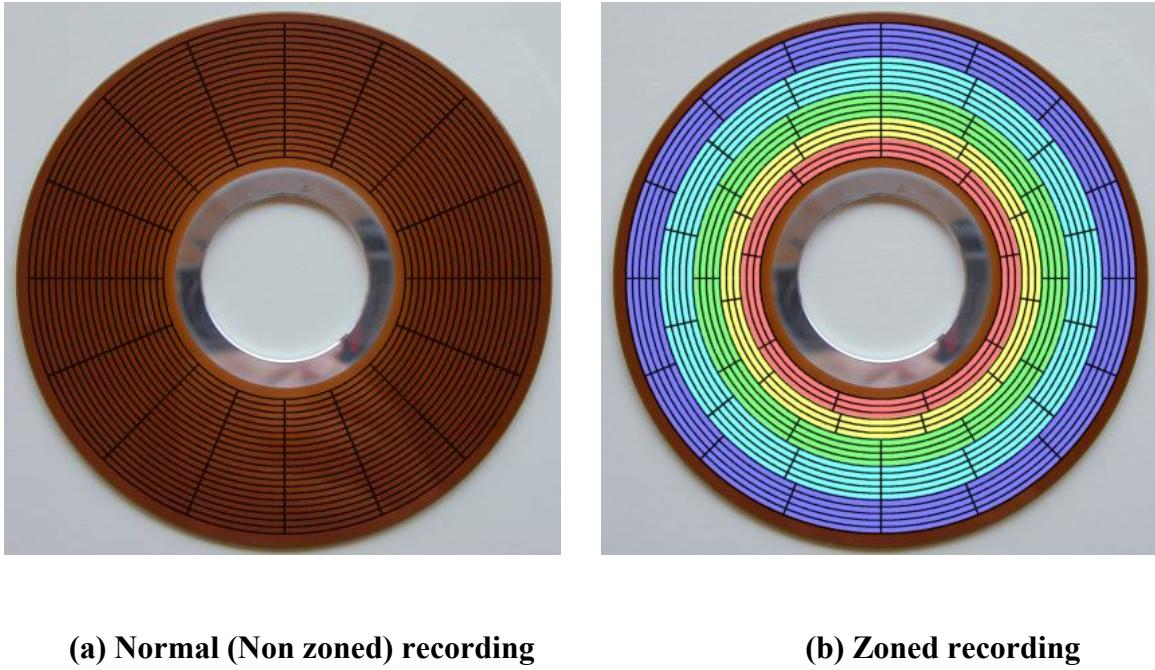


Figure 10.12: Difference between non-zoned and zoned recording⁹

To overcome this problem of under-utilization, modern disk drives use a technology called *zoned bit recording (ZBR)* that keeps the footprint of the sectors roughly constant on the entire surface. However, the outer tracks have more sectors than the inner ones (see Figure 10.12b). The disk surface is divided into zones; tracks in the different zones have different number of sectors thus resulting in a better utilization.

Let,

- p – number of platters,
- n – number of surfaces per platter (1 or 2),
- t – number of tracks per surface,
- s – number of sectors per track,
- b – number of bytes per sector,

The total capacity of the disk assuming non-zoned recording:

$$\text{Capacity} = (p * n * t * s * b) \text{ bytes} \quad (1)$$

⁹ Picture source: <http://www.pcguide.com/ref/hdd/geom/tracksZBR-c.html>

With zoned recording,

z – number of zones,

t_{zi} – number of tracks at zone z_i ,

s_{zi} – number of sectors per track at zone z_i ,

The total capacity of the disk with zoned recording:

$$\text{Capacity} = (p * n * (\sum (t_{zi} * s_{zi}), \text{ for } 1 \leq i \leq z) * b) \text{ bytes} \quad (2)$$

Example 1:

Given the following specifications for a disk drive:

- 256 bytes per sector
 - 12 sectors per track
 - 20 tracks per surface
 - 3 platters
- a) What is the total capacity of such a drive in bytes assuming normal recording?
- b) Assume a zoned bit recording with the following specifications:
- 3 zones
 - Zone 3 (outermost): 8 tracks, 18 sectors per track
 - Zone 2: 7 tracks, 14 sectors per track
 - Zone 1: 5 tracks, 12 sectors per track

What is the total capacity of this drive with the zoned-bit recording?

Answer:

a)

Total capacity

$$\begin{aligned} &= \text{number of platters} * \text{surfaces/platter} * \text{tracks/surface} * \text{sectors/track} * \text{bytes/sector} \\ &= 3 * 2 * 20 * 12 * 256 \text{ bytes} \\ &= \mathbf{360 \text{ Kbytes (where K = 1024)}} \end{aligned}$$

b)

Capacity of Zone 3 =

$$\begin{aligned} &= \text{number of platters} * \text{surfaces/platter} * \text{tracks in zone 3} * \text{sectors/track} * \text{bytes/sector} \\ &= 3 * 2 * 8 * 18 * 256 \\ &= \mathbf{216 \text{ Kbytes}} \end{aligned}$$

Capacity of Zone 2 =

$$\begin{aligned} &= \text{number of platters} * \text{surfaces/platter} * \text{tracks in zone 2} * \text{sectors/track} * \text{bytes/sector} \\ &= 3 * 2 * 7 * 14 * 256 \\ &= \mathbf{147 \text{ Kbytes}} \end{aligned}$$

Capacity of Zone 1 =

$$\begin{aligned} &= \text{number of platters} * \text{surfaces/platter} * \text{tracks in zone 1} * \text{sectors/track} * \text{bytes/sector} \\ &= 3 * 2 * 5 * 12 * 256 \\ &= \mathbf{90 \text{ Kbytes}} \end{aligned}$$

$$\begin{aligned}\text{Total capacity} &= \text{sum of all zonal capacities} = 216 + 147 + 90 \\ &= \mathbf{453 \text{ Kbytes (where } K = 1024\text{)}}\end{aligned}$$

An important side effect of ZBR is the difference in transfer rates between outer and inner tracks. The outer track has more sectors compared to the inner ones; and the angular velocity of the disk is the same independent of the track being read. Therefore, the head reads more sectors per revolution of the disk when it is over an outer track compared to an inner track. In allocating space on the disk, there is a tendency to use the outer tracks first before using the inner tracks.

The address of a particular data block on the disk is a triple: $\{\text{cylinder}\#, \text{surface}\#, \text{sector}\#\}$. Reading or writing information to or from the disk requires several steps. First, the head assembly has to move to the specific cylinder. The time to accomplish this movement is the *seek time*. We can see that seeking to a particular cylinder is the same as seeking to any specific track in that cylinder since a cylinder is just a logical aggregation of the corresponding tracks on all the surfaces (see Figure 10.11). Second, the disk must spin to bring the required sector under the head. This time is the *rotational latency*. Third, data from the selected surface is read and transferred to the controller as the sector moves under the head. This is the *data transfer time*.

Of the three components to the total time for reading or writing to a disk, the seek time is the most expensive, followed by the rotational latency, and lastly the data transfer time. Typical values for seek time and average rotational latency are 8 ms and 4 ms, respectively. These times are so high due to the electro-mechanical nature of the disk subsystem.

Let us understand how to compute the data transfer time. Note that the disk does not stop for reading or writing. Just as in a VCR the tape is continuously moving while the head is reading and displaying the video on the TV, the disk is continually spinning and the head is reading (or writing) the bits off the surface as they pass under the head. The data transfer time is derivable from the rotational latency and the recording density of the media. You are perhaps wondering if reading or writing the media itself does not cost anything. The answer is it does; however, this time is purely electro-magnetic and is negligible in comparison to the electro-mechanical delay in the disk spinning to enable all the bits of a given sector to be read.

We refer to the *data transfer rate* as the amount of data transferred per unit time once the desired sector is under the magnetic reading head. Circa 2008, data transfer rates are in the range of 200-300 Mbytes/sec.

Let,

- r – rotational speed of the disk in Revolutions Per Minute (RPM),
- s – number of sectors per track,
- b – number of bytes per sector,

Time for one revolution = $60/r$ seconds

Amount of data read in one revolution = $s * b$ bytes

The data transfer rate of the disk:

$$(\text{Amount of data in track}) / (\text{time for one revolution}) = (s * b) / (60/r)$$

$$\text{Data transfer rate} = (s * b * r) / 60 \text{ bytes/second} \quad (3)$$

Example 2:

Given the following specifications for a disk drive:

- 512 bytes per sector
- 400 sectors per track
- 6000 tracks per surface
- 3 platters
- Rotational speed 15000 RPM
- Normal recording

What is the transfer rate of the disk?

Answer:

Time for one rotation = $1/15000$ minutes

$$= 4 \text{ ms}$$

The amount of data in track = sectors per track * bytes per sector

$$= 400 * 512$$

$$= 204,800 \text{ bytes}$$

Since the head reads one track in one revolution of the disk, the transfer rate

$$= \text{data in one track/time per revolution}$$

$$= (204,800/4) * 1000 \text{ bytes/sec}$$

$$= \mathbf{51,200,000 \text{ bytes/sec}}$$

The seek time and rotational latency experienced by a specific request depends on the exact location of the data on the disk. Once the head is positioned on the desired sector the time to read/write the data is deterministic, governed by the rotational speed of the disk. It is often convenient to think of *average seek time* and *average rotational latency* in performance estimation of disk drives to satisfy requests. Assuming a uniform distribution of requests over all the tracks, the average seek time is the mean of the observed times to seek to the first track and the last track of the disk. Similarly, assuming a uniform distribution of requests over all the sectors in a given track, the average rotational latency is the mean of the access times to each sector in the track. This is half the rotational latency of the disk¹⁰.

¹⁰ In the best case the desired sector is already under the head when the desired track is reached; in the worst case the head just missed the desired sector and waits for an entire revolution.

Let,

a – average seek time in seconds

r – rotational speed of the disk in Revolutions Per Minute (RPM),

s – number of sectors per track,

$$\text{Rotational latency} = 60/r \text{ seconds} \quad (4)$$

$$\text{Average rotational latency} = (60 / (r * 2)) \text{ seconds} \quad (5)$$

Once the read head is over the desired sector, then the time to read that sector is entirely decided by the RPM of the disk.

Sector read time = rotational latency / number of sectors per track

$$\text{Sector read time} = (60 / (r * s)) \text{ seconds} \quad (6)$$

To read a random sector on the disk, the head has to seek to that particular sector, and then the head has to wait for the desired sector to appear under it, and then read the sector. Thus, there are three components to the time to read a random sector:

- Time to get to the desired track
 - = Average seek time
 - = a seconds
- Time to get the head over the desired sector
 - = Average rotational latency
 - = $(60 / (r * 2))$ seconds
- Time to read a sector
 - = Sector read time
 - = $(60 / (r * s))$ seconds

$$\text{Time to read a random sector on the disk} \quad (7)$$

$$\begin{aligned} &= \text{Time to get to the desired track} + \\ &\quad \text{Time to get the head over the desired sector} + \\ &\quad \text{Time to read a sector} \\ &= a + (60 / (r * 2)) + (60 / (r * s)) \text{ seconds} \end{aligned}$$

Example 3:

Given the following specifications for a disk drive:

- 256 bytes per sector
- 12 sectors per track
- 20 tracks per surface
- 3 platters
- Average seek time of 20 ms
- Rotational speed 3600 RPM
- Normal recording

- a) What would be the time to read 6 contiguous sectors from the same track?
- b) What would be the time to read 6 sectors at random?

Answer:

a)

Average seek time = 20 ms

Rotational latency of the disk

$$= 1/3600 \text{ minutes}$$

$$= 16.66 \text{ ms}$$

Average rotational latency

$$= \text{rotational latency}/2$$

$$= 16.66/2$$

Time to read 1 sector

$$= \text{rotational latency} / \text{number of sectors per track}$$

$$= 16.66/12 \text{ ms}$$

To read 6 contiguous sectors on the same track the time taken

$$= 6 * (16.66/12)$$

$$= 16.66/2$$

Time to read 6 contiguous sectors from the disk

$$= \text{average seek time} + \text{average rotational latency} + \text{time to read 6 sectors}$$

$$= 20 \text{ ms} + 16.66/2 + 16.66/2$$

$$= \mathbf{36.66 \text{ ms}}$$

b)

For the second case, we will have to seek and read each sector separately.

Therefore, each sector read will take

$$= \text{average seek time} + \text{average rotational latency} + \text{time to read 1 sector}$$

$$= 20 + 16.66/2 + 16.66/12$$

Thus total time to read 6 random sectors

$$= 6 * (20 + 16.66/2 + 16.66/12)$$

$$= \mathbf{178.31 \text{ ms}}$$

10.8.1 Saga of Disk Technology

The discussion until now has been kept simple just for the sake of understanding the basic terminologies in disk subsystems. The disk technology has seen an exponential growth in recording densities for the past two decades. For example, circa 1980, 20 megabytes was considered a significant disk storage capacity. Such disks had about 10 platters and were bulky. The drive itself looked like a clothes washer (see Figure 10.13) and the media was usually removable from the drive itself.



(a) Removable Media

(b) Disk drive

Figure 10.13: Magnetic media and Disk drive¹¹

Circa 2008, a desktop PC comes with several 100 Gigabytes of storage capacity. Such drives have the media integrated into them. Circa 2008, small disks for the desktop PC market (capacity roughly 100 GB to 1 TB, 3.5" diameter) have 2 to 4 platters, rotation speed in the range of 7200 RPM, 5000 to 10000 tracks per surface, several 100 sectors per track, and 256 to 512 bytes per sector (see Figure 10.14).



Figure 10.14: PC hard drive (circa 2008)¹²

¹¹ Picture source: unknown

¹² Picture of a Western Digital Hard Drive: variable RPM, 1 TB capacity, source: <http://www.wdc.com>

While we have presented fairly simple ways to compute the transfer rates of disks and a model of accessing the disk based on the cylinder-track-sector concept, modern technology has thrown all these models out of whack. The reason for this is fairly simple. Internally the disk remains the same as pictured in Figure 10.14. Sure, the recording densities have increased, the RPM has increased, and consequently the size of the platter as well as the number of platters in a drive has dramatically decreased. These changes themselves do not change the basic model of disk access and accounting for transfer times. The real changes are in three areas: advances in the drive electronics, advances in the recording technology, and advance in interfaces.

The first advance is in the internal drive electronics. The simple model assumes that the disk rotates at a constant speed. The head moves to the desired track and waits for the desired sector to appear under it. This is wasteful of power. Therefore, modern drives vary the RPM depending on the sector that needs to be read, ensuring that the sector appears under the head just as the head assembly reaches the desired track.

Another advance is in the recording strategy. Figure 10.15 shows a cross section of the magnetic surface to illustrate this advance in recording technology. Traditionally, the medium records the data bits by magnetizing the medium horizontally parallel to the magnetic surface sometimes referred to as *longitudinal recording* (Figure 10.15-(a)). A recent innovation in the recording technology is the *perpendicular magnetic recording (PMR)*, which as the name suggests records the data bits by magnetizing the medium perpendicular to the magnetic surface (Figure 10.15-(b)). Delving into the electromagnetic properties of these two recording techniques is beyond the scope of this book. The intent is to get across the point that this new technology greatly enhances the recording density achievable per unit area of the magnetic surface on the disk. Figure 10.15 illustrates this point. It can be seen that PMR results in doubling the recording density and hence achieves larger storage capacity than longitudinal recording for a given disk specification.

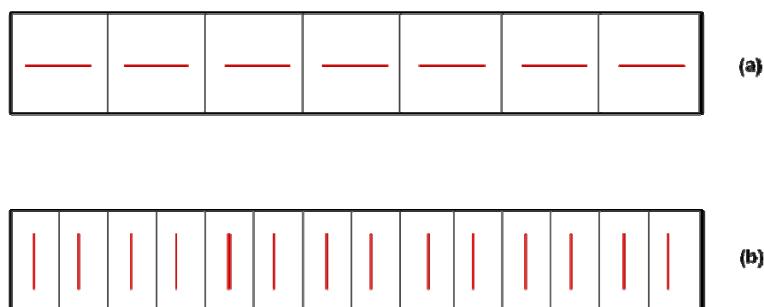


Figure 10.15: Disk recording: (a) Longitudinal recording; (b) PMR

The third change is in the computational capacity that has gotten into these modern drives. Today, a hard drive provides much more intelligent interfaces for connecting it to the rest of the system. Gone are the days when a disk controller sits outside the drive. Today the drive is integrated with the controller. You may have heard of terminologies such as IDE (which stands for *Integrated Drive Electronics*), ATA (which stands for *Advanced Technology Attachment*), SATA (which stands for *Serial Advanced*

*Technology Attachment) and SCSI (which stands for *Small Computer Systems Interface*).* These are some of the names for modern intelligent interfaces.

Conceptually, these advanced interfaces reduce the amount of work the CPU has to do in ordering its requests for data from the disk. Internal to the drive is a microprocessor that decides how logically contiguous blocks (which would be physically contiguous as well in the old world order) may be physically laid out to provide best access time to the data on the disk. In addition to the microprocessors, the drives include data buffers into which disk sectors may be pre-read in readiness for serving requests from the CPU. The microprocessor maintains an internal queue of requests from the CPU and may decide to reorder the requests to maximize performance.

Much of the latency in a disk drive arises from the mechanical motions involved in reading and writing. Discussion of the electromechanical aspects of disk technology is beyond the scope of this book. Our focus is on how the system software uses the disk drive for storing information. The storage allocation schemes should try to reduce the seek time and the rotational latency for accessing information. Since we know that seek time is the most expensive component of disk transfer latency, we can now elaborate on the concept of a logical cylinder. If we need to store a large file that may span several tracks, should we allocate the file to **adjacent tracks on the same surface or corresponding (same) tracks of a given cylinder?** The answer to this question has become quite complex with modern disk technology. As a first order, we will observe that the former will result in multiple seeks to access different parts of the given file while the latter will result in a single seek to access any part of a given file. We can easily see that the latter allocation will be preferred. This is the reason for recognizing a logical cylinder in the context of the disk subsystem. Having said that, we should observe that since the number of platters in a disk is reducing and is typically 1 or 2, the importance of the cylinder concept is diminishing.

File systems is the topic of the next chapter, wherein we elaborate on storage allocation schemes.

From the system throughput point of view, the operating system should schedule operations on the disk in such a manner as to reduce the overall mechanical motion of the disk. In modern drives, this kind of reordering of requests happens within the drive itself. Disk scheduling is the topic of discussion in the next section.

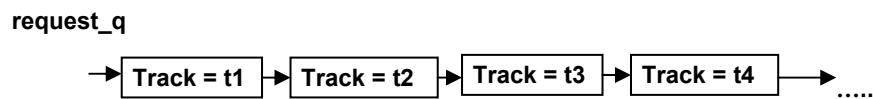


Figure 10.16: Disk request queue in the order of arrival

10.9 Disk Scheduling Algorithms

A device driver for a disk embodies algorithms for efficiently scheduling the disk to satisfy the requests that it receives from the operating system. As we have seen in earlier

chapters (see Chapters 7 and 8), the memory manager component of the operating system may make its own requests for disk I/O to handle demand paging. As we will see shortly in Chapter 11, disk drives host file systems for end users. Thus, the operating system (via system calls) may issue disk I/O requests in response to users' request for opening, closing, and reading/writing files. Thus, at any point of time, the device driver for a disk may be entertaining a number of I/O requests from the operating system (see Figure 10.16). The operating system queues these requests in the order of generation. The device driver schedules these requests commensurate with the disk-scheduling algorithm in use. Each request will name among other things, the specific track where the data resides on the disk. Since seek time is the most expensive component of I/O to or from the disk, the primary objective of disk scheduling is to minimize seek time.

We will assume for this discussion that we are dealing with a single disk that has received a number of requests and must determine the most efficient algorithm to process those requests. We will further assume that there is a single head and that seek time is proportional to the number of tracks crossed. Finally, we will assume a random distribution of data on the disk, and that reads and writes take equal amounts of time.

The typical measures of how well the different algorithms stack up against one another are the *average waiting time* for a request, *the variance in wait time* and the overall *throughput*. Average wait time and throughput are self-evident terms from our discussion on CPU scheduling (Chapter 6). These terms are system centric measures of performance. From an individual request point of view, variance in waiting time is more meaningful. This measure tells us how much an individual request's waiting time can deviate from the average. Similar to CPU scheduling, the *response time* or *turnaround time* is a useful metric from the point of view of an individual request.

In Table 10.3, t_i , w_i , and e_i , are the turnaround time, wait time, and actual I/O service time, for an I/O request i , respectively. Most of these metrics and their mathematical notation are similar to the ones given in Chapter 6 for CPU scheduling.

Name	Notation	Units	Description
Throughput	n/T	Jobs/sec	System-centric metric quantifying the number of I/O requests n executed in time interval T
Avg. Turnaround time (t_{avg})	$(t_1+t_2+\dots+t_n)/n$	Seconds	System-centric metric quantifying the average time it takes for a job to complete
Avg. Waiting time (w_{avg})	$((t_1-e_1) + (t_2-e_2) + \dots + (t_n-e_n))/n$ or $(w_1+w_2+\dots+w_n)/n$	Seconds	System-centric metric quantifying the average waiting time that an I/O request experiences
Response time/turnaround time	t_i	Seconds	User-centric metric quantifying the turnaround time for a specific I/O request i
Variance in Response time	$E[(t_i - t_{avg})^2]$	Seconds ²	User-centric metric that quantifies the statistical variance of the actual response time (t_i) experienced by an I/O request i from the expected value (t_{avg})
Variance in Wait time	$E[(w_i - w_{avg})^2]$	Seconds ²	User-centric metric that quantifies the statistical variance of the actual wait time (w_i) experienced by an I/O request i from the expected value (w_{avg})
Starvation	-	-	User-centric qualitative metric that signifies denial of service to a particular I/O request or a set of I/O request due to some intrinsic property of the I/O scheduler

Table 10.3: Summary of Performance Metrics

We review five different algorithms for disk scheduling. To make the discussion simple and concrete, we will assume that the disk has 200 tracks numbered 0 to 199 (with 0 being the outermost and 199 being the innermost). The head in its fully retracted position

is on track 0. The head assembly extends to its maximum span from its resting position when it is on track 199.

You will see a similarity between these algorithms and some of the CPU scheduling algorithms we saw in Chapter 6.

10.9.1 First-Come First Served

As the name suggests, this services the requests by the order of arrival. In this sense, it is similar to the FCFS algorithm for CPU scheduling. The algorithm has the nice property that a request incurs the least variance in waiting time regardless of the track from which it requests I/O. However, that is the only good news. From the system's perspective, this algorithm will result in poor throughput for most common workloads. Figure 10.17 illustrates how the disk head may have to swing back and forth across the disk surface to satisfy the requests in an FCFS schedule, especially when the FCFS requests are to tracks that are far apart.

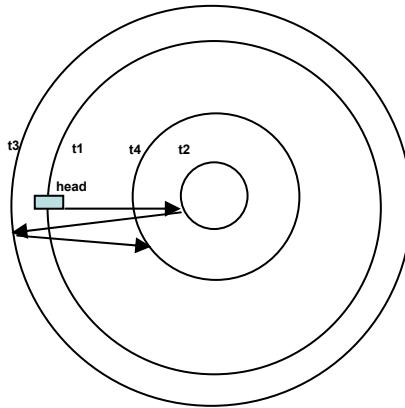


Figure 10.17: Movement of Disk Head with FCFS

10.9.2 Shortest Seek Time First

This scheduling policy has similarity to the SJF processor scheduling. The basic idea is to service the tracks that lead to minimizing the head movement (see Figure 10.18). As with SJF for processor scheduling, SSTF minimizes the average wait time for a given set of requests and results in good throughput. However, similar to SJF it has the potential of starving requests that happen to be far away from the current cluster of requests. Compared to FCFS, this schedule has high variance.

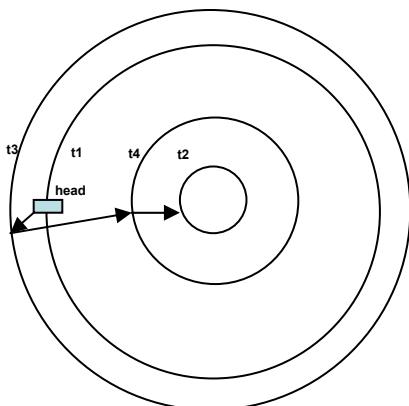


Figure 10.18: Movement of Disk Head with SSTF

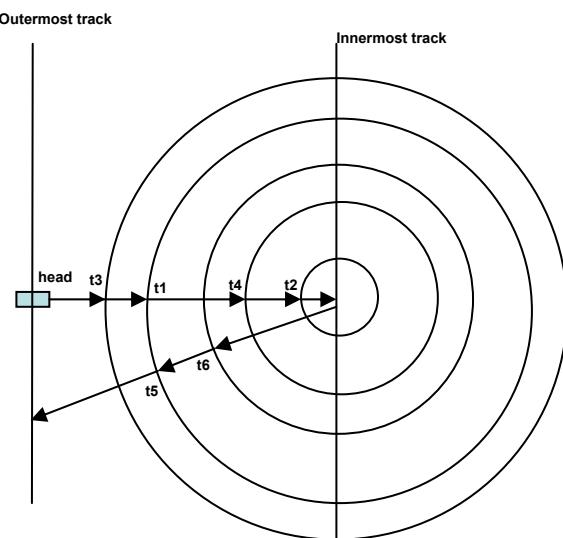


Figure 10.19: Movement of Disk Head with SCAN

10.9.3 Scan (elevator algorithm)

This algorithm is in tune with the electro-mechanical properties of the head assembly. The basic idea is as follows: the head moves from its position of rest (track 0) towards the innermost track (track 199). As the head moves, the algorithm services the requests that are *en route* from outermost to innermost track, regardless of the arrival order. Once the head reaches the innermost track, it reverses direction and moves towards the outermost track, once again servicing requests en route. As long as the request queue is non-empty, the algorithm continuously repeats this process. Figure 10.19 captures the spirit of this algorithm.

The requests t1, t2, t3, and t4 existed during the forward movement of the head (as in Figure 10.16). Requests t5 and t6 (in that order) appeared after the head reached the innermost track. The algorithm services these requests on the reverse traversal of the

head. This algorithm should remind you of what happens when you wait for an elevator, which uses a similar algorithm for servicing requests. Hence, the SCAN algorithm is often referred to as the elevator algorithm. SCAN has lower variance in wait time compared to SSTF, and overall has an average wait time that is similar to SSTF. Similar to SSTF, SCAN does not preserve the arrival order of the requests. However, SCAN differs from SSTF in one fundamental way. SSTF may starve a given request arbitrarily. On the other hand, there is an upper bound for SCAN's violation of the first-come-first-served fairness property. The upper bound is the traversal time of the head from one end to the other. Hence, SCAN avoids starvation of requests.

10.9.4 C-Scan (Circular Scan)

This is a variant of SCAN. The algorithm views the disk surface as logically circular. Hence, once the head reaches the innermost track, the algorithm retracts the head assembly to the position of rest, and the traversal starts all over again. In other words, the algorithm does not service any requests during the traversal of the head in the reverse direction. This pictorially shown for the same requests serviced in the SCAN algorithm in Figure 10.20.

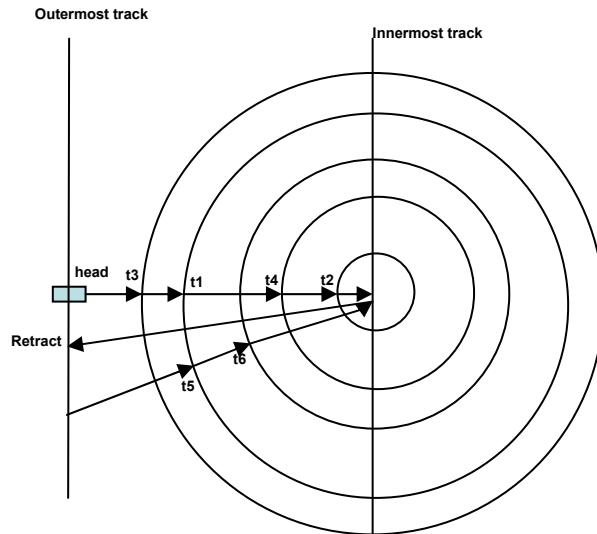


Figure 10.20: Movement of Disk Head with C-SCAN

By ignoring requests in the reverse direction, C-SCAN removes the bias that the SCAN algorithm has for requests clustered around the middle tracks of the disk. This algorithm reduces unfairness in servicing requests (notice how it preserves the order of arrival for t5 and t6), and overall leads to lowering the variance in waiting time compared to SCAN.

10.9.5 Look and C-Look

These two policies are similar to Scan and C-Scan, respectively; with the exception that if there are no more requests in the direction of head traversal, then the head assembly immediately reverses direction. That is, the head assembly does not unnecessarily traverse all the way to the end (in either direction). One can see the similarity to how an elevator works. Due to avoiding unnecessary mechanical movement, these policies tend to be even better performers than the Scan and C-Scan policies. Even though, historically, SCAN is referred to as the elevator algorithm, LOOK is closer to the servicing pattern observed in most modern day elevator systems. Figure 10.21 shows the same sequence of requests as in the SCAN and C-SCAN algorithms for LOOK and C-LOOK algorithms. Note the position of the head at the end of servicing the outstanding requests. The head stops at the last serviced request if there are no further requests. This is the main difference between LOOK and SCAN, and C-LOOK and C-SCAN, respectively.

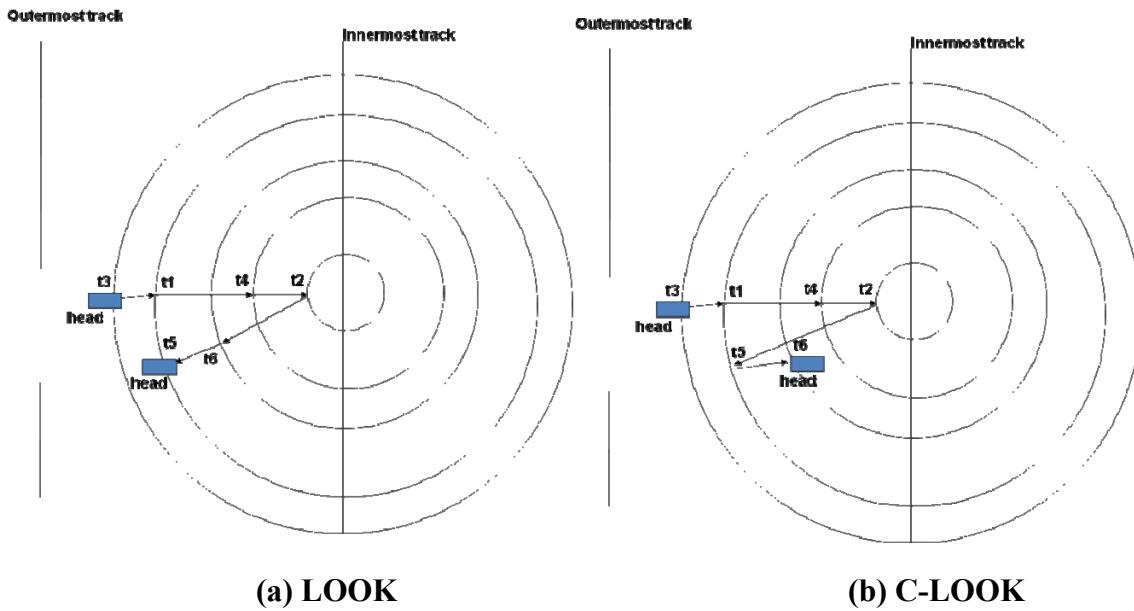


Figure 10.21: Movement of Disk Head with LOOK and C-LOOK

10.9.6 Disk Scheduling Summary

The choice of the scheduling algorithm depends on a number of factors including expected layout, the storage allocation policy, and the electromechanical capabilities of the disk drive. Typically, some variant of LOOK or C-LOOK is used for disk scheduling. We covered the other algorithms more from the point of completeness than as viable choices for implementation in a real system.

As we mentioned in Section 10.8.1, modern disk drives provide very sophisticated interface to the CPU. Thus, the internal layout of the blocks on the drive may not even be visible to the disk device driver, which is part of the operating system. Assuming that the

interface allows multiple outstanding requests from the device driver, the disk scheduling algorithms discussed in this section are in the controller itself.

Example 4 illustrates the difference in schedule of the various algorithms.

Example 4:

Given the following:

Total number of cylinders in the disk	=	200 (numbered 0 to 199)
Current head position	=	cylinder 23
Current requests in order of arrival	=	20, 17, 55, 35, 25, 78, 99

Show the schedule for the various disk scheduling algorithms for the above set of requests.

Answer:

- (a) Show the schedule for C-LOOK for the above requests

25, 35, 55, 78, 99, 17, 20

- (b) Show the schedule for SSTF

25, 20, 17, 35, 55, 78, 99

- (c) Show the schedule for LOOK

25, 35, 55, 78, 99, 20, 17

- (d) Show the schedule for SCAN

25, 35, 55, 78, 99, 199, 20, 17, 0

- (e) Show the schedule for FCFS

20, 17, 55, 35, 25, 78, 99

- (f) Show the schedule for C-SCAN

25, 35, 55, 78, 99, 199, 0, 17, 20

10.9.7 Comparison of the Algorithms

Let us analyze the different algorithms using the request pattern in Example 4. In the order of arrival, we have seven requests: R1 (cylinder 20), R2 (cylinder 17), R3 (cylinder 55), R4 (cylinder 35), R5 (cylinder 25), R6 (cylinder 78), and R7 (cylinder 99).

Let us focus on request R1. We will use the number of tracks traversed as the unit of response time for this comparative analysis. Since the starting position of the head in this example is 23, the response time for R1 for the different algorithms:

- $T_1^{\text{FCFS}} = 3$ (R1 gets serviced first)
- $T_1^{\text{SSTF}} = 7$ (service R5 first and then R1)
- $T_1^{\text{SCAN}} = 355$ (sum up the head traversal for (d) in the above example)
- $T_1^{\text{C-SCAN}} = 395$ (sum up the head traversal for (f) in the above example)
- $T_1^{\text{LOOK}} = 155$ (sum up the head traversal for (c) in the above example)
- $T_1^{\text{C-LOOK}} = 161$ (sum up the head traversal for (a) in the above example)

Table 10.4 shows a comparison of the response times (in units of head traversal) with respect to the request pattern in Example 4 for some chosen disk scheduling algorithms (FCFS, SSTF, and LOOK).

Requests	Response time		
	FCFS	SSTF	LOOK
R1 (cyl 20)	3	7	155
R2 (cyl 17)	6	10	158
R3 (cyl 55)	44	48	32
R4 (cyl 35)	64	28	12
R5 (cyl 25)	74	2	2
R6 (cyl 78)	127	71	55
R7 (cyl 99)	148	92	76
Average	66.4	36	70

Table 10.4: Quantitative Comparison of Scheduling Algorithms for Example 4

The throughput is computed as the ratio of the number of requests completed to the total number of tracks traversed by the algorithm to complete all the requests.

- FCFS = 7/148 = 0.047 requests/track-traversal
- SSTF = 7/92 = 0.076 requests/track-traversal
- LOOK = 7/158 = 0.044 requests/track-traversal

From the above analysis, it would appear that SSTF does the best in terms of average response time and throughput. But this comes at the cost of fairness (compare the response times for R5 with respect to earlier request R1-R4 in the column under SSTF). Further, as we observed earlier SSTF has the potential for starvation. At first glance, it would appear that LOOK has the worst response time of the three compared in the table. However, there are a few points to note. First, the response times are sensitive to the initial head position and the distribution of requests. Second, it is possible to come up with a pathological example that may be particularly well suited (or by the same token ill-suited) for a particular algorithm. Third, in this contrived example we started with a request sequence that did not change during the processing of this sequence. In reality,

new requests may join the queue that would have an impact on the observed throughput as well as the response time (see Exercise 11).

In general, with uniform distribution of requests on the disk the average response time of LOOK will be closer to that of SSTF. More importantly, something that is not apparent from the table is both the time and energy needed to change the direction of motion of the head assembly that is inherent in both FCFS and SSTF. This is perhaps the single most important consideration that makes LOOK a more favorable choice for disk scheduling.

Using Example 4, it would be a useful exercise for the reader to compare all the disk scheduling algorithms with respect to the other performance metrics that are summarized in Table 10.3.

Over the years, disk-scheduling algorithms have been studied extensively. As we observed earlier (Section 10.8.1), disk drive technology has been advancing rapidly. Due to such advances, it becomes imperative to re-evaluate the disk-scheduling algorithms with every new generation of disks¹³. So far, some variant of LOOK has proven to perform the best among all the candidates.

10.10 Solid State Drive

One of the fundamental limitations of the hard disk technology is its electromechanical nature. Over the years, several new storage technologies have posed a threat to the disk but they have not panned out primarily due to the low cost per byte of disk compared to these other technologies.

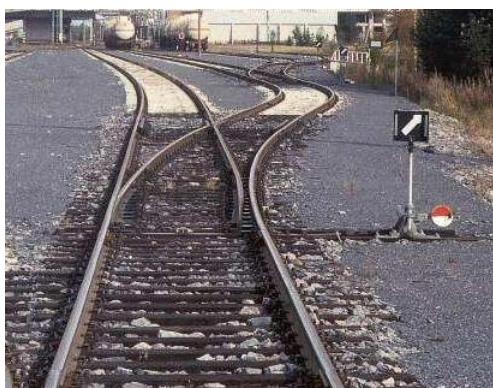


Figure 10.22: A Railroad Switch¹⁴

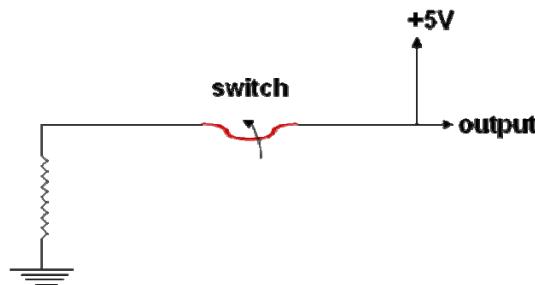


Figure 10.23: An Electrical Switch

A technology that is threatening the relative monopoly of the hard disk is *Solid State Drive (SSD)*. Origins of this technology can be traced back to *Electrically Erasable Programmable Read-Only Memory (EEPROM)*. In Chapter 3, we introduced ROM as a kind of solid state memory whose contents are *non-volatile*, i.e., the contents remain unchanged across power cycling. It will be easy to understand this technology with a simple analogy. You have seen a railroad switch shown in Figure 10.22. Once the

¹³ See for example, <http://www.ece.cmu.edu/~ganger/papers/sigmetrics94.pdf>

¹⁴ Source: http://upload.wikimedia.org/wikipedia/commons/d/da/Railway_turnout_-_Oulu_Finland.jpg

switch is thrown the incoming track (bottom part of the figure) remains connected to the chosen fork. A ROM works in exactly the same manner.

Figure 10.23 is a simple electrical analogy of the railroad switch. If the switch in the middle of the figure is opened then the output is a 1; otherwise the output is a 0. This is the basic building block of a ROM. The switches are realized using basic logic gates (NAND or NOR). Collection of such switches packed inside an integrated circuit is the ROM. Depending on the desired output, a switch can be “programmed” so that the output is a 0 or a 1. This is the reason such a circuit is referred to as *Programmable Read-Only Memory (PROM)*.

A further evolution in this technology allows the bit patterns in the ROM to be electrically erased and re-programmed to contain a different bit pattern. This is the EEPROM technology. While this technology appears very similar to the capability found in a RAM, there is an important difference. Essentially, the granularity of read/write in a RAM can be whatever we choose it to be. However, due to the electrical properties of the EEPROM technology, erasure in an EEPROM needs to happen a block of bits at a time. Further, such writes take several orders of magnitude more time compared to reading and writing a RAM. Thus, EEPROM technology cannot be used in place of the DRAM technology. However, this technology popularized as *Flash memory* found a place in portable memory cards, and for storage inside embedded devices such as cell phones and iPods.

A logical question that may perplex you is why Flash memory has not replaced the disk as permanent storage in desktops and laptops. After all, being entirely solid state, this technology does not have the inherent problems of the disk technology (slower access time for data due to the electromechanical nature of the disk).

Three areas where the hard disk still has an edge are density of storage (leading to lower cost per byte), higher read/write bandwidth, and longer life. The last point needs some elaboration. SSD, due to the nature of the technology, has an inherent limitation that any given area of the storage can be rewritten only a finite number of times. This means that frequent writes to the same block would lead to uneven wear, thus shortening the lifespan of the storage as a whole. Typically, manufacturers of SSD adopt *wear leveling* to avoid this problem. The idea with wear leveling is to redistribute frequently written blocks to different sections of the storage. This necessitates additional read/write cycles over and beyond the normal workload of the storage system.

Technological advances in SSD are continually challenging these gaps. For example, circa 2008, SSD devices with a capacity of 100 GB and transfer rates of 100 MB/second are hitting the market place. Several box makers are introducing laptops with SSD as the mass storage for low-end storage needs. Circa 2008, the cost of SSD-based mass storage is significantly higher than a comparable disk-based mass storage.

Only time will tell whether the hard drive technology will continue its unique long run as the ultimate answer for massive data storage.

10.11 Evolution of I/O Buses and Device Drivers

With the advent of the PC, connecting peripherals to the computer has taken a life of its own. While the data rates in the above table suggest how the devices could be interfaced to the CPU, it is seldom the case that devices are directly interfaced to the CPU. This is because peripherals are made by “third party vendors¹⁵,” a term that is used in the computer industry to distinguish “box makers” such as IBM, Dell, and Apple from the vendors (e.g., Seagate for disks, Axis for cameras, etc.) who make peripheral devices and device drivers that go with such devices. Consequently, such third party vendors are not privy to the internal details of the boxes and therefore have to assume that their wares could be interfaced to any boxes independent of the manufacturer. You would have heard the term “plug and play”. This refers to a feature that allows any peripheral device to be connected to a computer system without doing anything to the internals of the box. This feature has been the primary impetus for the development of standards such as PCI, which we introduced in Section 10.4.

In recent times, it is impossible for a computer enthusiast not to have heard of terminologies such as *USB* and *Firewire*. Let us get familiar with these terminologies. As we mentioned in Section 10.4, PCI bus uses 32-bit parallel bus that it multiplexes for address, data, and command. USB which stands for *Universal Serial Bus*, and *Firewire* are two competing standards for *serial* interface of peripherals to the computer. You may wonder why you would want a serial interface to the computer, since a parallel interface would be faster. In fact, if one goes back in time, only slow speed character-oriented devices (such as Cathode Ray Terminal or CRT, usually referred to as “dumb terminals”) used a serial connection to the computer system.

Well, ultimately, you know that the speed of the signals on a single wire is limited by the *speed of light*. The actual data rates as you can see from Table 10.2 are nowhere near that. Latency of electronic circuitry has been the limiting factor in pumping the bits faster on a single wire. Parallelism helps in overcoming this limitation and boosts the overall throughput by pumping the bits on parallel wires. However, as technology improves the latency of the circuitry has been reducing as well allowing signaling to occur at higher frequencies. Under such circumstances, serial interfaces offer several advantages over parallel. First, it is smaller and cheaper due to the reduced number of wires and connectors. Second, parallel interfaces suffer from cross talk at higher speeds without careful shielding of the parallel wires. On the other hand, with careful wave-shaping and filtering techniques, it is easier to operate serial signaling at higher frequencies. This has led to the situation now where serial interfaces are actually faster than parallel ones.

Thus, it is becoming the norm to connect high-speed devices serially to the computer system. This was the reason for the development of standards such as USB and Firewire for serially interfacing the peripherals to the computer system. Consequently, one may notice that most modern laptops do not support any parallel ports. Even a parallel printer

¹⁵ We already introduced this term without properly defining it in Section 10.4.

port has disappeared from laptops post 2007. Serial interface standards have enhanced the “plug and play” nature of modern peripheral devices.

You may wonder why there are two competing standards. This is again indicative of the “box makers” wanting to capture market share. Microsoft and Intel promoted USB, while Firewire came out of Apple. Today, both these serial interfaces have become industry standards for connecting both slow-speed and high-speed peripherals. USB 1.0 could handle transfer rates up to 1.5 MB/sec and was typically used for aggregating the I/O of slow-speed devices such as keyboard and mouse. Firewire could support data transfers at speeds up to a 100 MB/sec and was typically used for multimedia consumer electronics devices such as digital camcorders. USB 2.0 supports data rates up to 60 MB/sec and thus the distinction between Firewire and USB is getting a bit blurred.

To complete the discussion of I/O buses, we should mention two more enhancements to the I/O architecture that are specific to the PC industry. *AGP*, which stands for *Advanced Graphics Port*, is a specialized dedicated channel for connecting 3-D graphics controller card to the motherboard. Sharing the bandwidth available on the PCI bus with other devices was found inadequate, especially for supporting interactive games that led to the evolution of the AGP channel for 3-D graphics. In recent times, AGP has been largely replaced by *PCI Express (PCI-e)*, which is a new standard that also provides a dedicated connection between the motherboard and graphics controller. Detailed discussions of the electrical properties of these standards and the finer differences among them are beyond the scope of this book.

10.11.1 Dynamic Loading of Device Drivers

Just as the devices are plug and play, so are the device drivers for controlling them. Consider for a moment the device driver for a digital camera. The driver for this device need not be part of your system software (see Figure 10.9) all the time. In modern operating systems such as Linux and Microsoft Vista, the device drivers get “dynamically” linked into the system software when the corresponding device comes on line. The operating system recognizes (via device interrupt), when a new device comes on line (e.g., when you plug in your camera or flash memory into a USB port). The operating system looks through its list of device drivers and identifies the one that corresponds to the hardware that came on line (most likely the device vendor in cooperation with the operating system vendor developed the device driver). It then dynamically links and loads the device driver into memory for controlling the device. Of course, if the device plugged in does not have a matching driver, then the operating system cannot do much except to ask the user to provide a driver for the new hardware device plugged into the system.

10.11.2 Putting it all Together

With the advent of such high-level interfaces to the computer system, the devices be they slow-speed or high-speed are getting farther and farther away from the CPU itself. Thus programmed I/O is almost a thing of the past. In Section 10.5, we mentioned the IBM innovation in terms of I/O processors in the context of mainframes in the 60’s and 70’s. Now such concepts have made their way into your PC.

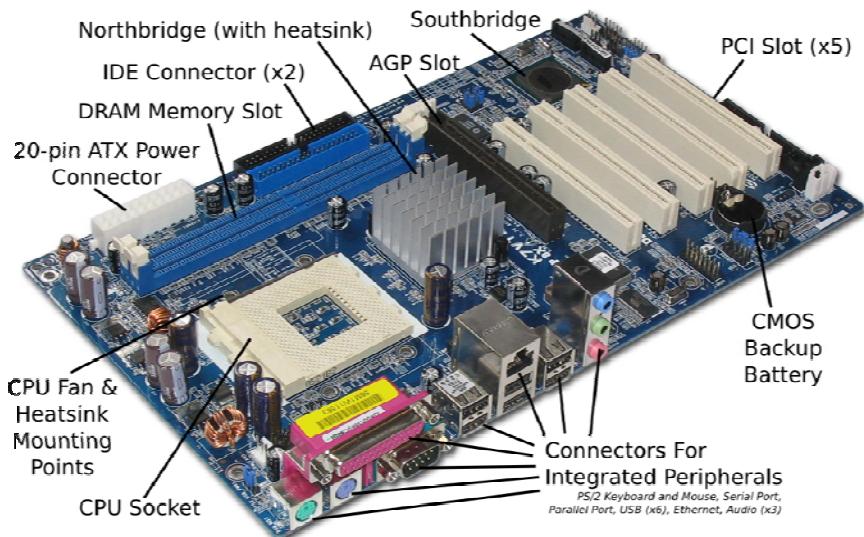


Figure 10.24: Picture of a Motherboard (ASRock K7VT4A Pro)¹⁶

Motherboard is a term coined in the PC era to signify the circuitry that is central to the computer system. It is single printed circuit board consisting of the processor, memory system (including the memory controller), and the I/O controller for connecting the peripheral devices to the CPU. The name comes from the fact that the printed circuit board contains slots into which expansion boards (usually referred to as *daughter cards*) may be plugged in to expand the capabilities of the computer system. For example, expansion of physical memory is accomplished in this manner. Figure 10.24 shows a picture of a modern motherboard. The picture is self-explanatory in terms of the components and their capabilities. The picture shows the slots into which daughter cards for peripheral controllers and DIMMS (see Chapter 9 for discussion of DIMMS) may be plugged in.

Figure 10.25 shows the block diagram of the important circuit elements inside a modern motherboard. It is worth understanding some of these elements. Every computer system needs some low-level code that executes automatically upon power up. As we already know, the processor simply executes instructions. The trick is to bring the computer system to the state where the operating system is in control of all the resources. You have heard the term *booting* up the operating system. The term is short for *bootstrapping*, an allusion to picking oneself up using the bootstraps of your shoes. Upon power up, the processor automatically starts executing a bootstrapping code that is in a well-known fixed location in read-only memory (ROM). This code does all the initialization of the system including the peripheral devices before transferring the control to the upper layers of the system software shown in Figure 10.9. In the world of PCs, this bootstrapping code is called *BIOS*, which stands for *Basic Input/Output System*.

¹⁶ Picture Courtesy:

http://en.wikipedia.org/wiki/Image:ASRock_K7VT4A_Pro_Mainboard_Labeled_English.png

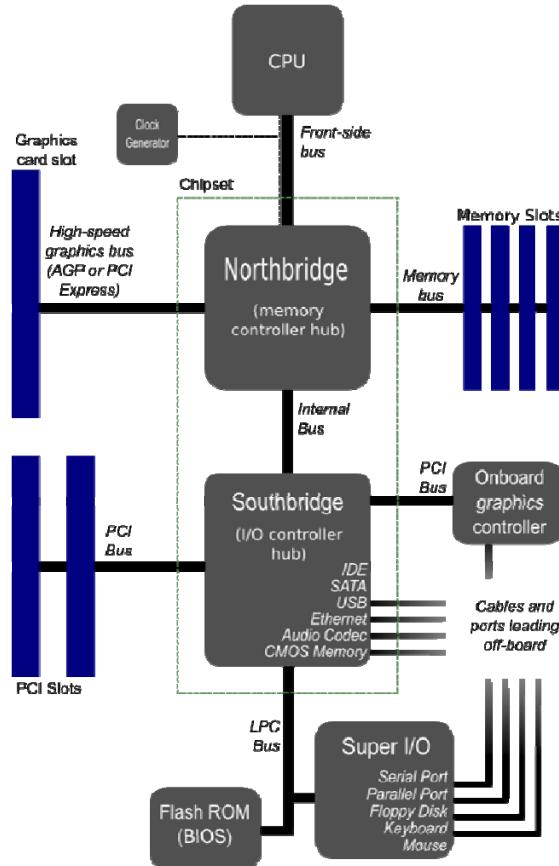


Figure 10.25: Block Diagram of a Motherboard¹⁷

The following additional points are worth noting with respect to Figure 10.25:

- The box labeled *Northbridge* is a chip that serves as the *hub* for orchestrating the communication between the CPU, and memory system as well as the I/O controllers.
- Similarly, the box labeled *Southbridge* is a chip that serves as the I/O controller hub. It connects to the standard I/O buses that we discussed in this section including PCI and USB, and arbitrates among the devices among the buses and their needs for direct access to the memory via the Northbridge as well as for interrupt service by the CPU. It embodies many of the functions that we discussed in the context of an I/O processor in Section 10.5.
- PCI Express is another bus standard that supports the high transfer rate and response time needs of devices such as a high resolution graphics display.
- LPC, which stands for *Low Pin Count* bus is another standard for connecting low bandwidth devices (such as keyboard and mouse) to the CPU.
- The box labeled *Super I/O* is a chip that serves the I/O controller for a number of slow-speed devices including keyboard, mouse, and printer.

¹⁷ Picture courtesy: http://en.wikipedia.org/wiki/Image:Motherboard_diagram.png

As can be seen from this discussion, the hardware inside the computer system is quite fascinating. While we have used the PC as a concrete example in this section, the functionalities embodied in the elements shown in Figure 10.25 generalizes to any computer system. There was a time when the inside of a computer system would be drastically different depending on the class of machine ranging from a personal computer such as an IBM PC to a Vector Supercomputer such as a Cray-1. With advances in single chip microprocessor technology that we discussed in Chapter 5, and the level of integration made possible by Moore's law (see Chapter 3), there is a commonality in the building blocks used to assemble computer systems ranging from PCs, to desktops, to servers, to supercomputers.

10.12 Summary

In this chapter, we covered the following topics:

1. Mechanisms for communication between the processor and I/O devices including programmed I/O and DMA.
2. Device controllers and device drivers.
3. Buses in general and evolution of I/O buses in particular, found in modern computer systems.
4. Disk storage and disk-scheduling algorithms.

In the next chapter, we will study file systems, which is the software subsystem built on top of stable storage in general, hard disk in particular.

10.13 Review Questions

1. Compare and contrast program controlled I/O with Direct Memory Access (DMA).
2. Given a disk drive with the following characteristics:
 - Number of surfaces = 200
 - Number of tracks per surface = 100
 - Number of sectors per track = 50
 - Bytes per sector = 256
 - Speed = 2400 RPM

What is the total disk capacity?

What is the average rotational latency?

3. A disk has 20 surfaces (i.e. 10 double sided platters). Each surface has 1000 tracks. Each track has 128 sectors. Each sector can hold 64 bytes. The disk space allocation policy allocates an integral number of contiguous cylinders to each file,

How many bytes are contained in one cylinder?

How many cylinders are needed to accommodate a 5-Megabyte file on the disk?

How much space is wasted in allocating this 5-Megabyte file?

4. A disk has the following configuration:

The disk has 310 MB

Track size: 4096 bytes

Sector Size: 64 bytes

A programmer has 96 objects each being 50 bytes in size. He decides to save each object as an individual file. How many bytes (in total) are actually written to disk?

5. Describe in detail the sequence of operations involved in a DMA data transfer.
6. What are the mechanical things that must happen before a disk drive can read data?
7. A disk drive has 3 double-sided platters. The drive has 300 cylinders. How many tracks are there per surface?
8. Given the following specifications for a disk drive:
 - 512 bytes per sector
 - 30 tracks per surface
 - 2 platters
 - zoned bit recording with the following specifications:
 - * 3 Zones
 - Zone 3 (outermost): 12 tracks, 200 sectors per track
 - Zone 2: 12 tracks, 150 sectors per track
 - Zone 1: 6 tracks, 50 sectors per track

What is the total capacity of this drive with the zoned-bit recording?

9. Given the following specifications for a disk drive:
 - 256 bytes per sector
 - 200 sectors per track
 - 1000 tracks per surface
 - 2 platters
 - Rotational speed 7500 RPM
 - Normal recording

What is the transfer rate of the disk?

10. Given the following specifications for a disk drive:
 - 256 bytes per sector
 - 100 sectors per track
 - 1000 tracks per surface
 - 3 platters
 - Average seek time of 8 ms
 - Rotational speed 15000 RPM

- Normal recording

- a) What would be the time to read 10 contiguous sectors from the same track?
- b) What would be the time to read 10 sectors at random?

11. What are the objectives of a disk scheduling algorithm?
12. Using the number of tracks traversed as a measure of the time; compare all the disk scheduling algorithms for the specifics of the disk and the request pattern given in Example 4, with respect to the different performance metrics summarized in Table 10.3.
13. Assume the same details about the disk as in Example 4. The request queue does not remain the same but it changes as new requests are added to the queue. At any point of time the algorithm uses the current requests to make its decision as to the next request to be serviced. Consider the following request sequence:
 - Initially (at time 0) the queue contains requests for cylinders: 99, 3, 25
 - At the time the next decision has to be taken by the algorithm, one new request has joined the queue: 46
 - Next decision point one more request has joined the queue: 75
 - Next decision point one more request has joined the queue: 55
 - Next decision point one more request has joined the queue: 85
 - Next decision point one more request has joined the queue: 73
 - Next decision point one more request has joined the queue: 50Assume that the disk head at time 0 is just completing a request at track 55.
 - a) Show the schedule for FCFS, SSTF, SCAN, C-SCAN, LOOK, and C-LOOK.
 - b) Compute the response time (in units of head traversal) for each of the above requests.
 - c) What is the average response time and throughput observed for each of the algorithms?

Chapter 11 File System (Revision number 10)

In this chapter, we will discuss issues related to mass storage systems. In particular, we will discuss the choices in designing a file system and its implementation on the disk (hard drive as it is popularly referred to in the personal computer parlance). Let us understand the role of the file system in the overall mission of this textbook in “unraveling the box”. Appreciating the capabilities of the computer system is inextricably tied to getting a grip on the way information is stored and manipulated inside the box. Therefore, getting our teeth into how the file system works is an important leg of the journey into unravelling the box.

We have all seen physical filing cabinets and manila file folders with papers in them. Typically, tags on the folders identify the content for easy retrieval (see Figure 11.1). Usually, we may keep a directory folder that tells us the organization of the files in the filing cabinet.

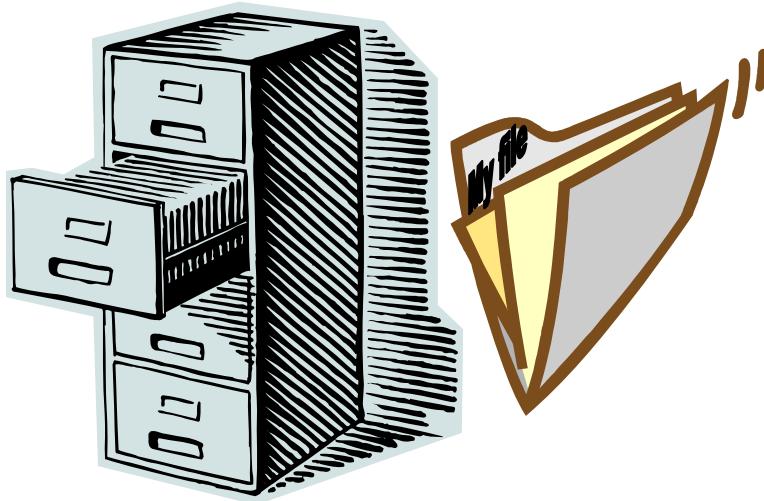


Figure 11.1: File cabinet and file folder

A computer file system is similar to a physical filing cabinet. Each file (similar to a manila file folder) is a collection of information with attributes associated with the information. Process is the software abstraction for the processor; data structure is the software abstraction for memory. Similarly, a file is the software abstraction for an input/output device since a device serves as either a source or sink of information. This abstraction allows a user program to interact with I/O in a device independent manner.

First, we will discuss the attributes associated with a file and the design choices therein, and then consider the design choices in its implementation on a mass storage device.

11.1 Attributes

We refer to the attributes associated with a file as *metadata*. The metadata represents *space overhead* and therefore requires careful analysis as to its utility.

Let us understand the attributes we may want to associate with a file.

- **Name:** This attribute allows logically identifying the contents of the file. For example, if we are storing music files we may want to give a *unique name* to each recording. To enable easy lookup, we may keep a *single* directory file that contains the names of all the music recordings. One can easily see that such a *single level* naming scheme used in early storage systems such as Univac Exec 8 computer systems (in the 70's), is restrictive. Later systems such as DEC TOPS-10 (in the early 80's) used a two-level naming scheme. A top-level directory allows getting to an individual user or a project (e.g. recordings of Billy Joel). The second level identifies a specific file for that user/project (e.g. a specific song).

However, as systems grew in size it became evident that we need a more hierarchical structure to a name (e.g. each user may want to have his/her own music collection of different artists). In other words, we may need a multi-level directory as shown in Figure 11.2.

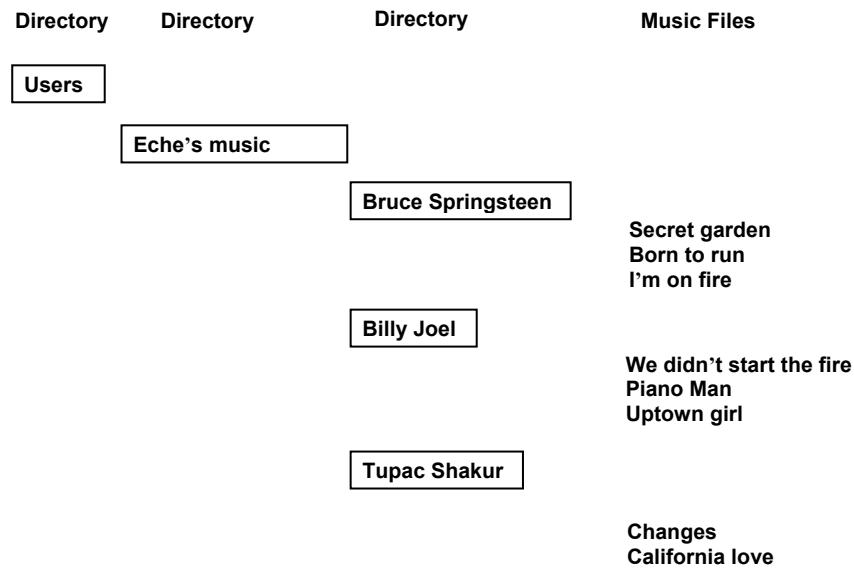


Figure 11.2: A multi-level directory

Most modern operating systems such as Windows XP, Unix, and MacOS implement a multi-part hierarchical name. Each part of the name is *unique* only with respect to the previous parts of the name. This gives a tree structure to the organization of the files in a file system as shown in Figure 11.3. Each node in the tree is a name that is unique with respect to its parent node. Directories are files as well. In the tree structure shown in Figure 11.3, the intermediate nodes are *directory files*, and the leaf nodes are *data files*. The contents of a directory file are information about the files in the next level of the subtree rooted at that directory file (e.g., contents of directory

users are **{students, staff, faculty}**, contents of directory **faculty** are member of the faculty such as **rama**).

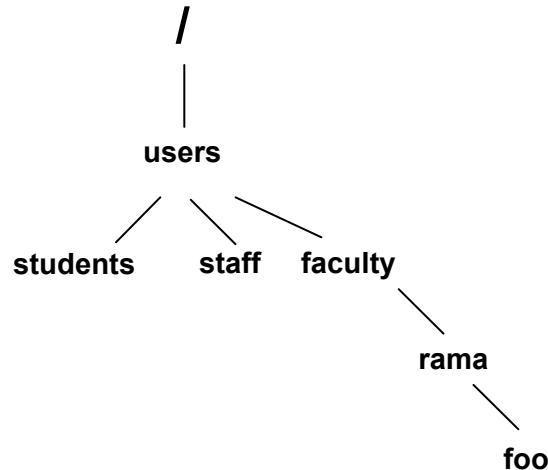


Figure 11.3: A tree structure for the file name /users/faculty/rama/foo

Some operating systems make extensions (in the form of suffixes) to a file name mandatory. For e.g. in DEC TOPS-10 operating system, text files automatically get the .TXT suffix appended to the user supplied name. In UNIX and Windows operating systems, such file name extensions are optional. The system uses the suffix to guess the contents of the file and launch the appropriate application program to manipulate the file (such as the C compiler, document editor, and photo editor).

Some operating systems allow *aliases* to a file. Such aliases may be at the level of the actual content of the file. Alternatively, the alias may simply be at the level of the names and not the actual contents. For example, the **ln** command (stands for link) in UNIX creates an alias to an existing file. The command

ln foo bar

results in creating an alias **bar** to access the contents of an existing file named **foo**. Such an alias, referred to as a *hard link*, gives an equal status to the new name **bar** as the original name **foo**. Even if we delete the file **foo**, the contents of the file are still accessible through the name **bar**.

i-node	access rights	hard links	size	creation time	name
3193357	-rw-----	2	rama	80 Jan 23 18:30	bar
3193357	-rw-----	2	rama	80 Jan 23 18:30	foo

We will explain what an i-node is shortly in Section 11.3.1. For now, suffice it to say that it is a data structure for the representation of a file. Note that both **foo** and **bar** share exactly the same internal representation since they both have the same i-node. This is how both the names have equal status regardless of the order of creation of

foo and **bar**. This is also the reason why both the names have the size, and the same timestamp despite the fact that the name **bar** was created later than **foo**. Contrast the above situation with the command

```
ln -s foo bar
```

This command also results in creating an alias named **bar** for **foo**.

<u>i-node</u>	<u>access rights</u>	<u>hard links</u>	<u>size</u>	<u>creation time</u>	<u>name</u>
3193495	lrwxrwxrwx	1	rama	3 Jan 23 18:52	bar -> foo
3193357	-rw----	1	rama	80 Jan 23 18:30	foo

However, the difference is in the fact that **bar** is *name equivalence* to **foo** and does not directly point to the file contents. Note that the i-nodes for the two names are different. Thus, the time of creation of **bar** reflects the time when the **ln** command was actually executed to create an alias. Also, the size of the files are different. The size of **foo** depicts the actual size of the file (80 bytes), while the size of **bar** is just the size of the name string **foo** (3 bytes). Such an alias is referred to as a *soft link*. It is possible to manipulate the contents of the file with equal privilege with either name. However, the difference arises in file deletion. Deletion of **foo** results in the removal of the file contents. The name **bar** still exists but its alias **foo** and the file contents do not. Trying to access the file named **bar** results in an access error.

One may wonder why the operating system may want to support two different aliasing mechanisms, namely, hard and soft links. The tradeoff is one of efficiency and usability. A file name that is a soft link immediately tells you the original file name, whereas a hard link obfuscates this important detail.

Therefore, soft links increase usability. On the other hand, every time the file system encounters a soft link it has to resolve the alias by traversing its internal data structures (namely, i-nodes in Unix). We will see shortly how this is done in the Unix file system. The hard link directly points to the internal representation of the original file name. Therefore, there is no time lost in name resolution and can lead to improved file system performance.

However, a hard link to a directory can lead to circular lists, which can make deletion operations of directories very difficult. For this reason, operating systems such as Unix disallow creating hard links to directories.

Writing to an existing file results in over-writing the contents in most operating systems (UNIX, Windows). However, such writes may create a new version of the file in a file system that supports versioning.

- **Access rights:** This attribute specifies *who* has access to a particular file and *what* kind of privilege each allowed user enjoys. The privileges generally provided on a

file include: *read, write, execute, change ownership, change privileges*. Certain privileges exist at the individual user level (e.g., either the creator or the users of a file), while some other privileges exist only for the system administrator (**root** in UNIX and **administrator** in Windows). For example, in UNIX the owner of a file may execute the “change the permission mode of a file” command

```
chmod u+w foo      /* u stands for user;
                      w stands for write;
                      essentially this command
                      says add write access
                      to the user;

*/
```

that gives write permission to the owner to a file named **foo**. On the other hand, only the system administrator may execute the “change ownership” command

```
chown rama foo
```

that makes **rama** the new owner of the file **foo**.

An interesting question that arises as a file system designer is deciding on the granularity of access rights. Ideally, we may want to give individual access rights to *each* user in the system to *each* file in the system. This choice results in an $O(n)$ metadata space overhead per file, where n is the number of users in the system. Operating systems exercise different design choices to limit space overhead. For example, UNIX divides the world into three categories: *user, group, all*. *User* is an authorized user of the system; *group* is a set of authorized users; and *all* refers to all authorized users on the system. The system administrator maintains the membership of different group names. For example, students in a CS2200 class may all belong to a group name **cs2200**. UNIX supports the notion of individual ownership and group ownership to any file. The owner of a file may change the group ownership for a file using the command

```
chgrp cs2200 foo
```

that makes **cs2200** the group owner of the file **foo**.

With the world divided into three categories as above, UNIX provides *read, write, and execute* privilege for each category. Therefore, 3 bits encode the access rights for each category (1 each for read, write, execute). The execute privilege allows a file to be treated as an executable program. For example, the compiled and linked output of a compiler is a binary executable file. The following example shows all the visible metadata associated with a file in UNIX:

```
rwxrw-r--    1 rama      fac   2364 Apr 18 19:13 foo
```

The file **foo** is owned by rama, group-owned by fac. The first field gives the access rights for the three categories. The first three bits (rwx) give read, write, and execute privilege to the user (rama). The next three bits (rw-) give read and write privilege (no execute privilege) to the group (fac). The last three bits (r--) give read privilege (no write or execute privilege) to all users. The “1” after the access rights states the number of hard links that exists to this file. The file is of size 2364 bytes and the modification timestamp of the contents of the file is April 18 at 19:13 hours.

Windows operating system and some flavors of UNIX operating systems allow a finer granularity of access rights by maintaining an *access control list (ACL)* for each file. This flexibility comes at the expense of increased metadata for each file.

Attribute	Meaning	Elaboration
Name	Name of the file	Attribute set at the time of creation or renaming
Alias	Other names that exist for the same physical file	Attribute gets set when an alias is created; system such as Unix provide explicit commands for creating aliases for a given file; Unix supports aliasing at two different levels (physical or hard, and symbolic or soft)
Owner	Usually the user who created the file	Attribute gets set at the time of creation of a file; systems such as Unix provide mechanism for the file’s ownership to be changed by the superuser
Creation time	Time when the file was created first	Attribute gets set at the time a file is created or copied from some other place
Last write time	Time when the file was last written to	Attribute gets set at the time the file is written to or copied; in most file systems the creation time attribute is the same as the last write time attribute; Note that moving a file from one location to another preserves the creation time of the file
Privileges <ul style="list-style-type: none">• Read• Write• Execute	The permissions or access rights to the file specifies who can do what to the file;	Attribute gets set to default values at the time of creation of the file; usually, file systems provide commands to modify the privileges by the owner of the file; modern file systems such NTFS provide an access control list (ACL) to give different levels of access to different users
Size	Total space occupied on the file system	Attribute gets set every time the size changes due to modification to the file

Table 11.1: File System Attributes

Unix command	Semantics	Elaboration
touch <name>	Create a file with the name <name>	Creates a zero byte file with the name <name> and a creation time equal to the current wall clock time
mkdir <sub-dir>	Create a sub-directory <sub-dir>	The user must have write privilege to the current working directory (if <sub-dir> is a relative name) to be able to successfully execute this command
rm <name>	Remove (or delete) the file named <name>	Only the owner of the file (and/or superuser) can delete a file
rmdir <sub-dir>	Remove (or delete) the sub-directory named <sub-dir>	Only the owner of the <sub-dir> (and/or the superuse) can remove the named sub-directory
ln -s <orig> <new>	Create a name <new> and make it symbolically equivalent to the file <orig>	This is name equivalence only; so if the file <orig> is deleted, the storage associated with <orig> is reclaimed, and hence <new> will be a dangling reference to a non-existent file
ln <orig> <new>	Create a name <new> and make it physically equivalent to the file <orig>	Even if the file <orig> is deleted, the physical file remains accessible via the name <new>
chmod <rights> <name>	Change the access rights for the file <name> as specified in the mask <rights>	Only the owner of the file (and/or the superuser) can change the access rights
chown <user> <name>	Change the owner of the file <name> to be <user>	Only superuser can change the ownership of a file
chgrp <group> <name>	Change the group associated with the file <name> to be <group>	Only the owner of the file (and/or the superuser) can change the group associated with a file
cp <orig> <new>	Create a new file <new> that is a copy of the file <orig>	The copy is created in the same directory if <new> is a file name; if <new> is a directory name, then a copy with the same name <orig> is created in the directory <new>
mv <orig> <new>	Renames the file <orig> with the name <new>	Renaming happens in the same directory if <new> is a file name; if <new> is a directory name, then the file <orig> is moved into the directory <new> preserving its name <orig>
cat/more/less <name>	View the file contents	

Table 11.2: Summary of Common Unix file system commands

Table 11.1 summarizes common file system attributes and their meaning. Table 11.2 gives a summary of common commands available in most Unix file systems. All the commands are with respect to the current working directory. Of course, an exception to this rule is if the command specifies an absolute Unix path-name (e.g. /users/r/rama).

11.2 Design Choices in implementing a File System on a Disk Subsystem

We started discussion on file systems by presenting a file as a software abstraction for an input/output device. An equally important reason for file systems arises from the need to keep information around beyond the lifetime of a program execution. A file serves as a convenient abstraction for meeting this need. Permanent read/write storage is the right answer for keeping such persistent information around. File system is another important software subsystem of the operating system. Using disk as the permanent storage, we will discuss the design choices in implementing a file system.

As we have seen in Chapter 10, physically a disk consists of platters, tracks, and sectors. A given disk has specific fixed values for these hardware parameters. Logically, the corresponding tracks on the various platters form a cylinder. There are four components to the latency for doing input/output from/to a disk:

- Seek time to a specific cylinder
- Rotational latency to get the specific sector under the read/write head of the disk
- Transfer time from/to the disk controller buffer
- DMA transfer from/to the controller buffer to/from the system memory

We know that a file is of arbitrary size commensurate with the needs of the program. For example, files that contain simple ASCII text may be a few Kilobytes in size. On the other hand, a movie that you may download and store digitally on the computer occupies several hundreds of Megabytes. The file system has to bridge the mismatch between the user's view of a file as a storage abstraction, and physical details of the hard drive. Depending on its size, a file may potentially occupy several sectors, several tracks, or even several cylinders.

Therefore, one of the fundamental design issues in file system is the physical representation of a file on the disk. Choice of a design point should take into consideration both end users' needs and system performance. Let us understand these issues. From a user's perspective, there may be two requirements. One, the user may want to view the contents of a file sequentially (e.g. UNIX **more**, **less**, and **cat commands**). Two, the user may want to search for something in a file (e.g. UNIX **tail** command). The former implies that the physical representation should lend itself to efficient *sequential access* and the latter to *random access*. From the system performance point of view, the file system design should lend itself to easily *growing* a file when needed and for *efficient allocation* of space on the disk for new files or growth of existing files.

Therefore the *figures of merit*¹ for a file system design are:

- Fast sequential access
- Fast random access
- Ability to grow the file
- Easy allocation of storage
- Efficiency of space utilization on the disk

In the next few paragraphs, we will identify several file allocation schemes on a hard drive. For each scheme, we will discuss the data structures needed in the file system, and the impact of the scheme on the figures of merit. We will establish some common terminology for the rest of the discussion. An *address* on the disk is a triple $\{cylinder\#, surface\#, sector\#\}$. The file system views the disk as consisting of *disk blocks*, a design parameter of the file system. Each disk block is a physically contiguous region of the disk (usually a set of sectors, tracks, or cylinders depending on the specific allocation scheme), and is the smallest granularity of disk space managed by the file system. For simplicity of the discussion we will use, *disk block address*, as a short hand for the disk address (the 4-tuple, $\{cylinder\#, surface\#, sector\#, size of disk block\}$) corresponding to a particular disk block, and designate it by a unique integer.

11.2.1 Contiguous Allocation

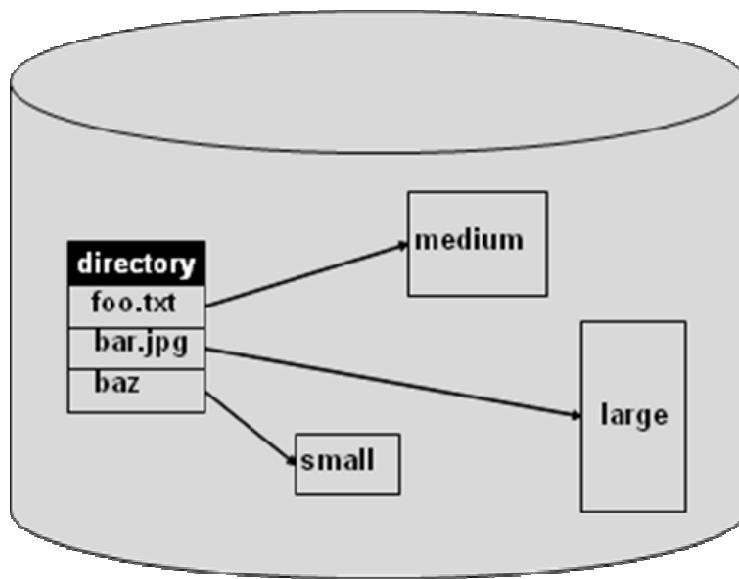


Figure 11.4 shows a disk with a directory that maps file name to a disk block address for contiguous allocation. The size of contiguous allocation matches the size of the file.

This disk allocation scheme has similarities to both the fixed and variable size partition based memory allocation schemes discussed in Chapter 8. At file creation time, the file system pre-allocates a fixed amount of space to the file. The amount of space allocated depends on the type of file (e.g., a text file versus a media file). Further, the amount of

¹ A figure of merit is a criterion used to judge the performance of the system.

space allocated is the maximum size that the file can grow to. Figure 11.4 shows the data structures needed for this scheme. Each entry in the *directory* data structure contains a mapping of the file name to the disk block address, and the number of blocks allocated to this file.

The file system maintains a *free list* of available disk blocks (Figure 11.5). Free list is a data structure that enables the file system to keep track of the currently unallocated disk blocks. In Chapter 8, we discussed how the memory manager uses the free list of physical frames to make memory allocation upon a page fault; similarly, the file system uses the free list of disk blocks to make disk block allocation for the creation of new files. As we will see, the details of this free list depend on the specific allocation strategy in use by the file system.

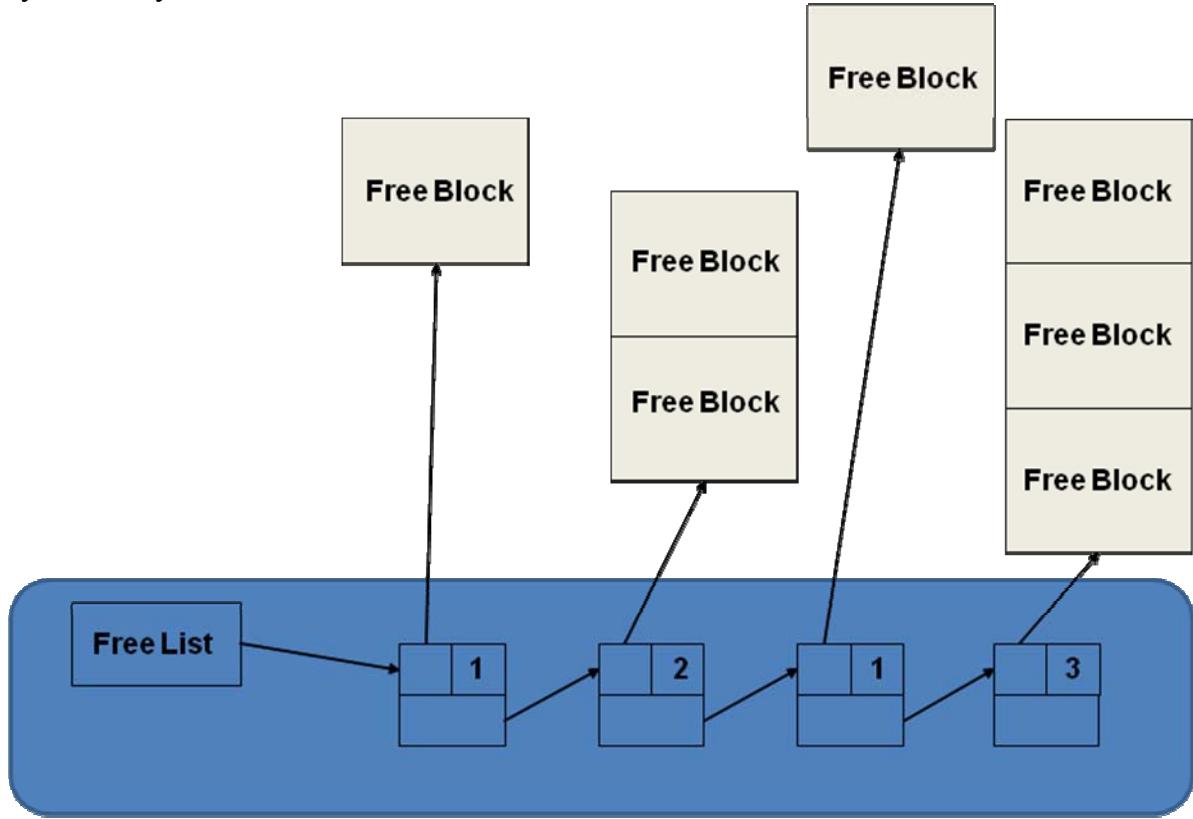


Figure 11.5 shows a free list, with each node containing {pointer to the starting disk block address, number of blocks} for contiguous allocation

For contiguous allocation, each node in this free list gives the starting disk block address and the number of available blocks. Allocation of disk blocks to a new file may follow one of *first fit* or *best fit* policy. Upon file deletion, the released disk blocks return to the free list. The file system combines adjacent nodes to form bigger contiguous disk block partitions. Of course, the file system does such *compaction* infrequently due to the overhead it represents. Alternatively, the file system may choose to do such compaction upon an explicit request from the user. Much of this description should sound familiar to the issues discussed in Chapter 8 for variable size memory partitions. Similar to that

memory management scheme, this disk allocation strategy also suffers from *external fragmentation*. Further, since the file system commits a fixed size chunk of disk blocks (to allow for the maximum expected growth size of that file) at file creation time, this scheme suffers from *internal fragmentation* (similar to the fixed size partition memory allocation scheme discussed in Chapter 8).

The file system could keep these data structures in memory or on the disk. The data structures have to be in persistent store (i.e. some disk blocks are used to implement these data structures) since the files are persistent. Therefore, these data structures reside on the disk itself. However, the file system caches this data structure in memory to enable quick allocation decisions as well as to speed up file access.

Let us analyze this scheme qualitatively with respect to the figures of merit. Allocation can be expensive depending on the algorithm used (first fit or best fit). Since a file occupies a fixed partition (a physically contiguous region on the disk), both sequential access and random access to a file is efficient. Upon positioning the disk head at the starting disk block address for a particular file, the scheme incurs very little additional time for seeking different parts of the file due to the nature of the allocation. There are two downsides to the allocation scheme:

1. The file cannot grow in place beyond the size of the fixed partition allocated at the time of file creation. One possibility to get around this problem is to find a free list node with a bigger partition and copy the file over to the new partition. This is an expensive choice; besides, such a larger partition has to be available. The file system may resort to compaction to create such a larger partition.
2. As we said already, there is potential for significant wastage due to internal and external fragmentation.

Example 1:

Given the following:

Number of cylinders on the disk	=	10,000
Number of platters	=	10
Number of surfaces per platter	=	2
Number of sectors per track	=	128
Number of bytes per sector	=	256
Disk allocation policy	=	contiguous cylinders

- (a) How many cylinders should be allocated to store a file of size 3 Mbyte?
- (b) How much is the internal fragmentation caused by this allocation?

Answer:

(a)

$$\begin{aligned} \text{Number of tracks in a cylinder} &= \text{number of platters} * \text{number of surfaces per platter} \\ &= 10 * 2 = 20 \end{aligned}$$

$$\begin{aligned} \text{Size of track} &= \text{number of sectors in track} * \text{size of sector} \\ &= 128 * 256 \end{aligned}$$

$$\begin{aligned} &= 2^{15} \text{ bytes} \end{aligned}$$

$$\begin{aligned}\text{Capacity in 1 cylinder} &= \text{number of tracks in cylinder} * \text{size of track} \\ &= 20 * 2^{15} \\ &= 10 * 2^{16} \text{ bytes}\end{aligned}$$

$$\text{Number of cylinders to host a 3 MB file} = \text{CEIL } ((3 * 2^{20}) / (10 * 2^{16})) = 5$$

(b)

$$\begin{aligned}\text{Internal fragmentation} &= 5 \text{ cylinders} - 3 \text{ MB} \\ &= 3276800 - 3145728 = 131072 \text{ bytes}\end{aligned}$$

11.2.2 Contiguous Allocation with Overflow Area

This strategy is the same as the previous one with the difference that the file system sets aside an *overflow* region that allows spillover of large files that do not fit within the fixed partition. The overflow region also consists of physically contiguous regions allocated to accommodate such spillover of large files. The file system needs an additional data structure to manage the overflow region. This scheme fares exactly similarly with respect to the figures of merit as the previous scheme. However, on the plus side this scheme allows file growth limited only by the maximum amount of space available in the overflow region without the need for any other expensive operation as described above. On the minus side, random access suffers slightly for large files due to the spill into the overflow region (additional seek time).

Despite some of the limitations, contiguous allocation has been used quite extensively in systems such IBM VM/CMS due to the significant performance advantage for file access times.

11.2.3 Linked Allocation

In this scheme, the file system deals with allocation at the level of individual disk blocks. The file system maintains a *free list* of all available disk blocks. A file occupies as many disk blocks as it takes to store it on the disk. The file system allocates the disk blocks from the free list as the file grows. The free list may actually be a linked list of the disk blocks with each block pointing to the next free block on the disk. The file system has the head of this list cached in memory so that it can quickly allocate a disk block to satisfy a new allocation request. Upon deletion of a file, the file system adds its disk blocks to the free list. In general having such a linked list implemented via disk blocks leads to expensive traversal times for free-list maintenance. An alternative is to implement the free list as a bit vector, one bit for each disk block. The block is free if the corresponding bit is 0 and busy if it is 1.

Note that the free list changes over time as applications produce and delete files, and as files grow and shrink. Thus, there is no guarantee that a file will occupy contiguous disk blocks. Therefore, a file is physically stored as a linked list of the disk blocks assigned to it as shown in Figure 11.6. As in the previous allocation schemes, some disk blocks hold the persistent data structures (free list and directory) of the file system.

On the plus side, the allocation is quick since it is one disk block at a time. Further, the scheme accommodates easy growth of files. There is no external fragmentation due to the on-demand allocation. Consequently, the scheme never requires disk compaction. On the minus side, since the disk blocks of a file may not be contiguous the scheme performs poorly for file access compared to contiguous allocation, especially for random access due to the need for retrieving next block pointers from the disk blocks. Even for sequential access, the scheme may be inefficient due to the variable seek time for the disk blocks that make up the list. The error-prone nature of this scheme is another downside: any bug in the list maintenance code leads to a complete breakdown of the file system.

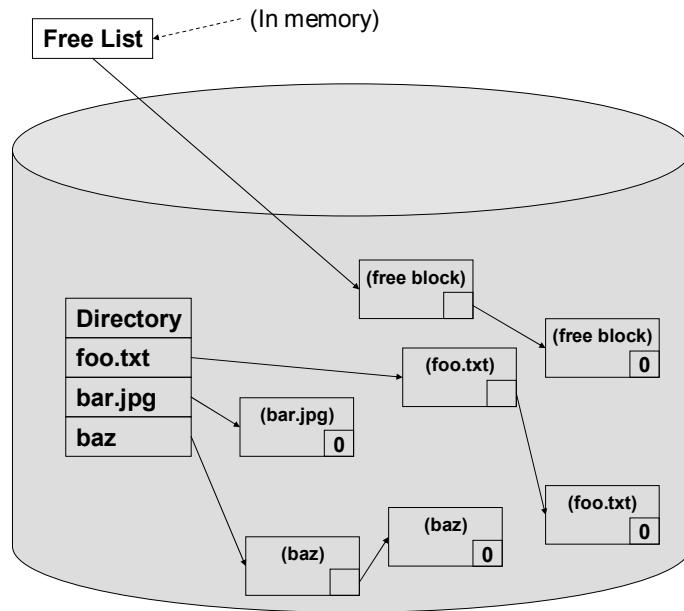


Figure 11.6: Linked Allocation

11.2.4 File Allocation Table (FAT)

This is a variation of the linked allocation. A table on the disk, *File Allocation Table (FAT)*, contains the linked list of the files currently populating the disk (see Figure 11.7). The scheme divides the disk logically into partitions. Each partition has a FAT, in which each entry corresponds to a particular disk block. The *free/busy* field indicates the availability of that disk block (0 for free; 1 for busy). The *next* field gives the next disk block of the linked list that represents a file. A distinguished value (-1) indicates this entry is the last disk block for that file. A single *directory* for the entire partition contains the *file name* to *FAT index* mapping as shown in Figure 11.7. Similar to the linked allocation, the file system allocates a disk block on demand to a file.

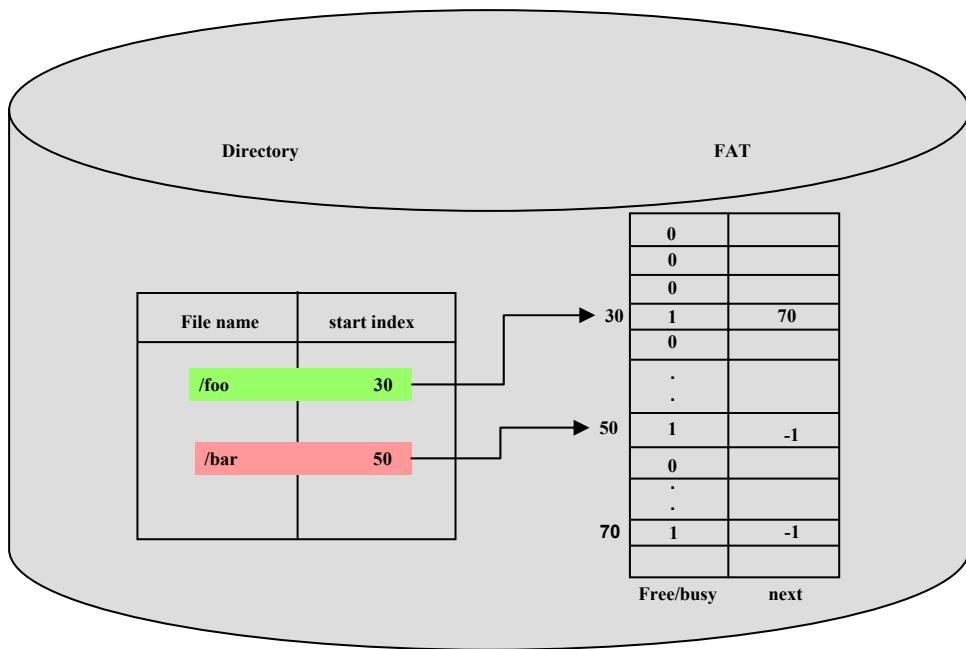


Figure 11.7: File Allocation Table (FAT)

For example, */foo* occupies two disk blocks: 30 and 70. The *next* field of entry 30 contains the value 70, the address of the next disk block. The next field of entry 70 contains -1 indicating this is the last block for */foo*. Similarly, */bar* occupies one disk block (50). If */foo* or */bar* were to grow, the scheme will allocate a free disk block and fix up the FAT accordingly.

Let us analyze the *pros* and *cons* of this scheme. Since FAT captures the linked list structure of the disk in a tabular data structure, there is less chance of errors compared to the linked allocation. By caching FAT in memory, the scheme leads to efficient allocation times compared to linked allocation. The scheme performs similar to linked allocation for sequential file access. It performs better than linked allocation for random access since the FAT contains the next block pointers for a given file.

One of the biggest downside to this scheme is the logical partitioning of a disk. This leads to a level of management of the space on the disk for the end user that is not pleasant. It creates artificial scarcity of space on the disk in a particular partition even when there is plenty of physical space on the disk. However, due to its simplicity (centralized data structures in the directory and FAT) this allocation scheme was popular in early personal computer operating systems such as MS-DOS and IBM OS/2.

Example 2:

This question is with respect to the disk space allocation strategy referred to as **FAT**. Assume there are 20 data blocks numbered 1 through 20.

There are three files currently on the disk:

foo occupies disk blocks 1, 2 and 3;

bar occupies disk blocks 10, 13, 15, 17, 18 and 19

gag occupies disk blocks 4, 5, 7 and 9

Show the contents of the **FAT** (show the free blocks and allocated blocks per convention used in this section).

Answer:

1	2
2	3
3	-1
4	5
5	7
6	0
7	9
8	0
9	-1
10	13
11	0
12	0
13	15
14	0
15	17
16	0
17	18
18	19
19	-1
20	0

11.2.5 Indexed Allocation

This scheme allocates an *index* disk block for each file. The index block for a file is a fixed-size data structure that contains addresses for data blocks that are part of that file. Essentially, this scheme aggregates data block pointers for a file scattered all over the FAT data structure (in the previous scheme) into one table per file as shown in Figure 11.8. This table called *index node* or *i-node*, occupies a disk block. The *directory* (also on the disk) contains the *file name* to *index node* mapping for each file. Similar to the linked allocation, this scheme maintains the *free list* as a bit vector of disk blocks (0 for free, 1 for busy).

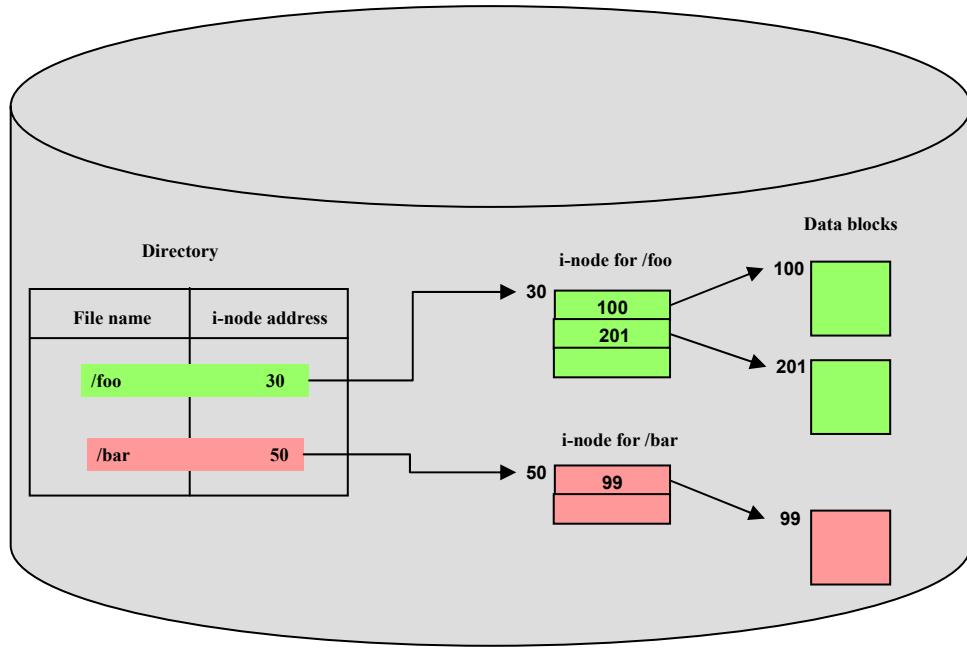


Figure 11.8: Indexed Allocation

Compared to FAT, this scheme performs better for random access since i-node aggregates all the disk block pointers into one concise data structure. The downside to this scheme is the limitation on the maximum size of a file, since i-node is a fixed size data structure per file with direct pointers to data blocks. The number of data block pointers in an i-node bounds the maximum size of the file.

We will explore other schemes that remove this restriction on maximum file size in the following subsections.

Example 3:

Consider an indexed allocation scheme on a disk

- The disk has 10 platters (2 surfaces per platter)
- There are 1000 tracks in each surface
- Each track has 400 sectors
- There are 512 bytes per sector
- Each i-node is a fixed size data structure occupying one sector.
- A data block (unit of allocation) is a contiguous set of 2 cylinders
- A pointer to a disk data block is represented using an 8-byte data structure.

- a) What is the minimum amount of space used for a file on this system?
- b) What is the maximum file size with this allocation scheme?

Answer:

$$\begin{aligned} \text{Size of a track} &= \text{number of sectors per track} * \text{size of sector} \\ &= 400 * 512 \text{ bytes} = 200 \text{ Kbytes (K = 1024)} \end{aligned}$$

$$\text{Number of tracks in a cylinder} = \text{number of platters} * \text{number of surfaces per platter}$$

$$= 10 * 2 = 20$$

$$\begin{aligned}\text{Size of a cylinder} &= \text{number of tracks in a cylinder} * \text{size of track} \\ &= 20 * 200 \text{ Kbytes} \\ &= 4000 \text{ Kbytes}\end{aligned}$$

$$\begin{aligned}\text{Unit of allocation (data block)} &= 2 \text{ cylinders} \\ &= 2 * 4000 \text{ Kbytes} \\ &= 8000 \text{ Kbytes}\end{aligned}$$

Size of i-node = size of sector = 512 bytes

a)

$$\begin{aligned}\text{Minimum amount of space for a file} &= \text{size of i-node} + \text{size of data block} \\ &= 512 + (8000 * 1024) \text{ bytes} \\ &= \mathbf{8,192,512 \text{ bytes}}\end{aligned}$$

$$\begin{aligned}\text{Number of data block pointers in an i-node} &= \text{size of i-node} / \text{size of data block pointer} \\ &= 512/8 = 64\end{aligned}$$

b)

$$\begin{aligned}\text{Maximum size of file} &= \text{Number of data block pointers in i-node} * \text{size of data block} \\ &= 64 * 8000 \text{ K bytes (K = 1024)} \\ &= \mathbf{5,24,288,000 \text{ bytes}}\end{aligned}$$

11.2.6 Multilevel Indexed Allocation

This scheme fixes the limitation in the indexed allocation by making the i-node for a file an indirection table. For example, with one level indirection, each i-node entry points to a first level table, which in turn points to the data blocks as shown in Figure 11.9. In this figure, the i-node for foo contains pointers to a first level indirection index blocks. The number of first level indirection tables equals the number of pointers that an i-node can hold. These first level indirection tables hold pointers to the data blocks of the file.

The scheme can be extended to make an i-node a two (or even higher) level indirection table depending on the size of the files that need to be supported by the file system. The downside to this scheme is that even small files that may fit in a few data blocks have to go through extra levels of indirection.

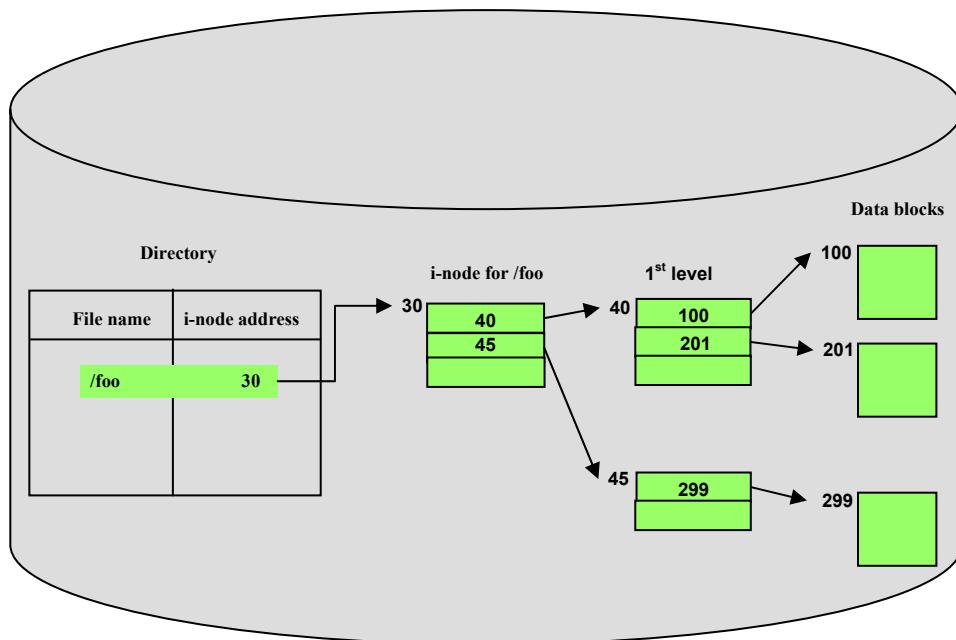


Figure 11.9: Multilevel Indexed Allocation (with one level of indirection)

11.2.7 Hybrid Indexed Allocation

This scheme combines the ideas in the previous two schemes to form a hybrid scheme. Every file has an i-node as shown in Figure 11.10. The scheme accommodates all the data blocks for small file with direct pointers. If the file size exceeds the capacity of direct blocks, then the scheme uses single or more levels of indirection for the additional data blocks. Figure 11.10 shows */foo* using direct, single indirect, and double indirect pointers. The i-node for */foo* is a complex data structure as shown in the figure. It has pointers to two direct data blocks (100 and 201); one pointer to an indirect index block (40); one pointer to a double indirect index block (45) which in turn has pointers to single indirect index blocks (60 and 70); and one pointer to a triple indirect index block (currently unassigned). The scheme overcomes the disadvantages of the previous two schemes while maintaining their good points. When a file is created with this scheme only the i-node is allocated for it. That is, the single, double, and triple indirect pointers are initially null. If the file size does not exceed the amount of space available in the direct blocks then there is no need to create additional index blocks. However, as the file grows and exceeds the capacity of the direct data blocks, space is allocated on a need basis for the first level index block, second level index block, and third level index block.

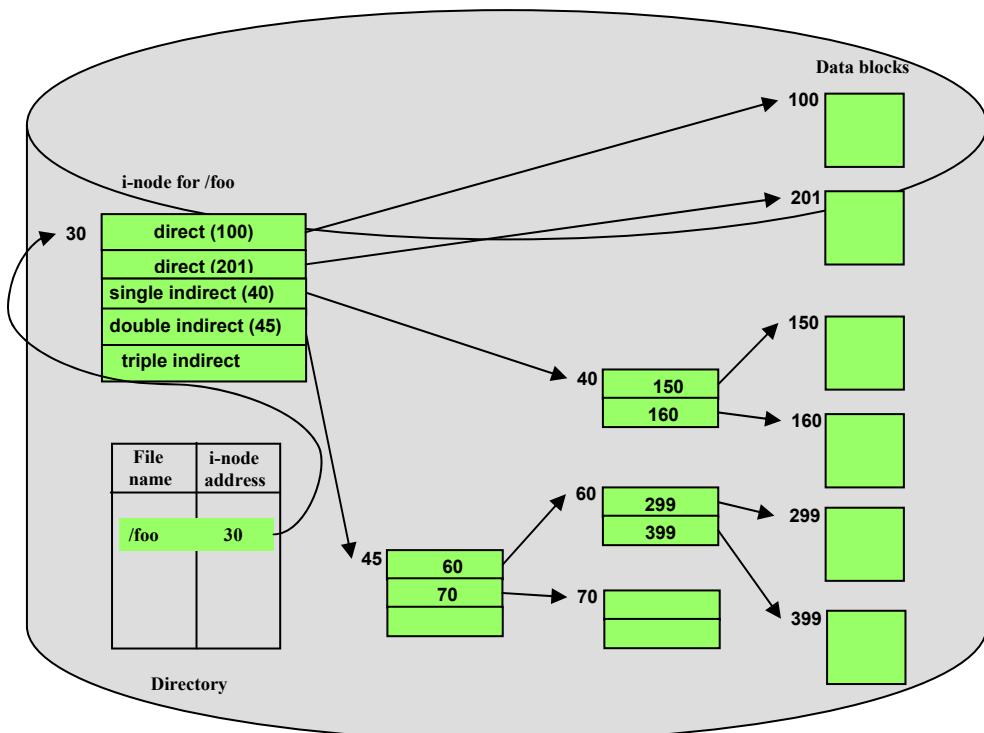


Figure 11.10: Hybrid Indexed Allocation

Example 4:

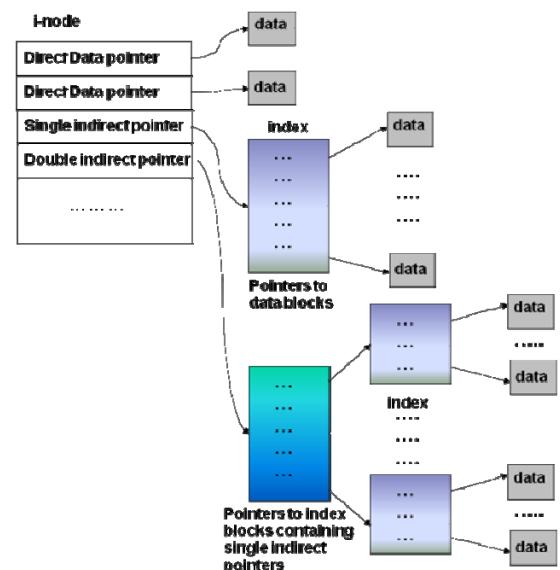
Given the following:

- Size of index block = 512 bytes
- Size of Data block = 2048 bytes
- Size of pointer = 8 bytes (to index or data blocks)

The i-node consists of

- 2 direct data block pointers,
- 1 single indirect pointer, and
- 1 double indirect pointer.

An index block is used for the i-node as well as for the index blocks that store pointers to other index blocks and data blocks. Pictorially the organization is as shown in the figure on the right. Note that the index blocks and data blocks are allocated on a need basis.



- (a) What is the maximum size (in bytes) of a file that can be stored in this file system?
- (b) How many data blocks are needed for storing a data file of 266 KB?
- (c) How many index blocks are needed for storing the same data file of size 266 KB?

Answer:

(a)

Number of single indirect or double indirect pointers in an index block
 $= 512/8$
 $= 64$

Number of direct data blocks = 2

An i-node contains 1 single indirect pointer that points to an index block (we call this *single indirect index block*) containing pointers to data blocks.

Number of data blocks with one level of indirection

$$\begin{aligned} &= \text{number of data block pointers in an index block} \\ &= 64 \end{aligned}$$

An i-node contains 1 double indirect pointer that points to an index block (we call this *double indirect index block*) containing 64 single indirect pointers. Each such pointer point to a single indirect index block that in turn contains 64 pointers to data blocks. This is shown in the figure.

Number of data block with two levels of indirection

$$\begin{aligned} &= \text{number of single indirect pointers in an index block *} \\ &\quad \text{number of data block pointers index node} \\ &= 64*64 \end{aligned}$$

Max file size in blocks

$$\begin{aligned} &= \text{number of direct data blocks +} \\ &\quad \text{number of data blocks with one level of indirection +} \\ &\quad \text{number of data blocks with two levels of indirection} \\ &= 2 + 64 + 64*64 = 4162 \text{ data blocks} \end{aligned}$$

Max file size

$$\begin{aligned} &= \text{Max file size in blocks * size of data block} \\ &= 4162 * 2048 \text{ bytes} \\ &= \mathbf{8,523,776 \text{ bytes}} \end{aligned}$$

(b)

Number of data blocks needed

$$\begin{aligned} &= \text{size of file / size of data block} \\ &= 266 * 2^{10} / 2048 \\ &= \mathbf{133} \end{aligned}$$

(c)

To get 133 data blocks we need:

- 1 i-node** (gets us 2 direct data blocks)
- 1 single indirect index block** (gets us 64 data blocks)
- 1 double indirect index block**
- 2 single indirect index blocks** (gets us the remaining
off the double indirect index block $64 + 3$ data blocks)

Therefore, totally we need

5 index blocks

11.2.8 Comparison of the allocation strategies

Table 11.3 summarizes the relative advantages and disadvantages of the various allocation strategies.

Allocation Strategy	File representation	Free list maintenance	Sequential Access	Random Access	File growth	Allocation Overhead	Space Efficiency
Contiguous	Contiguous blocks	complex	Very good	Very good	messy	Medium to high	Internal and external fragmentation
Contiguous With Overflow	Contiguous blocks for small files	complex	Very good for small files	Very good for small files	OK	Medium to high	Internal and external fragmentation
Linked List	Non-contiguous blocks	Bit vector	Good but dependent on seek time	Not good	Very good	Small to medium	Excellent
FAT	Non-contiguous blocks	FAT	Good but dependent on seek time	Good but dependent on seek time	Very good	Small	Excellent
Indexed	Non-contiguous blocks	Bit vector	Good but dependent on seek time	Good but dependent on seek time	limited	Small	Excellent
Multilevel Indexed	Non-contiguous blocks	Bit vector	Good but dependent on seek time	Good but dependent on seek time	Good	Small	Excellent
Hybrid	Non-	Bit vector	Good but	Good	Good	Small	Excellent

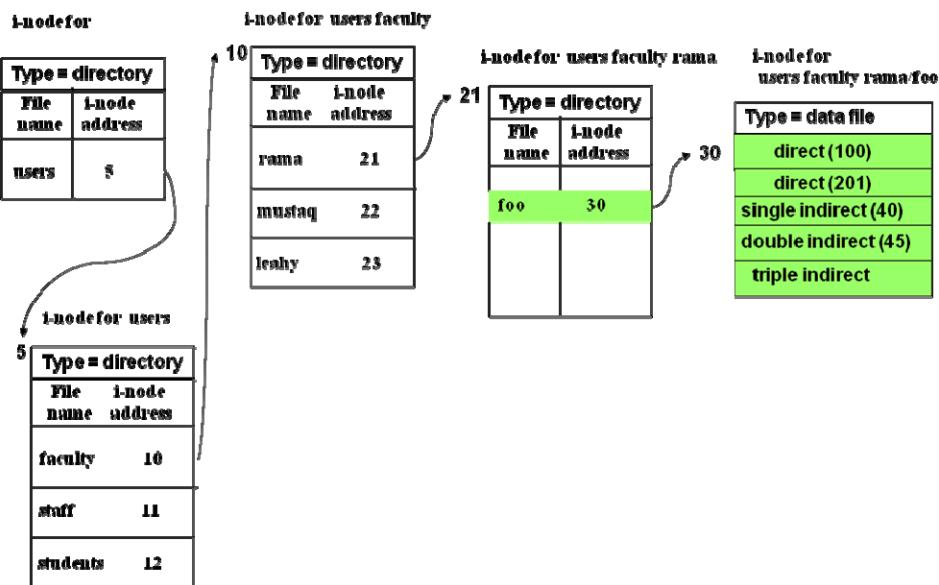
	contiguous blocks		dependent on seek time	but dependent on seek time			
--	-------------------	--	------------------------	----------------------------	--	--	--

Table 11.3: Comparison of Allocation Strategies

For the integrity of the file system across power failures, the persistent data structures of the file system (e.g., i-nodes, free list, directories, FAT, etc.) have to reside on the disk itself. The number of disk blocks devoted to hold these data structures represents space overhead of the particular allocation strategy. There is a time penalty as well for accessing these data structures. File systems routinely cache these critical data structures in main memory to avoid the time penalty.

11.3 Putting it all together

As a concrete example, UNIX operating system uses a hybrid allocation approach with hierarchical naming. With hierarchical naming, the directory structure is no longer centralized. Each i-node represents a part of the multipart hierarchical name. Except for the leaf nodes, which are data files, all the intermediate nodes are directory nodes. The type field in the i-node identifies whether this node is a directory or a data file. The i-node data structure includes fields for the other file attributes we discussed earlier such as access rights, timestamp, size, and ownership. In addition, to allow for aliasing, the i-node also includes a *reference count* field.



**Figure 11.11: A simplified i-node structure for a hierarchical name in UNIX
(a sequence of commands have resulted in creating the file /users/faculty/rama/foo)**

Figure 11.11 shows the complete i-node structure for a file `/users/faculty/rama/foo`. For the sake of simplicity, we do not show the data blocks.

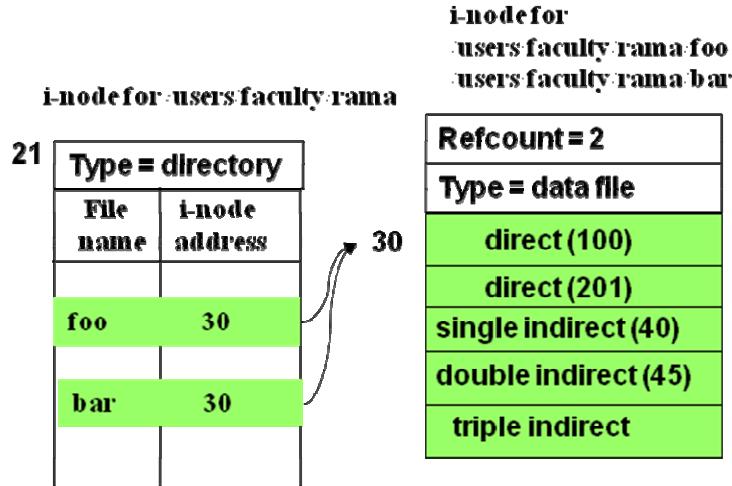


Figure 11.12: Two files `foo` and `bar` share an i-node since they are hard links to each other (the command “`ln foo bar`” results in this structure)

Figure 11.12 shows the i-node structure with `bar` as a hard link to `foo`. Both the files share the same i-node. Hence the *reference count* field in the i-node is 2. Figure 11.13 shows the i-node structure when a third file `baz` is symbolically linked to the name `/users/faculty/rama/foo`. The i-node for `baz` denotes it as a symbolic link and contains the name `/users/faculty/rama/foo`. The file system starts traversing the i-node structure given by the symbolic name (i.e. in this case, upon accessing `baz`, the traversal will start from the i-node for `/`).

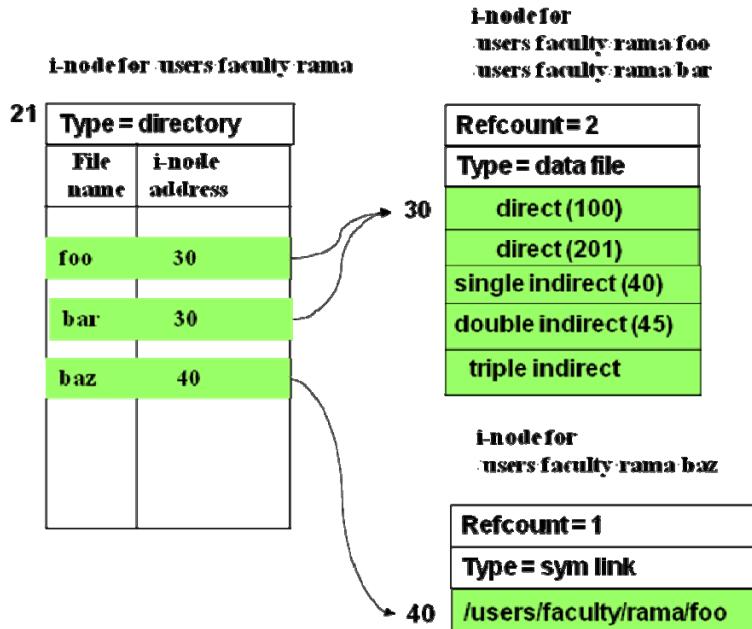


Figure 11.13: File `baz` is a symbolic link to `/users/faculty/rama/foo`

(the command “ln -s /users/faculty/rama/foo baz”
results in this structure)

Example 5:

Current directory /tmp
I-node for /tmp 20

The following Unix commands are executed in the current directory:

```
touch foo                         /* creates a zero byte file
                                      in the current directory */
ln foo bar                        /* create a hard link */
ln -s /tmp/foo baz            /* create a soft link */
ln baz gag                        /* create a hard link */
```

Note:

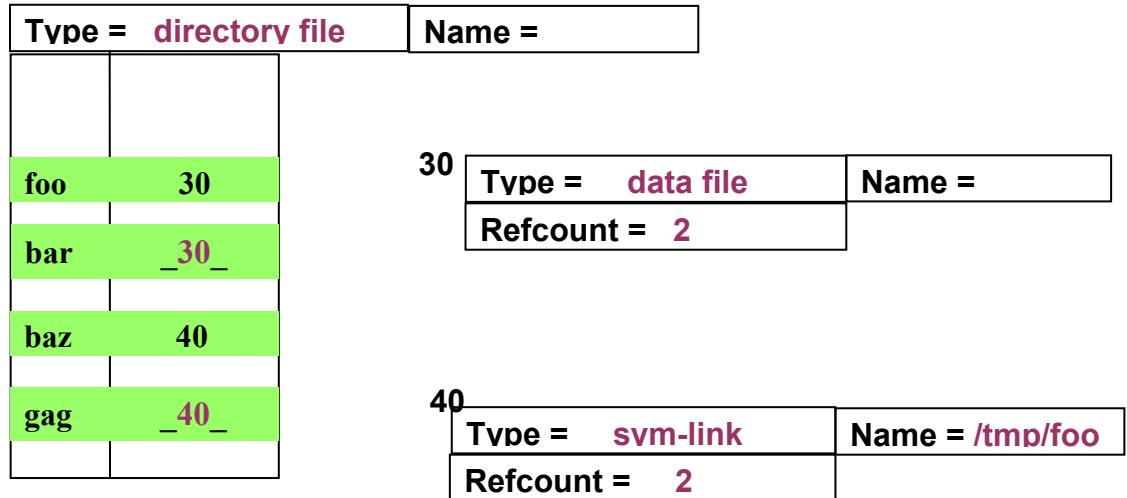
- Type of i-node can be one of directory-file, data-file, sym-link
 - if the type is sym-link then you have to give the name associated with that sym-link; otherwise the name field in the i-node is blank
- reference count is a non-zero positive integer

In the picture below fill in all the blanks to complete the contents of the various i-nodes.

i-node for /tmp			
20	Type = _____	Name = _____	
foo	30	Type = _____	Name = _____
bar	_____	RefCount = _____	
baz	40	Type = _____	Name = _____
gag	_____	RefCount = _____	

Answer:

20 i-node for /tmp



Example 6:

Given the following commands pictorially show the i-nodes and their contents. You can fabricate disk block addresses for the i-nodes to make the pictures simpler. Where relevant show the reference count for the i-nodes.

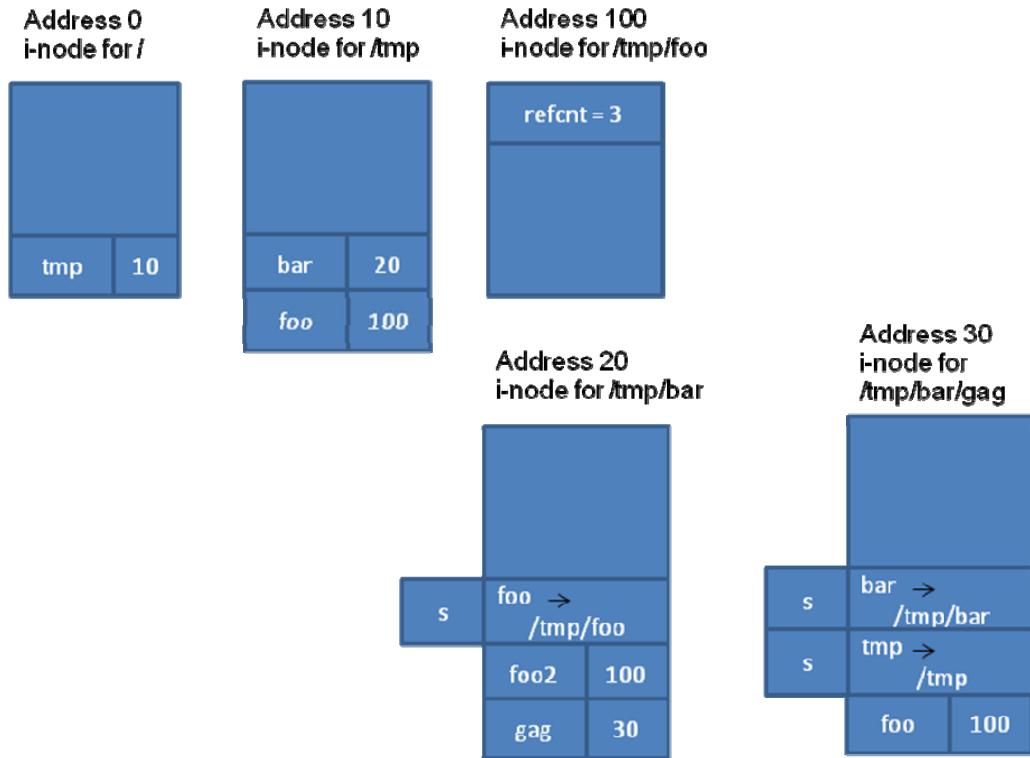
```
touch /tmp/foo
mkdir /tmp/bar
mkdir /tmp/bar/gag
ln /tmp/foo /tmp/bar/foo2
ln -s /tmp/foo /tmp/bar/foo
ln /tmp/foo /tmp/bar/gag/foo
ln -s /tmp/bar /tmp/bar/gag/bar
ln -s /tmp /tmp/bar/gag/tmp
```

Note:

mkdir creates a directory; touch creates a zero byte file; ln is link command (-s denotes symbolic link).

Assume that the above files and directories are the only ones in the file system

Answer



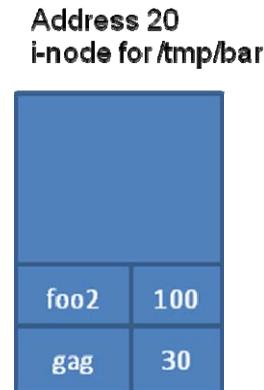
If we now execute the command:

```
rm /tmp/bar/foo
```

Show the new contents of the i-nodes. (SHOW ONLY THE AFFECTED I-NODES).

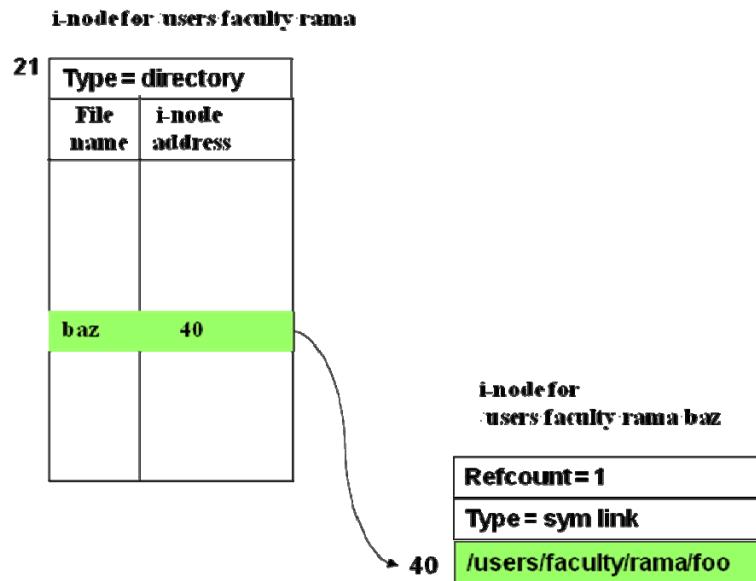
Answer:

Only i-node for /tmp/bar changes as shown below. The symbolic link entry is removed from the i-node.



With reference to Figure 11.13, first let us understand what would happen to the i-node structure if the file *bar* was deleted. The entry bar will be removed from the i-node for */user/faculty/rama* and the *reference count* field in the i-node for */users/faculty/rama/foo* (block 30) will be decremented by 1 (i.e. the new *refcount* will be 1).

Next, let us see what would happen to the i-node structure once both *bar* and *foo* are deleted. Figure 11.14 depicts this situation. Since the *refcount* for i-node at block 30 goes to 0 with the deletion of both *foo* and *bar*, the file system returns the disk block 30 to the free list. Notice that *baz* still exists as a file in */users/faculty/rama*. This is because the file system only checks if the name being aliased at the time of creation of the symbolic link is valid. However, notice that no information about the symbolic link is kept in the i-node of the name (“*foo*” in this case). Therefore, there is no way for the file system to check for existence of symbolic links to a name at the time of deleting a name (“*foo*” in this case). However, any attempt to examine the contents of *baz* results in an error since the name */users/faculty/rama/foo* is non-existent so far as the file system is concerned. This shows illustrates the need for exercising care while using symbolic links.



**Figure 11.14: State of i-node structure after both foo and bar are deleted.
(the commands “rm foo bar” results in this structure)**

Example 7:

Consider the following:

```

touch foo;          /* creates a zero byte file */
ln foo bar;        /* creates a hard link called bar
                     to foo */
ln -s foo baz;    /* creates a soft link called baz
                     to foo */
rm foo;            /* removes the file foo */

```

What would happen if we now execute the command

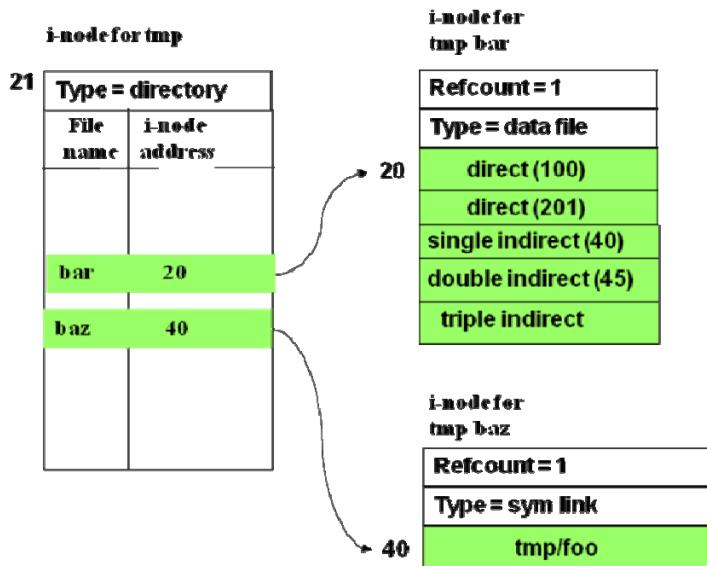
```
cat baz;          /* attempt to view the content of baz
                     */
```

Answer:

Let the current directory be tmp where all the action shown above takes place. In this case, foo, bar, and baz are names created in the i-node for tmp.

Let the i-node for foo be 20. When we create the hard link bar, the refcount for i-node 20 goes up to 2. When foo is removed, the refcount for i-node drops to 1 but the i-node is not removed since bar has a hard link to it; however, the name foo is removed from the i-node of the current directory.

Therefore, when we attempt to view the contents of baz using the cat command, the file system flags an error since the name foo does not exist in the current directory any more. This is shown in the following figure:



11.3.1 i-node

In Unix, every file has a *unique* number associated with it, called the *i-node* number. You can think of this number as an *index* into a table that has all the information associated with the file (owner, size, name, etc.). As Unix systems evolved the complexity of the information associated with each file increased as well (the name itself can be arbitrarily long, etc.). Thus in modern Unix systems, each file is represented by a unique i-node data structure that occupies an entire disk block. The collection of i-nodes thus forms a logical table and the i-node number is simply the unique disk block address that indexes into this logical table and contains the information about that particular file. For the sake of convenience in implementing the file system, all the i-nodes occupy spatially adjacent disk blocks in the layout of the file system on the physical media. Typically, the file system reserves a sufficiently large number of disk blocks as i-nodes.

This implicitly becomes the limit of the maximum number of files in a given file system. It is customary for the storage allocation algorithm to maintain a bit vector (one bit for each i-node) that indicates whether a particular i-node is currently in use or not. This allows for efficient allocation of i-nodes upon file creation requests. Similarly, the storage manager may implement the free list of data blocks on the disk as a bit vector (one bit to signify the use or otherwise of a data block on the media) for efficient storage allocation.

11.4 Components of the File System

While it is possible to implement file systems at the user level, typically it is part of the operating system. Figure 11.15 shows the layers of the file system for the disk. We refer to the part of the operating system that manages the file system as the *File System (FS) Manager*. We break down the FS manager into the following layers for the sake of exposition.

- **Media independent layer:** This layer consists of the user interface, i.e. the *Application Program Interface (API)* provided to the users. The API module gives the file system commands for the user program to open and close files, read and write files, etc. This layer also consists of the *Name Resolver* module that translates the user-supplied name to an internal representation that is meaningful to the file system. For instance, the name resolver will be able to map the user specific name (e.g., E:\myphotos) to the specific device on which the file exists (such as the disk, CD, and Flash drive).
- **Media specific storage space allocation layer:** This layer embodies the space allocation (on file creation), space reclamation (on file deletion), free-list maintenance, and other functions associated with managing the space on the physical device. For instance, if the file system exists on disk, then the data structures and algorithms will reflect the discussions in Section 11.2.
- **Device driver:** This is the part that deals with communicating the command to the device and effecting the data transfer between the device and the operating system buffers. The device driver details (see Chapter 10 for a general description of device drivers) depend on the specifics of the mass storage that hosts the file system.
- **Media specific requests scheduling layer:** This layer is responsible for scheduling the requests from the OS commensurate with the physical properties of the device. For example, in the case of a disk, this layer may embody the disk scheduling algorithms that we learned in Chapter 10. As we observed in Chapter 10, even in the case of a disk, the scheduling algorithms may in fact be part of the device controller that is part of the drive itself. The scheduling algorithm may be quite different depending on the nature of the mass storage device.

There will be distinct instances of the bottom three layers of the software stack shown in Figure 11.15 for each mass storage device that provides a file system interface to the user. Thus, file is a powerful abstraction for hiding the details of the actual physical storage on which the data pertaining to the file is kept.

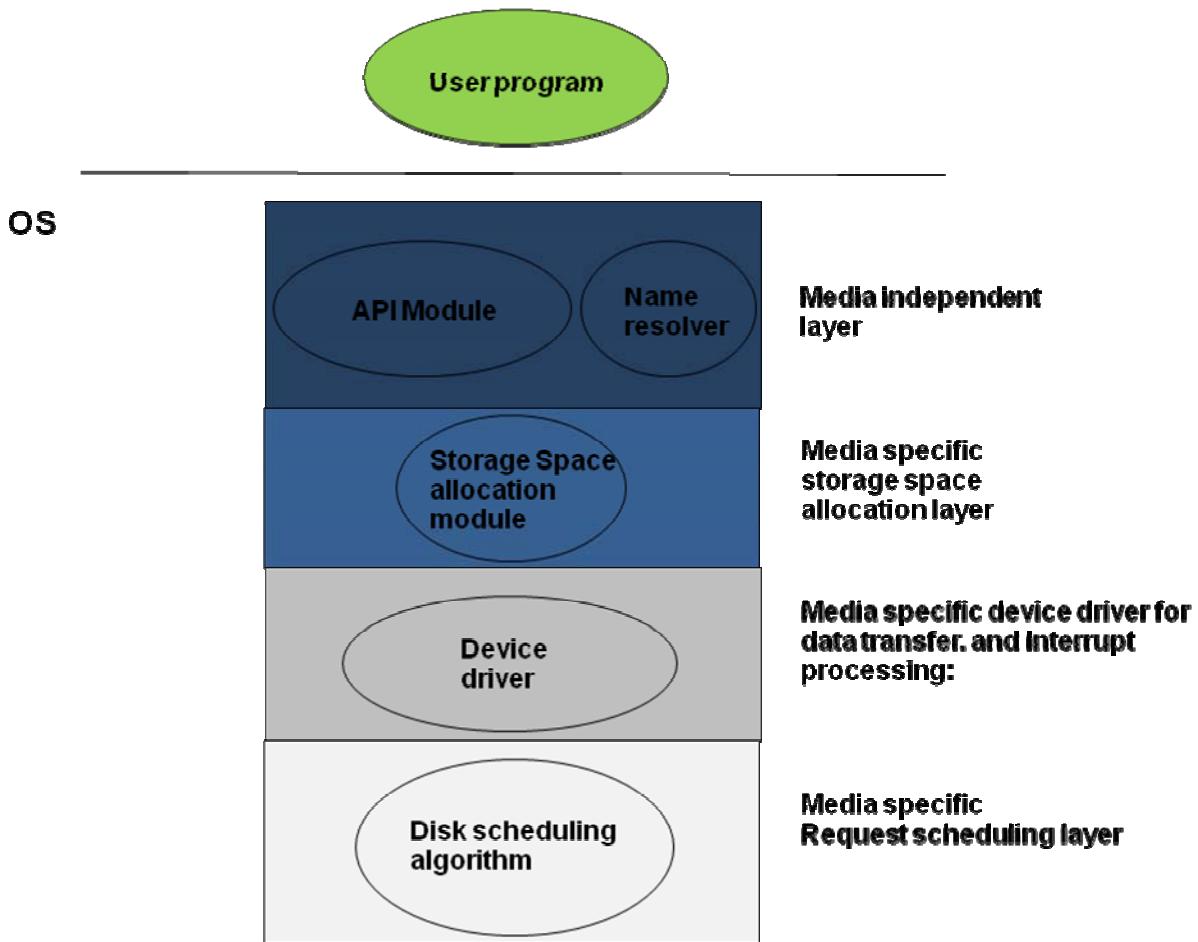


Figure 11.15: Layers of a disk-specific file system manager

11.4.1 Anatomy of creating and writing files

As a concrete example, let us say your program makes an I/O call to create a file on the hard disk. The following steps trace the path of such an I/O call through the software layers shown in Figure 11.15.

1. The API routine for creating a file calls validates the call by checking the permissions, access rights, and other related information for the call. After such validation, it calls the name resolver.
2. The name resolver contacts the storage allocation module to allocate an i-node for the new file.
3. The storage allocation module gets a disk block from the free list and returns it to the name resolver. The storage allocation module will fill in the i-node commensurate with the allocation scheme (see Section 11.2). Let us assume the allocation in effect is the hybrid scheme (Section 11.2.7). Since the file has been created without any data as yet, no data blocks would have been allocated to the file.
4. The name resolver creates a directory entry and records the name to i-node mapping information for the new file in the directory.

Notice that these steps do not involve actually making a trip to the device since the data structures accessed by the file system (directory and free list) are all in memory.

Now let us say, your program writes to the file just created. Let us trace the steps through the software layers for this operation.

1. As before the API routine for file write will do its part in terms of validating the request.
2. The name resolver passes the memory buffer to the storage allocation module along with the i-node information for the file.
3. The storage allocation module allocates data blocks from the free list commensurate with the size of the file write. It then creates a request for disk write and hands the request to the device driver.
4. The device driver adds the request to its request queue. In concert with the disk-scheduling algorithm, the device driver completes the write of the file to the disk.
5. Upon completion of the file write, the device driver gets an interrupt from the disk controller that is passed back up to the file system, which in turn communicates with the CPU scheduler to continue the execution of your program from the point of file write.

It should be noted that as far as the OS is concerned, the file write call is complete as soon as the request is handed to the device driver. The success or failure of the actual call will be known later when the controller interrupts. The file system interacts with the CPU scheduler in exactly the same manner, as did the memory manager. The memory manager, in order to handle a page fault, makes an I/O request on behalf of the faulting process. The memory manager gets notified when the I/O completes so that it can tell the CPU scheduler to resume execution of the faulting process (see Chapter 8, Section 8.2). This is exactly what happens with the file system as well.

11.5 Interaction among the various subsystems

It is interesting to review the various software subsystems we have come across within the operating system so far: CPU scheduler, VM manager, File System (FS) Manager, and device drivers for the various I/O devices.

All of these subsystems are working for the user programs of course. For example, the VM manager is servicing page faults that a user program incurs implicitly performing I/O on behalf of the faulting process. The FS manager explicitly performs I/O catering to the requests of a user program to read and write files. In both cases, some mass storage device (hard disk, Flash memory, CD, DVD, etc.) is the target of the I/O operation. The device driver pertinent to the specific I/O request performs the I/O request and reports the result of the I/O operation back to the requestor.

The device driver has to figure out somehow whose I/O request was completed upon an I/O completion interrupt from the device controller. One simple and unifying method is to use the PCB data structure that we introduced in Chapter 6 (Section 6.4) as a communication vehicle among the subsystems.

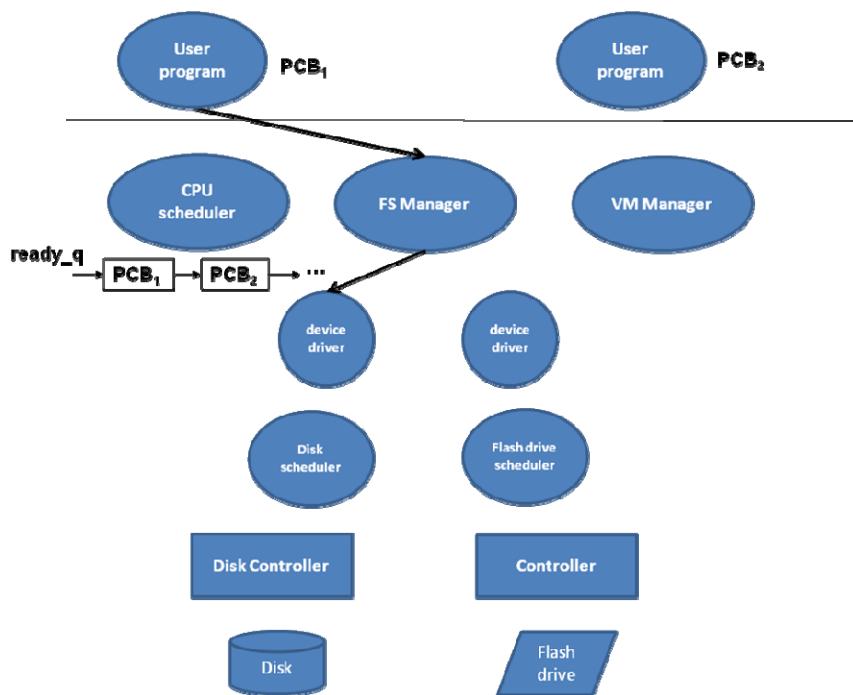


Figure 11.16-(a): Information flow on a file system call

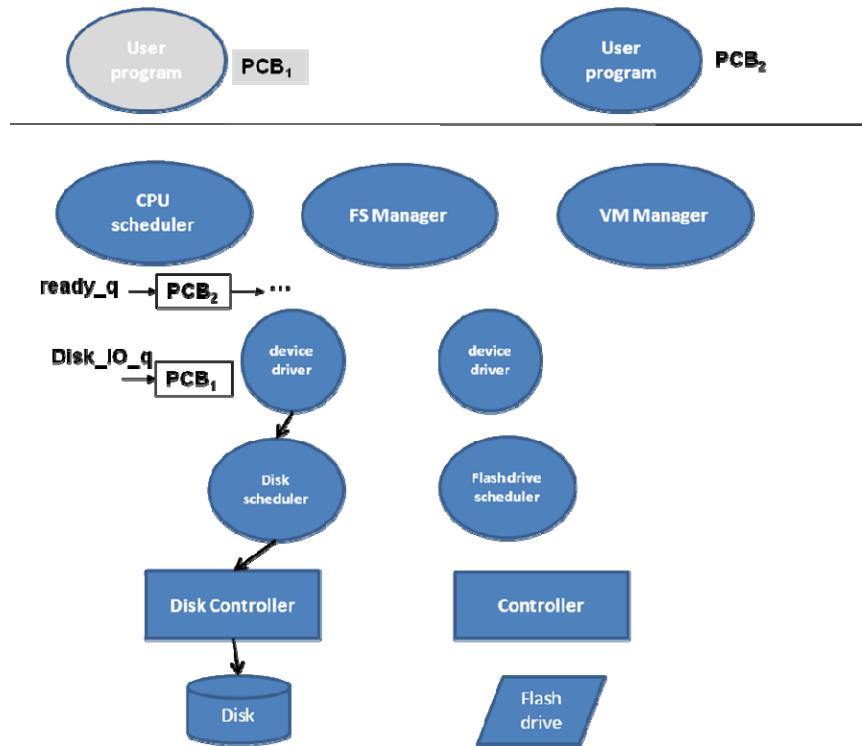


Figure 11.16-(b): Disk Driver handling the file system call from PCB₁

For example, the top layer of the FS manager passes the file I/O request to the device driver via the PCB of the process. This is straightforward and similar to a procedure call,

except that the procedure call is across layers of the software stack shown in Figure 11.15. This information flow is shown in Figure 11.16-(a). Let PCB_1 represent the PCB of the process that makes the file system call. Once the call has been handed over to the device driver, the process is no longer runnable and this is depicted in Figure 11.16-(b). PCB_1 is now in the $Disk_IO_q$ while being serviced by the device driver.

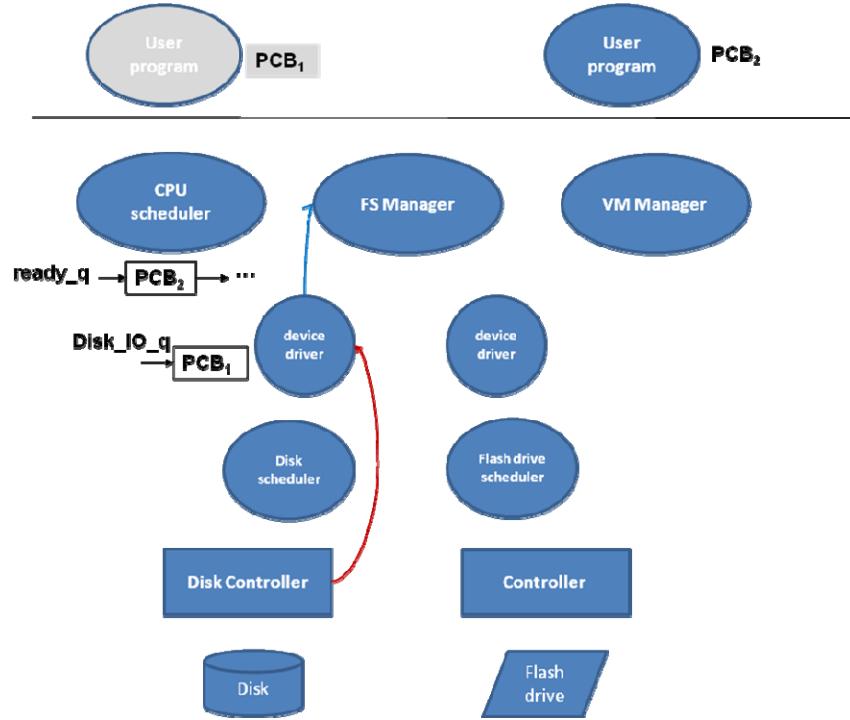


Figure 11.16-(c): A series of upcalls upon completion of the file I/O request (the red arrow is in hardware; the blue arrow is in software)

The device driver has to notify the upper layers of the system software stack upon completion of the I/O. In Chapter 6 (see Section 6.7.1.1), we introduced the concept of an *upcall*. This is essentially a mechanism for a lower layer in the system software stack to call a function in the upper layers. In fact, there is a continuum of upcalls as can be seen in Figure 11.16-(c). First, the disk controller makes an upcall in the form of an interrupt to the interrupt handler in the device controller. The interrupt handler knows exactly the process on whose behalf this I/O was initiated (from the $Disk_IO_q$). Of course, the interrupt handler in the device driver needs to know exactly who to call. For this reason, every upper layer in the system stack registers a handler with the lower layers for enabling such upcalls. Using this handler, the device driver makes an upcall to the FS manager to indicate completion of the file I/O request. One can see the similarity to this upcall mechanism and the way hardware interrupts are handled by the CPU (see Chapter 4). Due to the similarity both in terms of the function (asynchronously communicating events to the system) as well as the mechanism used to convey such events, upcalls are often referred to as *software interrupts*.

Upon receiving this upcall, the FS manager restores the PCB of the requesting process (PCB_1) back into the ready_q of the CPU scheduler as shown in Figure 11.16-(d). As can be seen, the interactions among the components of the operating system are enabled smoothly through the abstraction of the executing program in the form of a PCB. This is the power of the PCB abstraction.

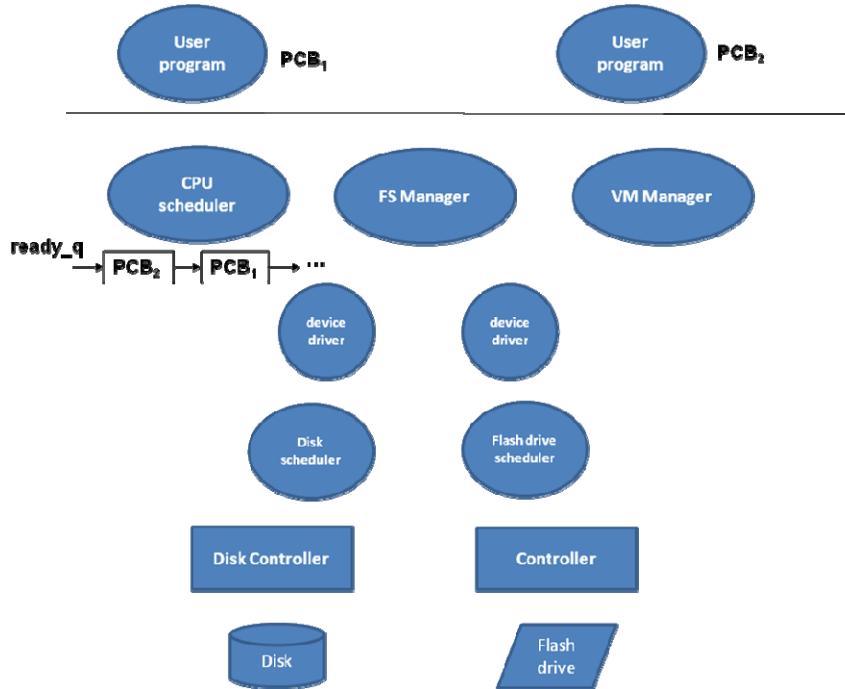


Figure 11.16-(d): FS Manager puts the process (PCB₁) that made the I/O call back in the CPU ready_q

A similar sequence of events would take place upon a page fault by a process; the only difference being that the VM Manager will be the initiator of the actions up and down the software stack shown in Figures 11.16-(a-d).

11.6 Layout of the file system on the physical media

Let us now understand how the operating system takes control of the resources in the system on power up.

In Chapter 10, we mentioned the basic idea behind booting an operating system. We mentioned how the BIOS performs the basic initialization of the devices and hands over control to the higher layers of the system software. Well actually, the process of booting up the operating system is a bit more involved. The image of the operating system is in the mass storage device. In fact, the BIOS does not even know what operating system needs to be booted up. Therefore, the BIOS has to have a clear idea as to where and how the information is kept in the mass storage device so that it can read in the operating system and hand over control to it. In other words, the layout of the information on the mass storage device becomes a *contract* between the BIOS and the operating system, be it Windows or Linux or whatever else.

To make this discussion concrete, let us assume that the mass storage device is a disk. At the very beginning of the storage space on the disk, say $\{platter\ 0, track\ 0, sector\ 0\}$, is a special record called the *Master Boot Record (MBR)*. Recall that when you create an executable through the compilation process it lives on the disk until loaded into memory by the loader. In the same manner, MBR is just a program at this well-defined location on the disk. BIOS is serving as a loader to load this program into memory and it will transfer control to MBR.

The MBR program knows the layout of the rest of the disk and knows the exact location of the operating system on the disk. The physical disk itself may be made up of several partitions. For example, on your desktop or laptop, you may have seen several “drives” with distinct names (Microsoft Windows gives them names such as C, D, etc.). These may be distinct physical drives but they may also be logical drives, each of which corresponds to a distinct partition on the same physical drive. Let us say you have a single disk but have a dual boot capability that allows either Linux or Windows to be your operating system depending on your choice. The file systems for each of these operating systems necessarily have to be distinct. This is where the partitions come into play.

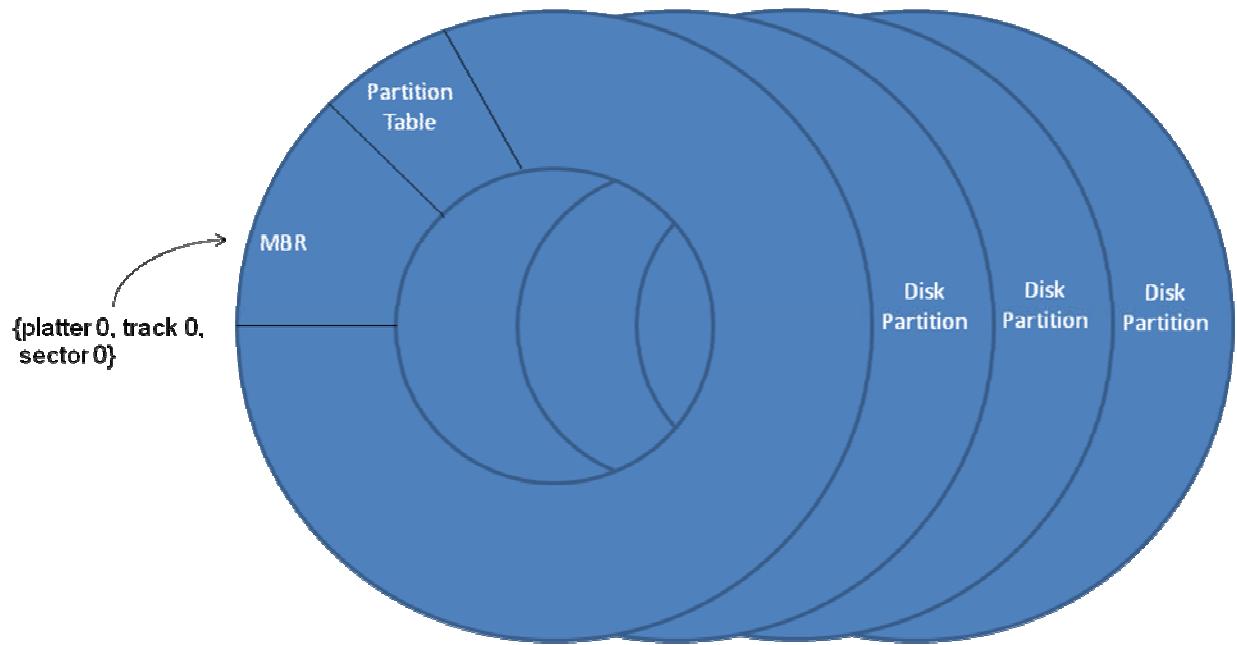


Figure 11.17: Conceptual layout of information on the disk

Figure 11.17 shows a conceptual layout of the disk. For the sake of clarity, we have shown each partition on a different disk platter. However, it should be emphasized that several partitions may live on the same platter depending on the disk capacity. The key data structure of the MBR program is the partition table. This table gives the start and end device address (i.e., the triple $\{platter, track, sector\}$) of each partition on the disk (Table 11.4). MBR uses this table to decide which partition has to be activated depending on the choice exercised by the user at boot time. Of course, in some systems

there may be no choice (e.g., there is a single partition or only one partition has an operating system associated with it).

Partition	Start address {platter, track, sector}	End address {platter, track, sector}	OS
1	{1, 10, 0}	{1, 600, 0}	Linux
2	{1, 601, 0}	{1, 2000, 0}	MS Vista
3	{1, 2001, 0}	{1, 5000, 0}	None
4	{2, 10, 0}	{2, 2000, 0}	None
5	{2, 2001, 0}	{2, 3000, 0}	None

Table 11.4: Partition Table Data Structure

Table 11.4 shows several partitions. Depending on the OS to be booted up, the MBR program will activate the appropriate partition (in this case either partition 1 or 2). Note that partitions 3-5 do not have any OS associated with it. They may simply be logical “drives” for one or the other operating system.

Figure 11.18 shows the layout of information in each partition. Other than the very first entry (boot block), the actual layout of information in the partition varies from file system to file system. However, to keep the discussion concrete, we will assume a particular layout and describe the functionality of each entry. The chosen layout of information is close to traditional Unix file systems.

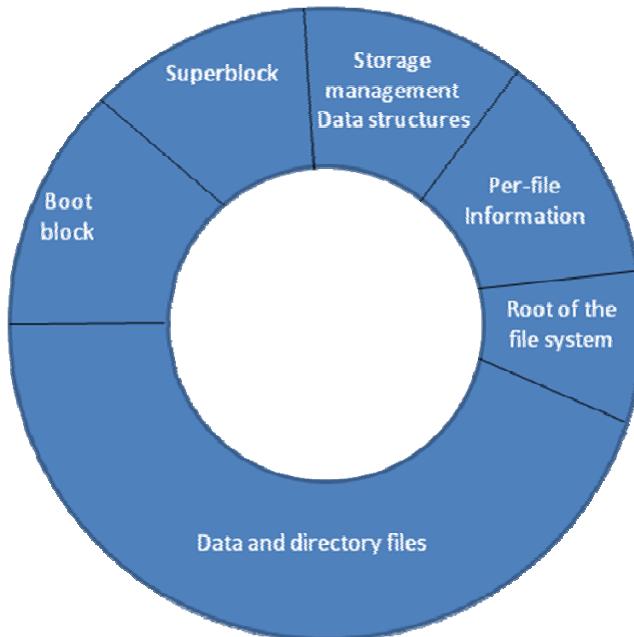


Figure 11.18: Layout of each partition

Let us review each entry in the partition.

- *Boot block* is the very first item in every partition. MBR reads in the boot block of the partition being activated. The boot block is simply a program (just like MBR) that is responsible for loading in the operating system associated with this partition. For uniformity, every partition starts with a boot block even if there is no OS associated with a particular partition (entries 3-5 in Table 11.4).
- *Superblock* contains all the information pertinent to the file system contained in this partition. This is the key to understanding the layout of the rest of this partition. The boot program (in addition to loading the operating system or in lieu thereof) reads in the superblock into memory. Typically, it contains a *code* usually referred to as the *magic number* that identifies the type of file system housed in this partition, the number of disk blocks, and other administrative information pertaining to the file system.
- The next entry in the partition contains the *data structures for storage management* in this partition. The data structures will be commensurate with the specific allocation strategy being employed by the file system (see Section 11.2). For example, it may contain a bit map to represent all the available disk data blocks (i.e., the free list). As another example, this entry may contain the FAT data structure that we mentioned earlier (see Section 11.2.4).
- The next entry in the partition corresponds to the *per-file information* maintained by the file system. This data structure is unique to the specifics of the file system. For example, as we mentioned earlier (see Section 11.3.1), in Unix file system, every file has a unique number associated with it called the i-node number. In such a system, this entry may simply be a collection of all the i-nodes in the file system.
- Modern day file systems are all hierarchical in nature. The next entry in the partition points to the *root directory* of the hierarchical tree-structured file system. For example, in Unix this would correspond to the information pertaining to the file named “/”.
- The last entry in the partition is just the collection of disk blocks that serve as containers for *data and directory files*. The data structures in the storage management entry of the partition are responsible for allocating and de-allocating these disk blocks. These disk blocks may be used to hold data (e.g., a JPEG image) or a directory containing other files (i.e., data files and/or other directories).

As should be evident, the superblock is a critical data structure of the file system. If it is corrupted for some reason then it is very difficult to recover the contents of the file system.

11.6.1 In memory data structures

For efficiency, a file system would read in the critical data structures (the superblock, free list of i-nodes, and free list of data blocks) from the disk at the time of booting. The file system manipulates these in-memory data structures as user programs create and delete files. In fact, the file system does not even write the data files created to the mass storage device immediately. The reason for the procrastination is two-fold. First, many files, especially in a program development environment, have a short lifetime (less than 30 seconds). Consider the lifetime of intermediate files created during a program

compilation and linking process. The programs (such as the compiler and linker) that create such files will delete them upon creation of the executable file. Thus, waiting for a while to write to the mass storage device is advantageous since such procrastination would help reduce the amount of I/O traffic. The second reason is simply a matter of convenience and efficiency. The file system may batch the I/O requests to reduce both the overhead of making individual requests as well as for dealing with interrupts from the device signaling I/O completion.

There is a downside to this tactic, however. The in-memory data structures may be inconsistent with their counterparts on the disk. One may have noticed the message on the computer screen that says, “It is not safe to remove the device,” when one tries to unplug a memory stick from the computer. This is because the OS has not committed the changes it has made to the in-memory data structures (and perhaps even the data itself) to the mass storage device. Many operating systems offer commands to help you force the issue. For example, the Unix command *sync* forces a write of all the in-memory buffers to the mass storage device.

11.7 Dealing with System Crashes

Let us understand what exactly is meant by an operating system crash. As we have said often, an operating system is simply a program. It is prone to bugs just as any other piece of software that you may write. A user program may be terminated by the operating system if it does something it should not do (try to access a portion of memory beyond its bounds), or may simply hang because it is expecting some event that may never occur. An operating system may have such bugs as well and could terminate abruptly. In addition, a power failure may force an abnormal termination of the operating system. This is what is referred to as a *system crash*.

The file system is a crucial component of the computer system. Since it houses persistent data, the health of the file system is paramount to productivity in any enterprise. Therefore, it is very important that the file system survive machine crashes. Operating systems take tremendous care to keep the integrity of the file system. If a system crashes unexpectedly, there may be inconsistencies between the in-memory data structures at the time of the crash and on-disk versions.

For this reason, as a last ditch effort, the operating system will dump the contents of the memory on to the mass storage device at a well-known location before terminating (be it due to a bug in the OS or due to power failure). On booting the system, one of the first things that the boot program will do is to see if such a crash image exists. If so, it will try to reconstruct the in-memory data structures and reconcile them with the on-disk versions. One of the reasons why your desktop takes time when you boot up your machine is due to the consistency check that the operating system performs to ensure file system integrity. In UNIX, the operating system automatically runs *file system consistency check (fsck)* on boot up. Only if the system passes the consistency check the boot process continues. Any enterprise does periodic backup of the disk on to tapes to ward off against failures.

11.8 File systems for other physical media

Thus far, we have assumed the physical media to be the disk. We know that a file system may be hosted on a variety of physical media. Let us understand to what extent some of the discussion we have had thus far changes if the physical media for the mass storage is different. A file system for a CD-ROM and CD-R (a recordable CD) are perhaps the simplest. Once recorded, the files can never be erased in such media, which significantly reduces the complexity of the file system. For example, in the former case, there is no question of a free list of space on the CD; while in the latter all the free space is at the end of the CD where the media may be appended with new files. File system for a CD-RW (rewritable CD) is a tad more complex in that the space of the deleted files need to be added to the free space on the media. File systems for DVD are similar.

As we mentioned in Chapter 10, solid-state drives (SSD) are competing with disk as mass storage media. Since SSD allows random access, seek time to disk blocks – a primary concern in disk-based file systems – is less of a concern in SSD-based file systems. Therefore, there is an opportunity to simplify the allocation strategies. However, there are considerations specific to SSD while implementing a file system. For example, a given area of an SSD (usually referred to as a *block*) may be written to only a finite number of times before it becomes unusable. This is due to the nature of the SSD technology. Consequently, file systems for SSD adopt a *wear leveling* allocation policy to ensure that all the areas of the storage are equally used over the lifetime of the SSD.

11.9 A summary of modern file systems

In this section, we will review some of the exciting new developments in the evolution of modern file systems. In Chapter 6, we gave a historical perspective of the evolution of Unix paving the way for both Mac OS X and Linux. Today, of course, OS X, Linux and Microsoft Windows rule the marketplace for a variety of application domains. We limit our discussion of modern file systems to Linux and Microsoft families of operating systems.

11.9.1 Linux

The file system interface (i.e., API) provided by Linux has not changed from the days of early Unix systems. However, internally the implementation of the file system has been undergoing enormous changes. This is akin to the architecture vs. implementation dichotomy that we discussed in Chapters 2 and 3 in the context of processor design.

Most of the evolutionary changes are to accommodate multiple file system partitions, longer file names, larger files, and hide the distinction between files present on local media Vs. the network.

One important change in the evolution of Unix file systems is the introduction of *Virtual File System (VFS)*. This abstraction allows multiple, potentially different, file systems to co-exist under the covers. The file system may be on a local device, an external device accessible through the network, and on different media types. VFS does not impact you as a user. You would open, read, or write a file just the same way as you would in a traditional Unix file system. Abstractions in VFS know exactly which file system is

responsible for your file and through one level of indirection (basically, function pointers stored in VFS abstraction layer) redirect your call to the specific file system that can service your request.

Now let us take a quick look at the evolution of file system implementation in Linux itself. In Chapter 6, we mentioned how MINIX served as a starting point for Linux. The very first file system for Linux used the MINIX file system. It had its limitation in the length of file names and maximum size of individual files. So this was quickly replaced by *ext* (which stands for extended file system) that circumvented those limitations. However, ext had several performance inefficiencies giving rise the *ext2*, a second version of ext, which is still in widespread use in the Linux community.

11.9.1.1 ext2

The disk layout of an ext2 file system partition is not remarkably different from the generic description we gave in Section 11.6. A few things are useful to mention regarding ext2.

- **Directory file**

The first is the contents of an i-node structure for a directory file. A directory file is made up of an integral number of contiguous disk blocks to facilitate writing the directory file to the disk in one I/O operation. Each entry in the directory names a file or another sub-directory. Since a name can be of arbitrary length the size of each entry is variable. Figure 11.19-(a) shows the format of an entry in the directory.

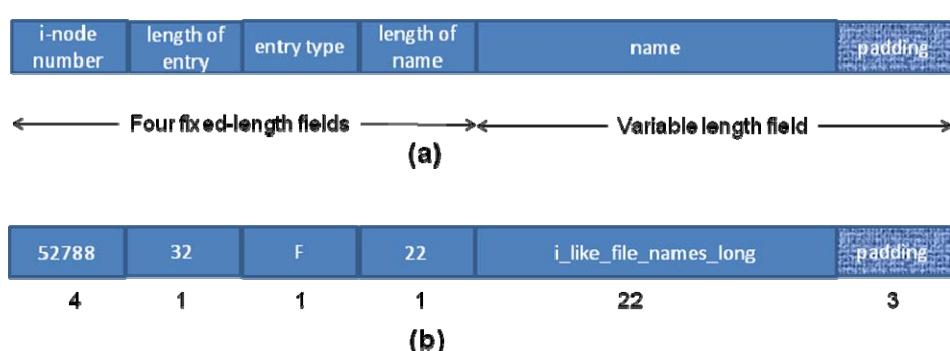


Figure 11.19: Format of each individual entry in a directory file

Each entry is composed of:

- **i-node number:** a fixed length field that gives the i-node number for the associated entry
- **length of entry:** a fixed length field that gives the length of the entry in bytes (i.e., the total amount of space occupied by the entry on the disk); this is needed to know where in the directory structure the next entry begins
- **type of entry:** a fixed length field that specifies if the entry is a data file (f) or a directory file (d)
- **length of name:** a fixed length field that specifies the length of the file name
- **name:** a variable length field that gives the name of the file in ASCII

- **padding:** an optional variable length field perhaps to make the total size of an entry some multiple of a binary power; as we will see shortly, such padding space may also be created or expanded due to file deletions

Figure 11.19-(b) shows an example entry for a file named “`i_like_my_file_names_long`” with the values filled in. The size of each field in bytes is shown below the figure.

Entries in the directory are laid out contiguously in the textual order of creation of the files. For example, if you create the files “`datafile`”, “`another_datafile`”, and a “`my_subdirectory`”, in that order in a directory; the first two being data files, and the third a directory file. The directory entries will be as shown in Figure 11.20-(a). Let us say, you delete one of the files, say, “`another_datafile`”. In this case, there will simply be an empty space in the directory layout (see Figure 11.20-(b)). Basically, the space occupied by the deleted file entry will become part of the padding of the previous entry. This space can be used for the creation of a new file entry in the directory.

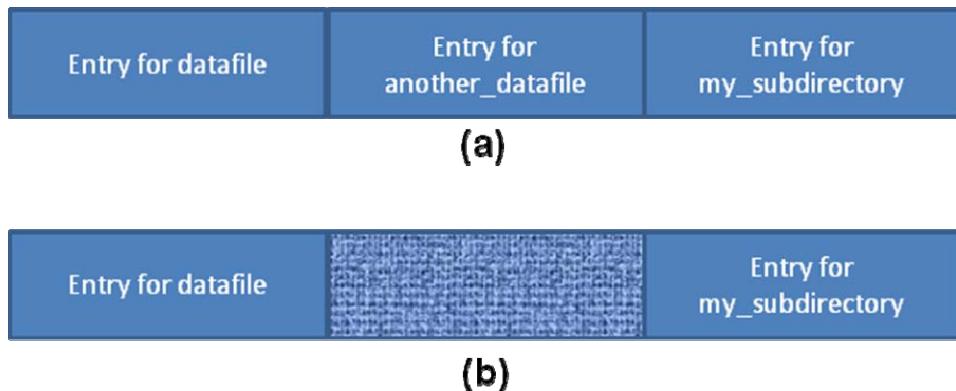


Figure 11.20: Layout of multiple entries in a directory file

- **Data file**

The second point of interest is the i-node structure for a data file. As we said earlier, ext2 removes the limitation of MINIX with respect to the maximum size of files. The i-node structure for a data file reflects this enhancement. The scheme used is the hybrid allocation scheme we discussed earlier in Section 11.2.7. An i-node for a data file contains the following fields:

- **12 data block addresses:** The first 12 blocks of a data file can be directly reached from the i-node. This is very convenient for small files. The file system will also try to allocate these 12 data blocks contiguously so that the performance can be optimized.
- **1 single indirect pointer:** This points to a disk block that serves as a container for pointers to data blocks. For example, with a disk block size of 512 bytes, and a data block pointer size of 4 bytes, this first level indirection allows expanding the file size by an additional 128 data blocks.
- **1 double indirect pointer:** This points to a disk block that serves as a container for pointers to tables that contain single level indirect pointers. Continuing the same example, this second level indirection allows expanding the file size by an additional 128×128 (i.e., 2^{14}) data blocks.

- **1 triple indirect pointer:** This adds one more level of indirection over the double indirect pointer, allowing a file to be expanded by an additional $128*128*128$ (i.e., 2^{21}) data blocks.

Figure 11.21 pictorially shows such an i-node with all the important fields filled in.

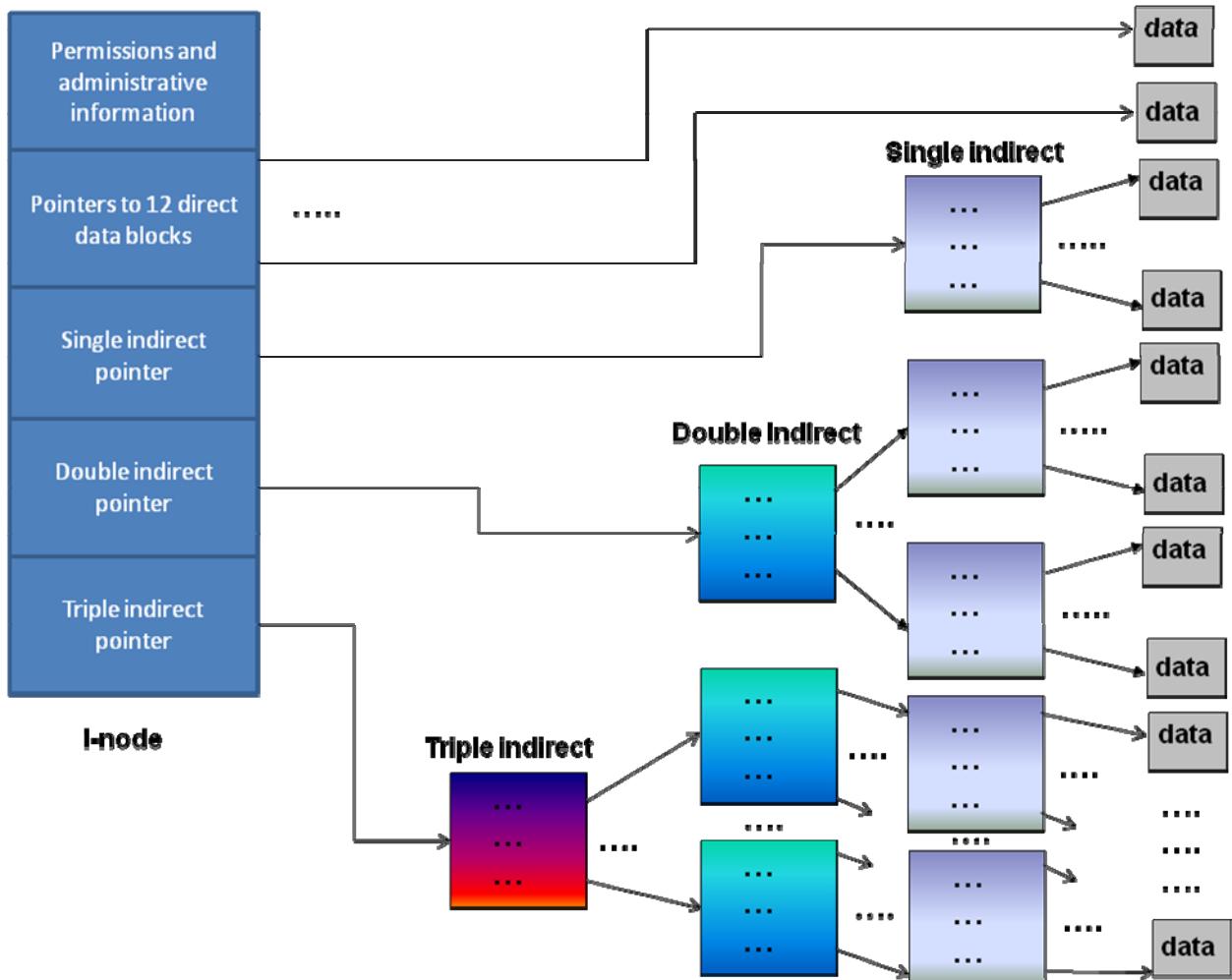


Figure 11.21: i-node structure of a data file in Linux ext2

Example 8:

What is the maximum file size in bytes possible in the ext2 system? Assume a disk block is 1 KB and pointers to disk blocks are 4 bytes.

Answer:

Direct data blocks

$$\begin{aligned}
 \text{Number of direct data blocks} &= \text{number of direct pointers in an i-node} \\
 &= 12 \quad (1)
 \end{aligned}$$

Data blocks via the single indirect pointer

Next, let us count the number of data blocks available to a file via the single indirect pointer in an i-node. The single indirect pointer points to a single indirect table that contains pointers to data blocks.

Number of data blocks via the single indirect pointer in an i-node

$$\begin{aligned}
 &= \text{Number of pointers in a single indirect table} \\
 &= \text{Number of pointers in a disk block} \\
 &= \text{size of disk block/size of pointer} \\
 &= 1\text{KB}/4\text{bytes} \\
 &= 256
 \end{aligned} \tag{2}$$

Data blocks via double indirect pointer

Next, let us count the number of data blocks available to a file via the double indirect pointer in an i-node. The double indirect pointer points to a disk block that contains pointers to tables of single indirect pointers.

Number of single indirect tables via the double indirect pointer

$$\begin{aligned}
 &= \text{Number of pointers in a disk block} \\
 &= \text{size of disk block/size of pointer} \\
 &= 1\text{KB}/4\text{bytes} \\
 &= 256
 \end{aligned} \tag{3}$$

Number of data blocks via each of these single indirect tables (from equation (2) above)

$$= 256 \tag{4}$$

From (3) and (4), the number of data blocks via the double indirect pointer in an i-node

$$= 256*256 \tag{5}$$

Data blocks via triple indirect pointer

By similar analysis the number of data blocks via the triple indirect pointer in an i-node

$$= 256*256*256 \tag{6}$$

Putting (1), (2), (5), and (6) together, the total number of disk blocks available to a file

$$\begin{aligned}
 &= 12 + 256 + 256*256 + 256*256*256 \\
 &= 12 + 2^8 + 2^{16} + 2^{24}
 \end{aligned}$$

The maximum file size in bytes for a data file (given the data block is of size 1 KB)

$$\begin{aligned}
 &= (12 + 2^8 + 2^{16} + 2^{24}) \text{ KB} \\
 &> \mathbf{16 \text{ GBytes}}
 \end{aligned}$$

11.9.1.2 Journaling file systems

Most modern file systems for Linux are *journaling* file systems. Normally, you think of writing to a file as writing to the data block of the file. Logically, this is correct.

However, practically there are issues with this approach. For example, if the file size is

too small then there is both space and more importantly time overhead in writing such small files to the disk. Of course, operating systems buffer writes to files in memory and write to disk opportunistically. Despite such optimizations, ultimately, these files have to be written to disk. Small files not only waste space on the disk (due to internal fragmentation) but also result in time overhead (seek time, rotational latency, and metadata management).

Another complication is system crashes. If the system crashes in the middle of writing to a file, the file system will be left in an inconsistent state. We alluded to this problem already (see Section 11.7).

To address both the problem of overcoming the inefficiency of writing small files and to aid in the recovery from system crashes, modern file systems use a journaling approach. The idea is simple and intuitive and uses the metaphor of keeping a personal journal or a diary. One's personal diary is a time-ordered record of events of importance in one's everyday activities. The *journal* serves the same purpose for the file system. Upon file writes, instead of writing to the file and/or creating a small file, a log record (similar to database record) that would contain the information corresponding to the file write (meta-data modifications to i-node and superblock, as well as modifications to the data blocks of the file). Thus, the journal is a *time-ordered* record of all the changes to the file system. The nice thing about this approach is that the journal is distinct from the file system itself, and allows the operating system to optimize its implementation to best suit the file system needs. For example, the journal may be implemented as a sequential data structure. Every entry in this sequential data structure represents a particular file system operation.

For example, let us say you modified specific data blocks three files (X, Y, and Z) that are scattered all over the disk in that order. The journal entries corresponding to these changes will appear as shown in Figure 11.22. Note that each log record may of different size depending on the number of data blocks of a file that is modified at a time by the corresponding write operation.

File name = X Data block = d_x New Contents = c_x	File name = Y Data block = d_y New Contents = c_y	File name = Z Data block = d_z New Contents = c_z
Log record for changes to file X	Log record for changes to file Y	Log record for changes to file Z

Figure 11.22: Journal entries of changes to the file system

The journal data structure is a composite of several *log segments*. Each log segment is of finite size (say for example, 1 MByte). As soon as a log segment fills up, the file system writes this out to a contiguous portion of the disk and starts a new log segment for the subsequent writes to the file system. Note that the file X, Y, and Z do not reflect the changes made to them yet. Every once in a while, the file system may *commit* the changes to the actual files by reading the log segment (in the order in which they were generated) and applying the log entries to the corresponding files. Once the changes have

been committed then the log segments may be discarded. If the file X is read before the changes have been applied to it, the file system is smart enough to apply the changes from the log segment to the file before allowing the read to proceed.

As should be evident, the log segment serves to aggregate small writes (to either small files, or small parts of a large file) into coarser-grain operations (i.e., larger writes to contiguous portion of the disk) to the journal.

Journaling overcomes the small write problem. As a bonus journaling also helps coping with system crashes. Remember that as a last ditch effort (see Section 11.7), the operating system stores everything in memory to the disk at the time of the crash or power failure. Upon restart, the operating system will recover the in-memory log segments. The file system will recognize that the changes in the log segments (both the ones on the disk as well as the in-memory log segments that were recovered from the crash) were not committed successfully. It will simply reapply the log records to the file system to make the file system consistent.

Ext3 is the next iteration of the Linux file system. Compared the ext2, the main extension in the ext3 file system is support for journaling. In this sense, a file partition built using ext2 may be accessed using ext3 file system since the data structures and internal abstractions are identical in both. Since creating a journal of all file system operations may be expensive in terms of space and time, ext3 may be configured to journal only the changes to the meta-data (i.e., i-nodes, superblock, etc.). This optimization will lead to better performance in the absence of system crashes but cannot make guarantees about file data corruption due to crashes.

There are a number of new Unix file systems. *ReiserFS* is another file system with metadata only journaling for Linux. *jFS* is a journaling file system from IBM for use with IBM's AIX (a version of Unix) and with Linux. *xFS* is a journaling file system primarily designed for SGI Irix operating system. *zFS* is a high performance file system for high-end enterprise systems built for the Sun Solaris operating system. Discussion of such file systems is beyond the scope of this textbook.

11.9.2 Microsoft Windows

Microsoft file systems have an interesting history. As one may recall, MS-DOS started out as an operating system for PC. Since disks in the early stages of PC evolution had fairly small capacity (on the order of 20-40 MBytes), the file systems for such computers were limited as well. For example, FAT-16 (a specific instance of the allocation scheme discussed in Section 11.2.4) uses 16-bit disk addresses, with a limit of 2Gbytes per file partition. FAT-32, by virtue of using 32-bit disk addresses, extends the limit of a file partition to 2 TBytes. These two file systems have mostly been phased out with the advent of Microsoft XP and Vista operating systems. Currently (circa 2008), NT file system, which was first introduced with the Microsoft NT operating system, is the *de facto* standard Microsoft file system. FAT-16 may still be in use for removable media such as floppy disk. FAT-32 also still finds use, especially for inter-operability with older versions of Windows such as 95/98.

NTFS, as the file system is referred to, supports most of the features that we discussed in the context of Unix file systems earlier. It is a complex modern file system using 64-bit disk address, and hence can support very large disk partitions. *Volume* is the fundamental unit of structuring the file system. A volume may occupy a part of the disk, the whole disk, or even multiple disks.

API and system features. A fundamental difference between NTFS and Unix is the view of a file. In NTFS, a file is an *object* composed of *typed attributes* rather than a stream of bytes in Unix. This view of a file offers some significant flexibility at the user level. Each typed attribute is an independent byte stream. It is possible to create, read, write, and/or delete each attributed part without affecting the other parts of the same file. Some attribute types are standard for all files (e.g., name, creation time, and access control). As an example, you may have an image file with multiple attributes: raw image, thumbnail, etc. Attributes may be created and deleted at will for a file.

NTFS supports long file names (up to 255 characters in length) using Unicode character encoding to allow non-English file names. It is a hierarchical file system similar to Unix, although the hierarchy separator in Unix, namely, “/”, is replaced by “\” in NTFS. Aliasing a file through hard and soft links, which we discussed in Section 11.1, is a fairly recent addition to the NTFS file system.

Some of the features of NTFS that are interesting include on the fly compression and decompression of files as they are written and read, respectively; an optional encryption feature; and support for small writes and system crashes via journaling.

Implementation. Similar to the i-node table in Unix (see Section 11.3.1), the main data structure in NTFS is the *Master File Table (MFT)*. This is also stored on the disk and contains important meta-data for the rest of the file system. A file is represented by one or more records in the MFT depending on both the number of attributes as well as the size of the file. A bit map specifies which MFT records are free. The collection of MFT records describing a file is similar in functionality to an i-node in the Unix world but the similarity stops there. An MFT record (or records) for a file contains the following attributes:

- File name
- Timestamp
- Security information (for access control to the file)
- Data or pointers to disk blocks containing data
- An optional pointer to other MFT records if the file size is too large to be contained in one record or if the file has multiple attributes all of which do not fit in one MFT record

Each file in NTFS has a unique ID called an *Object reference*, a 64-bit quantity. The ID is an index into the MFT for this file and serves a similar function to the i-node number in Unix.

The storage allocation scheme tries to allocate contiguous disk blocks to the data blocks of the file to the extent possible. For this reason, the storage allocation scheme maintains available disk blocks in *clusters*, where each cluster represents a number of contiguous disk blocks. The cluster size is a parameter chosen at the time of formatting the drive. Of course, it may not always be possible to allocate all the data blocks of a file to contiguous disk blocks. For example, consider a file that has 13 data blocks. Let us say that the cluster size is 4 blocks, and the free list contains clusters starting at disk block addresses 64, 256, 260, 408. In this case, there will be 3 non-contiguous allocations made for this file as shown in Figure 11.23.

File name and other standard attributes	File size = 13	Cluster address = 64 Size = 4	Cluster address = 256 Size = 8	Cluster address = 408 Size = 1
---	----------------	----------------------------------	-----------------------------------	-----------------------------------

Figure 11.23: An MFT record for a file with 13 data blocks

Note that internal fragmentation is possible with this allocation since the remaining 3 disk blocks in cluster 408 are unused.

An MFT record is usually 1Kbytes to 4 Kbytes. Thus, if the file size is small then the data for the file is contained in the MFT record itself, thereby solving the small write problem. Further, the clustered disk allocation ensures good sequential access to files.

11.10 Summary

In this chapter, we studied perhaps one of the most important components of your system, namely, the file system. The coverage included

- attributes associated with a file,
- allocation strategies and associated data structure for storage management on the disk,
- meta-data managed by the file system,
- implementation details of a file system,
- interaction among the various subsystems of the operating system,
- layout of the files on the disk,
- data structures of the file system and their efficient management,
- dealing with system crashes, and
- file system for other physical media

We also studied some modern file systems in use circa 2008.

File systems is a fertile of research and development. One can see an analogy similar to processor architecture vs. implementation in the context of file systems. While the file system API remains mostly unchanged across versions of operating systems (be they Unix, Microsoft, or anything else), the implementation of the file system continually evolves. Some of the changes are due to the evolution in the disk technology, and some are due to the changes in the types of files used in applications. For example, modern workloads generate multimedia files (audio, graphics, photos, video, etc.) and the file system implementation has to adapt to supporting such diverse file types efficiently.

In this chapter, we have only scratched the surface of the intricacies involved in implementing a file system. We hope that we have stimulated your interest enough to get you to take a more advanced course in operating systems.

11.11 Review Questions

1. Where can the attribute data for a file be stored? Discuss the pros and cons of each choice.
2. Examine the following directory entry in Unix:

```
-rwxrwxrwx 3 rama      0 Apr 27 21:01 foo
```

After executing the following commands:

```
chmod u-w foo  
chmod g-w foo
```

What are the access rights of the file "foo"?

3. Linked allocation results in
 - Good sequential access
 - Good random access
 - Ability to grow the file easily
 - Poor disk utilization
 - Good disk utilization

Select all that apply

4. Fixed contiguous allocation of disk space results in
 - Good sequential access
 - Good random access
 - Ability to grow the file easily
 - Poor disk utilization
 - Good disk utilization

Select all that apply.

5. Given the following:

Number of cylinders on the disk	=	6000
Number of platters	=	3
Number of surfaces per platter	=	2
Number of sectors per track	=	400
Number of bytes per sector	=	512
Disk allocation policy	=	contiguous cylinders

- (a) How many cylinders should be allocated to store a file of size 1 Gbyte?
 (b) How much is the internal fragmentation caused by this allocation?
6. What are the problems with FAT allocation policy?
7. Compare linked allocation policy with FAT.
8. How does indexed allocation overcome the problem of FAT and linked allocation?
9. Consider an indexed allocation scheme on a disk
 - The disk has 3 platters (2 surfaces per platter)
 - There are 4000 tracks in each surface
 - Each track has 400 sectors
 - There are 512 bytes per sector
 - Each i-node is a fixed size data structure occupying one sector.
 - A data block (unit of allocation) is a contiguous set of 4 cylinders
 - A pointer to a disk data block is represented using an 8-byte data structure.
 a) What is the minimum amount of space used for a file on this system?
 b) What is the maximum file size with this allocation scheme?
10. Given the following for a hybrid allocation scheme:
 - size of index block = 256 bytes
 - size of data block = 8 Kbytes
 - size of disk block pointer = 8 bytes (to either index or data block)
 - An i-node consists of 2 direct block pointers, 1 single indirect pointer, 1 double indirect pointer, and 1 triple indirect pointer.
 (a) What is the maximum size (in bytes) of a file that can be stored in this file system?
 (b) How many data blocks are needed for storing a data file of 1 Gbytes?
 (c) How many index blocks are needed for storing the same data file of size 1 Gbytes?
11. What is the difference between hard and soft links?
12. Consider:


```

touch f1          /* create a file f1 */
ln -s f1 f2      /* sym link */
ln -s f2 f3
ln f1 f4          /* hard link */
ln f4 f5
    
```

 (a) How many i-nodes will be created by the above set of commands?
 (b) What is the reference count on each node thus created?

13. For ext2 file system with a disk block size of 8 KB and a pointer to disk blocks of 4 bytes, what is the largest file that can be stored on the system? Sketch the structure of an i-node and show the calculations to arrive at the maximum file size (see Example 8).

Chapter 12 Multithreaded Programming and Multiprocessors **(Revision number 16)**

Multithreading is a technique for a program to do multiple tasks, perhaps concurrently. Since the early stages of computing, exploiting parallelism has been a quest of computer scientists. As early as the 70's, languages such as *Concurrent Pascal* and *Ada* have proposed features for expressing program-level concurrency. Humans think and do things in parallel all the time. For example, you may be reading this book while you are listening to some favorite music in the background. Often, we may be having an intense conversation with someone on some important topic, while working on something with our hands, maybe fixing a car or folding our laundry. Given that computers extend the human capacity to compute, it is only natural to provide the opportunity for the human to express concurrency in the tasks that they want the computer to do on their behalf. Sequential programming forces us to express our computing needs in a sequential manner. This is unfortunate since humans think in parallel but end up coding up their thoughts sequentially. For example, let us consider an application such as video surveillance. We want the computer to gather images from ten different cameras continuously, analyze each of them individually for any suspicious activity, and raise an alarm in case of a threat. There is nothing sequential in this description. If anything, the opposite is true. Yet if we want to write a computer program in C to carry out this task, we end up coding it sequentially.

The intent of this chapter is to introduce concepts in developing multithreaded programs, the operating system support needed for these concepts, and the architectural support needed to realize the operating system mechanisms. It is imperative to learn parallel programming and the related system issues since single chip processors today incorporate multiple processor cores. Therefore, parallel processing is becoming the norm these days from low-end to high-end computing systems. The important point we want to convey in this chapter is that the threading concept and the system support for threading are simple and straightforward.

12.1 Why Multithreading?

Multithreading allows expressing the inherent parallelism in the algorithm. A *thread* is similar to a process in that it represents an active unit of processing. Later in this chapter, we will discuss the semantic differences between a thread and a process. Suffice it to say at this point that a user level process may comprise multiple threads.

Let us understand how multithreading helps at the programming level. First, it allows the user program to express its intent for concurrent activities in a modular fashion, just as the procedure abstraction helps organize a sequential program in a modular fashion. Second, it helps in overlapping computation with inherent I/O activities in the program. We know from Chapter 10, that DMA allows a high-speed I/O device to communicate directly with the memory without the intervention of the processor. Figure 12.1 shows this situation for a program that periodically does I/O but does not need the result of the

I/O immediately. Expressing the I/O activity as a separate thread helps in overlapping computation with communication.

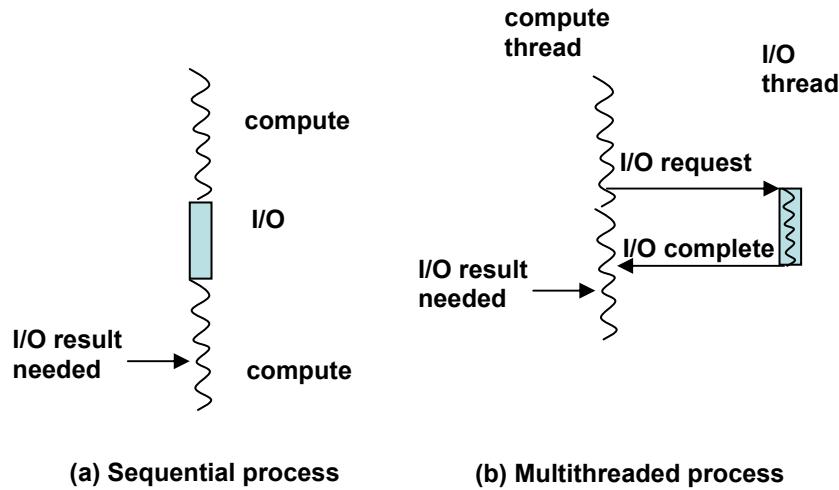


Figure 12.1: Overlapping Computation with I/O using threads

Next, let us see how multithreading helps at the system level. It is common to have multiple processors in the same computer or even in a single chip these days. This is another important reason for multithreading since any expression of concurrency at the user level can serve to exploit the inherent hardware parallelism that may exist in the computer. Imagine what the program does in a video surveillance application. Figure 12.2 shows the processing associated with a single stream of video in such an application. The digitizer component of the program continually converts the video into a sequence of frames of pixels. The tracker component analyzes each frame for any content that needs flagging. The alarm component takes any control action based on the tracking. The application pipeline resembles the processor pipeline, albeit at a much coarser level. Thus if we have multiple processors in the computer that work autonomously, then they could be executing the program components in parallel leading to increased performance for the application.

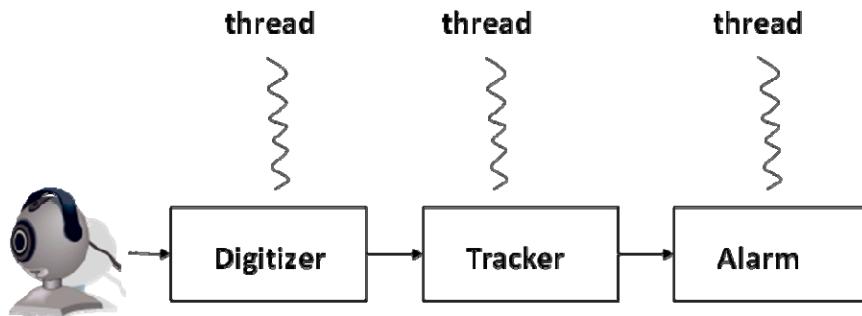


Figure 12.2: Video processing pipeline

Thus, multithreading is attractive from the point of view of program modularity, opportunity for overlapping computation with I/O, and the potential for increased performance due to parallel processing.

We will use the application shown in Figure 12.2 as a running example to illustrate the programming concepts we develop for multithreading in this chapter.

12.2 Programming support for threads

Now that we appreciate the utility of threads as a way of expressing concurrency, let us understand what it takes to support threads as a programming abstraction. We want to be able to dynamically *create* threads, *terminate* threads, *communicate* among the threads, and *synchronize* the activities of the threads,

Just as the system provides a math library for common functions that programmers may need, the system provides a library of functions to support the threads abstraction. We will explore the facilities provided by such a library in the next few subsections. In the discussion, it should be understood that the syntax used for data types and library functions are for illustration purposes only. The actual syntax and supported data types may be different in different implementations of thread libraries.

12.2.1 Thread creation and termination

A thread executes some program component. Consider the relationship between a process and the program. A process starts executing a program at the *entry* point of the program, the *main* procedure in the case of a C program. In contrast, we want to be able to express concurrency, using threads, at *any* point in the program *dynamically*. This suggests that the entry point of a thread is *any user-defined procedure*. We define *top-level* procedure as a procedure name that is visible (through the visibility rules of the programming language) wherever such a procedure has to be used as a target of thread creation. The top-level procedure may take a number of arguments to be passed in.

Thus, a typical thread creation call may look as shown below:

```
thread_create (top-level procedure, args);
```

Essentially, the thread creation call names the top-level procedure and passes the arguments (packaged as args) to initiate the execution of a thread in that procedure. From the user program's perspective, this call results in the creation of a *unit of execution* (namely a *thread*), concurrent with the current thread that made the call (Figure 12.3 before/after picture).

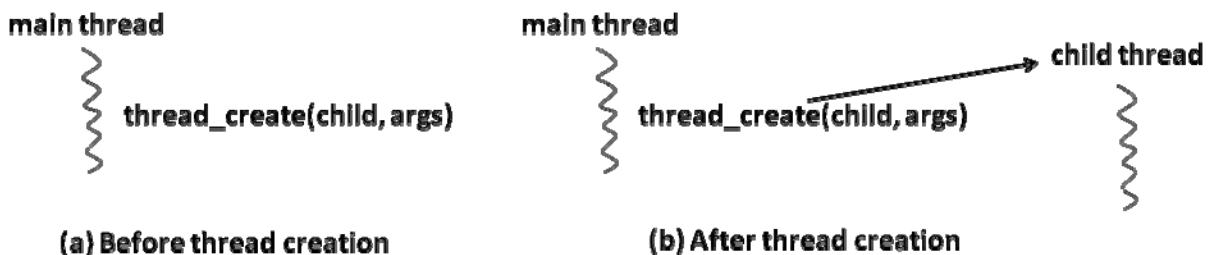


Figure 12.3: Thread creation

Thus, a **`thread_create`** function *instantiates* a new and independent entity called a *thread* that has a life of its own.

This is analogous to parents giving birth to children. Once a child is born, he or she roams around the house doing his/her own thing (within the limits set by the parents) independent of the parent. This is exactly what happens with a thread. Recall that a program in execution is a process. In a sequential program there is only one *thread of control*, namely, the process. By a *thread of control*, we mean an active entity that is roaming around the memory footprint of the program carrying out the intent of the programmer. Now that it has a life of its own, a thread (within the limits set by the parent process) can do its own thing. For example, in Figure 12.3, once created, “child” can make its own procedure calls programmed into the body of its code independent of what “main” is doing in its code body past the thread creation point.

Let us understand the limits set for the child thread. In the human analogy, a parent may place a gate near the stairs to prevent the child from going up the stairs, or child proof cabinets and so on. For a thread, there are similar limits imposed by both the programming language and the operating system. A thread starts executing in a top-level procedure. In other words, the starting point for a threaded program is a normal sequential program. Therefore, the visibility and scoping rules of the programming language applies to constrain the data structures that a thread can manipulate during its lifetime. The operating system creates a unique and distinct memory footprint for the program called an *address space*, when it instantiates the program as a process. As we saw in earlier chapters, the address space contains the code, global data, stack, and heap sections specific to each program. A process plays in this “sandbox”. This is exactly the same “sandbox” for children of this process to play in as well.

The reader may be puzzled as to the difference between a “process” and a “thread”. We will elaborate on this difference more when we get to the section on operating system support for threads (Section 12.6). Suffice it to say at this point that the amount of state that is associated with a process is much more than that with a thread. On the other hand, a thread shares the parent process’s address space, and in general has lesser state information associated with it than a process. This makes a process a heavier-weight entity compared to a thread. However, both are independent threads of control within the process’s address space and have lives of their own.

One fundamental difference between a process and thread is memory protection. The operating system turns each program into a process, each with its own address space that acts as a wall around each process. However, threads execute within a single address space. Thus, they are not protected from each other. In human terms, one cannot walk into a neighbor’s house and start scribbling on the walls. However, one’s children (if not supervised) can happily go about scribbling on the walls of the house with their crayons. They can also get into fistfights with one another. We will see shortly how we can enforce some discipline among the threads to maintain a sense of decorum while executing within the same address space.

A thread automatically terminates when it exits the top-level procedure that it started in. Additionally, the library may provide an explicit call for terminating a thread in the same process:

```
thread_terminate (tid);
```

Where, **tid** is the system-supplied identifier of the thread we wish to terminate.

Example 1:

Show the code fragment to instantiate the digitizer and tracker parts of the application shown in Figure 12.2

Answer:

```
digitizer()
{
    /* code for grabbing images from camera
     * and share the images with the tracker
     */
}

tracker()
{
    /* code for getting images produced by the digitizer
     * and analyzing an image
     */
}

main()
{
    /* thread ids */
    thread_type digitizer_tid, tracker_tid;

    /* create digitizer thread */
    digitizer_tid = thread_create(digitizer, NULL);

    /* create tracker thread */
    tracker_tid = thread_create(tracker, NULL);

    /* rest of the code of main including
     * termination conditions of the program
     */
}
```

Figure 12.4: Code fragment for thread creation

Note that the shaded box is all that is needed to create the required structure.

12.2.2 Communication among threads

Threads may need to share data. For example, the digitizer in Figure 12.2 shares the buffer of frames that it creates with the tracker.

Let us see how the system facilitates this sharing. As it turns out, this is straightforward. We already mentioned that a threaded program is created out of a sequential program by turning the top-level procedures into threads. Therefore, the data structures that are visible to multiple threads (i.e., top-level procedures) within the scoping rules of the original program become shared data structures for the threads. Concretely, in a programming language such as C, the global data structures become shared data structures for the threads.

Of course, if the computer is a multiprocessor, there are system issues (at both the operating system and hardware levels) that need to be dealt with to facilitate this sharing. We will revisit these issues in a later section (Section 12.8).

Example 2:

Show the data structure definition for the digitizer and tracker threads of Figure 12.2 to share images.

Answer:

```
#define MAX 100          /* maximum number of images */

image_type frame_buf[MAX];  /* data structure for
                           * sharing images between
                           * digitizer and tracker
                           */

digitizer()
{
    loop {
        /* code for putting images into frame_buf */
    }
}

tracker()
{
    loop {
        /* code for getting images from frame_buf
         * and analyzing them
         */
    }
}
```

Note: The shaded box shows the data structure created at the global level to allow both the tracker and digitizer threads to share.

12.2.3 Read-write conflict, Race condition, and Non-determinism

In a sequential program, we never had to worry about the integrity of data structures since there are no concurrent activities in the program by definition. However, with multiple threads working concurrently within a single address space it becomes essential to ensure that the threads do not step on one another. We define *read-write conflict* as a condition in which multiple concurrent threads are simultaneously trying to access a shared variable with at least one of the threads trying to write to the shared variable. A *race* condition is defined as the situation wherein a read-write conflict exists in a program without an intervening synchronization operation separating the conflict. Race conditions in a program may be intended or unintended. For example, if a shared variable is used for synchronization among threads, then there will be a race condition. However, such a race condition is an intended one.

Consider the following code fragment:

Scenario #1:

```
int flag = 0; /* shared variable initialized to zero */

Thread 1                                Thread 2

while (flag == 0) {
    /* do nothing */
}
.
.
.
if (flag == 0) flag = 1;
.
.
```

Threads 1 and 2 are part of the same process. Per definition, threads 1 and 2 are in a read-write conflict. Thread 1 is in a loop continuously reading shared variable flag, while thread 2 is writing to it in the course of its execution. On the surface, this setup might appear to be a problem, since there is a race between the two threads. However, this is an intentional race (sometimes referred to as *synchronization race*), wherein thread 1 is awaiting the value of flag to change by thread 2. Thus, a race condition does not always imply that the code is erroneous.

Next, we will define a particular kind of race condition that could lead to erroneous program behavior. A *data race* is defined as a read-write conflict without an intervening synchronization operation wherein the variable being accessed by the threads is *not* a synchronization variable. That is, a data race occurs when there is unsynchronized access to arbitrary shared variables in a parallel program.

Consider, the following code fragment:

Scenario #2:

```
int count = 0; /* shared variable initialized to zero */

Thread 1 (T1)           Thread 2 (T2)           Thread 3 (T3)
.
.
.
count = count+1;    count = count+1;    count = count+1;
.
.
.

Thread 4 (T4)
.
.
.

printf("count = %d\n", count);
```

There is a data race (for the variable **count**) among all the four threads. What value will thread 4 print? Each of the threads 1, 2, and 3, are adding 1 to the current value of count. However, what is the current value of count seen by each of these threads? Depending on the order of execution of the increment statement (**count = count+1**), the printf statement in thread 4 will result in different values being printed.

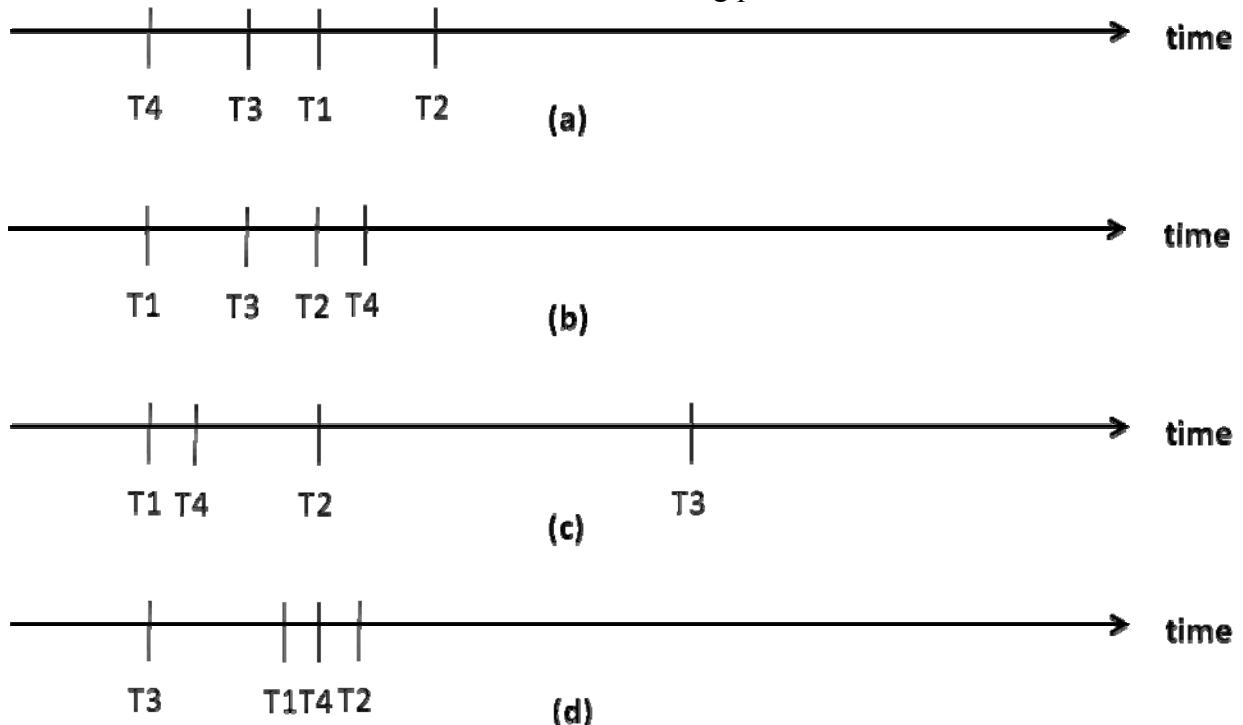


Figure 12.5: Examples of possible executions of Scenario 2 on a uniprocessor using non-preemptive scheduling of threads

This is illustrated in Figure 12.5 wherein four different order of execution are captured just to illustrate the point.

The first question that should arise in the mind of the reader is why there are many different possible executions of the code shown in Scenario 2. The answer is straightforward. The threads are concurrent and execute asynchronously with respect to each other. Therefore, once created, the order of execution of these threads is simply a function of the available number of processors in the computer, any dependency among the threads, and the scheduling algorithm used by the operating system.

The timeline of execution shown in Figure 12.5 for scenario #2 assumes that there is a single processor to schedule the threads and that there is no dependency among the threads. That is, once spawned the threads may execute in any order. Further, a non-preemptive scheduling discipline is assumed. The following example illustrates that there are more than 4 possible executions for these threads.

Example 3:

Assume there are 4 threads in a process. Assume that the threads are scheduled on a uniprocessor. Once spawned, each thread prints its thread-id and terminates. Assuming a non-preemptive thread scheduler, how many different executions are possible?

Answer:

As stated in the problem, the threads are independent of one another. Therefore, the operating system may schedule them in any order.

The number of possible executions of the 4 threads = **4!**

Thus, execution of a parallel program is a fundamental departure from the sequential execution of a program in a uniprocessor. The execution model presented to a sequential program is very simple: the instructions of a sequential program execute in the *program order* of the sequential program¹. We define *program order* as the combination of the textual order in which the instructions of the program appear to the programmer, and the logical order in which these instructions will be executed in every run of the program. The logical order of course depends on the intended semantics for the program as envisioned by the programmer. For example, if you write a high-level language program the source code listing gives you a textual order the program. Depending on the input and the actual logic you have in the program (in terms of conditional statements, loops, procedure calls, etc.) the execution of the program will result in taking a specific path through your source code. In other words, the behavior of a sequential program is *deterministic*, which means that for a given input the output of the program will remain unchanged in every execution of the program.

¹ Note that, as we mentioned in Chapter 5 (Section 5.13.2.4), processor implementation might choose to reorder the execution of the instructions, so long as the appearance of program order to the programmer is preserved despite such reordering.

It is instructive to understand the execution model for a parallel program that is comprised of several threads. The individual threads of a process experience the same execution model as a sequential program. However, there is no guarantee as to the order of execution of the different threads of the same process. That is, the behavior of a parallel program is *non-deterministic*. We define a non-deterministic execution as one in which the output of the program for a given input may change from one execution of the program to the next.

Let us return to scenario #2 and the 4 possible non-preemptive executions of the program shown in Figure 12.5. What are the values that would be printed by T4 in each of the 4 executions?

Let us look at Figure 12.5-(a). Thread T4 is the first one to complete execution. Therefore, the value it would print for count is zero. On the other hand, in Figure 12.5-(b), thread T4 is the last to execute. Therefore, the value printed by T4 would be 3 (each of the executions of T1, T2, and T3 would have incremented count by 1). In Figures 12.5-(c) and 12.5-(d), T4 would print 1, and 2, respectively.

Further, if the scheduler is preemptive, many more interleavings of the threads of an application are possible. Worse yet, a statement such as

count = count + 1

would get compiled into a sequence of machine instructions. For example, compilation of this statement would result in series of instructions that includes a load from memory, increment, and a store to memory. The thread could be preempted after the load instructions before the store. This has serious implications on the expected program behavior and the actual execution of a program that has such data races. We will revisit this issue shortly (Section 12.2.6) and demonstrate the need for atomicity of a group of instructions in the context of a programming example.

Figure 12.6 demonstrates how due to the non-determinism inherent in the parallel programming model, instructions of threads of the same program may get arbitrarily interleaved in an actual run of the program on a uniprocessor.

<u>Thread 1 (T₁)</u>	<u>Thread 2 (T₂)</u>	<u>Thread 3 (T₃)</u>	
I ₁	I ₁	I ₁	T ₃ :I ₁
I ₂	I ₂	I ₂	T ₂ :I ₁
I ₃	I ₃	I ₃	T ₃ :I ₂
I ₄	I ₄	I ₄	T ₁ :I ₂
I ₅	I ₅	I ₅	T ₁ :I ₃
I ₆	I ₆	I ₆	T ₁ :I ₄
I ₇	I ₇	I ₇	T ₂ :I ₃
I ₈	I ₈	I ₈	T ₃ :I ₄
I ₉	I ₉	I ₉	T ₂ :I ₄
			T ₂ :I ₅
			T ₂ :I ₆
			T ₁ :I ₅
			T ₁ :I ₆
			T ₁ :I ₇
			T ₃ :I ₅
			T ₃ :I ₆

(a) A parallel program with three threads

(b) An arbitrary interleaving of the instructions from the three threads

Figure 12.6: A parallel program and a possible trace of its execution on a uniprocessor

There are a few points to take away from Figure 12.6. Instructions of the *same* thread execute in program order (e.g., T₁:I₁, T₂:I₂, T₃:I₃, T₄:I₄,). However, instructions of different threads may get arbitrarily interleaved (Figure 12.6-(b)), while preserving the program order of the individual threads. For example, if you observe the instructions of thread 2 (T₂), you will see that they are in the program order for T₂.

Example 4:

Given the following threads and their execution history, what is the final value in memory location x? Assume that the execution of each instruction is atomic. Assume that Mem[x] = 0 initially.

<u>Thread 1 (T1)</u>
Time 0: R1 <- Mem[x]
Time 2: R1 <- R1+2
Time 4: Mem[x] <- R1

<u>Thread 2 (T2)</u>
Time 1: R2 <- Mem[x]
Time 3: R2 <- R2+1
Time 5: Mem[x] <- R2

Answer:

Each of the threads T1 and T2, load a memory location, add a value to it, and write it back. With the presence of data race and preemptive scheduling, unfortunately, x will contain the value written to it by the last store operation.

Since T2 is the last to complete its store operation, the final value in x is 1.

To summarize, non-determinism is at the core of the execution model for a parallel program. As an application programmer, it is important to understand and come to terms with this concept to be able to write correct parallel programs. Table 12.1 summarizes the execution models of sequential and parallel programs.

	Execution model
Sequential program	The program execution is deterministic, i.e., instructions execute in program order. The hardware implementation of the processor may reorder instructions for efficiency of pipelined execution so long as the appearance of program order is preserved despite such reordering.
Parallel program	The program execution is non-deterministic, i.e., instructions of each individual thread execute in program order. However, the instructions of the different threads of the same program may be arbitrarily interleaved.

Table 12.1: Execution Models of Sequential and Parallel Programs

12.2.4 Synchronization among threads

Let us understand how a programmer could expect a deterministic behavior for her program given that the execution model for a parallel program is non-deterministic.

As an analogy, let us watch some children at play shown in Figure 12.7. There are activities that they can do independently and concurrently (Figure 12.7-(a)) without stepping on each other. However, if they are sharing a toy, we tell them to take turns so that the other child gets a chance (Figure 12.7-(b)).



(a) Children playing independently²



(b) Children sharing a toy³

Figure 12.7: Children playing in a shared “sandbox”

Similarly, if there are two threads, one a *producer* and the other a *consumer* then it is essential that the producer not modify the shared buffer while the consumer is reading it (see Figure 12.8). We refer to this requirement as *mutual exclusion*. The producer and consumer work concurrently except when either or both have to modify or inspect shared data structures. In that case, necessarily they have to execute sequentially to ensure data integrity.

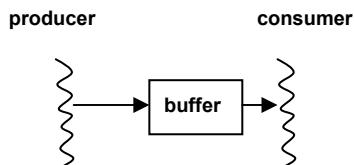


Figure 12.8: Shared buffer between threads

1. Mutual Exclusion Lock and Critical Section

The library provides a *mutual exclusion lock* for this purpose. A *lock* is a data abstraction that has the following semantics. A program can declare any number of these locks just as it declared variables of any other data type. The reader can see the analogy to a physical lock. Only one thread can hold a particular lock at a time. Once a thread *acquires* a lock, other threads cannot get that same lock until the first thread *releases* the lock. The following declaration creates a variable of type *lock*:

```
mutex_lock_type mylock;
```

The following calls allow a thread to acquire and release a particular lock:

```
thread_mutex_lock (mylock);
thread_mutex_unlock(mylock);
```

² Picture source: www.fotosearch.com/BNS238/cca021/

³ Picture source: www.liveandlearn.com/toys/tetherball.html

Successful return from the first function above is an indication to the calling thread that it has successfully acquired the lock. If another thread currently holds the lock, then the calling thread *blocks* until the lock becomes free. In general, we define the *blocked* state of a thread as one in which the thread cannot proceed in its execution until some condition is satisfied. The second function shown above releases the named lock.

Sometimes, a thread may not want to block but go on to do other things if the lock is unavailable. The library provides a non-blocking version of the lock acquisition call:

```
{success, failure} <- thread_mutex_trylock (mylock);
```

This call returns a success or failure indication for the named lock acquisition request.

Example 5:

Show the code fragment to allow the producer and consumer threads in Figure 12.8 to access the buffer in a mutually exclusive manner to place an item and retrieve an item, respectively.

Answer:

```
item_type buffer;
mutex_lock_type buflock;

int producer()
{
    item_type item;

    /* code to produce item */
    .....
    thread_mutex_lock(buflock);
        buffer = item;
        thread_mutex_unlock(buflock);
    .....
    .....

}

int consumer()
{
    item_type item;

    .....
    .....

    thread_mutex_lock(buflock);
        item = buffer;
        thread_mutex_unlock(buflock);
    .....

    /* code to consume item */
}
```

Note: Only **buffer** and **buflock** are shared data structures.
"item" is a local variable within each thread.

In example 5, the producer and consumer execute concurrently for the most part. When either the producer or the consumer executes code in their respective shaded boxes, what is the other thread doing? The answer to this question depends on where the other thread is in its execution. Let us say the producer is executing within its shaded box. Then the consumer is in one of two situations:

- Consumer is also actively executing if it is outside its shaded box
- If the consumer is trying to get into its shaded box, then it has to *wait* until the producer is out of its shaded box; similarly, the producer has to wait until the consumer is already in the shaded box

That is, the execution of the producer and the consumer in their respective shaded box is *mutually exclusive* of each other. Such code that is executed in a mutually exclusive manner is referred to as *critical section*. We define critical section as a region of the program in which the execution of the threads is serialized. That is, exactly one thread can be executing the code *inside* a critical section at a time. If multiple threads arrive at a critical section at the same time, one of them will succeed in entering and executing the code inside the critical section, while the others wait at the entry point. Many of us would have encountered a similar situation with an ATM machine, where we have to wait for our turn to withdraw cash or deposit a check.

As an example of a critical section, we present the code for implementing updates to a shared counter.

```
mutex_lock_type lock;
int counter; /* shared counter */

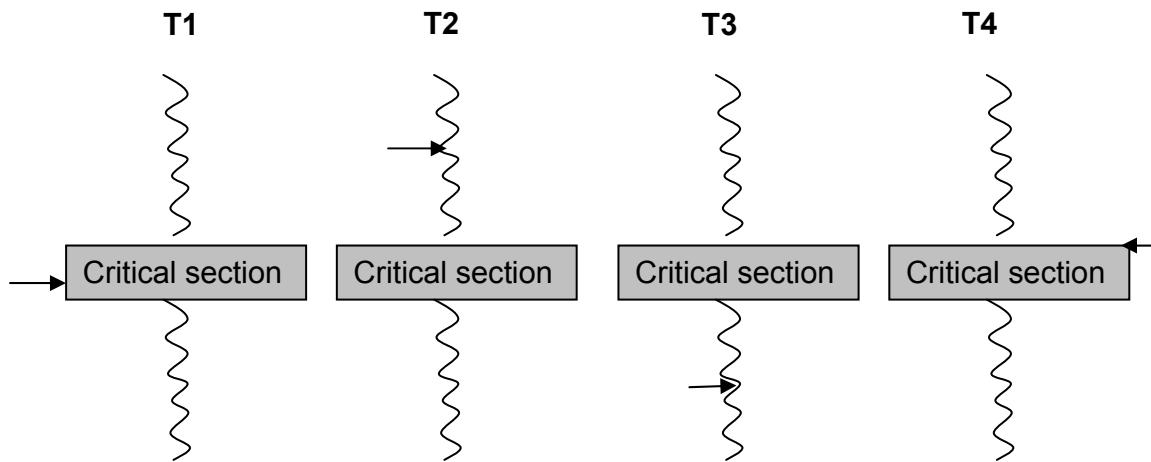
int increment_counter()
{
    /* critical section for
     * updating a shared counter
     */
    thread_mutex_lock(lock);
    counter = counter + 1;
    thread_mutex_unlock(lock);
    return 0;
}
```

Any number of threads may call increment_counter simultaneously. Due to the mutually exclusive nature of thread_mutex_lock, exactly one thread can be inside the critical section updating the shared counter.

Example 6:

Given the points of execution of the threads (indicated by the arrows) in the figure below, state which ones are active and which ones are blocked and why. Assume that the critical

sections are mutually exclusive (i.e., they are governed by the same lock). T1-T4 are threads of the same process.



Answer:

T1 is **active** and executing code **inside its critical section**.

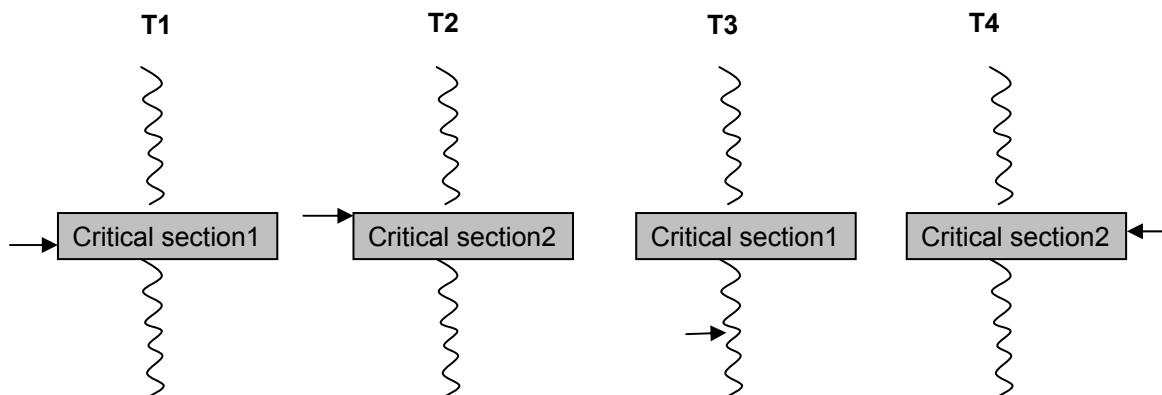
T2 is **active** and executing code **outside its critical section**.

T3 is **active** and executing code **outside its critical section**.

T4 is **blocked** and **waiting** to get into its **critical section**. (It will get in once the lock is released by T1).

Example 7:

Given the points of execution of the threads (indicated by the arrows) in the figure below, state which ones are active and which ones are blocked and why. Note that distinct locks govern each of critical sections 1 and 2, respectively.



Answer:

T1 is **active** and executing code **inside its critical section 1**.

T2 is **blocked** and **waiting** to get into its **critical section 2**. (It will get in once the lock is released by T4).

T3 is **active** and executing code **outside its critical section 1**.

T4 is **active** and executing code **inside its critical section 2**.

2. Rendezvous

A thread may want to wait another thread in the same process. The most common usage of such a mechanism is for a parent to wait for the children that it spawns. Consider for example, a thread is spawned to read a file from the disk while main has some other concurrent activity to perform. Once main completes this concurrent activity, it cannot proceed further until the spawned thread completes its file read. This is a good example where main may wait for its child to terminate which would be an indication that the file read is complete.

The library provides such a *rendezvous* mechanism through the function call:

```
thread_join (peer_thread_id);
```

The function blocks the caller until the named peer thread terminates. Upon the peer thread's termination, the calling thread resumes execution.

More formally, we define *rendezvous* as a meeting point between threads of the same program. A rendezvous requires a minimum of two threads but in the limit may include all the threads of a given program. Threads participating in a rendezvous continue with their respective executions once all the other threads have also arrived at the rendezvous point. Figure 12.9 shows an example of a rendezvous among threads. T1 is the first to arrive at the rendezvous point, awaits the arrival of the other two threads. T3 arrives next; finally, once T2 arrives the rendezvous is complete, and the three threads proceed with their respective executions. As should be evident, rendezvous is a convenient way for threads of a parallel program to coordinate their activities with respect to one another in the presence of the non-deterministic execution model. Further, any subset of the threads of a process may decide to rendezvous amongst them. The most general form of rendezvous mechanism is often referred to in the literature as *barrier synchronization*. This mechanism is extremely useful in parallel programming of scientific applications. All the threads of a given application that would like to participate in the rendezvous execute the barrier synchronization call. Once all the threads have arrived at the barrier, the threads are allowed to proceed with their respective executions.

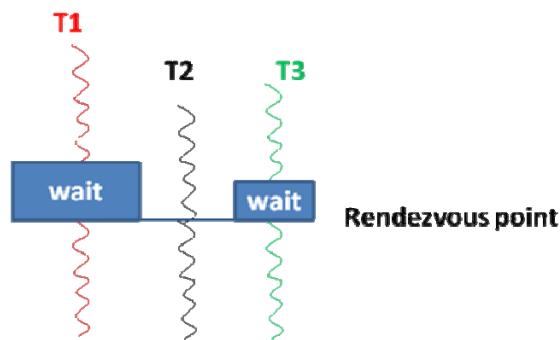


Figure 12.9: Rendezvous among threads

The `thread_join` call is a special case of the general rendezvous mechanism. It is a one-sided rendezvous. Only the thread that executes this call may experience any waiting (if

the peer thread has not already terminated). The peer thread is completely oblivious of the fact that another thread is waiting on it. Note that this call allows the calling thread to wait on the termination of exactly one thread. For example, if a main thread spawns a set of children threads, and would like to be alive until all of them have terminated, then it has to execute multiple `thread_join` calls one after the other for each of its children. In Section 12.2.8 (see Example 10), we will show a symmetric rendezvous between two threads using conditional variables.

There is one way in which the real-life analogy breaks down in the context of threads. A child usually outlives the parent in real life. Unfortunately, this is not necessarily true in the context of threads. In particular, recall that all the threads execute within an address space. As it turns out, not all threads are equal in status. There is a distinction between a parent thread and a child thread. In Figure 12.4 (Example 1), when the process is instantiated there is only one thread within the address space, namely, “main”. Once “main” spawns the “digitizer” and “tracker” threads, the address space has three active threads: “main”, “digitizer”, and “tracker”. What happens when “main” exits? This is the parent thread and is **synonymous** with the process itself. The usual semantics implemented by most operating systems is that when the parent thread in a process terminates, then the entire process terminates. However, note that if a child spawns its own children, immediate parent does not determine these children’s life expectancy; the main thread that is synonymous with the process determines it. This is another reason why the `thread_join` call comes in handy, wherein the parent thread can wait on the children before exiting.

Example 8:

“main” spawns a top-level procedure “foo”. Show how we can ensure that “main” does not terminate prematurely.

Answer:

```
int foo(int n)
{
    .....
    return 0;
}

int main()
{
    int f;
    thread_type child_tid;

    .....

    child_tid = thread_create (foo, &f);

    .....
```

```
    thread_join(child_tid);  
}
```

Note: “main” by executing `thread_join` on the `child_tid` essentially waits for the child to be finished before itself terminating.

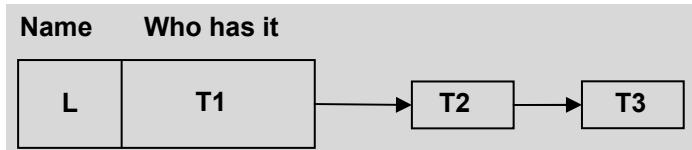
12.2.5 Internal representation of data types provided by the threads library

We mentioned that a thread that blocks when a lock it wants is currently in use by another thread. Let us understand the exact meaning of this statement. As should be evident by now, in contrast to data types (such as “int” and “float”) supported by a programming language such as C, the threads library supports the data types we have been introducing in this chapter.

The `thread_type` and the `mutex_lock_type` are opaque data type (i.e., the user has no direct access to the internal representation of these data types). Internally, the threads library may have some accounting information associated with a variable of the `thread_type` data type. The `mutex_lock_type` data type is interesting and worth knowing more about from the programmer’s perspective. Minimally, the internal representation for a variable of this data type will have two things:

- The thread (if any) that is currently holding the lock
- A queue of waiting requestors (if any) for this lock

Thus, if we have a lock variable `L`, currently a thread `T1` has it, and there are two other threads `T2` and `T3` waiting to get the lock then the internal representation in the threads library for this variable `L` will look as follows:



When `T1` releases the lock, `T2` gets the lock since that is the first thread in the waiting queue for this lock. Note that every lock variable has a distinct waiting queue associated with it. A thread can be on exactly one waiting queue at any point of time.

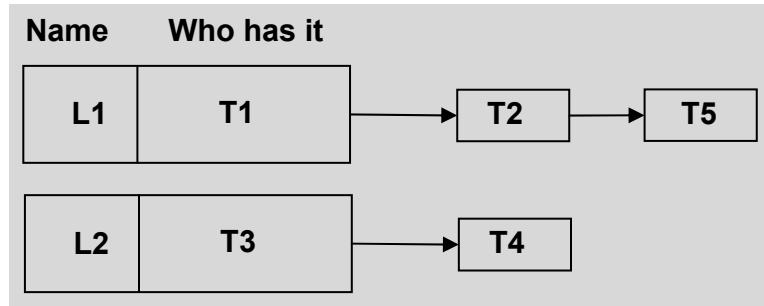
Example 9:

Assume that the following events happen in the order shown (T1-T5 are threads of the same process):

```
T1 executes thread_mutex_lock(L1);  
T2 executes thread_mutex_lock(L1);  
T3 executes thread_mutex_lock(L2);  
T4 executes thread_mutex_lock(L2);  
T5 executes thread_mutex_lock(L1);
```

Assuming there has been no other calls to the threads library prior to this, show the state of the internal queues in the threads library after the above five calls.

Answer:



12.2.6 Simple programming examples

1. Basic code with no synchronization

First, let us understand why we need the synchronization constructs. Consider the following sample program (#1) to show the interaction between the digitizer and the tracker threads of Figure 12.2. Suffice it to say at this point that we will progressively refine the sample program to ensure that it delivers the desired semantics for this application. For the benefit of advanced readers, the sample program #5 appearing a little later in the text delivers the desired semantics (see Section 12.2.9).

```

/*
 * Sample program #1:
 */
#define MAX 100

int bufavail = MAX;
image_type frame_buf[MAX];

digitizer()
{
    image_type dig_image;
    int tail = 0;

    loop { /* begin loop */
        if (bufavail > 0) {
            grab(dig_image);
            frame_buf[tail mod MAX] =
                dig_image;
            bufavail = bufavail - 1;
            tail = tail + 1;
        }
    } /* end loop */
}

tracker()
{
    image_type track_image;
    int head = 0;

    loop { /* begin loop */
        if (bufavail < MAX) {
            track_image =
                frame_buf[head mod MAX];
            bufavail = bufavail + 1;
            head = head + 1;
            analyze(track_image);
        }
    } /* end loop */
}

```

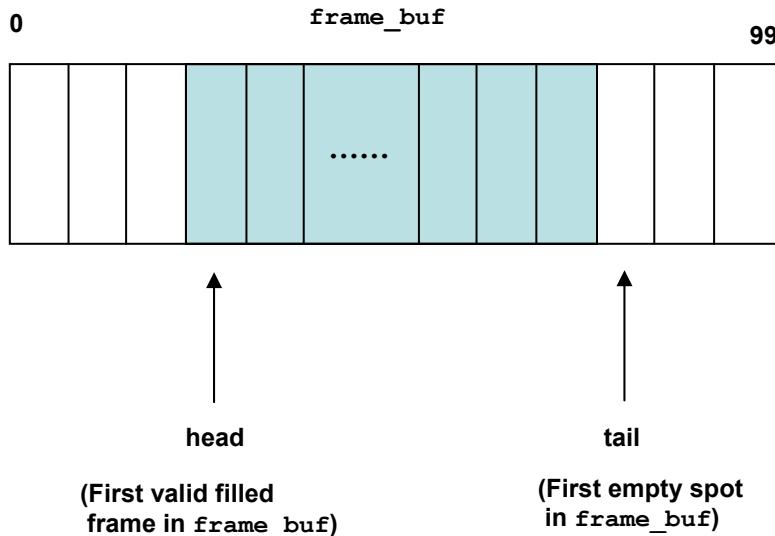


Figure 12.10: `frame_buf` implemented as a circular queue with head and tail pointers

In this sample program, `bufavail` and `frame_buf` are the shared data structures between the digitizer and the tracker threads. The sample program shows an implementation of the `frame_buf` as a circular queue with a `head` and a `tail` pointer,

with insertion at the tail and deletion at the head (see Figure 12.10; the shaded region contains valid items in **frame_buf**). The availability of space in the buffer is indicated by the **bufavail** variable.

The **head** and **tail** pointers themselves are local variables inside the digitizer and tracker, respectively. The digitizer code continuously loops grabbing an image from the camera, putting it in the frame buffer, advancing its **tail** pointer to point to the next open slot in **frame_buf**. Availability of space in the frame buffer (**bufavail > 0**) predicates this execution within the loop. Similarly, the tracker code continuously loops getting an image from the frame buffer (if one is available), advancing its **head** pointer to the next valid frame in **frame_buf**, and then analyzing the frame for items of interest. The two threads are independent of one another **except** for the interaction that happens in the **shaded boxes** in the two threads shown in sample program (#1). The code in the shaded boxes manipulates the shared variables, namely, **frame_buf** and **bufavail**.

2. Need for atomicity for a group of instructions

The problem with the above code fragment is that the digitizer and the tracker are concurrent threads and they could be reading and writing to the shared data structures *simultaneously*, if they are each executing on a distinct processor. Figure 12.11 captures this situation. Both the digitizer and the tracker are modifying **bufavail** at the same time.

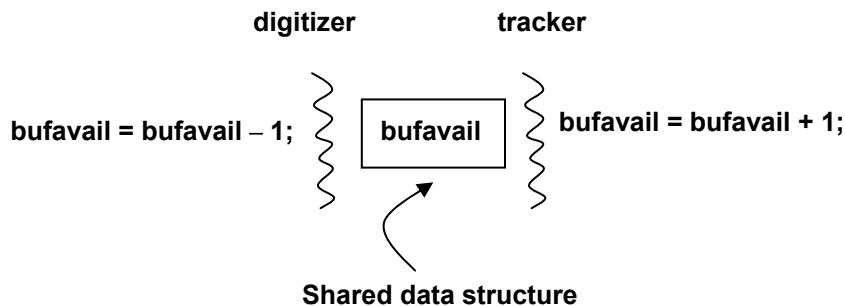


Figure 12.11: Problem with unsynchronized access to shared data

Let us drill down into this situation a little more. The statement

$$\text{bufavail} = \text{bufavail} - 1; \quad (1)$$

is implemented as a set of instructions on the processor (load **bufavail** into a processor register; do the decrement; store the register back into **bufavail**).

Similarly, the statement

$$\text{bufavail} = \text{bufavail} + 1; \quad (2)$$

is implemented as a set of instructions on the processor (load **bufavail** into a processor register; do the increment; store the register back into **bufavail**).

A correct execution of the program requires the *atomic* execution of each of the two statements (1 and 2). That is, either statement (1) executes and then (2), or vice versa. Interleaved execution of the instruction sequences for (1) and (2) could lead to erroneous and unanticipated behavior of the program. This is what we referred to as data race in Section 12.2.3. As we already mentioned in Section 12.2.3, such an interleaving could happen even on a uniprocessor due to context switches (see Example 4). The processor guarantees atomicity of an instruction at the level of the instruction-set architecture. However, the system software (namely, the operating system) has to guarantee atomicity for a group of instructions.

Therefore, to ensure atomicity we need to encapsulate accesses to shared data structures within *critical sections* to ensure mutually exclusive execution. However, we have to be careful in deciding how and when to use synchronization constructs. Indiscriminate use of these constructs, while ensuring atomicity could lead to restricting concurrency and worse yet introduce incorrect program behavior.

3. Code refinement with coarse grain critical sections

We will now proceed to use the synchronization constructs **thread_mutex_lock** and **thread_mutex_unlock** in the sample program to achieve the desired mutual exclusion.

Sample program #2 is another attempt at writing a threaded sample program for the same example in Figure 12.2. This program illustrates the use of mutual exclusion lock. The difference from the earlier one (sample program #1) is the addition of the synchronization constructs inside the shaded boxes in sample program #2. In each of digitizer and tracker, the code between lock and unlock is the work done by the respective threads to access shared data structures. Note that the use of synchronization constructs ensures atomicity of the entire code block between lock and unlock calls for the digitizer and tracker, respectively. This program is “correct” in terms of the desired semantics but has a serious performance problem that we elaborate next.

```

/*
 * Sample program #2:
 */
#define MAX 100

int bufavail = MAX;
image_type frame_buf[MAX];
mutex_lock_type buflock;

digitizer()
{
    image_type dig_image;
    int tail = 0;

    loop { /* begin loop */
        thread_mutex_lock(buflock);
        if (bufavail > 0) {
            grab(dig_image);
            frame_buf[tail mod MAX] =
                dig_image;
            tail = tail + 1;
            bufavail = bufavail - 1;
        }
        thread_mutex_unlock(buflock);
    } /* end loop */
}

tracker()
{
    image_type track_image;
    int head = 0;

    loop { /* begin loop */
        thread_mutex_lock(buflock);
        if (bufavail < MAX) {
            track_image =
                frame_buf[head mod MAX];
            head = head + 1;
            bufavail = bufavail + 1;
            analyze(track_image);
        }
        thread_mutex_unlock(buflock);
    } /* end loop */
}

```

4. Code refinement with fine grain critical sections

A close inspection of sample program #2 will reveal that it has no synchronization problem but there is no concurrency in the execution of the digitizer and the tracker. Let us analyze what needs mutual exclusion in this sample program. There is no need for mutual exclusion for either grabbing the image by the digitizer or for analyzing the image by the tracker. Similarly, once the threads have ascertained the validity of operating on **frame_buf** by checking **bufavail**, insertion or deletion of the item can proceed concurrently. That is, although **frame_buf** is a shared data structure, the way it is used in the program obviates the need for serializing access to it. Therefore, we modify the program as shown below to increase the amount of concurrency between the tracker and the digitizer. We limit the mutual exclusion to the checks and modifications done to **bufavail**. Unfortunately, the resulting code has a serious problem that we elaborate next.

```

/*
 * Sample program #3:
 */
#define MAX 100

int bufavail = MAX;
image_type frame_buf[MAX];
mutex_lock_type buflock;

digitizer()
{
    image_type dig_image;
    int tail = 0;

    loop { /* begin loop */
        grab(dig_image);
        thread_mutex_lock(buflock);
        while (bufavail == 0) do nothing;
        thread_mutex_unlock(buflock);
        frame_buf[tail mod MAX] =
            dig_image;
        tail = tail + 1;
        thread_mutex_lock(buflock);
        bufavail = bufavail - 1;
        thread_mutex_unlock(buflock);
    } /* end loop */
}

tracker()
{
    image_type track_image;
    int head = 0;

    loop { /* begin loop */
        thread_mutex_lock(buflock);
        while (bufavail == MAX) do nothing;
        thread_mutex_unlock(buflock);
        track_image =
            frame_buf[head mod MAX];
        head = head + 1;
        thread_mutex_lock(buflock);
        bufavail = bufavail + 1;
        thread_mutex_unlock(buflock);
        analyze(track_image);
    } /* end loop */
}

```

12.2.7 Deadlocks and livelocks

Let us dissect sample program #3 to see if it has any problems. Consider the **while** statement in the digitizer code. It is checking **bufavail** for an empty spot in the **frame_buf**. Let us assume that **frame_buf** is full. In this case, the digitizer is continuously executing the **while** statement waiting for space to free up in **frame_buf**. The tracker has to make space in the **frame_buf** by removing an image and incrementing **bufavail**. However, the digitizer has **buflock** and hence the tracker is stuck trying to acquire **buflock**. A similar situation arises when **frame_buf** is empty (the **while** statement in the tracker code).

The problem we have described above, called *deadlock* is the bane of all concurrent programs. Deadlock is a situation wherein a thread is waiting for an event that will never happen. For example, the digitizer is waiting for **bufavail** to become non-zero in the **while** statement, but that event will not happen since the tracker cannot get the lock. The situation captured above is a special case of deadlock, often referred to as a *livelock*. A thread involved in a deadlock may be waiting actively or passively. Livelock is the

situation wherein a thread is actively checking for an event that will never happen. Let us say in this case, the digitizer is holding the **buflock** and checking for **bufavail** to become non-zero. This is a livelock since it is wasting processor resource to check for an event that will never happen. On the other hand, the tracker is waiting for the lock to be released by the digitizer. Tracker's waiting is passive since it is blocked in the operating system for the lock release. Regardless of whether the waiting is passive or active, the threads involved in a deadlock are stuck forever. It should be evident to the reader that deadlocks and livelocks are yet another manifestation of the basic non-deterministic nature of a parallel program execution.

One could make a case that the **while** statements in the above code do not need mutual exclusion since they are only inspecting buffer availability. In fact, removing mutual exclusion around the **while** statements will remove the *deadlock* problem. Sample program #4 takes this approach. The difference from sample program #3 is that the lock around the while statement has been removed for both the digitizer and the tracker.

```

/*
 * Sample program #4:
 */
#define MAX 100

int bufavail = MAX;
image_type frame_buf[MAX];
mutex_lock_type buflock;

digitizer()
{
    image_type dig_image;
    int tail = 0;

    loop { /* begin loop */
        grab(dig_image);
        while (bufavail == 0) do nothing;
        frame_buf[tail mod MAX] =
            dig_image;
        tail = tail + 1;
        thread_mutex_lock(buflock);
        bufavail = bufavail - 1;
        thread_mutex_unlock(buflock);
    } /* end loop */
}

tracker()
{
    image_type track_image;
    int head = 0;

    loop { /* begin loop */
        while (bufavail == MAX) do nothing;
        track_image =
            frame_buf[head mod MAX];
        head = head + 1;
        thread_mutex_lock(buflock);
        bufavail = bufavail + 1;
        thread_mutex_unlock(buflock);
        analyze(track_image);
    } /* end loop */
}

```

This solution is correct and has concurrency as well for the digitizer and the tracker. However, the solution is grossly inefficient due to the nature of waiting. We refer to the

kind of waiting that either the digitizer or the tracker does in the while statement as *busy waiting*. The processor is busy doing nothing. This is inefficient since the processor could have been doing something more useful for some other thread or process.

12.2.8 Condition variables

Ideally, we would like the system to recognize that the condition that the digitizer is waiting for (`bufavail > 0`) is not satisfied, and therefore release the lock on behalf of the digitizer, and reschedule it later on when the condition is satisfied.

This is exactly the semantics of another data abstraction commonly provided by the library, called a *condition variable*.

The following declaration creates a variable of type of *condition variable*:

```
cond_var_type buf_not_empty;
```

The library provides calls to allow threads to *wait* and *signal* one another using these condition variables:

```
thread_cond_wait(buf_not_empty, buflock);
thread_cond_signal(buf_not_empty);
```

The first call allows a thread (tracker in our example) to wait on a condition variable. *Waiting* on a condition variable amounts to the library de-scheduling the thread that made that call. Notice that the second argument to this call is a mutual exclusion lock variable. Implicitly, the library performs `unlock` on the named lock variable before de-scheduling the calling thread. The second call *signals* any thread that may be waiting on the named condition variable. A *signal* as the name suggests is an indication to the waiting thread that it may resume execution. The library knows the specific lock variable associated with the *wait* call. Therefore, the library performs an implicit `lock` on that variable prior to scheduling the waiting thread. Of course, the library treats as a NOP any signal on a condition variable for which there is no waiting thread. Multiple threads may wait on the same condition variable. The library picks one of them (usually First-Come-First-Served) among the waiting threads to deliver the signal. One has to be very careful in using wait and signal for synchronization among threads. Figure 12.12 (a) shows a correct use of the primitives. Figure 12.12 (b) shows an incorrect use, creating a situation wherein T1 starts waiting after T2 has signaled, leading to a deadlock.

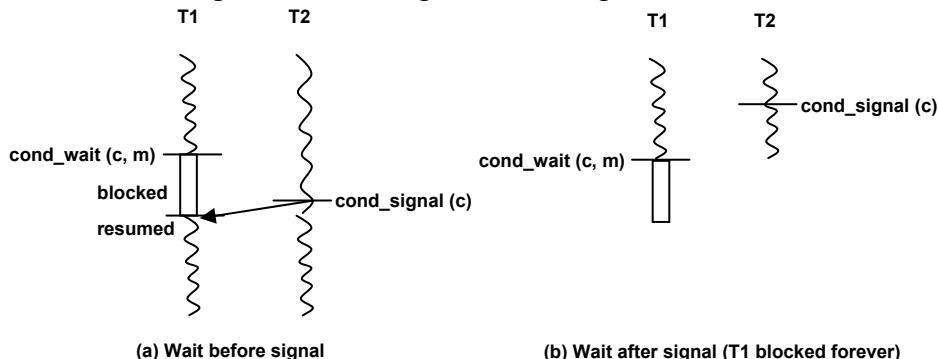
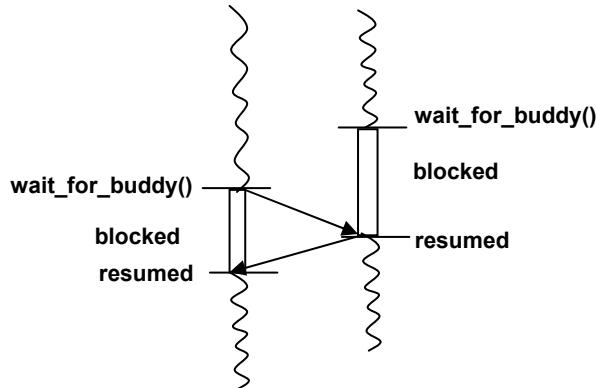


Figure 12.12: Wait and Signal with condition variable: “c” is a condition variable, and “m” is a mutex lock with which is “c” is associated through the cond_wait call

Figure 12.12(b) illustrates the situation that a signal sent prematurely will land a peer in a deadlock. The following example shows how one can devise a rendezvous mechanisms between two peers independent of the order of arrival.

Example 10:

Write a function **wait_for_buddy()** to be used by EXACTLY 2 threads to rendezvous with each other as shown in the figure below. The order of arrival of the two thread should be immaterial. Note that this is a general example of accomplishing a rendezvous (described earlier in Section 12.2.4) among independent threads of the same process.



Answer:

The solution uses a boolean (`buddy_waiting`), a mutex lock (`mtx`), and a condition variable (`cond`). The basic idea is the following:

- Whichever thread arrives first (the “if” section of the code below), sets the `buddy_waiting` flag to be true and waits
- The second arriving thread (the “else” section of the code below), sets the `buddy_waiting` to be false, signals the first thread, and waits
- The first arriving thread is unblocked from its conditional wait, signals the second thread, unlocks the mutex and leaves the procedure
- The second arriving thread is unblocked from its conditional wait, unlocks the mutex and leaves the procedure
- It is important to observe the wait-signal ordering in the “if” and “else” code blocks; not following the order shown will result in deadlock.

```

boolean buddy_waiting = FALSE;
mutex_lock_type mtx; /* assume this has been initialized properly */
cond_var_type cond; /* assume this has been initialized properly */

wait_for_buddy()
{
    /* both buddies execute the lock statement */
    thread_mutex_lock(mtx);

    if (buddy_waiting == FALSE) {
        /* first arriving thread executes this code block */
        buddy_waiting = TRUE;

        /* the following order is important */
        /* ... */
    }
}

```

```

/* the first arriving thread will execute a wait statement */
thread_cond_wait (cond, mtx);

/* the first thread wakes up due to the signal from the second
 * thread, and immediately signals the second arriving thread
 */
thread_cond_signal(cond);
}

else {
    /* second arriving thread executes this code block */
    buddy_waiting = FALSE;

    /* the following order is important */
    /* signal the first arriving thread and then execute a wait
     * statement awaiting a corresponding signal from the first thread
     */
    thread_cond_signal (cond);
    thread_cond_wait (cond, mtx);
}

/* both buddies execute the unlock statement */
thread_mutex_unlock (mtx);
}

```

12.2.8.1 Internal representation of the condition variable data type

It is instructive from a programming standpoint to understand the internal representation within the threads library of the **cond_var_type** data type. Minimally, a variable of this data type has:

- a queue of threads (if any) waiting for signal on this variable
- for each thread waiting for a signal, the associated mutex lock

A thread that calls **thread_cond_wait** names a mutex lock. The threads library unlocks this lock on behalf of this thread before placing it on the waiting queue. Similarly, upon receiving a signal on this condition variable, when a thread is unblocked from the waiting queue, it is the responsibility of the threads library to re-acquire the lock on behalf of the thread before resuming the thread. This is the reason why the threads library remembers the lock associated with a thread on the waiting queue.

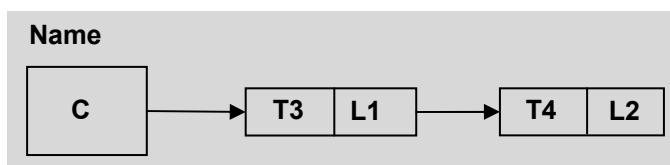
Thus, for instance if two threads T3 and T4 execute conditional wait calls on a condition variable C; let T3's call be

thread_cond_wait (C, L1)

and let T4's call be

thread_cond_wait (C, L2)

The internal representation of C after the above two calls will look as follows:



Note that it is not necessary that all wait calls on a given condition variable name the same lock variable.

Example 11:

Assume that the following events happen in the order shown (T1-T7 are threads of the same process):

```

T1 executes thread_mutex_lock(L1) ;
T2 executes thread_cond_wait(C1, L1) ;
T3 executes thread_mutex_lock(L2) ;
T4 executes thread_cond_wait(C2, L2) ;
T5 executes thread_cond_wait(C1, L2) ;

```

(a) Assuming there has been no other calls to the threads library prior to this, show the state of the internal queues in the threads library after the above five calls.

(b) Subsequently the following event happens:

```

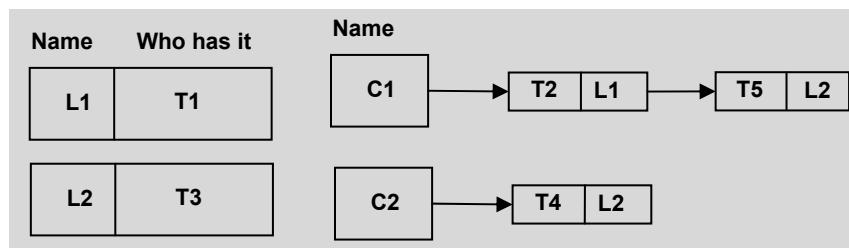
T6 executes thread_cond_signal(C1) ;
T7 executes thread_cond_signal(C2) ;

```

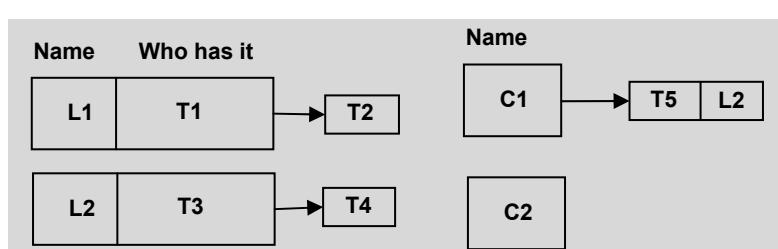
Show the state of the internal queues in the threads library after these two calls.

Answer:

(a)



(b)



The library moves T2 to the waiting queue of L1, and T4 to the waiting queue of L2 upon receiving the signals on C1 and C2, respectively.

12.2.9 A complete solution for the video processing example

Now we will return to our original video processing example of Figure 12.2. Shown below is a program sample using wait and signal semantics for the same example. Note that each thread waits after checking a condition that is not true currently; the other thread enables this condition eventually ensuring that there will not be a deadlock. Note

also that each thread performs the signaling while holding the mutual exclusion lock. While this is strictly not necessary, nevertheless, it is a good programming practice, and results in less erroneous parallel programs.

```

/*
 * Sample program #5: This solution delivers the expected
 * semantics for the video processing
 * pipeline shown in Figure 12.2, both
 * in terms of performance and
 * correctness for a single digitizer
 * feeding images to a single tracker.
*/
#define MAX 100

int bufavail = MAX;
image_type frame_buf[MAX];
mutex_lock_type buflock;
cond_var_type buf_not_full;
cond_var_type buf_not_empty;

digitizer()
{
    image_type dig_image;
    int tail = 0;

    loop { /* begin loop */
        grab(dig_image);
        thread_mutex_lock(buflock);
        if (bufavail == 0)
            thread_cond_wait(buf_not_full,
                             buflock);
        thread_mutex_unlock(buflock);
        frame_buf[tail mod MAX] = dig_image;
        tail = tail + 1;
        (1)   thread_mutex_lock(buflock);
        bufavail = bufavail - 1;
        thread_cond_signal(buf_not_empty);
        thread_mutex_unlock(buflock);
    } /* end loop */
}

tracker()
{
    image_type track_image;
    int head = 0;

    loop { /* begin loop */
        (2)   thread_mutex_lock(buflock);
        if (bufavail == MAX)
            thread_cond_wait(buf_not_empty,
                             buflock);
        thread_mutex_unlock(buflock);
        track_image = frame_buf[head mod MAX];
        head = head + 1;
        thread_mutex_lock(buflock);
        bufavail = bufavail + 1;
        thread_cond_signal(buf_not_full);
        thread_mutex_unlock(buflock);
        analyze(track_image);
    } /* end loop */
}

```

(3)

(4)

The key point to note in this program sample is the *invariant* maintained by the library on behalf of each thread. An invariant is some indisputable truth about the state of the program. At the point of making the **thread_cond_wait** call, the invariant is that the calling thread is holding a lock. The library implicitly releases the lock on behalf of the calling thread. When the thread resumes it is necessary to re-establish the invariant. The library re-establishes the invariant on behalf of the blocked thread prior to resumption by implicitly re-acquiring the lock.

12.2.9.1 Discussion of the solution

1. Concurrency

Let us analyze the solution presented in program sample #5 and convince ourselves that there is no lack of concurrency.

- First, notice that code blocks (1) and (3) hold the lock only for checking the value in **bufavail**. If the checking results in a favorable outcome, the release the lock and go on to either putting an image or getting an image, respectively. What if the checking results in an unfavorable outcome? In that case, the code blocks execute a conditional wait statement on **buf_not_full** and **buf_not_empty**, respectively. In either case, the library releases the associated lock immediately.
- Second, notice that code blocks (2) and (4) hold the lock only for updating the **bufavail** variable, and signaling to unblock the other thread (if it is waiting).

Given the above two points, there is **no lack of concurrency** since the lock is never held for any extended period by either thread.

Example 12:

Assume that the digitizer is in code block (2) and is about to signal on the **buf_not_empty** condition variable in Sample program #5.

Explain if the following statement is **True** or **False** with justification:

The tracker is guaranteed to be waiting for a signal on **buf_not_empty inside code block (3).**

Answer:

False. Tracker could be waiting but not always. Note that the signaling in code block (2) is unconditional. Therefore, we do not know what the value of **bufavail** is. The only way for the tracker to be blocked inside code block (2) is if **bufavail = MAX**. We know it is non-zero since the digitizer is able to put a frame in, but we do not know that it is = MAX.

2. Absence of deadlock

Next, let us convince ourselves that the solution is correct and does not result in deadlock. First, we will informally show that at any point of time **both** the threads do not block leading to a deadlock.

- Let the digitizer be waiting inside code block (1) for a signal. Given this, we will show that the tracker will not also block leading to a deadlock. Since the digitizer is blocked, we know the following to be true:

- ❖ **bufavail = 0**
- ❖ digitizer is blocked waiting for a signal on **buf_not_full**
- ❖ **buflock** has been implicitly released by the thread library on behalf of the digitizer.

There are three possible places for the tracker to block:

- ❖ Entry to code block (3): Since the digitizer does not hold **buflock** tracker will not block at the entry point.

- ❖ Entry to code block (4): Since the digitizer does not hold **buflock** tracker will not block at the entry point.
- ❖ Conditional wait statement inside code block (3): The digitizer is blocked waiting for a signal inside code block (1). Therefore, **bufavail** = 0; hence, the “if” statement inside code block (3) will return a favorable result and the tracker is guaranteed not to block.
- With a similar line of argument as above, we can establish that if the tracker is waiting inside code block (3) for a signal, the digitizer will not also block leading to a deadlock.

Next, we will show that if one thread is blocked it will eventually unblock thanks to the other.

- Let us say that the digitizer is blocked waiting for a signal inside code block (1). As we argued earlier, the tracker will be able to execute its code without blocking. Therefore, eventually it will get to the signal statement inside code block (4). Upon receiving this signal, the digitizer waits for re-acquiring the lock (currently held by the tracker inside code block (4)). Note that the thread library does this lock re-acquisition implicitly for the digitizer. Tracker leaves code block (4) releasing the lock; digitizer re-acquires the lock and moves out of code block (1) releasing the lock on its way out.
- With a similar line of argument as above, we can establish that if the tracker is waiting inside code block (3) for a signal, the digitizer will issue the signal that will unblock the tracker.

Thus, we have shown informally that the solution is correct and does not suffer from lack of concurrency.

12.2.10 Rechecking the predicate

The program sample #5 works correctly for the specific example where there is one tracker and one digitizer. However, in general, programming with condition variables needs more care to avoid synchronization errors. Consider the program fragment shown below for using a shared resource. Any number of threads can execute the procedure **use_shared_resource**.

```

/*
 * Sample program #6:
 */
enum state_t {BUSY, NOT_BUSY} res_state = NOT_BUSY;
mutex_lock_type cs_mutex;
cond_var_type res_not_busy;

/* helper procedure for acquiring the resource */
acquire_shared_resource()
{
    thread_mutex_lock(cs_mutex);
    if (res_state == BUSY)

```

T3 is here

```

        thread_cond_wait (res_not_busy, cs_mutex); T2 is here
        res_state = BUSY;
        thread_mutex_unlock(cs_mutex);
    }

/* helper procedure for releasing the resource */
release_shared_resource()
{
    thread_mutex_lock(cs_mutex);
    res_state = NOT_BUSY; T1 is here
    thread_cond_signal(res_not_busy);
    thread_mutex_unlock(cs_mutex);
}

/* top level procedure called by all the threads */
use_shared_resource()
{
    acquire_shared_resouce();
    resource_specific_function();
    release_shared_resource();
}

```

As shown above,

- T1 has just finished using the resource and has set the **res_state** as **NOT_BUSY**;
- T2 is in conditional wait; and
- T3 is waiting to acquire the **cs_mutex**.

Figure 12.13 shows the state of the waiting queues in the library for **cs_mutex** and **res_not_busy**:

- T2 is in the **res_not_busy** queue while T3 is in the **cs_mutex** queue, respectively (Figure 12.13 (a)).
- T1 signals, resulting in the library moving T2 to the **cs_mutex** queue since the library has to re-acquire the **cs_mutex** before resuming T2 (Figure 12.13 (b)).

When T1 releases **cs_mutex**:

- The lock is given to T3, the first thread in the waiting queue for **cs_mutex**.
- T3 tests **res_state** to be **NOT_BUSY** and releases **cs_mutex**, and goes on to use the resource.
- T2 resumes from the **thread_cond_wait** (since **cs_mutex** is now available for it), releases **cs_mutex** and goes on to use the resource as well.

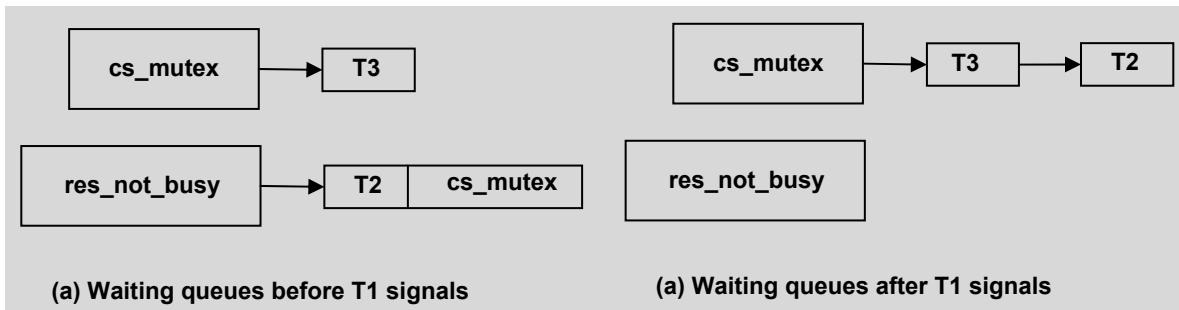


Figure 12.13: State of the waiting queues

Now we have violated the mutual exclusion condition for using the shared resource. Let us investigate what led to this situation. T1 enables the condition that T2 is waiting on prior to signaling, but T3 negates it before T2 resumes execution. Therefore, re-checking the predicate (i.e., the condition that needs to be satisfied in the program) upon resumption is a defensive coding technique to avoid such synchronization errors.

The program fragment shown below fixes the above problem by changing the **if** statement associated with the **thread_cond_wait** to a **while** statement. This ensures that a thread tests the predicate again upon resumption and blocks again on **thread_cond_wait** call if necessary.

```

/*
 * Sample program #7:
 */
enum state_t {BUSY, NOT_BUSY} res_state = NOT_BUSY;
mutex_lock_type cs_mutex;
cond_var_type res_not_busy;

acquire_shared_resource()
{
    thread_mutex_lock(cs_mutex);
    while (res_state == BUSY)                                T3 is here
        thread_cond_wait (res_not_busy, cs_mutex);           T2 is here
    res_state = BUSY;
    thread_mutex_unlock(cs_mutex);
}

release_shared_resource()
{
    thread_mutex_lock(cs_mutex);
    res_state = NOT_BUSY;                                  T1 is here
    thread_cond_signal(res_not_busy);
    thread_mutex_unlock(cs_mutex);
}
  
```

```

use_shared_resource()
{
    acquire_shared_resouce();
    resource_specific_function();
    release_shared_resource();
}

```

Example 13:

Rewrite sample program #5 to allow multiple digitizers and multiple trackers to work together. This is left as an exercise for the reader.

[Hint: Rechecking the predicate after awakening from a conditional wait becomes important. In addition, now the instances of digitizers share the head pointer and instances of trackers share the tail pointer. Therefore, modifications of these pointers require mutual exclusion among the digitizer instances and tracker instances, respectively. To ensure concurrency and reduce unnecessary contention among the threads, use distinct locks to provide mutual exclusion for the head and tail pointers, respectively.]

12.3 Summary of thread function calls and threaded programming concepts

Let us summarize the basic function calls we proposed in the earlier sections for programming multithreaded applications. Note that this is just an illustrative set of basic calls and not meant to be comprehensive. In Section 12.6, we give a comprehensive set of function calls proposed as an IEEE POSIX⁴ standard thread library.

- **thread_create (top-level procedure, args);**
creates a new thread that starts execution in the top-level procedure with the supplied args as actual parameters for the formal parameters specified in the procedure prototype.
- **thread_terminate (tid);**
terminates the thread with id given by **tid**.
- **thread_mutex_lock (mylock);**
when the thread returns it has **mylock**; the calling thread blocks if the lock is in use currently by some other thread.
- **thread_mutex_trylock (mylock);**
call does not block the calling thread; instead it returns **success** if the thread gets **mylock**; **failure** if the lock is in use currently by some other thread.
- **thread_mutex_unlock (mylock);**
if the calling thread currently has **mylock** it is released; error otherwise.

⁴ IEEE is an international society and stands for **Institute of Electrical and Electronics Engineers, Inc.**, and POSIX stands for **Portable Operating System Interface (POSIX®)**.

- **thread_join (peer_thread_tid);**
the calling thread blocks until thread given by **peer_thread_tid** terminates.
- **thread_cond_wait(buf_not_empty, buflock);**
calling thread blocks on the condition variable **buf_not_empty**; the library implicitly releases the lock **buflock**; error if the lock is not currently held by the calling thread.
- **thread_cond_signal(buf_not_empty);**
a thread (if any) waiting on the condition variable **buf_not_empty** is woken up; the awakened thread either is ready for execution if the lock associated with it (in the wait call) is currently available; if not, the thread is moved from the queue for the condition variable to the appropriate lock queue.

Concept	Definition and/or Use
Top level procedure	The starting point for execution of a thread of a parallel program
Program order	This is the execution model for a sequential program that combines the textual order of the program together with the program logic (conditional statements, loops, procedures, etc.) enforced by the intended semantics of the programmer.
Execution model for a parallel program	The execution model for a parallel program preserves the program order for individual threads, but allows arbitrary interleaving of the individual instructions of the different threads.
Deterministic execution	Every run of a given program results in the same output for a given set of inputs. The execution model presented to a sequential program has this property.
Non-deterministic execution	Different runs of the same program for the same set of inputs could result in different outputs. The execution model presented to a parallel program has this property.
Data race	Multiple threads of the same program are simultaneously accessing an arbitrary shared variable without any synchronization, with at least one of the accesses being a write to the variable.
Mutual exclusion	Signifies a requirement to ensure that threads of the same program execute serially (i.e., not concurrently). This requirement needs to be satisfied in order to avoid data races in a parallel program.
Critical section	A region of a program wherein the activities of the threads are serialized to ensure mutual exclusion.
Blocked	Signifies the state of a thread in which it is simply waiting in a queue for some condition to be satisfied to make it runnable.
Busy waiting	Signifies the state of a thread in which it is continuously checking for a condition to be satisfied before it can proceed further in its execution.
Deadlock	One or more threads of the same program are blocked awaiting a condition that will never be satisfied.
Livelock	One or more threads of the same program are busy-waiting for a condition that will never be satisfied.
Rendezvous	Multiple threads of a parallel program use this mechanism to coordinate their activities. The most general kind of rendezvous is barrier synchronization. A special case of rendezvous is the thread <code>join</code> call.

Table 12.2: Summary of Concepts Relating to Threads

Table 12.2 summarizes the important concepts we have introduced in the context of multithreaded programming as a quick reference.

12.4 Points to remember in programming with threads

There are several points to take care of while programming with threads:

1. It is important to design the data structures in such a way to enhance concurrency among threads.

2. It is important to minimize both the granularity of data structures that need to be locked in a mutually exclusive manner as well as the duration for which such locks need to be held.
3. It is important to avoid busy waiting since it is wasteful of the processor resource.
4. It is important to carefully understand the invariant that is true for each critical section in the program and ensure that this invariant is preserved while in the critical section.
5. It is important to make the critical section code as simple and concise as possible to enable manual verification that there are no deadlocks or livelocks.

12.5 Using threads as software structuring abstraction

Figure 12.14 shows some of the models for using threads as a structuring mechanism for system software.

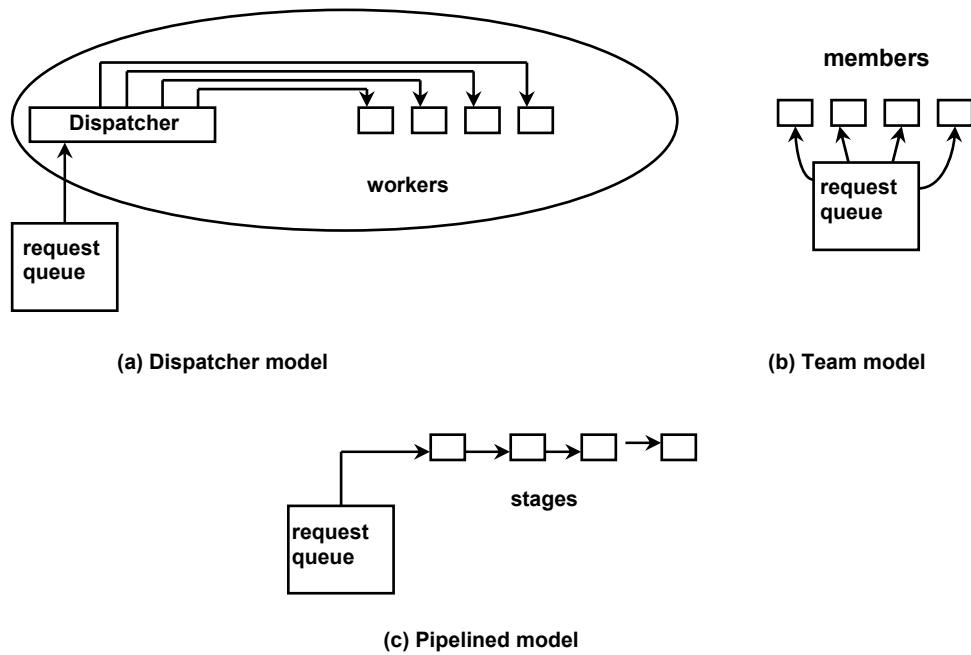


Figure 12.14: Structuring servers using threads

Software entities such as file servers, mail servers, and web servers typically execute on multiprocessors. Figure 12.14 (a) shows a dispatcher model for such servers. A *dispatcher* thread dispatches requests as they come in to one of a pool of *worker* threads. Upon completion of the request, the worker thread returns to the free pool. The *request queue* serves to smooth the traffic when the burst of requests exceeds server capacity. The dispatcher serves as a workload manager as well, shrinking and growing the number of worker threads to meet the demand. Figure 12.14 (b) shows a *team* model in which all the members of the team directly access the request queue for work. Figure 12.14 (c) shows a pipelined model that is more appropriate for continuous applications such as video surveillance that we discussed earlier in the chapter. Each stage of the pipeline handles a specific task (e.g. digitizer, tracker, etc.).

Client programs benefit from multithreading as well. Threads increase modularity and simplicity of client programs. For example, a client program may use threads to deal with exceptions, handling signals, and for terminal input/output.

12.6 POSIX pthreads library calls summary

IEEE has standardized the Application Programming Interface (API) for threads with the POSIX *pthreads* library. Every flavor of Unix operating system implements this standard. Program portability is the main intent of such standardization efforts. Microsoft Windows does not follow the POSIX standard for its thread library⁵. Summarized below are some of the most commonly used pthread library calls with a brief description of their purpose. For more information, see appropriate documentation sources (e.g. man pages on any flavor of Unix systems⁶).

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                      const pthread_mutex_attr_t *mutexattr);
```

Arguments

mutex	address of the mutex variable to be initialized
mutexattr	address of the attribute variable used to initialize the mutex. see <code>pthread_mutexattr_init</code> for more information

Semantics

Each mutex variable must be declared (`pthread_mutex_t`) and initialized.


```
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *cond_attr);
```

Arguments

cond	address of the condition variable being initialized
cond_attr	address of the attribute variable used to initialize the condition variable. Not used in Linux.

Semantics

Each condition variable must be declared (`pthread_cond_t`) and initialized.


```
int pthread_create(pthread_t *thread,
                    pthread_attr_t *attr,
                    void *(*start_routine)(void *),
                    void *arg);
```

Arguments

thread	Address of the thread identifier (tid)
attr	Address of the attributes to be applied to the new thread
start_routine	Function that new thread will start executing
arg	address of first argument passed to <code>start_routine</code>

Semantics

This function will create a new thread; establish the starting address (passed as the name of a function) and pass arguments to be passed to the

⁵ Though Microsoft does not directly support the POSIX standard, the thread library available in WIN32 platforms for developing multithreaded applications in C have, for the most part, a semantically equivalent function call for each of the POSIX standard thread calls presented in this section.

⁶ For example see: <http://linux.die.net/man/>

function where the thread starts. The thread id (*tid*) of the newly created thread will be placed in the location pointed to by *thread*.

```
int pthread_kill(pthread_t thread,  
                  int signo);
```

Arguments

thread	thread id of the thread to which the signal will be sent
signo	signal number to send to thread

Semantics

Used to send a signal to a thread whose *tid* is known.

```
int pthread_join(pthread_t th,  
                  void **thread_return);7
```

Arguments

th	tid of thread to wait for
thread_return	If <i>thread_return</i> is not NULL, the return value of <i>th</i> is stored in the location pointed to by <i>thread_return</i> . The return value of <i>th</i> is either the argument it gave to <i>pthread_exit(3)</i> , or PTHREAD_CANCELED if <i>th</i> was cancelled.

Semantics

pthread_join suspends the execution of the calling thread until the thread identified by *th* terminates, either by calling *pthread_exit(3)* or by being cancelled.

```
pthread_t pthread_self(void);
```

Arguments

None

Semantics

pthread_self returns the thread identifier for the calling thread.

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Arguments

mutex	address of mutex variable to be locked
-------	--

Semantics

Waits until the specified mutex is unlocked and then locks it and returns.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Arguments

mutex	address of mutex variable to be unlocked
-------	--

Semantics

Unlocks the specified mutex if, the caller is the thread that locked the mutex.

⁷ The pthreads library also supports general barrier synchronization, which is especially useful in parallel scientific applications. The interested reader is referred to Unix reference source such as <http://linux.die.net/man/>

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

Arguments

cond	address of condition variable to wait upon.
mutex	address of mutex variable associated with cond

Semantics

pthread_cond_wait atomically unlocks the mutex (as per pthread_unlock_mutex) and waits for the condition variable cond to be signaled. The thread execution is suspended and does not consume any CPU time until the condition variable is signaled. The mutex must be locked by the calling thread on entrance to pthread_cond_wait. Before returning to the calling thread, pthread_cond_wait re-acquires mutex (as per pthread_lock_mutex).


```
int pthread_cond_signal(pthread_cond_t *cond);
```

Arguments

cond	address of condition variable
------	-------------------------------

Semantics

Wakes up one thread waiting for a signal on the specified condition variable.


```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Arguments

cond	address of condition variable
------	-------------------------------

Semantics

Variant of the previous call, wherein the signal wakes up *all* threads waiting for a signal on the specified condition variable.


```
void pthread_exit(void *retval);
```

Arguments

retval	address of the return value of the thread
--------	---

Semantics

Terminates the execution of the calling thread.

12.7 OS support for threads

In operating systems such as MS-DOS, an early bare bones OS for the PC, there is no separation between the user program and the kernel (Figure 12.15). Thus, the line between the user program and the kernel is imaginary and hence it costs very little (in terms of time, equivalent to a procedure call) to go between user and kernel spaces. The downside to this structure is the fact that there is no memory protection among user programs, and an errant or malicious program can easily corrupt the memory space of the kernel.

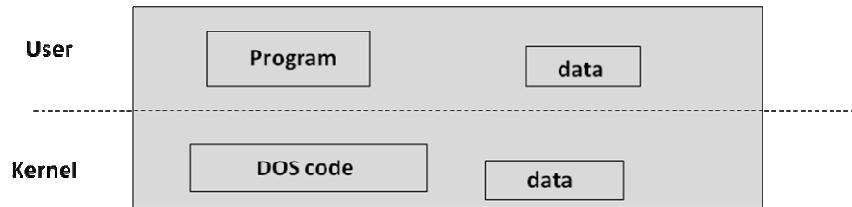


Figure 12.15: MS-DOS user-kernel boundary. MS-DOS was an early OS for the PC. The shading indicates that there is no enforced separation between user and kernel

Modern operating systems such as MS-Windows^{xp}, Linux, Mac OS X, and Unix provide true memory protection through virtual memory mechanisms we have discussed in earlier chapters. Figure 12.16 shows the memory spaces of user processes and the kernel in such operating systems.

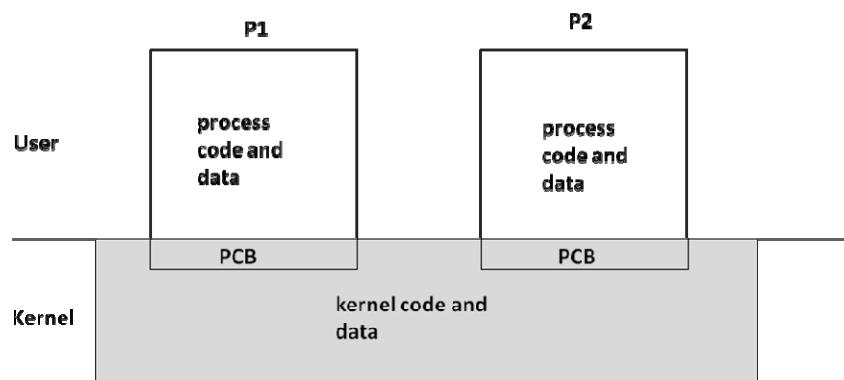


Figure 12.16: Memory protection in traditional operating systems. There is an enforced separation between the user and kernel

Every process is in its own address space. There is a clear dividing line between the user space and kernel space. System calls result in a change of protection domain. Now that we are familiar with memory hierarchy, we can see that the working set (which affects all the levels of the memory hierarchy from virtual memory to processor caches) changes on each such address space crossing. Consequently, frequent crossing of the boundary is detrimental to performance. A process control block (PCB) defines a particular process. In previous chapters, we have seen how the various components of the operating system such as the scheduler, memory system, and the I/O subsystem use the PCB. In a traditional operating system (i.e. one that is not multithreaded), the process is single-threaded. Hence, the PCB contains information for fully specifying the activity of this single thread on the processor (current PC value, stack pointer value, general-purpose register values, etc.). If a process makes a system call that blocks the process (e.g. read a file from the disk), then the program as a whole does not make any progress.

Most modern operating systems (Windows^{xp}, Sun Solaris, HP Tru64, etc.) are multithreaded. That is, the operating system recognizes that the state of a running program is a composite of the states of all its constituent threads. Figure 12.17 shows the distribution of memory spaces in modern operating systems supporting both single-

threaded as well as multithreaded programs. All threads of a given process share the address space of the process.

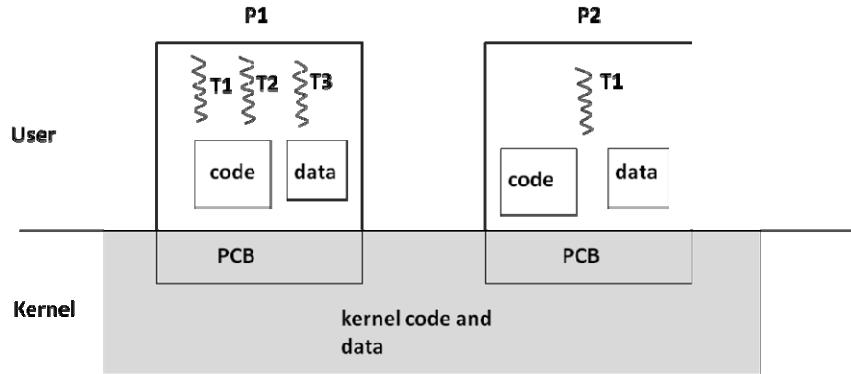


Figure 12.17: Memory protection in modern operating systems. A process may encompass multiple threads.

A given process may have multiple threads but since all the threads share the same address space, they share a common page table in memory. Let us elaborate on the computational state of a thread. A *thread control block (TCB)* contains all the state information pertaining to a thread. However, the information contained in a TCB is minimal compared to a PCB. In particular, the TCB contains the PC value, stack pointer value, and the GPR contents.

It is interesting to contrast the memory layout of multithreaded processes and single threaded processes (Figure 12.18). As we mentioned earlier, all the threads of a given process share the code, global data, and the heap. Thus, the stack is the only portion of memory that is unique for a particular thread. Due to the similarity in the visual picture of the stack layout of a multithreaded process to the cactus plant (Figure 12.18 (c)), we refer to this stack as a *cactus stack*.

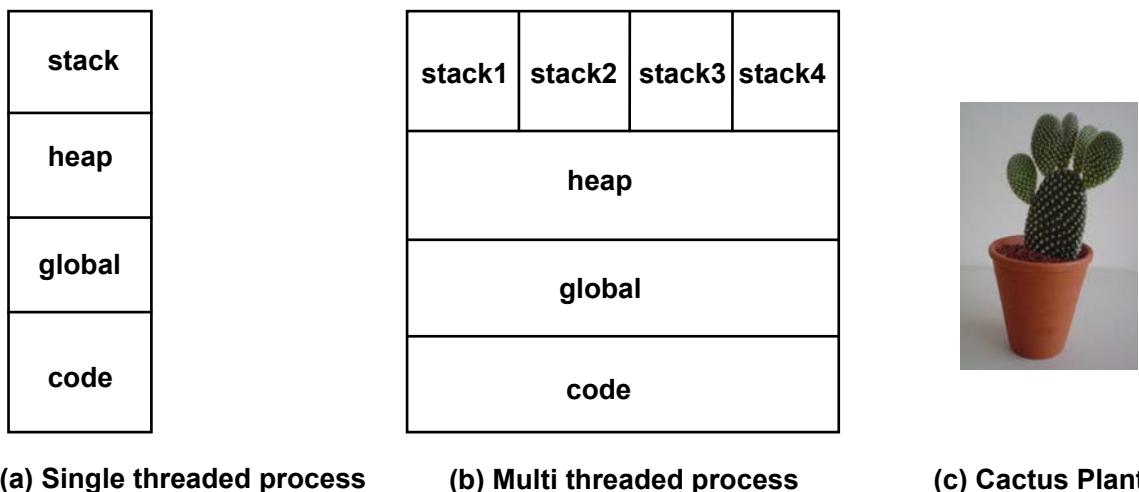


Figure 12.18: Memory layout of single threaded and multithreaded processes⁸

⁸ Picture source for cactus plant: unknown.

Next, we will see what it takes to implement a threads library: first at the user level and then at the kernel level.

12.7.1 User level threads

First, we will consider the implementation to be entirely at the user level. That is, the operating system only knows the existence of processes (which are single threaded). However, we could still have threads implemented at the user level. That is, the threads library exists as functionality above the operating system just as you have math libraries available for use in any program. The library provides thread creation calls we discussed earlier and supports data types such as **mutex** and **cond_var**, and operations on them. A program that wishes to use threads links in the threads library that becomes part of the program as shown in Figure 12.19.

The operating system maintains the traditional ready queue with the set of schedulable processes, which is the unit of scheduling at the operating system level. The threads library maintains a list of ready to run *threads* in each process with information about them in their respective *thread control blocks (TCBs)*. A TCB contains minimal information about each thread (PC value, SP value, and GPR values). Processes at the user level may be single threaded (e.g. P3) or multithreaded (P1 and P2).

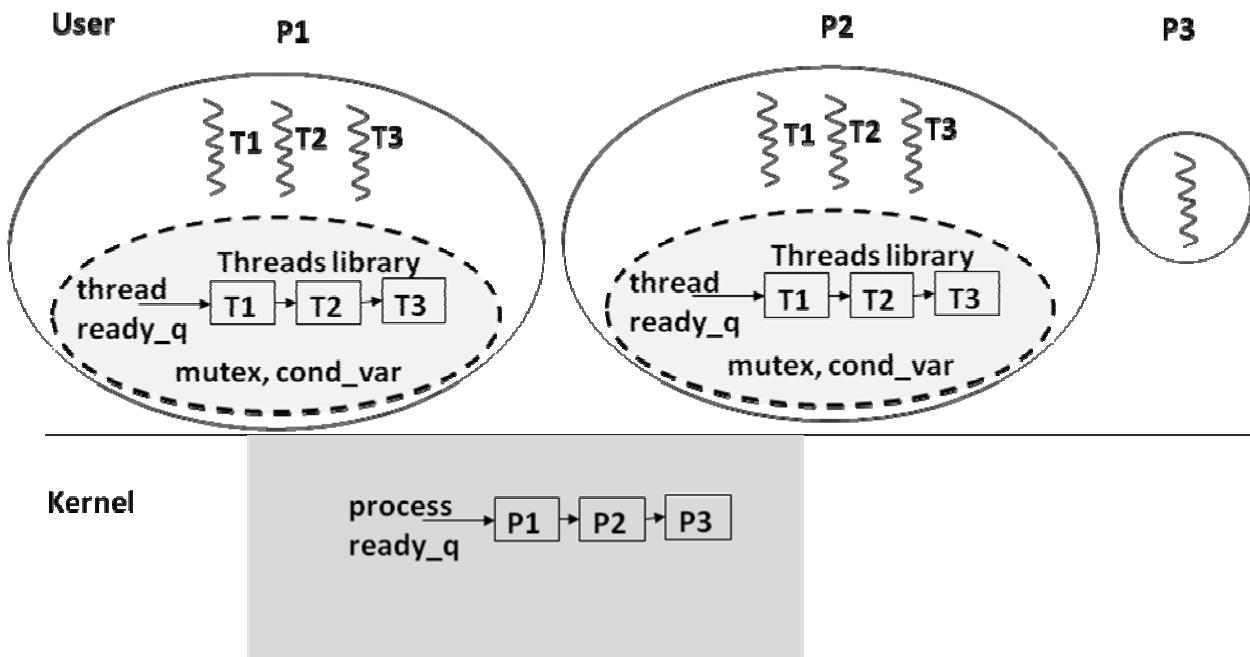


Figure 12.19: User level threads. Threads library is part of the application process's address space. In this sense, there is no enforced memory protection between the application logic and the threads library functionalities. On the other hand, there is an enforced separation between the user and the kernel.

The reader may wonder as to the utility of user level threads if process is the unit of scheduling by the operating system. Even if the underlying platform is a multiprocessor, the threads at the user level in a given process cannot execute concurrently. Recall, that thread is a structuring mechanism for constructing software. This is the primary purpose served by the user level threads. They operate as *co-routines*, that is, when one thread makes a thread synchronization call that blocks it, the thread scheduler picks some other thread within the same process to run. The operating system is oblivious to such thread level context switching that the thread scheduler performs using the TCBs. It is cheap to switch threads at the user level since it does not involve the operating system. The cost of a context switch approximates making a procedure call in a program. Thus, user level threads provide a structuring mechanism without the high cost of context switch involving the operating system. User level threads incur minimal direct and indirect cost for context switching (see Chapter 9.14 for details on direct and indirect costs). Further, thread level scheduler is customizable for a specific application.

It is interesting to understand what happens when one of the threads in a multithreaded process makes a blocking system call. In this case, the operating system blocks the whole process since it has no knowledge of other threads within the same process that are ready to run. This is one of the fundamental problems with user level threads. There are a number of different solution approaches to this problem:

1. One possibility is to wrap all OS calls with an envelope (for e.g. *fopen* becomes *thread_fopen*) that forces all the calls to go through the thread library. Therefore, when some thread (e.g. T1 of P1 in Figure 12.19) makes such a call, the thread library recognizes that issuing this call to the OS will block the whole process. Therefore, it defers making the call until all the threads in the process are unable to run anymore. At that point the library issues the blocking call to the OS on behalf of the thread.
2. A second approach is for an *upcall* mechanism (Figure 12.20) from the operating system that warns the thread scheduler that a thread in that process is about to make a blocking system call. This warning allows the thread scheduler (in the library) to perform a thread switch and/or defer the blocking call by the thread to a later more opportune time. Of course, this approach requires extension to the operating system for supporting such upcalls.

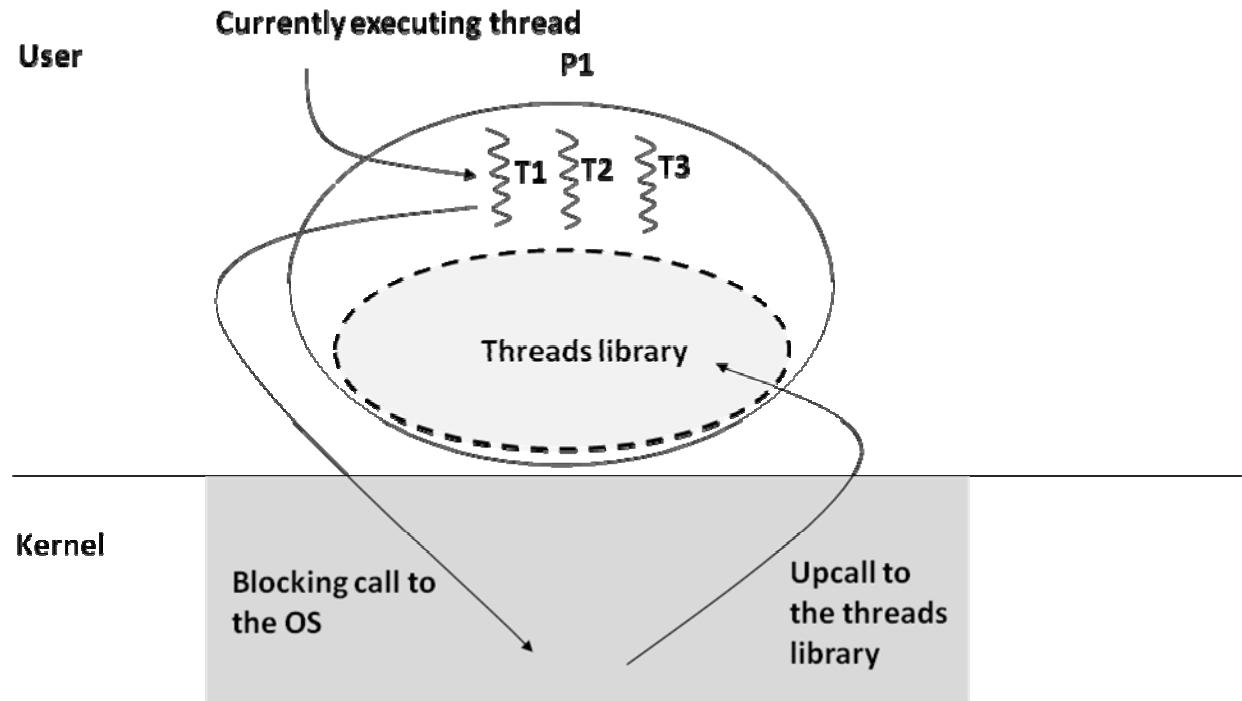


Figure 12.20: Upcall mechanism. Thread library registers a handler with the kernel. Thread makes a blocking call to the OS. The OS makes an upcall to the handler, thus alerting the threads library of the blocking system call made by a thread in that process.

In the chapter on CPU scheduling, we explored different CPU scheduling policies. Given that background, let us see how the thread scheduler switches among user level threads. Of course, if a thread makes a synchronization call into the threads library that is an opportune moment for the threads scheduler to switch among the threads of this process. Similarly, threads library may provide a *thread_yield* call to let a thread voluntarily give up the processor to give a chance for other threads in the same process to execute. One of the scheduling disciplines we studied in Chapter 6 is preemptive scheduling. If it is necessary to implement a preemptive thread scheduler at the user level, the thread scheduler can request a timer interrupt from the kernel and use that as a trigger to perform preemptive scheduling among the threads.

12.7.2 Kernel level threads

Let us understand the operating system support needed for implementing threads at the kernel level:

1. All the threads of a process live in a single address space. Therefore, the operating system should ensure that the threads of a process share the same page table.
2. Each thread needs its own stack, but share other portions of the memory footprint.
3. The operating system should support the thread level synchronization constructs discussed earlier.

First, let us consider a simple extension to the process level scheduler to support threads in the kernel. The operating system may implement a two-level scheduler as shown in Figure 12.21. A process level scheduler manages PCBs with information that is common

to all the threads in that process (page table, accounting information, etc.). A thread level scheduler manages TCBs. The process level scheduler allocates *time quanta* for processes, and performs preemptive scheduling among processes. Within a time quantum, the thread-level scheduler schedules the threads of that process in a round robin fashion or in a co-routine mode with the threads voluntarily yielding the processor. Since the operating system recognizes threads, it can switch among the threads of a process when the currently executing thread makes a blocking system call for I/O or thread synchronization.

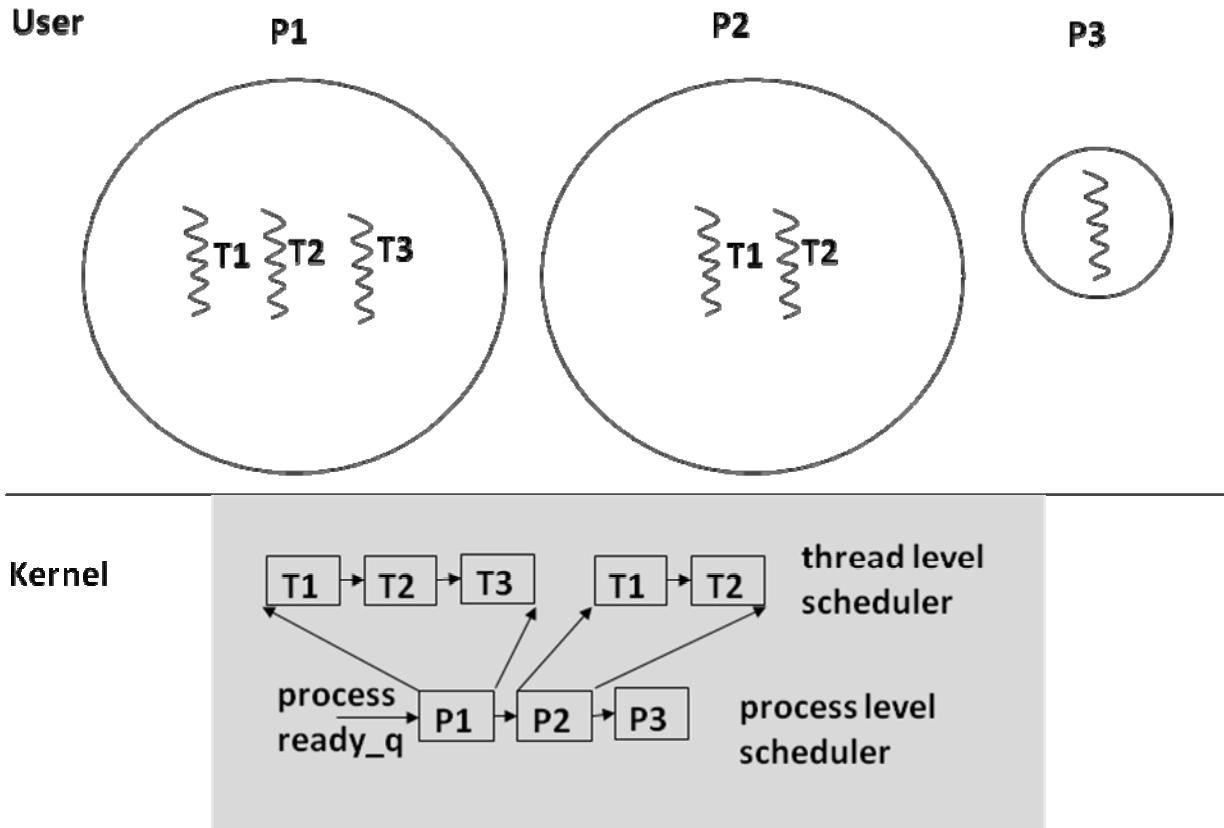


Figure 12.21: Kernel level threads. The process level scheduler uses the process ready_q. Even if a thread in the currently scheduled process makes a blocking system call, the OS uses the thread level scheduler to pick a ready thread from the currently running process to run for the remainder of the time quantum.

With computers and chips these days becoming multiprocessors, it is a serious limitation if the threads of a process cannot take advantage of the available hardware concurrency. The above structure allows threads of a given process to overlap I/O with processing, a definite step forward from user level threads. However, to fully exploit available hardware concurrency in a multiprocessor, a thread should be the unit of scheduling in the operating system. Next, we will discuss Sun Solaris threads as a concrete example of kernel level threads.

12.7.3 Solaris threads: An example of kernel level threads

Figure 12.22 shows the organization of threads in the Sun Solaris operating system.

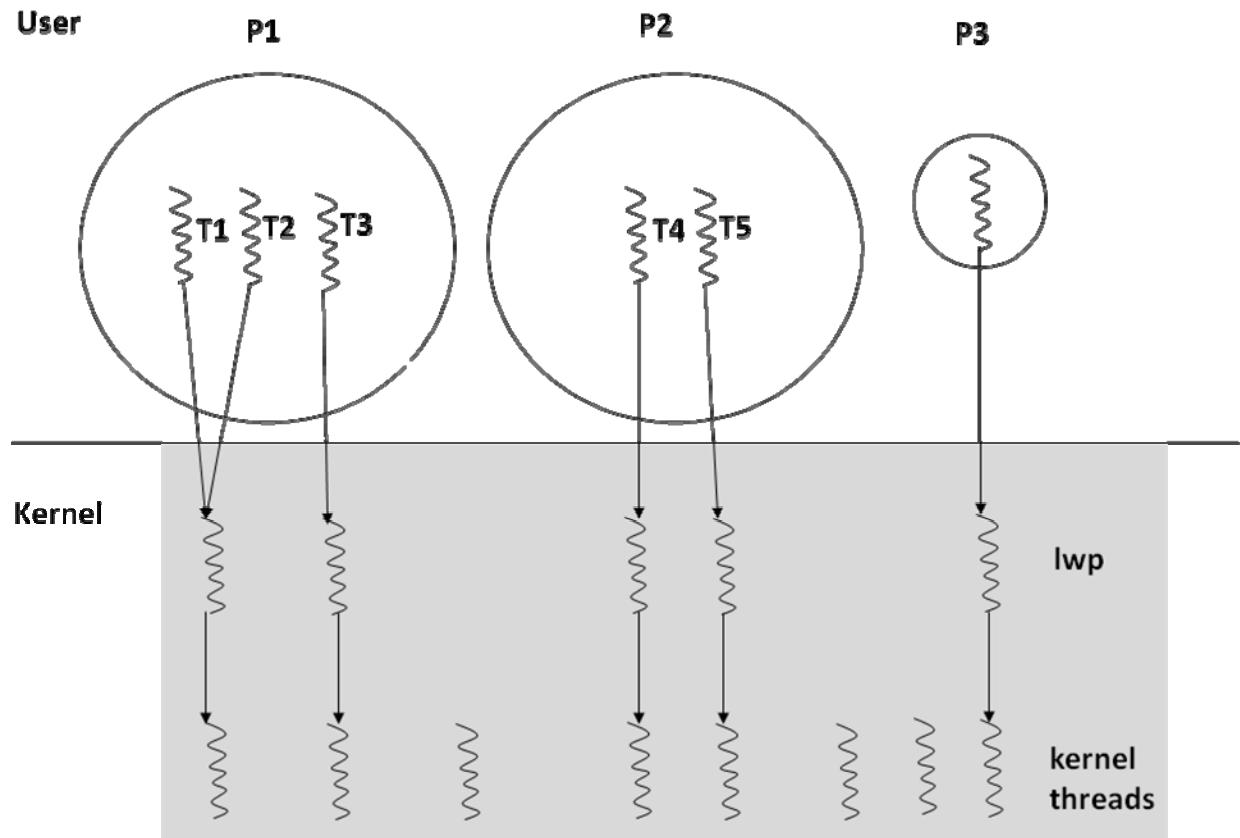


Figure 12.22: Sun Solaris threads structure. A thread in a process is bound to a light-weight process (lwp). Multiple threads of the same process may be bound to the same lwp. There is a one-to-one mapping between an lwp and a kernel thread. The unit of processor scheduling is a kernel thread.

A process is the entity that represents a program in execution. A process may create any number of user level threads that are contained within that process. The operating system allows the creator of threads to have control on the scheduling semantics of threads within that process. For example, threads may be truly concurrent or execute in a co-routine manner. To support these different semantics, the operating system recognizes three kinds of threads: *user*, *lightweight process (lwp)*, and *kernel*.

1. **Kernel:** The kernel threads are the unit of scheduling. We will see shortly its relation to the lwp and user threads.
2. **lwp:** An lwp is the representation of a process within the kernel. Every process upon startup is associated with a distinct lwp. There is a one-to-one association between an lwp and a kernel thread as shown in Figure 12.22. On the other hand, a kernel thread may be unattached to any lwp. The operating system uses such threads for functions that it has to carry out independent of user level processes. For example, kernel threads serve as vehicles for executing device specific functions.

3. User: As the name suggests, these threads are at the user level.

Thread creation calls supported by the operating system result in the creation of user level threads. The thread creation call specifies if the newly thread should be attached to an existing lwp of the process or allocated to a new lwp. See for example the threads T1 and T2 in process P1. They both execute in a co-routine manner due to being bound to the same lwp. Any number of user level threads may bind themselves to an lwp. On the other hand, thread T3 of P1 executes concurrently with one of T1 or T2. Threads T4 and T5 of P2 execute concurrently as well.

The ready queue of the scheduler is the set of kernel threads that are ready to run. Some of them, due to their binding to an lwp result in executing a user thread or a process. If a kernel thread blocks, the associated lwp and therefore the user level thread blocks as well. Since the unit of scheduling is a kernel thread, the operating system has the ability to schedule these threads concurrently on parallel processors if the underlying platform is a multiprocessor.

It is interesting to understand the inherent cost of a thread switch given this structure. Remember that every context switch is between one kernel thread to another. However, the cost of the switch changes dramatically depending on what is bound to a kernel thread. The cheapest form of context switch is between two user level threads bound to the same lwp (T1 and T2 in Figure 12.22). Presumably, the process has a thread library available at the user level. Thus, the thread switch is entirely at the user level (similar to user level threads we discussed earlier). Switching between lwps in the same process (for example T1 and T3 in Figure 12.22) is the next higher cost context switch. In this case, the direct cost is a trip through the kernel that performs the switch (in terms of saving and loading TCBs). There are no hidden costs in the switch due to memory hierarchy effects since both the threads are in the same process. The most expensive context switch is between two lwps that are in different processes (for example T3 and T4 in Figure 12.22). In this case, there are both direct and indirect costs involved since the threads are in different processes.

12.7.4 Threads and libraries

Irrespective of the choice of implementing threads at the user or the kernel level, it is imperative to ensure the safety of libraries that multithreaded programs use. For example, all the threads of a process share the heap. Therefore, the library that supports dynamic memory allocation needs to recognize that threads may request memory from the heap simultaneously. It is usual to have *thread-safe* wrappers around such library calls to ensure atomicity for such calls in anticipation of concurrent calls from the threads. Figure 12.23 shows an example. The library call implicitly acquires a mutual exclusion lock on behalf of the thread that makes that call.

```

/* original version */      | /* thread safe version */
|                         |
|                         | mutex_lock_type cs_mutex;
void *malloc(size_t size) | void *malloc(size_t size)
{                         |
|                         | {
|                         |     thread_mutex_lock(cs_mutex);
|
|     .....               |     .....
|     .....               |     .....
|                         |     thread_mutex_unlock(cs_mutex);
|
|     return(memory_pointer); |     return (memory_pointer);
}                         | }

```

Figure 12.23: Thread safe wrapper for library calls

12.8 Hardware support for multithreading in a uniprocessor

Let us understand what is required in hardware for supporting multithreading. There are three things to consider:

1. Thread creation and termination
2. Communication among threads
3. Synchronization among threads

12.8.1 Thread creation, termination, and communication among threads

First, let us consider a uniprocessor. Threads of a process share the same page table. In a uniprocessor each process has a unique page table. On a thread context switch within the same process, there is no change to the TLB or the caches since all the memory mapping and contents of the caches remain relevant for the new thread. Thus, creation and termination of threads, or communication among the threads do not require any special hardware support.

12.8.2 Inter-thread synchronization

Let us consider what it takes to implement the *mutual exclusion lock*. We will use a memory location **mem_lock** initialized to zero. The semantics are as follows. If **mem_lock** is zero, then the lock is available. If **mem_lock** is 1, then some thread currently has the lock. Here are the algorithms for **lock** and **unlock**.

```

Lock:
    if (mem_lock == 0)
        mem_lock = 1;
    else
        block the thread;

Unlock:
    mem_lock = 0;

```

The lock and unlock algorithms have to be *atomic*. Let us examine these algorithms. Unlock algorithm is a single memory store instruction of the processor. Assuming that each instruction is atomic, unlock is atomic.

The datapath actions necessary to implement the lock algorithm are as follows:

- Read a memory location
- Test if the value read is 0
- Set the memory location to 1
-

12.8.3 An atomic test-and-set instruction

We know that LC-2200 (see Chapter 2) does not provide any single instruction that performs the above datapath actions. Therefore, to make the lock algorithm atomic, we introduce a new instruction:

Test-And-Set memory-location

The semantics of this instruction is as follows:

- Read the current value of memory-location into some processor register
- Set the memory-location to a 1

The key point of this instruction is that if a thread executes this instruction, the above two actions (getting the current value of the memory location and setting the new value to 1) happen *atomically*, i.e., no other instruction (by any other thread) intervenes the execution of **Test-and-set**.

Example 14:

Given the following procedure called **binary-semaphore**:

```
static      int shared-lock = 0; /* global variable to
                                both T1 and T2 */
/* shared procedure for T1 and T2 */
int binary-semaphore(int L)
{
    int x;

    x = test-and-set (L);

    /* x = 0 for successful return */
    return(x);
}
```

Two threads **T1** and **T2** execute the following statement simultaneously:

MyX = binary_semaphore(shared-lock);

where **MyX** is a local variable in each of **T1** and **T2**.

What are the possible values returned to T1 and T2?

Answer:

Note that the instruction **test-and-set** is atomic. Therefore, although T1 and T2 execute the procedure simultaneously, the semantics of this instruction ensures that one or the other (whichever happens to be the winner) gets to execute the instruction first.

So, possible outcomes:

- 1) T1 is the winner.**
T1's **MyX** = 0; T2's **MyX** = 1
- 2) T2 is the winner.**
T1's **MyX** = 1; T2's **MyX** = 0

Note that it will never be the case that **both** T1 and T2 will get a 0 or 1 as the return value.

You may have seen and heard of the *semaphore* signaling system used extensively in railroads. In olden times (and even to this day in some developing countries), mechanical arms (see Figure 12.24) on a high pole cautioned an engine driver in a train to either stop or proceed with caution when approaching shared railroad tracks.



Figure 12.24: Railroad Semaphore⁹

Computer scientists have borrowed that term, semaphore, into computer science parlance. The procedure shown in Example 13 is a *binary semaphore*, i.e., it signals one among many threads that it is safe to proceed into a critical section.

Edsger Dijkstra, a well-known computer scientist of Dutch origin, was the proponent of semaphore as a synchronization mechanism for coordinating the activities of concurrent threads. He proposed two versions of this semaphore. Binary semaphore is the one we just saw wherein the semaphore grants or denies access to a single resource to a set of competing threads. *Counting semaphore* is a more general version wherein there are n

⁹ Picture source: <http://www.stuorg.iastate.edu/railroad/images/FortDodge03/120703-UP2085nose.JPG>

instances of a given resource; the semaphore grants or denies access to **an** instance of these n resources to competing threads. At most n threads can enjoy these resources **simultaneously** at any point of time.

12.8.4 Lock algorithm with test-and-set instruction

Having introduced the atomic **test-And-Set** instruction, we are now ready to review the implementation of the mutual exclusion lock primitive, which is at the core of programming support for multithreaded applications. As it turns out, we can implement the lock and unlock algorithm building on the binary semaphore as follows.

```
#define SUCCESS 0
#define FAILURE 1

int lock(int L)
{
    int x;
    while ( (x = test-and-set (L)) == FAILURE ) {
        /* current value of L is 1
         * implying that the lock is
         * currently in use
        */
        block the thread;

        /* the threads library puts the
         * the thread in a queue; when
         * lock is released it allows
         * this thread to check the
         * availability of the lock again
        */
    }

    /* falling out of the while loop implies that
     * the lock attempt was successful
    */

    return(SUCCESS);
}

int unlock(int L)
{
    L = 0;
    return(SUCCESS);
}
```

By design, a thread calling the lock algorithm has the lock when it returns. Using this basic lock and unlock algorithm, we can build the synchronization primitives we discussed in Section 12.3 and the POSIX thread library in Section 12.6.

Therefore, the minimal hardware support for multithreading is an atomic **Test-And-Set** (**TAS** for short) instruction. The key property of this instruction is that it *atomically reads, modifies, and writes* a memory location. There are other instructions that implement the same property. The point is that most modern processor architectures include one or more instructions in its repertoire that have this property.

Note that if the operating system deals with threads directly, then it can simply turn off interrupts while executing the lock algorithm to ensure atomicity. The **TAS** instruction allows implementing locks at the **user** level.

12.9 Multiprocessors

As the name suggest, a multiprocessor consists of multiple processors in a single computer sharing all the resources such as memory, bus, and input/output devices (see Figure 12.25). This is a *Symmetric multiprocessor (SMP)* since all the processors have an identical view of the system resources. An SMP is a cost effective approach to increasing the system performance at a nominal increase in total system cost. Many servers that we use on an everyday basis (web server, file server, mail server) run on 4-way or 8-way SMPs.

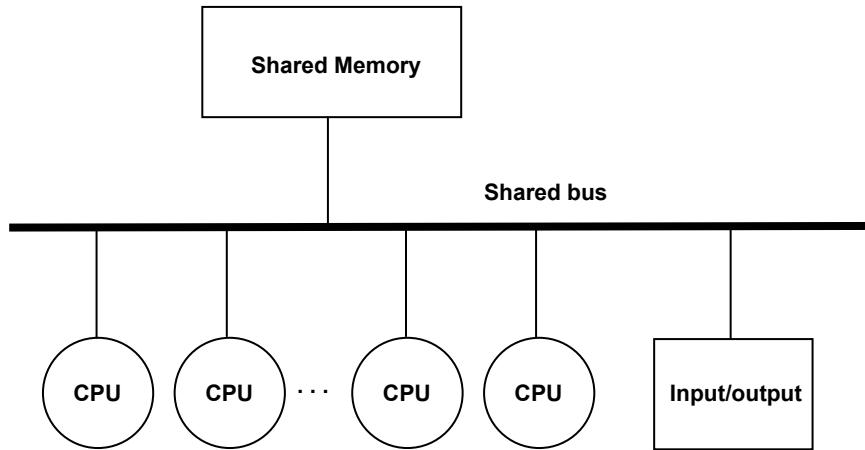


Figure 12.25: A Symmetric Multiprocessor (SMP)

Complication arises at the system level if the program is running on a multiprocessor. In this case, the threads of a given program may be on different physical processors. Thus, the system software (meaning the operating system and runtime libraries) and the hardware have to work in partnership to provide the semantics expected at the user program level for data structures shared by the threads.

We have seen that there are several intricacies associated with preserving the semantics of a sequential program with the hardware and software entities even in a single processor: TLB, page tables, caches, and memory manager just to name a few. The reader

can envision the complications that arise when the system software and hardware have to preserve the semantics of a multithreaded program. We will discuss these issues in this section.

The system (hardware and the operating system together) has to ensure three things:

1. Threads of the same process share the same page table.
2. Threads of the same process have identical views of the memory hierarchy despite being on different physical processors.
3. Threads are guaranteed atomicity for synchronization operations while executing concurrently.

12.9.1 Page tables

The processors share physical memory as shown in Figure 12.25. Therefore, the operating system satisfies the first requirement by ensuring that the page table in shared memory is the same for all the threads of a given process. However, there are a number of issues associated with an operating system for a multiprocessor. In principle, each processor *independently* executes the *same* operating system. However, they coordinate some of their decisions for overall system integrity. In particular, they take specific coordinated actions to preserve the semantics of multithreaded programs. These include scheduling threads of the same process simultaneously on different processors, page replacement, and maintaining the consistency of the TLB entries in each CPU. Such issues are beyond the scope of this discussion. Suffice it to say, these are fascinating issues and the reader is encouraged to take advanced courses in operating systems to study them.

12.9.2 Memory hierarchy

Each CPU has its own TLB and cache. As we said earlier, the operating system worries about the consistency of TLBs to ensure that all threads have the same view of the shared process address space. The hardware manages the caches. Each per-processor cache may currently be encaching the same memory location. Therefore, the hardware is responsible for maintaining a consistent view of shared memory that may be encached in the per-processor caches (see Figure 12.26). We refer to this problem as *multiprocessor cache coherence*.

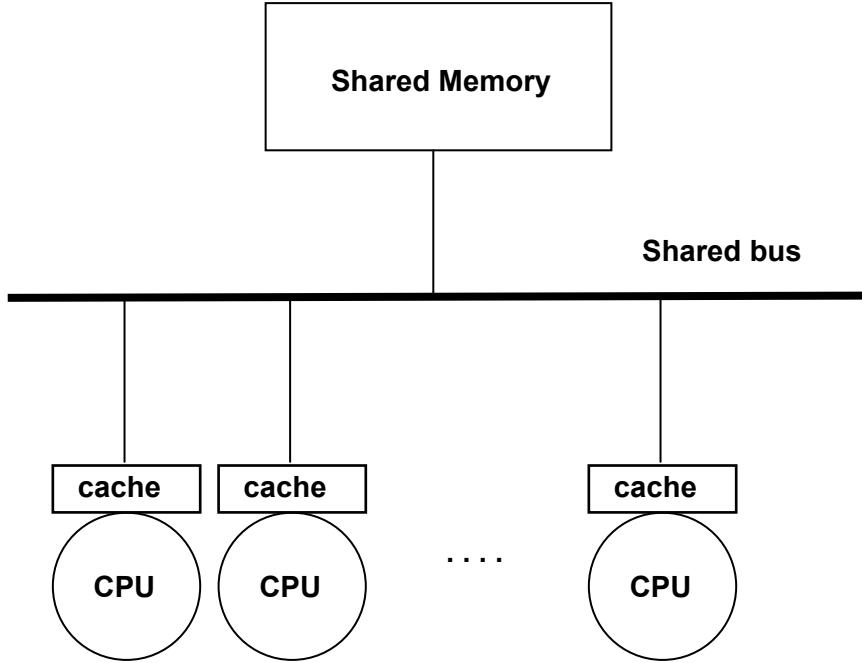


Figure 12.26: SMP with per-processor caches

Figure 12.27 illustrates the cache coherence problem. Threads T1, T2, and T3 (of the same process) execute on processors P1, P2, and P3, respectively. All three of them currently have location X cached in their respective caches (Figure 12.27-a). T1 writes to X. At this point, the hardware has one of two choices:

- It can *invalidate* copies of X in the peer caches as shown in Figure 12.27-b. This requires enhancement to the shared bus in the form of an *invalidation line*. Correspondingly, the caches *monitor* the bus by *snooping* on it to watch out for invalidation requests from peer caches. Upon such a request on the bus, every cache checks if this location is cached locally; if it is then it invalidates that location. Subsequent misses for the same location is either satisfied by the cache that has the most up to date copy (for example P1 in this example) or by the memory, depending on the write policy used. We refer to this solution as the *write-invalidate* protocol.
- It can *update* copies of X in the peer caches as shown in Figure 12.27-c. This may manifest simply as a memory write on the bus. The peer caches that *observe* this bus request update their copies of X (if present in the cache). We refer to this solution as the *write-update* protocol.

Snoopy caches is a popular term used for bus-based cache coherence protocols. In this section, we have presented a very basic intuition to the multiprocessor cache coherence problem and snoopy cache solution approaches to the problem. Snoopy caches do not work if the processors do not have a shared bus (a broadcast medium) to snoop on. In Section 12.10.2.4, we will discuss a different solution approach called *directory-based scheme* that does not rely on a shared bus for communication among the processors. Investigation of scalable solution approaches to the cache coherence problem was a fertile area of research in the mid to late 80's and resulted in several doctoral

dissertations. The reader is encouraged to take advanced courses in computer architecture to learn more on this topic.

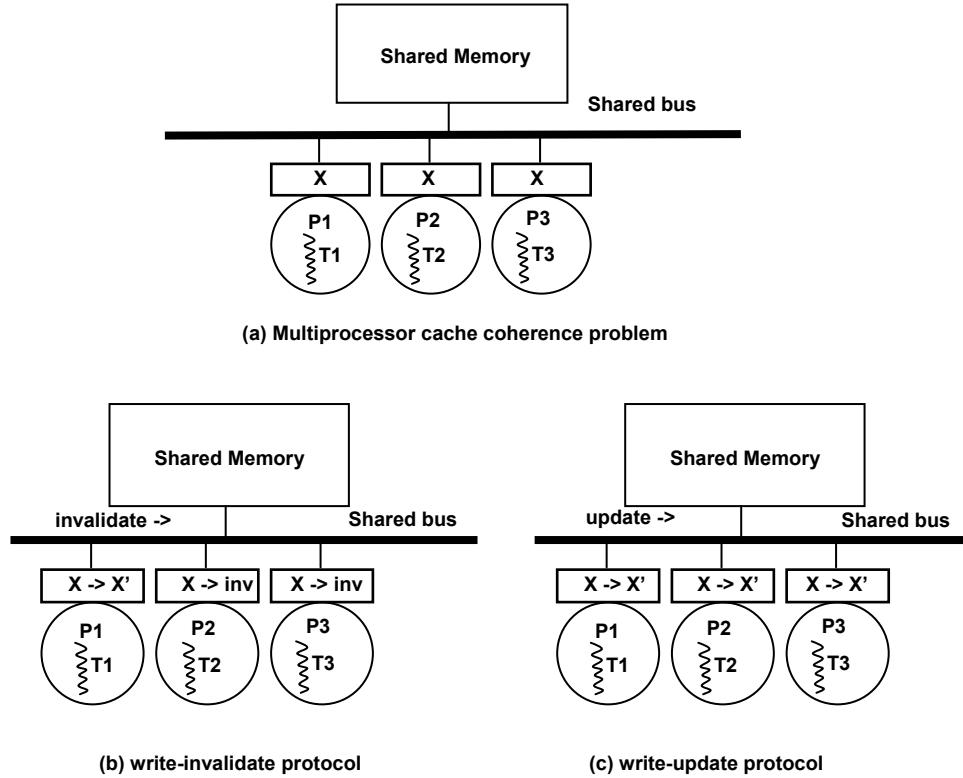


Figure 12.27: Multiprocessor cache coherence problem and solutions

Example 15:

Given the following details about an SMP (symmetric multiprocessor):

Cache coherence protocol:	write-invalidate
Cache to memory policy:	write-back

Initially:

The caches are empty

Memory locations:

A contains 10

B contains 5

Consider the following timeline of memory accesses from processors P1, P2, and P3.

Time (in increasing order)	Processor P1	Processor P2	Processor P3
T1	Load A		
T2		Load A	
T3			Load A
T4		Store #40, A	
T5	Store #30, B		

Using the table below, summarize the activities and the values in the caches.

Answer:

(I indicates the cache location is invalid. NP indicates not present)

Time	Variables	Cache of P1	Cache of P2	Cache of P3	Memory
T1	A	10	NP	NP	10
T2	A	10	10	NP	10
T3	A	10	10	10	10
T4	A	I	40	I	10
T5	A B	I 30	40 NP	I NP	5

12.9.3 Ensuring atomicity

Since the CPUs share memory, the lock and unlock algorithms, we presented in Chapter 12.8 work fine in a multiprocessor. The key requirement is ensuring atomicity of these algorithms in the presence of concurrent execution of the threads in different processors. An instruction such as TAS introduced in Chapter 12.8 that has the ability to atomically *read-modify-write* a shared memory location serves this purpose.

12.10 Advanced Topics

We will introduce the reader to some advanced topics of special relevance in the context of multiprocessors and multithreading.

12.10.1 OS topics

12.10.1.1 Deadlocks

In Section 12.2.7, we introduced the concept of deadlocks and livelocks. We will generalize and expand on the concept of deadlocks. We gave a very simple and intuitive definition of deadlock as a situation where in a thread is waiting for an event that will never happen. There are several reasons that lead to deadlocks in a computer system. One reason is that there are concurrent activities in the system, and there are finite resources be they hardware or software. For example, let us consider a uniprocessor running multiple applications. If the scheduler uses a non-preemptive algorithm, and the application currently running on the processor goes into an infinite loop, all the other applications are deadlocked. In this case, the processes are all waiting for a physical resource, namely, a processor. This kind of deadlock is usually referred to as a *resource deadlock*. The conditions that led to the deadlock are two-fold: the need for *mutual exclusion* for accessing the shared resource (i.e., the processor) and the *lack of preemption* for yanking the resource away from the current user.

A similar situation arises due to locks governing different data structures of a complex application. Let us first consider a fun analogy. Nick and his partner arrive in the recreation center to play squash. There is only one court and exactly two rackets in the recreation center. There is one service desk to check out rackets and another to claim the court. Nick and his partner go first to the former and check out the rackets. Alex and his partner have similar plans. However, they first claim the court and then go to the service desk to check out the rackets. Now Nick and Alex are deadlocked. This is also a resource deadlock. In addition to the two conditions we identified earlier, there are two conditions that led to the deadlock in this case: *circular wait* (Alex is waiting for Nick to give up the rackets, and Nick is waiting for Alex to give up the court), and the fact that each of them can *hold a resource and wait for another one*. Complex software systems use fine-grain locking to enhance the opportunity for concurrent execution of threads. For example, consider payroll processing. A check issuing process may be locking the records of all the employees to generate the paychecks; while a merit-raise process may be sweeping through the database locking all the employee records to add raises to all the employees at the same time. *Hold and wait*, and *circular wait* by each of these processes will lead to a deadlock.

To sum up there are four conditions that *must hold simultaneously* for processes to be involved in resource deadlock in a computer system:

- **Mutual exclusion:** a resource can be used only in a mutually exclusive manner
- **No preemption:** the process holding a resource has to give it up voluntarily
- **Hold and wait:** a process is allowed to hold a resource while waiting for other resources
- **Circular wait:** there is a cyclic dependency among the processes waiting for resources (A is waiting for resource held by B; B is waiting for a resource held by C; C....X; X is waiting for a resource held by A)

These are called *necessary* conditions for a deadlock. There are three strategies for dealing with deadlocks. Deadlock *avoidance*, *prevention*, and *detection*. A reader interested in detailed treatment of these strategies is referred to advanced textbooks in Operating Systems¹⁰. Here we will give the basic intuition behind these strategies. Deadlock avoidance algorithm is ultra-conservative. It basically assumes that the request pattern for resources are all known *a priori*. With this global knowledge, the algorithm will make resource allocation decisions that are guaranteed to never result in a deadlock. For example, if you have \$100 in hand, and you know that in the worst case you need a maximum of \$80 to see you through the rest of the month, you will be able to loan \$20 to help out a friend. If your friend needs \$30, you will say no since you could potentially get into a situation where you don't have enough to see you through the rest of the month. However, it could turn out that this month you get some free lunches and dinners and may end up not needing \$80. So, you are making a conservative decision as to how much you can loan a friend based on the worst case scenario. As you probably guessed, deadlock avoidance will lead to poor resource utilization due to its inherent conservatism.

More importantly, deadlock avoidance is simply not practical since it requires prior knowledge of future resource requests. A better strategy is deadlock prevention, which goes after each of the four *necessary* conditions for deadlock. The basic idea is to break one of the necessary conditions and thus *prevent* the system from deadlocking. Using the same analogy, you could loan your friend \$30. However, if it turns out that this month you do need \$80 to survive, you go to your friend and get back \$10 out of the \$30 you loaned him. Basically, this strategy breaks the necessary condition "no preemption." Of course, the same prevention strategy may not be applicable for all types of resources. For example, if processes need to access a single shared physical resource in a mutually exclusive manner, then one way to avoid deadlock is to pretend as though there are as many instances of that shared resource as the number of requestors. This may seem like a crazy idea but if you think about it, this is precisely how a departmental printer is shared. Basically, we spool our print jobs that are then buffered awaiting the physical printer. Essentially, "spooling" or "buffering" is a way to break the necessary condition "mutual exclusion". Similarly, to break the necessary condition, "hold and wait," we can mandate that all resources need to be obtained simultaneously before starting the process. Returning to our squash player analogy, Nick (or Alex) would have to get both the court and the rackets together, not one after another. Lastly, to break the necessary condition "circular wait" we could order the resources and mandate that the requests be made *in order*. For example, we could mandate that you first have to claim the squash court (resource #1), before you request the rackets (resource #2). This would ensure that there will not be any circular wait.

Deadlock prevention leads to better resource utilization compared to avoidance. However, it is still conservative. For example, consider mandating that a process get all the resources *a priori* before it starts. This for sure prevents deadlock. However, if the process does not need all the resources for the entire duration, then the resources are getting under-utilized. Therefore, a more liberal policy is deadlock detection and recovery. The idea is to be liberal with resource requests and grants. However, if a

¹⁰ A. S. Tannenbaum, "Modern Operating Systems," Prentice-Hall.

deadlock does occur, have a mechanism for detecting it and recovering from it. Returning to our squash players analogy, when the desk clerk at the court claim counter notices the deadlock, she takes the racket from Nick, calls up her cohort at the racket claim counter, and tells her to send Alex over to her counter to resolve the deadlock.

Example 16 :

Consider a system with three resources:

1 display, 1 KBD, 1 printer.

There are four processes:

P1 needs all three resources
P2 needs KBD
P3 needs display
P4 need KBD and display.

Explain how deadlock avoidance, prevention, and detection will work to cater to the needs of the four processes, respectively.

Answer:

Let us consider solutions for each strategy.

Avoidance: Allocate all needed resources as a bundle at the start to a process. Essentially, this amounts to not starting P2, P3, or P4 if P1 is running; not starting P1 if any of the others are running.

Prevention: Have an artificial ordering of the resources, say KBD, display, printer. Make the processes always request the three resources in the above order and release all the resources a process is holding at the same time upon completion. This ensures no circular wait (P4 cannot be holding display and ask for a KBD; P1 cannot ask for printer without already having the display, etc).

Detection: Allow resources to be requested individually and in any order. We will assume all processes are re-startable. If a process (P2) requests a resource (say KBD), which is currently assigned to another process (P4) and if P4 is waiting for another resource, then force a release of KBD by aborting P4, assign the KBD to P2, and restart P4.

A topic that is closely related to deadlocks in the context of resource allocation is *starvation*. This is the situation wherein some process is indefinitely blocked awaiting a resource. For example, if the resource allocation uses some sort of a priority scheme, then lower priority processes may be starved, if there is a steady stream of higher priority processes making requests for the same resource. Returning to our squash court example, consider the situation where faculty member are given priority over students.

This could lead to starvation of students. We give another example of starvation when we discuss classic problems in synchronization (see Section 12.10.1.4).

Beyond resource deadlocks, computer systems are susceptible to other kinds of deadlocks as well. Specifically, the kinds of deadlocks we discussed earlier in this chapter have to do with errors in writing correct parallel program that could lead to deadlocks and livelocks. This problem gets exacerbated in distributed systems (see Chapter 13), where messages may be lost in transit due to a variety of reasons, leading to deadlocks. All of these can be lumped under the heading of *communication deadlocks*.

12.10.1.2 Advanced synchronization algorithms

In this chapter, we studied basic synchronization constructs. Such constructs have been incorporated into IEEE standards such as POSIX threading library. As we observed earlier, these libraries or their variants have been incorporated into almost all modern operating systems. Most application software for parallel systems is built using such multithreading libraries.

Programming mutual exclusion locks and condition variables is error-prone. The primary reason is that the logic of synchronized access to shared data structures is strewn all over the program and makes it hard from a software-engineering point of view. This has implications on the design, development, and maintenance of large complex parallel programs.

We can boil down the needs of concurrent programming to three things:

- (1) an ability for a thread to execute some sections of the program (which we called critical sections in Section 12.2.4) in a mutually exclusive manner (i.e., serially),
- (2) an ability for a thread to wait if some condition is not satisfied, and
- (3) an ability for a thread to notify a peer thread who may be waiting for some condition to become true.

Monitor is a programming construct proposed by Brinch Hansen and Tony Hoare in the 70's that meet the above needs. Essentially, a monitor is abstract data type that contains the data structures and procedures for manipulating these data structures. In terms of modern programming languages such as Java or C++, one can think of the monitor as syntactically similar to an object. Let us understand the difference between a Java object and a monitor. The principal difference is that there can be exactly one active thread inside a monitor at any point of time. In other words, if you have a need for a critical section in your program, you will write that portion of the program as a monitor. A program structure that requires multiple independent critical sections (as we saw in Example 7), will be constructed using multiple monitors, each with a distinct name for each of the critical sections. A thread inside the monitor may have to block for a resource. For this purpose the monitor provides *condition variables*, with two operations *wait* and *notify* on such a variable. The reader can see an immediate parallel to the condition variable inside a monitor and the condition variable available in the pthreads library. The monitor construct meets all the three needs for writing concurrent programs that we laid out above. To verify this is true, we will do an example.

Example 17:

Write a solution using monitors for the video processing example developed in this chapter.

Answer:

The digitizer and tracker codes are written assuming the existence of a monitor called FrameBuffer. The procedures grab and analyze used by the digitizer and tracker, respectively, are outside the monitor.

```
digitizer()
{
    image_type dig_image;

    loop {
        grab(dig_image);
        FrameBuffer.insert(dig_image);
    }
}

tracker()
{
    image_type track_image;

    loop {
        FrameBuffer.remove_image(&track_image);
        analyze(track_image);
    }
}
```

```

monitor FrameBuffer
{
#define MAX 100

image_type frame_buf[MAX];
int bufavail = MAX;
int head = 0, tail = 0;

condition not_full, not_empty;

void insert_image(image_type image)
{
    if (bufavail == 0)
        wait(not_full);
    frame_buf[tail mod MAX] = image;
    tail = tail + 1;
    bufavail = bufavail - 1;
    if (bufavail == (MAX-1)) {
        /* tracker could be waiting */
        notify(not_empty);
    }
}

void remove_image(image_type *image)
{
    if (bufavail == MAX)
        wait(not_empty);
    *image = frame_buf[head mod MAX];
    head = head + 1;
    bufavail = bufavail + 1;
    if (bufavail == 1) {
        /* digitizer could be waiting */
        notify(not_full);
    }
}

} /* end monitor */

```

A number of things are worth commenting on the solution presented in Example 17. The most important point to note is that all the details of synchronization and buffer management that were originally strewn in the digitizer and tracker procedures in the pthreads version are now nicely tucked away inside the monitor named **FrameBuffer**. This simplifies the tracker and digitizer procedures to be just what is intuitively needed for those functions. This also ensures that the resulting program will be less error-prone

compared to sprinkling synchronization constructs all over the program. Another elegant aspect of this solution is that there can be any number of digitizer and tracker threads in the application. Due to the semantics of the monitor construct (mutual exclusion), all the calls into the monitor from these various threads get serialized. Overall, the monitor construct adds significantly to the software-engineering of parallel programs.

The reader may be wondering if the monitor is such a nice construct why we are not using it. The main catch is that it is a programming construct. In the above example, we have written the monitor using a C-style syntax to be compatible with the earlier solutions of the video processing application developed in this chapter. However, C does not support the monitor construct. One could of course “simulate” the monitor construct by writing the monitor using facilities available in the operating system (e.g., pthreads library, please see Exercise 16).

Some programming languages have adopted the essence of the monitor idea. For example, Java is an object-oriented programming language. It supports user-level threads and allows methods (i.e., procedures) to be grouped together into what are called classes. By prefixing a method with the keyword “synchronized,” the Java runtime will ensure that exactly one user-level thread is able to execute a synchronized method at any time in a given object. In other words, as soon as a thread starts executing a synchronized method, no other thread will be allowed to another synchronized method within the same object. Other methods that do not have the keyword synchronized may of course be executed concurrently with the single execution of a synchronized method. While Java does not have an in-built data type similar to the monitor’s condition variable, it does provide wait and notify functions that allow blocking and resumption of a thread inside a synchronized method (please see Exercise 17).

12.10.1.3 Scheduling in a Multiprocessor

In Chapter 6, we covered several processor scheduling algorithms. All of these apply to parallel systems as well. However, with multiple processors the scheduler has an opportunity to run multiple threads of the same application or threads of different applications. This gives rise to some interesting options.

The simplest way to coordinate the scheduling of the threads on the different processors is to have a single data structure (a run queue) that is shared by the scheduler on each of the processors (Figure 12.28). This also ensures that all the processors share the computational load (in terms of the runnable threads) equally.

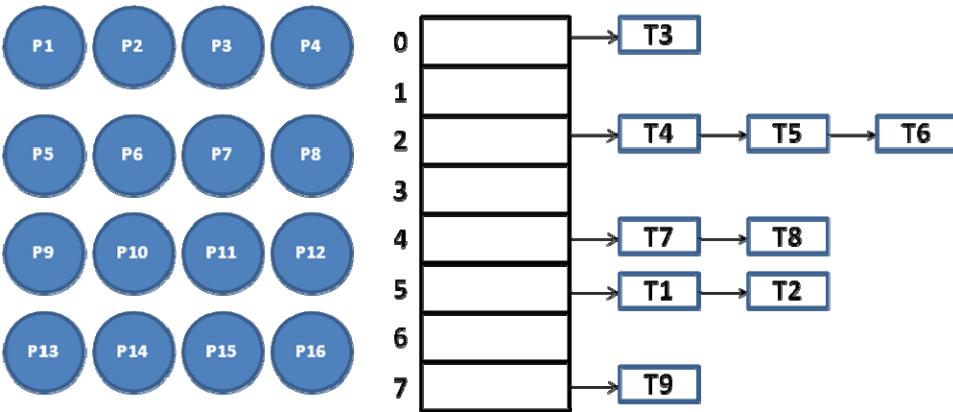


Figure 12.28: Shared scheduling queue with different priority levels

There are some downsides to this approach. The first issue is pollution of the memory hierarchy. On modern processors with multiple levels of caches, there is significant time penalty for reaching into levels that are farther away from the processor (see Chapter 9 for more details). Consider a thread T1 that was running on processor P2. Its time quantum ran out and it was context switched out. With a central queue, T1 may be picked up by some other processor, say P5, the next time around. Let us understand why this may not be a desirable situation. Well, perhaps most of the memory footprint of T1 is still in the nearer levels of processor P2's memory hierarchy. Therefore, T1 would have incurred less cache misses if it was run on P2 the second time around. In other words, T1 has an *affinity* for processor P2. Thus, one embellishment to the basic scheduling algorithm when applied to multiprocessors is to use cache affinity, a technique first proposed by Vaswani and Zahorjan. A per-processor scheduling queue (shown in Figure 12.29) would help in managing the threads and their affinity for specific processors than a shared queue. For load balancing, processors who find themselves out of work (due to their queues empty) may do what is usually referred to as *work stealing* from the scheduling queues of other processors. Another embellishment is to *lengthen the time quantum* for a thread that is currently holding a mutual exclusion lock. The intuition behind this is the fact that other threads of the same application cannot really run until this thread relinquishes the lock. Zahorjan proposed this technique as well. Figure 12.29 shows how a conceptual picture of a per processor scheduling queue.

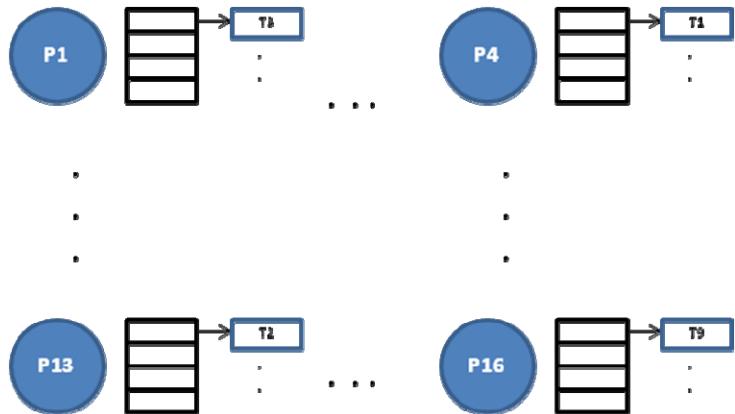


Figure 12.29: Per processor scheduling queue

We will introduce the reader to two additional techniques for making multiprocessor scheduling more effective, especially for multithreaded applications. The first one is called *space sharing*. The idea here is to dedicate a set of processors for an application for its lifetime. At application start up, as many processors as there are threads in the application are allocated to this application. The scheduler will delay starting the application until such time as that many idle processors are available. Since a processor is dedicated to a thread, there is no context switching overhead (and affinity is preserved as well). If a thread is blocked on synchronization or I/O, then the processor cycles are simply wasted. This technique provides excellent service to applications at the risk of wasting resources. Modifications to this basic idea of space sharing is for the application to have the ability to scale up or down its requirements for CPUs depending on the load on the system. For example, a web server may be able to shrink the number of threads to 10 instead of 20 if the system can provide only 10 processors. Later on, if the system has more processors available, the web server can claim them and create additional threads to run on those processors. A space sharing scheduler would divide up the total number of processors in the system into different sized partitions (see Figure 12.30) and allocate partitions at a time to applications instead of individual processors. This reduces the amount of book-keeping data structures that the scheduler needs to keep. This should remind the reader of fixed sized partition memory allocation we studied in Chapter 7. Similar to that memory management scheme, space sharing may result in internal fragmentation. For example, if an application requires 6 processors, it would get a partition with 8 processors. Two of the 8 processors will remain idle for the duration of the application.

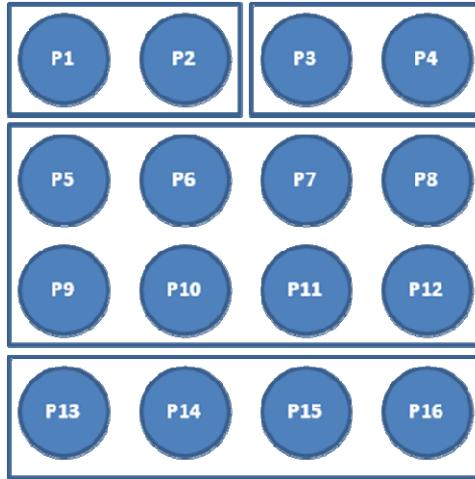


Figure 12.30: The scheduler has created 4 partitions to enable space sharing. There are two 2-processor partitions; and one each of a 4-processor and 8-processor partitions.

The last technique we introduce is *gang scheduling*. It is a complementary technique to space sharing. Consider the following. Thread T1 is holding a lock; T2 is waiting for it. T1 releases the lock but T2 is not currently scheduled to take advantage of the lock that just became available. The application may have been designed to use fine-grain locks such that a thread holds a lock only for a very short amount of time. In such a situation, there is tremendous loss of productivity for this application due to the fact that the scheduler is unaware of the close coupling that exists among the threads of the application. Gang scheduling alleviates this situation. Related threads of an application are scheduled as a group, hence the name gang scheduling. Each thread runs on a different CPU. However, in contrast to space sharing a CPU is not dedicated to a single thread. Instead, each CPU is time-shared.

Gang scheduling works as follows:

- Time is divided into fixed size quanta
- All the CPUs are scheduled at the beginning of each time quantum
- The scheduler uses the principle of gangs to allocate the processors to the threads of a given application
- Different gangs may use the same set of processors in different time quanta
- Multiple gangs may be scheduled at the same time depending on the availability of the processors
- Once scheduled the association of a thread to a processor remains until the next time quantum even if the thread blocks (i.e., the processor will remain idle)

Gang scheduling may incorporate cache affinity in its scheduling decision. Figure 12.31 shows how three gangs may share 6 CPUs both in space and time using the gang scheduling principle.

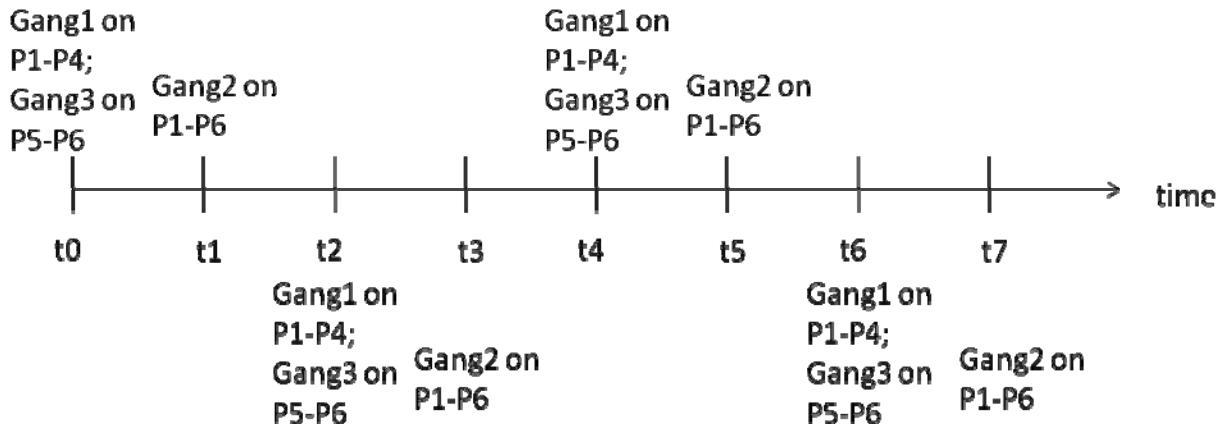


Figure 12.31: Gang1 needs 4 processors; Gang2 needs 6 processors; Gang3 needs 2 processors. Timeline of gang scheduling these three different gangs.

To summarize, we introduced the reader to four techniques that are used to increase the efficiency of CPU scheduling in a multiprocessor:

- Cache affinity scheduling
- Lock-based time quantum extension
- Space sharing
- Gang scheduling

A multiprocessor scheduler might use a combination of these above techniques to maximize efficiency depending on the workload for which the system is intended. Finally, these techniques are on top of and work in concert with the short-term scheduling algorithm that is used in each processor (see Chapter 6 for coverage of short-term scheduling algorithms).

12.10.1.4 Classic problems in concurrency

We will conclude this discussion by mentioning some classic problems in concurrency that have been used as a way of motivating the synchronization issues in parallel systems.

(1) Producer-consumer problem: This is also referred to as the bounded-buffer problem. The video processing application that has been used as running example in this chapter is an instance of the producer-consumer problem. Producers keep putting stuff into a shared data structure and consumers keep taking stuff out of it. This is a common communication paradigm that occurs in several applications. Any application that can be modeled as a pipeline uses this communication paradigm.

(2) Readers-writers problem: Let us say you are trying to get tickets to a ball game. You would go to a website such as Ticketmaster.com. You would go to the specific date on which you want to get the tickets and check availability of seats. You may look at different options in terms of pricing and seating before you finally decide to lock in on a specific set of seats and purchase the tickets. While you are doing this, there are probably 100's of others who are also looking to purchase tickets for the same game on the same

day. Until you actually purchase the seats you want, those seats are up for grabs by anyone. Basically, a database contains all the information concerning availability of seats on different dates, etc. Looking for availability of seats is a read operation on the database. Any number of concurrent readers may browse the database for availability. Purchasing tickets is a write operation on the database. This requires exclusive access to the database (or at least part of the database), implying that there should be no readers present while during the writing.

The above example is an instance of the classic readers-writers problem first formulated in 1971 by Courtois, et al. A simple minded solution to the problem would go like this. So long as readers are in the database, allow new readers to come in since none of them needs exclusive access. When a writer comes along, make him wait if there are readers currently active in the database. As soon as all the readers have exited the database, let the writer in exclusively. Similarly, if a writer is in the database, block the readers until the writer exits the database. This solution using mutual exclusion locks is shown in Figure 12.32.

```

mutex_lock_type readers_count_lock, database_lock;
int readers_count = 0;

void readers()
{
    lock(read_count_lock); /* get exclusive lock
                           * for updating readers
                           * count
                           */
    if (readers_count == 0) {
        /* first reader in a new group,
         * obtain lock to the database
         */
        lock(database_lock);
        /* note only first reader does this,
         * so in effect this lock is shared by
         * all the readers in this current set
         */
    }
    readers_count = readers_count + 1;
    unlock(read_count_lock);
    read_database();
    lock(read_count_lock); /* get exclusive lock
                           * for updating readers
                           * count
                           */
    readers_count = readers_count - 1;
    if (readers_count == 0) {
        /* last reader in current group,
         * release lock to the database
         */
        unlock(database_lock);
    }
    unlock(read_count_lock);
}

void writer()
{
    lock(database_lock); /* get exclusive lock */
    write_database();
    unlock(database_lock); /* release exclusive lock */
}

```

Figure 12.32: Solution to readers-writers problem using mutual exclusion locks

Pondering on this simple solution, one can immediately see some of its shortcomings. Since readers do not need exclusive access, so long as at least one reader is currently in the database, the solution will continue to let new readers in. This will lead to starvation of writers. The solution can be fixed by checking if there are waiting writers when a new reader wants to join the party, and simply enqueueing the new reader behind the waiting writer (please see Exercises 18-21).

(3) Dining Philosophers problem: This is a famous synchronization problem due to Dijkstra (1965). Since the time Dijkstra posed and solved this problem in 1965, any new synchronization construct that is proposed used this problem as a litmus test to see how efficiently the proposed synchronization construct solves it. Five philosophers are sitting around a round table. They alternate between eating and thinking. There is a bowl of spaghetti in the middle of the table. Each philosopher has his individual plate. There are totally five forks, one in between every pair of philosophers as shown in Figure 12.33. When a philosopher wants to eat, he picks up the two forks nearest to his on either side, gets some spaghetti from the bowl on to his plate and eats. Once done eating, he puts down his forks and continues thinking¹¹.

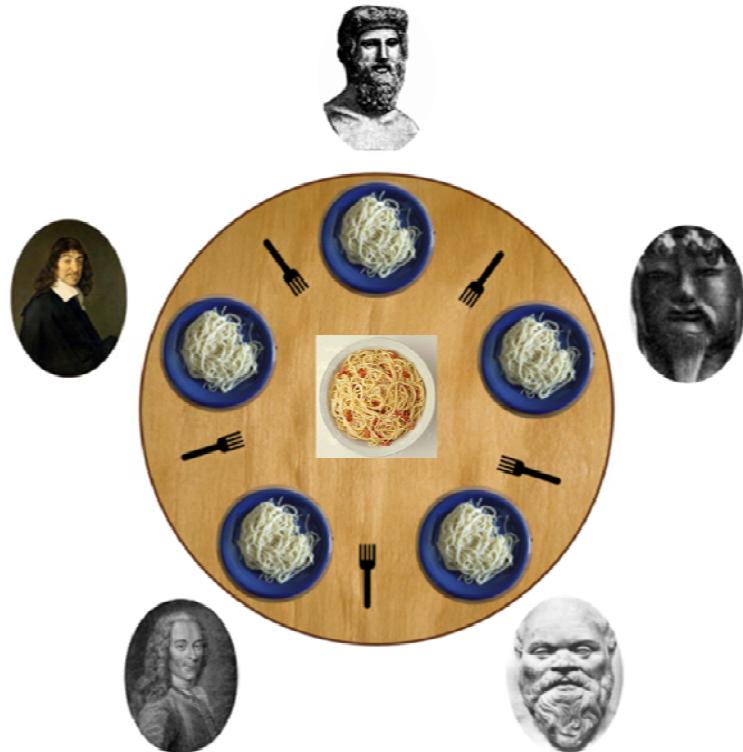


Figure 12.33: Dining Philosophers¹²

¹¹ It would appear that these philosophers did not care much about personal hygiene reusing forks used by neighbors!

¹² Picture source:

http://upload.wikimedia.org/wikipedia/commons/thumb/6/6a/Dining_philosophers.png/578px-Dining_philosophers.png, and <http://www.dkimages.com>

The problem is to make sure each philosopher does his bit of eating and thinking without bothering anyone else. In other words, we want each philosopher to be an independent thread of execution with maximal concurrency for the two tasks that each has to do, namely, eating and thinking. Thinking requires no coordination since it is an individual effort. On the other hand, eating requires coordination since there is a shared fork between every two philosophers.

A simple solution is for each philosopher to agree up front to pick up one fork first (say the left fork), and then the right fork and start eating. The problem with this solution is that it does not work. If every philosopher picks up the left fork simultaneously, then they are each waiting for the right fork, which is held by their right neighbor and this leads to the circular wait condition of deadlock.

Let us sketch a possible correct solution. We will introduce an intermediate state between *thinking* and *eating*, namely, *hungry*. In the hungry state, a philosopher will try to pick up both the forks. If successful, he will proceed to the eating state. If he cannot get both forks simultaneously, he will keep trying until he succeeds in getting both forks. What would allow a philosopher to pick up both forks simultaneously? Both his immediate neighbors *should not be* in the *eating* state. Further, he will want to ensure that his neighbors do not change states while he is doing the testing to see if he can pick up the forks. Once done eating, he will change his state to thinking, and simply notify his immediate neighbors so that they may attempt to eat if they are hungry. Figure 12.34 gives a solution to the dining philosophers' problem using monitors. Note that when a philosopher tries to get the forks using *take_forks*, monitor due to its guarantee that there is only one active thread in it at any point of time, ensures that no other philosopher can change his state.

```
void philosopher(int i)
{
    loop /* forever */
        do_some_thinking();
        DiningPhilosophers.take_forks(i);
        eat();
        DiningPhilosophers.put_down_forks(i);
    }
}
```

Figure 12.34-(a): Code that each philosopher thread executes.

```

monitor DiningPhilosophers
{
#define N 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2
#define LEFT ((i+N-1) mod N)
#define RIGHT ((i+1) mod N)

condition phil_waiting[N]; /* one for each philosopher */
int phil_state[N]; /* state of each philosopher */

    void take_forks (int i)
    {
        phil_state[i] = HUNGRY;
        repeat
            if ( (phil_state[LEFT] != EATING) &&
                (phil_state[RIGHT] != EATING) )
                phil_state[i] = EATING;
            else
                wait(phil_waiting[i]);
        until (phil_state[i] == EATING);
    }

    void put_down_forks (int i)
    {
        phil_state[i] = THINKING;
        notify(phil_waiting[LEFT]); /* left neighbor
                                      notified */
        notify(phil_waiting[RIGHT]); /* right neighbor
                                      notified */
    }

/* monitor initialization code */
init:
{
    int i;
    for (i = 1; i < N; i++) {
        phil_state[i] = THINKING;
    }
}
} /* end monitor */

```

Figure 12.34-(b): Monitor for Dining Philosophers problem

12.10.2 Architecture topics

In Section 12.9, we introduced multiprocessors. We will delve a little deeper into advanced topics relating to parallel architectures.

12.10.2.1 Hardware Multithreading

A pipelined processor exploits instruction level parallelism (ILP). However, as we have discussed in earlier chapters (see Chapters 5 and 9), there are limits to exploiting instruction-level parallelism due to a variety of factors including branches, limited functional units, and the growing gap between processor cycle time and memory cycle time. For example, cache misses result in pipeline stalls and the severity grows with the level at which cache misses occur. A cache miss that requires off-chip servicing could result in the CPU waiting for several tens of clock cycles. With multithreaded programs, there is another avenue to use the processor resources efficiently, namely across all the active threads that are ready to run. This is referred to as *thread level parallelism (TLP)*. *Multithreading* at the hardware level comes in handy to help reduce the ill effects of pipeline stalls due to ILP limitations by exploiting TLP.

A simple analogy will help here. Imagine a queue of people awaiting service in front of a single bank teller. A customer walks up to the teller. The teller finds that the customer needs to fill out a form before the transaction can be completed. The teller is smart. She asks the customer to step aside and fill out the form, and starts dealing with the next customer. When the first customer is ready again, the teller will deal with him to finish the original transaction. The teller may temporarily sideline multiple customers for filling out forms, and thus keep the waiting line moving efficiently.

Hardware multithreading is very much like this real life example. The uniprocessor is the teller. The customers are the independent threads. A long latency operation that could stall the processor is akin to filling out a form by the customer. Upon a thread stall on a long latency operation (e.g., a cache miss that forces main memory access), the processor picks the next instruction to execute from another ready thread, and so on. Thus, just like the bank teller, the processor utilizes its resources efficiently even if one or more ready threads are stalled on long latency operations. Naturally, this raises several questions. How does the processor know there are multiple threads ready to run? How does the operating system know that it can schedule multiple threads simultaneously to run on the same physical processor? How does the processor keep the *state* (PC, register file, etc.) of each thread distinct from one another? Does this technique work only for speeding up multithreaded applications or does it also work for sequential applications? We will answer these questions in the next couple of paragraphs.

Multithreading in hardware is yet another example of a contract between hardware and system software. The processor architecture specifies how many concurrent threads it can handle in hardware. In the Intel architecture, this is referred to as the number of *logical processors*. Essentially, this represents the level of *duplication* of the hardware resources needed to keep the states of the threads distinct. Each logical processor has its own distinct PC, and register file. The operating system allows an application to bind a thread to a logical processor. Earlier we discussed the operating system support for

thread level scheduling (Section 12.7). With multiple logical processors at its disposal, the operating system can simultaneously schedule multiple ready threads for execution on the processor. The physical processor maintains distinct persona for each logical processor through duplicate sets of register files, PC, page tables, etc. Thus, when an instruction corresponding to a particular logical processor goes through the pipeline, the processor knows the thread-specific hardware resource that have to be accessed for the instruction execution.

A multithreaded application stands to gain with hardware support for multithreading, since even if one thread is blocked due to limitations of ILP, some other thread in the same application may be able to make forward progress. Unfortunately, if an application is single-threaded then it cannot benefit from hardware multithreading to speed up its execution. However, hardware multithreading would still help improve the *throughput* of the system as a whole. This is because hardware multithreading is agnostic as to whether the threads it is pushing through the pipeline belong to independent processes or part of the same process.

This discussion begs another question: can exploitation of ILP by the processor using superscalar design coexist with exploitation of TLP? The answer is yes, and this is precisely what most modern processors do to enhance performance. The basic fact is that modern multiple-issue processors have more functional units than can be gainfully used up by a single thread. Therefore, it makes perfect sense to combine ILP exploitation using multiple-issue with TLP exploitation using the logical processors.

Each vendor gives a different name for this integrated approach to exploiting ILP and TLP. Intel calls it *hyperthreading* and is pretty much a standard feature on most Intel line of processors. IBM calls is *simultaneous multithreading (SMT)* and uses it in the IBM Power5 processor.

12.10.2.2 Interconnection Networks

Before we look at different types of parallel architectures, it is useful to get a basic understanding of how to interconnect the elements inside a computer system. In the case of a uniprocessor, we already learned in earlier chapters (see Chapter 4, 9, and 10) the concept of buses that allow the processor, memory, and peripherals to be connected to one another. The simplest form of interconnection network for a parallel machine is a shared bus (Section 12.9). However, large-scale parallel machines may have on the order of 1000's of processors. We will call each node in such a parallel machine a *processing element – PE* for short. A shared bus would become a bottleneck for communication among the PEs in such a large-scale machine. Therefore, such machines use more sophisticated interconnection networks such as a *mesh* (each processor is connected to its four neighbors, north, south, east, and west), or a *tree*. Such sophisticated interconnection networks allow for simultaneous communication among different PEs. Figure 12.35 shows some examples of such sophisticated interconnection networks. Each PE may locally have buses to connect to the memory and other peripheral devices.

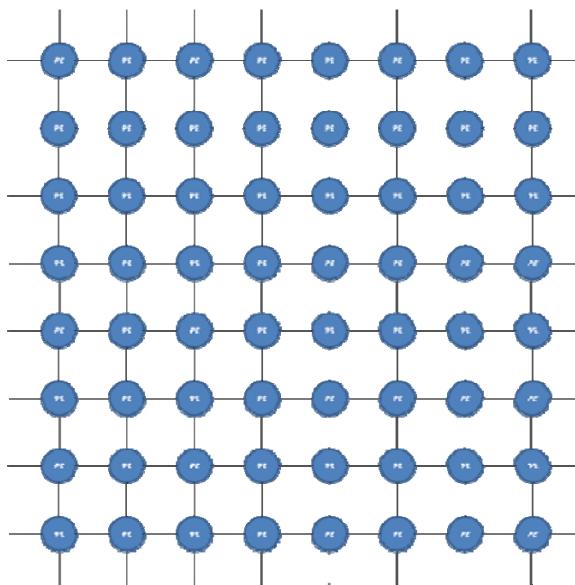


Figure 12.35-(a): A Mesh Interconnection Network

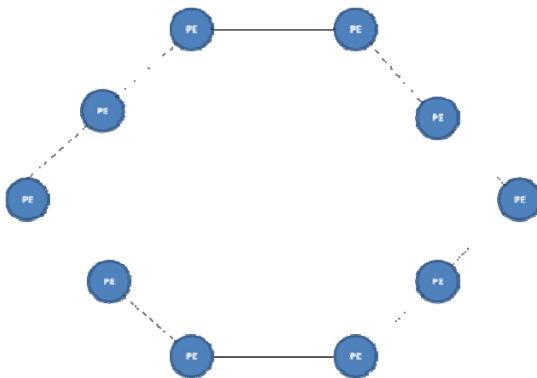


Figure 12.35-(b): A Ring Interconnection Network

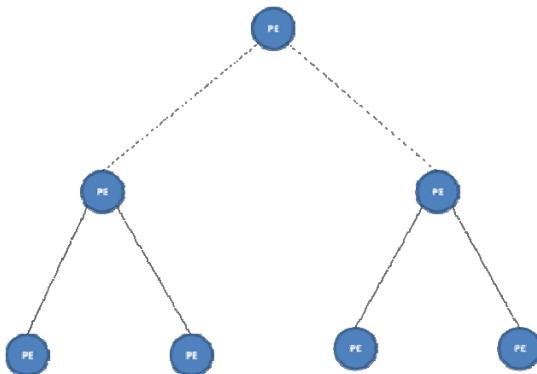


Figure 12.35-(c): A Tree Interconnection Network

12.10.2.3 Taxonomy of Parallel Architectures

Flynn in 1966 proposed a simple taxonomy for characterizing all architectures based on the number of independent instruction and data streams concurrently processed. The taxonomy, while useful for understanding the design space and the architectural choices, also aided the understanding of the kinds of applications that are best suited for each architectural style. Figure 12.36 shows the taxonomy graphically.

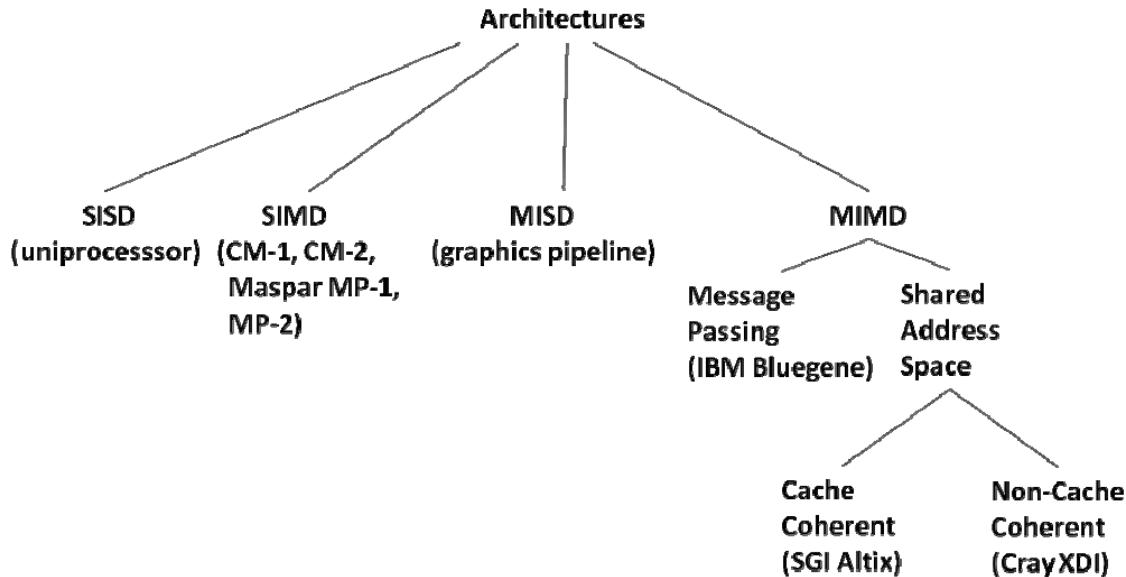


Figure 12.36: A Taxonomy of Parallel Architectures

Single Instruction Single Data (SISD): This is the classic uniprocessor. Does a uniprocessor exploit any parallelism? We know that at the programming level, the uniprocessor deals with only a single instruction stream and executes these instructions sequentially. However, at the implementation level the uniprocessor exploits what is called *instruction-level parallelism* (ILP for short). It is ILP that allows pipelined and superscalar implementation of an instruction-set architecture (see Chapter 5).

Single Instruction Multiple Data (SIMD): All the processors execute the same instruction in a lock-step fashion, on independent data streams. Before the killer micros made this style of architecture commercially non-competitive (in the early 90's), several machines were built to cater to this architecture style. Examples include Thinking Machine Corporation's Connection Machine CM-1 and CM-2; and Maspar MP-1 and MP-2. This style of architecture was particularly well-suited to image processing applications (for example, applying the same operation to each pixel of a large image). Figure 12.37 shows a typical organization of an SIMD machine. Each processor of the SIMD machine is called a *processing element (PE)*, and has its own data memory that is pre-loaded with a distinct data stream. There is a single instruction memory contained in the control unit that fetches and broadcasts the instructions to the PE array. The instruction memory is also pre-loaded with the program to be executed on the PE array. Each PE executes the instructions with the distinct data streams from the respective data

memories. SIMD model promotes parallelism at a very *fine granularity*. For example, a **for** loop may be executed in parallel with each iteration running on a different PE. We define *fine-grained parallelism* as one in which each processor executes a small number of instructions (say less than 10) before communicating with other processors.

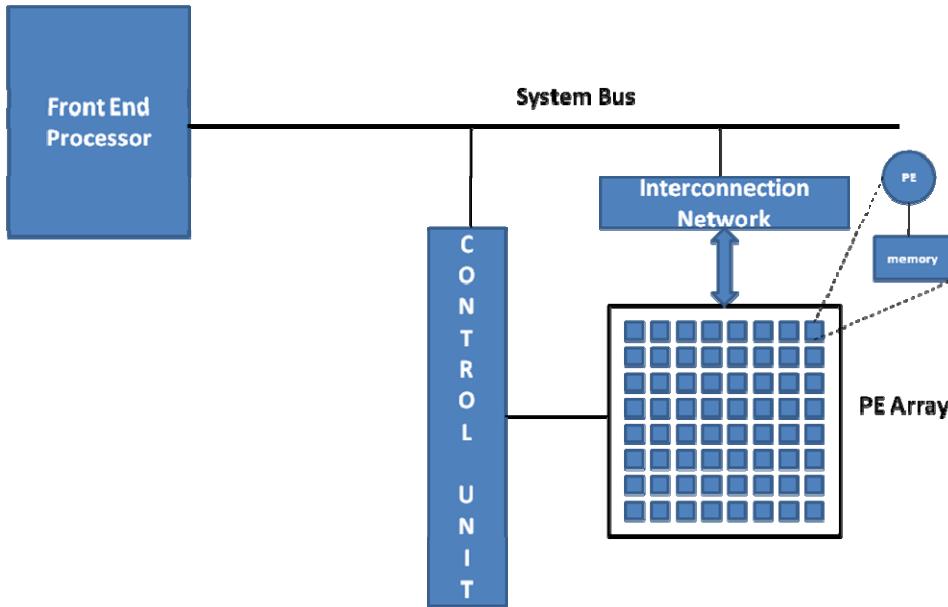


Figure 12.37: Organization of an SIMD machine. Front-end processor is typically a powerful workstation running some flavor of Unix. PE stands for processing element and is the basic building block of an SIMD machine

An SIMD machine is a workhorse to carry out some fine-grained compute-intensive task. Program development for the SIMD machine is usually via a front-end processor (a workstation class machine). The front-end is also responsible for pre-loading the data memories of the PEs and the instruction memory of the array control unit. All I/O is orchestrated through the front-end processor as well. The PEs communicate with one another using an interconnection network. Since SIMD machines may have on the order of 1000's of processors, they use sophisticated interconnection networks as discussed in Section 12.10.2.2 (e.g., mesh or a tree). Though no commercial machines of this flavor are in the market today, the Intel MMX instructions are inspired by the parallelism exploited by this architectural style. There has been a resurgence in this style of computation as exemplified by stream accelerators such as nVidia graphics card (referred to as *Graphics Processing Unit* or *GPU* for short). Further, as stream-oriented processing (audio, video, etc.) are becoming more mainstream, there is a convergence of the traditional processor architecture and the stream accelerators. For example, Intel's Larrabee *General Purpose GPU (GPGPU)* architecture represents one such integrated architecture meant to serve as the platform for future supercomputers.

Multiple Instructions Single Data (MISD): At an algorithmic level, one could see a use for this style of architecture. Let us say we want to run several different face detection algorithms on the same image stream. Each algorithm represents a different instruction

stream working on the same data stream (i.e., the image stream). While MISD serves to round out the classification of parallel architecture styles, there is not a compelling argument for this style. It turns out that most computations map more readily to the MIMD or the SIMD style. Consequently, no known architecture exactly matches this computation style. Systolic arrays¹³ may be considered a form of MISD architecture. Each cell of a systolic array works on the data stream and passes the transformed data to the next cell in the array. Though it is a stretch, one could say that a classic instruction processing pipeline represents an MISD style at a very fine grain if you consider each instruction as “data” and what happens to the instruction as it moves from stage to stage.

Multiple Instructions Multiple Data (MIMD): This is the most general architectural style, and most modern parallel architectures fall into this architectural style. Each processor has its own instruction and data stream, and work asynchronously with respect to one another. The processors themselves may be off-the-shelf processors. The programs running on each processor may be completely independent, or may be part of a complex application. If it is of the latter kind, then the threads of the same application executing on the different processors need to synchronize explicitly with one another due to the inherent asynchrony of the architectural model. This architectural style is best suited for supporting applications that exhibit medium and coarse-grained parallelism. We define *medium-grained parallelism* as one in which each processor executes approximately 10-100 instructions before communicating with other processors. We define *coarse-grained parallelism* as one in which each processor executes on the order of 1000’s of instructions before communicating with other processors.

Early multiprocessors were of the SIMD style. Necessarily, each processor of an SIMD machine has to be specially designed for that machine. This was fine so long as general-purpose processors were built out of discrete circuits. However, as we said already, the arrival of the killer micros made custom-built processors increasingly unviable in the market place. On the other hand, the basic building block in an MIMD machine is a general-purpose processor. Therefore, such architectures benefit from the technology advances of the off-the-shelf commercial processors. Further, the architectural style allows using such a parallel machine for running several independent sequential applications, threads of a parallel application, or some combination thereof. It should be mentioned that with the advent of stream accelerators (such as nVidia GeForce family of GPUs), hybrids of the parallel machine models are emerging.

12.10.2.4 Message-passing Vs. Shared Address Space Multiprocessors

MIMD machines may be subdivided into two broad categories: message-passing and shared address space. Figure 12.38 shows a picture of a message-passing multiprocessor. Each processor has its private memory, and processors communicate with each other using messages over an interconnection network. There is no shared memory among the processors. For this reason, such architectures are also referred to as *distributed memory* multiprocessors.

¹³ H. T. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," in Proc. SIAM Sparse Matrix Symp., 1978, pp. 256-282.

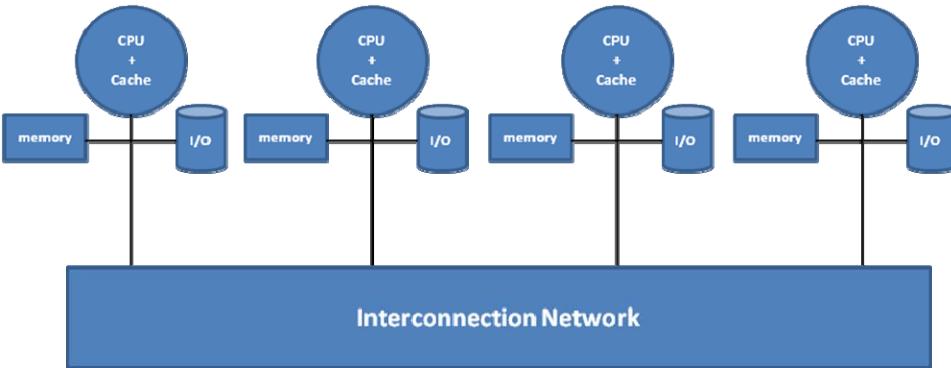


Figure 12.38: Message passing multiprocessor

IBM's Bluegene series is an example of a modern day message-passing machine that fits this model. Several examples of message passing machines of yester years include TMC's CM-5, Intel Paragon, and IBM SP-2. The previous generation of parallel machines relied on special interconnection network technologies to provide low-latency communication among the processors. However, with advances in computer networking (see Chapter 13), local area networking technology is offering viable low-latency high bandwidth communication among the processors in a message-passing multicomputer. Consequently, *cluster parallel machine*, a collection of computers interconnected by local area networking gear such as Gigabit Ethernet (see Section 13.6.1) has become a popular platform for parallel computing. Just as **pthreads** library offers a vehicle for parallel programming in a shared memory environment, *message passing interface* (*MPI* for short) is a programming library for the message-passing computing environment.

Shared address space multiprocessor is the second category of MIMD machines, which as the name suggests provides address equivalence for a memory location irrespective of which processor is accessing that location. In other words, a given memory address (say 0x2000) refers to the same physical memory location in whichever processor that address is generated. As we have seen, a processor has several levels of caches. Naturally, upon access to a memory location it will be brought into the processor cache. As we have already seen in Section 12.9, this gives rise to the cache coherence problem in a multiprocessor. Shared address space machines may be further subdivided into two broad categories based on whether this problem is addressed in hardware or software. Non-Cache Coherent (NCC) multiprocessors provide shared address space but no cache coherence in hardware. Examples of such machines from yester years include BBN Butterfly, Cray T3D, and Cray T3E. A current day example is Cray XDI.

Cache coherent (CC) multiprocessors provide shared address space as well as hardware cache coherence. Examples of this class of machines from yester years include KSR-1, Sequent Symmetry, and SGI Origin 2000. Modern machines of this class include SGI Altix. Further, the individual nodes of any high-performance cluster system (such as IBM Bluegene) is usually a cache coherent multiprocessor.

In Section 12.9, we introduced the reader to a bus-based shared memory multiprocessor. A large-scale parallel machine (regardless of whether it is a message-passing or a shared address space machine) would need a more sophisticated interconnection network than a bus. A large-scale parallel machine that provides a shared address space is often referred to as *distributed shared memory (DSM)* machine, since the physical memory is distributed and associated with each individual processor.

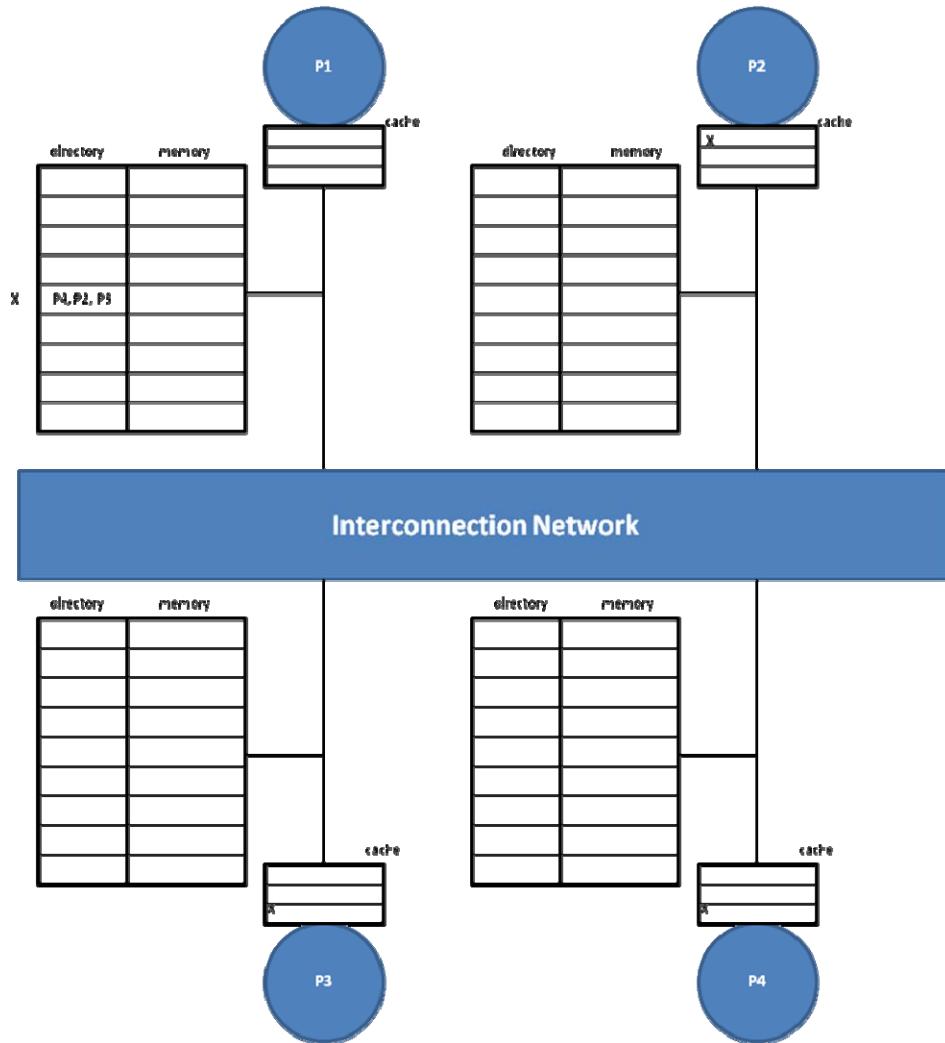


Figure 12.39: DSM Multiprocessor with directory based coherence

The cache coherence mechanisms we discussed in Section 12.9 assume a broadcast medium, namely, a shared bus so that a change to a memory location is seen simultaneously by all the caches of the respective processors. With an arbitrary interconnection network such as a tree, a ring, or a mesh (see Figure 12.35), there is no longer a broadcast medium. The communication is point-to-point. Therefore, some other mechanism is needed to implement cache coherence. Such large-scale shared memory multiprocessors use a *directory-based scheme* to implement cache coherence.

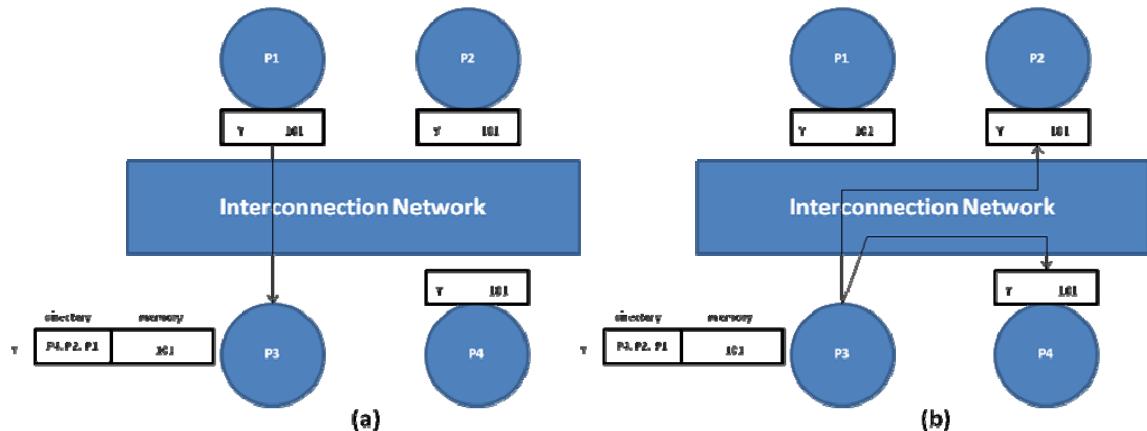
The idea is quite simple. Shared memory is already physically distributed; simply associate a directory along with each piece of the shared memory. There is a directory entry for each memory location. This entry contains the processors that currently have this memory location encached. The cache coherence algorithm may be invalidation or update based. The directory has the book-keeping information for sending either the invalidations or updates. Figure 12.39 shows a directory-based DSM organization. Location X is currently encaches in P2, P3, and P4. If P4 wishes to write to location X, then the directory associated with X sends an invalidation to P2 and P3, before P4 is allowed to write to X. Essentially, the directory orders the access to memory locations that it is responsible ensuring that the values seen by the processors are coherent.

Example 18:

In a 4-processor DSM similar to that shown in Figure 12.39, memory location Y is in physical memory of P3. Currently P1, P2, P4 have a copy of Y in their respective caches. The current values in Y is 101. P1 wishes to write the value 108 to Y. Explain the sequence of steps before the value of Y changes to 108. Assume that the cache write policy is write-back.

Answer:

- Since memory location Y is in P3, P1 sends a request through the interconnection network to P3 and asks write permission for y (Figure 12.40-(a)).
- P3 notices that P2 and P4 have copies of Y in their caches by looking up directory entry for Y. It sends invalidation request for Y to P2 and P4 (Figure 12.40-(b)).
- P2 and P4 invalidate their respective cache entry for Y and send back acknowledgements to P3 via the interconnection network. P3 removes P2 and P4 from the directory entry for Y (Figure 12.40-(c)).
- P3 sends write permission to P1 (Figure 12.40-(d)).
- P1 writes 108 in the cache entry for Y (Figure 12.40-(e)).



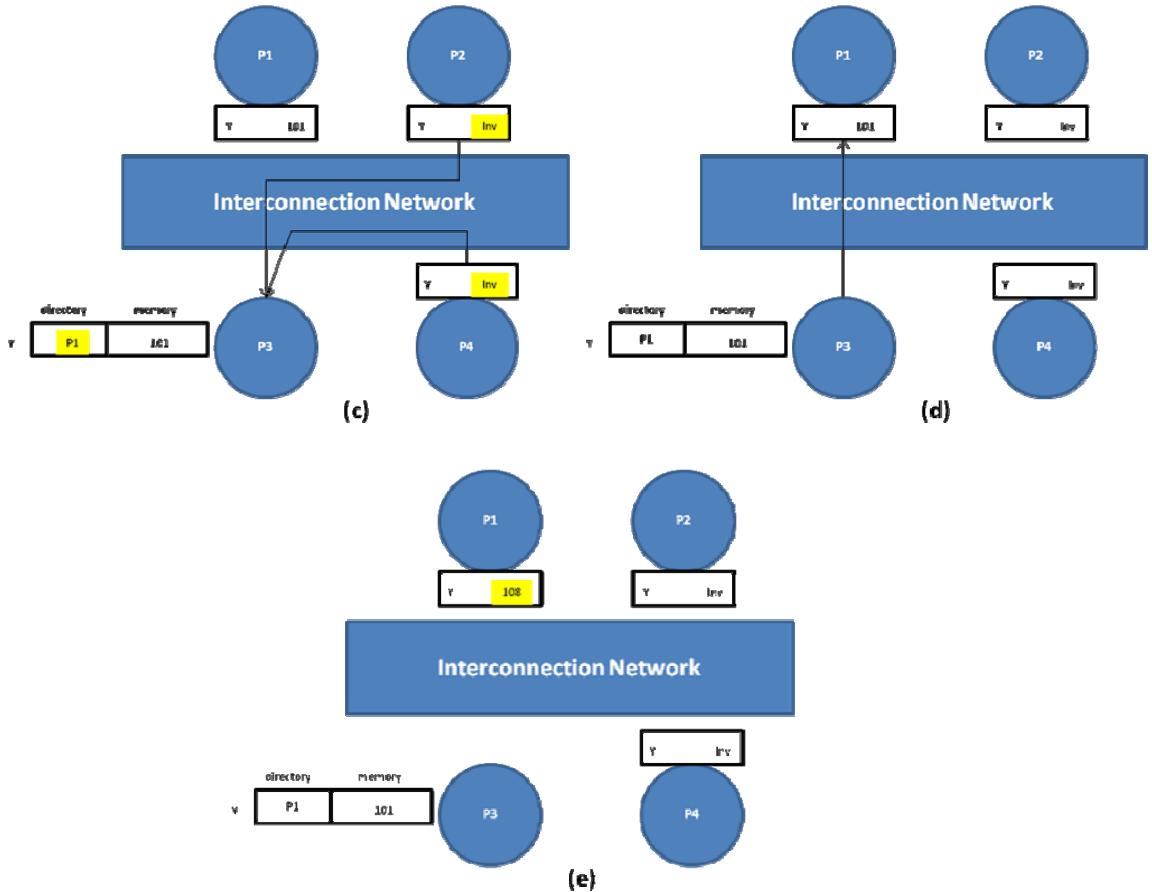


Figure 12.40: Sequence of steps for Example 18

12.10.2.5 Memory Consistency Model and Cache Coherence

A cache-coherent multiprocessor ensures that once a memory location (that may be currently cached) is modified the value will propagate to all the cached copies and the memory as well. One would expect that there will be some delay before all the cached copies have been updated with the new value that has just been written. This begs the question what is the view from the programmer's perspective? *Memory consistency model* is the contract between the programmer and the memory system for defining this view. We have seen repeatedly in this textbook that system design is all about establishing a contract between the hardware and the software. For example, ISA is a contract between the compiler writer and the processor architect. Once the ISA is set, the implementer of the ISA can exercise his/her choice in the hardware realization of the ISA. Memory consistency model is a similar contract between the programmer and the memory system architect.

Let us motivate this with an example. The following is the execution history on processors P1 and P2 of an SMP. The memory is shared by definition but the processor registers are private to each processor.

Initially,

Mem[X] = 0

	<u>Processor P1</u>	<u>Processor P2</u>
Time 0:	R1 <- 1	
Time 1:	Mem[X] <- R1	R2 <- 0
Time 2:
Time 3:	R2 <- Mem[X]
Time 4:

What would you expect the contents of R2 on P2 to be at time 4? The intuitive answer is 1. However, this really depends on the implementation of the memory system. Let us say that each of the instructions shown in the execution history is atomic with respect to that processor. Thus at time 1, processor P1 has written a value of 1 into Mem[X]. What this means as far as P1 is concerned is that if it tries to read Mem[X] after time 1, the memory system guarantees that it will see the value 1 in it. However, what guarantee does the memory system give for values to the same memory location when accessed from other processors? Is it legal for the memory system to return a value 0 for the access at time 3 on P2? The cache coherence mechanism guarantees that both the caches on P1 and P2 as well as the memory will *eventually* contain the value 1 for Mem[x]. But it does not specify *when* the values will become consistent. The memory consistency model answers the *when* question.

An intuitive memory consistency model proposed by Leslie Lamport is called *sequential consistency (SC)*. In this model, memory reads and writes are atomic with respect to the *entire system*. Thus, if we say that P1 has written to Mem[X] at time 1, then the new value for this memory location is *visible* to *all* the processors, henceforth. Thus, when P2 reads Mem[X] at time 3 it is also guaranteed by the memory system that it will see the value 1. As far as the programmer is concerned, this is all the detail about the memory system that he/she needs to write correct programs. The implementation of the memory system with caches and the details of the cache coherence mechanisms are irrelevant to the programmer. This is akin to the separation between architecture and implementation that we have seen in the context of processor ISA design.

Let us give a more precise definition of the SC memory model. It basically says that the effects of the memory accesses from an individual processor will be exactly the same as if there are no other processors accessing the memory. That is, each memory access (read or write) will be atomic. One would expect this behavior from a uniprocessor and therefore it is natural to expect this from a multiprocessor as well. As far as memory accesses from different processors, the model is saying that the observed effect will be an arbitrary interleaving of these individually atomic memory accesses emanating from the different processors.

One way to visualize the SC memory model, is to think of memory accesses from different processors as the cards dealt to a number of players in a card game. You may have seen a card sharp take two splits of a card deck and do a shuffle merge as shown in

the adjoining picture, while preserving the original order of the cards in the two splits. The SC memory model does an n -way shuffle merge of all the memory accesses coming from the n processors that comprise the multiprocessor.

Returning to the example above, we showed a particular post-mortem execution history in which the access to $\text{Mem}[X]$ from P2 *happened after* the access to $\text{Mem}[X]$ from P1. The processors are asynchronous with respect to each other. Therefore, another run of the same program could have an execution in which the accesses are reversed. In this case, the access by P2 to $\text{Mem}[X]$ would return a value of 0. In other words, for the above program, the SC model would allow the memory system to return a 0 or a 1 for the access by P2 to $\text{Mem}[X]$.



One can see that the SC memory model could result in parallel programs that have data races (see Section 12.2.3 for a discussion of data races). Fortunately, this does not affect correct parallel program development as long as the programmer uses synchronization primitives as discussed in earlier sections to coordinate the activities of the threads that comprise an application. In fact, as we have seen in earlier sections, the application programmer is completely oblivious to the memory consistency model since he/she is programming using libraries such as *pthreads*. Just as the ISA is a contract between the compiler writer and the architect, the memory consistency model is a contract between the library writer and the system architect.

The SC memory model says nothing about how the processors may coordinate their activities using synchronization primitives to eliminate such data races. As we have discussed in earlier sections, the system may provide synchronization primitives in hardware or software for the purposes of coordination among the threads. Therefore, it is possible to include such synchronization primitives along with the normal memory accesses to specify the contract between hardware and software. This would give more implementation choices for the system architect to optimize the performance of the memory system. The area of memory consistency models for shared memory systems has been a fertile area of research in the late 80's and the early 90's resulting in several doctoral dissertations. Detailed treatment of this topic is beyond the scope of this textbook. The interested reader is referred to advanced computer architecture textbooks for a more complete exposure to this exciting topic.

12.10.3 The Road Ahead: Multi- and Many-core Architectures

The road ahead is exciting for parallel computing. The new millennium has ushered in the age of multi-core processors. The name multi-core comes from the simple fact that the chip now consists of several independently clocked processing cores that constitute the processor. Moore's law fundamentally predicts increase in chip density with time. Thus far, this increase in chip density has been exploited to increase processor

performance. Several modern processors including AMD Phenom II, IBM Power5, Intel Pentium D and Xeon-MP, and Sun T1 have embraced the multi-core technology.

However, with the increase in processor performance the power consumption of single-chip processors has also been increasing steadily (see Section 5.15.2). The power consumption is directly proportional to the clock frequency of the processor. This trend is forcing processor architects to focus attention on ways to reduce power consumption while increasing chip density. The answer is to have multiple processing cores on a single-chip, each of which may be clocked at a lower clock frequency. The basic strategy for conserving power would then involve selectively turning off the power to parts of the chip. In other words, power consumption consideration is forcing us to choose parallelism over faster uniprocessors. With single-chip processors now containing multiple independent processing cores, parallel computing is no longer an option but a must as we move forward. The next step in the evolution of single-chip processors is *many-core* processors, wherein a single-chip may contain 100's or even 1000's of cores. Each of these processing cores will likely be a very simple processor, not that different from the basic pipelined processor we discussed in Chapter 5. Thus, even though we did not get into the micro-architectural intricacies of modern processors in the earlier chapters, the simple implementations discussed in these chapters may become more relevant with the many-core processors of tomorrow.

One might be tempted to think that a multi-core architecture is no different from shrink-wrapping an SMP into a single chip; and that a many-core architecture is similarly a shrink-wrapped version of a large-scale multiprocessor with 100's or 1000's of PEs. However, it is not business as usual, considering the range of issues that requires a radical rethinking including electrical engineering, programming paradigm, and resource management at the system level.

At the electrical engineering level, we already mentioned the need for reducing the power dissipation. The reason for conserving power is a pragmatic matter. It is just impossible to cool a processor that is smaller than the size of a penny but consumes several hundred watts of power. Another concern is distribution of electrical signals, especially the clock, on the chip. Normally, one would think that placing a 1 or 0 on a wire results in this signal being propagated as is to the recipients. This is safe assumption as long as there is sufficient time for the signal to travel to the recipient on the wire, referred to as the wire delay. We have seen in an earlier chapter (see Chapter 3) that the wire delay is a function of the resistance and capacitance of the electrical wire. As clock speed increases, the wire delay may approach the clock cycle time causing a wire inside a chip to behave like a transmission line, i.e., the signal strength deteriorates with distance. The upshot is that 1 may appear like a 0 at the destination, and vice versa leading to erroneous operation. Consequently, chip designers have to rethink and re-evaluate some basic assumptions about digital signals as the chip density increases in the multi- and many-core era. This is giving rise to new trends in integrated circuit (IC) design, namely, three dimensional design popularly referred to as *3D ICs*. Basically, to reduce the wire length, the chips are being designed with active elements (transistors) in several parallel planes in the z-dimension. Wires, which hitherto were confined to two dimensions, will now be run in

three dimensions, thus giving an opportunity to reduce the wire delay between active elements. A 3D chip is akin to building skyscrapers! Mainstream general-purpose processors are yet to embrace this new technology but it is just a matter of time.

At the architecture level, there are several differences between an SMP and a multi-core processor. In an SMP, at the hardware level, the only shared hardware resource is the bus. On the other hand, in a multi-core processor, several hardware resources may be common to all the cores. For example, we showed the memory hierarchy of AMD Barcelona chip in Chapter 9 (see Figure 9.45). The on-chip L3 cache is shared across all the four cores on the chip. There could be other shared resources such as I/O and memory buses coming out of the chip. It is architectural issue to schedule the efficient use of such shared resources.

At the programming level, there are important differences in the way we think about a parallel program running on a multi-core versus on an SMP. A good rule of thumb in designing parallel programs is to ensure that there is a good balance between computation and communication. In an SMP, threads should be assigned a significant amount of work before they are required to communicate with other threads that are running on other processors. That is, the computations have to be *coarse-grained* to amortize for the cost of inter-processor communication. Due to the proximity of the PEs in a multi-core processor, it is conceivable to achieve *finer-grain* parallelism than would be feasible in an SMP.

At the operating system level, there is a fundamental difference between an SMP and a multi-core processor. Scheduling threads of a multithreaded application on a multi-core versus an SMP requires rethinking to take full advantage of the shared hardware resources between the cores. In an SMP, each processor has an independent image of the operating system. In a multi-core processor, it is advantageous for the cores to share the same image of the operating system. The OS designers have to rethink what operating systems data structures should be shared across the cores, and what should be kept independent.

Many-core processors may bring forth a whole new set of problems at all of the above levels and add new ones such as coping with partial software and hardware failures.

12.11 Summary

In this chapter, we covered key concepts in parallel programming with threads, the operating system support for threads, and the architectural assists for threads. We also reviewed advanced topics in operating systems and parallel architectures.

The three things that an application programmer has to worry about in writing threaded parallel programs are thread creation/termination, data sharing among threads, and synchronization among the threads. Section 12.3 gives a summary of the thread function calls and Table 12.2 gives the vocabulary the reader should be familiar with in the context of developing threaded programs. Section 12.6 gives a summary of the important thread programming API calls supported by the *pthreads* library.

In discussing implementation of threads, we covered the possibility of threads being implemented above the operating system as a user level library with minimal support from the operating system itself in Section 12.7.1. Most modern operating systems such as Linux and Microsoft XP and Vista support threads as a basic unit of CPU scheduling. In this case, the operating system implements the functionality expected at the programming level. We covered kernel level threads in Section 12.7.2 as well as an example of how threads are managed in the Sun Solaris operating system in Section 12.7.3.

The fundamental architectural assist needed for supporting threads is an atomic read-modify-write memory operation. We introduced Test-And-Set instruction in Section 12.8, and showed how using this instruction, it is possible to implement higher-level synchronization support in the operating system. In order to support data sharing among the processors, the cache coherence problem needs to be solved which we discussed in Section 12.9.

We considered advanced topics in operating systems and architecture as they pertain to multiprocessors in Section 12.10. In particular, we introduced the reader to a formal treatment of deadlocks, sophisticated synchronization constructs such as monitor, advanced scheduling techniques, and classic problems in concurrency and synchronization (see Section 12.10.1). In Section 12.10.2, we presented a taxonomy of parallel architectures (SISD, SIMD, MISD, and MIMD) and discussed in depth the difference between message-passing style and shared memory style MIMD architectures.

The hardware and software issues associated with multiprocessors and multithreaded programs are intriguing and deep. We have introduced the reader to a number of exciting topics in this area. We have barely scratched the surface of some these issues in this Chapter. We hope we have raised the reader's curiosity level enough through this Chapter to take him/her through further exploration of these issues in future courses. A more in-depth treatment of some of the topics covered in this chapter may be found in the textbook by Culler, et al.¹⁴

12.12 Historical Perspective

Parallel computing and multiprocessors have been of interest to computer scientists and electrical engineers from the very early days of computing.

In Chapter 5, we saw that pipelined processor design exploits instruction-level parallelism or ILP. In contrast, multiprocessors exploit a different kind of parallelism, namely, thread level parallelism or TLP. Exploitation of ILP does not require the end user having to do anything different – he/she continues to write sequential programs. The compiler in concert with the architect does all the magic to exploit the ILP in the sequential program. However, exploitation of TLP requires more work. Either the program has to be written as an explicitly parallel program that uses multiple threads as

¹⁴ Culler, Singh, and Gupta, “Parallel Computer Architecture: A Hardware/Software Approach,” Morgan Kaufmann.

developed in this chapter, or the sequential program has to be converted automatically into a multithreaded parallel program. For the latter approach, a sequential programming language such as FORTRAN is extended with directives that the programmer inserts to indicate opportunities for automatic parallelization. The compiler exploits such directives to parallelize the original sequential program. This approach was quite popular in the early days of parallel computing but met with diminishing returns since its applicability is restricted usually to exploiting loop-level parallelism and not function parallelism. Just as a point of clarification, loop-level parallelism notices that successive iterations of a “for” loop in a program are independent of one another and turns every iteration or groups of iterations into a parallel thread. Function parallelism or task parallelism deals with conceptual units of work that the programmer has deemed to be parallel (similar to the vision example used in developing the thread programming constructs in this chapter).

One can easily understand the lure of harnessing parallel resources to get more work done. It would appear that an application that achieves a given single processor performance could in theory achieve an N -fold improvement in performance if parallelized. However, we know from Chapter 5 that Amdahl’s law limits this linear increase in performance due to the inherent serial component in the application. Further, there is usually a lag between the performance of a single processor and that of the individual processors that comprise a parallel machine. This is because constructing a parallel machine is not simply a matter of replacing the existing processors with faster ones as soon as they become available. Moore’s law (Please see Section 3.1) has been giving us increased single processor performance almost continuously for the past 30 years. Thus, a parallel machine becomes quickly outdated as the next generation of higher-performance microprocessors hit the market. For example, a modern day notebook computer costing a few thousand dollars has more processing power than a multi-million dollar Cray machine of the 70’s and 80’s. The obsolescence factor has been the main reason software for parallel machines has not seen the same rapid growth as that for uniprocessors.

Parallel architectures were a high-end niche market reserved for applications that demanded such high performance, mainly from the scientific and engineering domains. Examples of companies and the parallel machines marketed by them include TMC (connection machine line of parallel machines starting with CM-1 to CM-5); Maspar (MP-1 and MP-2); Sequent (Symmetry); BBN (Butterfly); Kendall Square Research (KSR-1 and KSR-2); SGI (Origin line of machines and currently Altix); and IBM (SP line of machines). Typical characteristics of such machines include either an off-the-shelf processor (e.g., TMC’s CM-5 used Sun SPARC, and SGI’s Origin series used MIPS), or a custom-built processor (e.g., KSR-1, CM-1, CM-2, MP-1, and MP-2); a proprietary interconnection network, and glue logic relevant to the taxonomical style of the architecture. The interconnection network was a key component of such architectures since efficient data sharing and synchronization among the processors were dependent on capabilities in the interconnection network.

The advent of the powerful single-chip microprocessor of the 90's (dubbed "Killer micros") turned out to be a disruptive technology that shook up the high-performance computing market. Once the single-chip microprocessor performance surpassed that of a custom crafted processor for a parallel machine, the economic viability of constructing such parallel machines came into question. While a few have survived by reinventing themselves, many parallel computing vendors that thrived in the niche market disappeared (for example, TMC and Maspar). In parallel with the advance of single-chip microprocessor performance, local area network technology was advancing at a rapid pace as well. We will visit the evolution of local area networks (LAN) in Chapter 13, but suffice it to say here that with advent of switched Gigabit Ethernet (see Section 13.11), the need for proprietary interconnection networks to connect processors of a parallel machine became less relevant. This breakthrough in LAN technology spurred a new class of parallel machines, namely, clusters. A cluster is essentially a set of compute nodes interconnected by off-the-shelf LAN technology. This has been the workhorse for high-performance computing since the mid to late 90's to the present day. Clusters promote a message-passing style of programming and as we said earlier, the MPI communication library has become a *de facto* standard for programming on clusters. What is inside a compute node changes of course with advances in technology. For example, at present it is not uncommon to have each computing node as an n -way SMP, where n may be 2, 4, 8, or 16 depending on the vendor. Further, each processor inside an SMP may be a hardware multithreaded multi-core processor. This gives rise to a mixed parallel programming model: shared memory style programming within a node and message-passing style programming across the nodes.

12.13 Review Questions

1. Compare and contrast processes and threads
2. Where does a thread start executing?
3. When does a thread terminate?
4. How many threads can acquire a mutex lock at the same time.
5. Does a condition variable allow a thread to wait conditionally or unconditionally?
6. Define deadlock and explain how it can happen and how it can be prevented.
7. What problems could be encountered with the following construct

```
if(state == BUSY)
    pthread_cond_wait(c, m);
state = BUSY;
```

8. Compare and contrast the contents of a PCB and a TCB.
9. Is there any point to using user threads on a system, which is only scheduling processes, and not threads. Will the performance be improved?

10. User level thread synchronization in a uniprocessor (select one from the following to complete the sentence)
- needs no special hardware support since turning off interrupts will suffice
 - needs some flavor of a *read modify write* instruction
 - can be implemented simply by using load/store instruction
11. Ensuring that all the threads of a given process share an address space in an SMP is (select one from the following to complete the sentence)
- impossible
 - trivially achieved since the page table is in shared memory
 - achieved by careful replication of the page table by the operating system for each thread
 - achieved by the hardware providing cache coherence
12. Keeping the TLBs consistent in an SMP (select one from the following to complete the sentence)
- is the responsibility of the user program
 - is the responsibility of the hardware
 - is the responsibility of the operating system
 - is impossible
13. Select all choices that apply regarding threads created in the same address space.
- they share code
 - they share global data
 - they share the stack
 - they share the heap
14. From the following sentences regarding threads, select the ones that are True.
- An operating system that provides no support for threads will block the entire process if one of the (user level) threads in that process makes a blocking system call.
 - In pthreads, a thread that does a pthreadcondwait will always block.
 - In pthreads, a thread that does a pthreadmutexlock will always block.
 - In Solaris, all user level threads in the same process compete equally for CPU resources.
 - All the threads within a single process in Solaris share the same page table.
15. Discuss the advantages of kernel threads in Sun Solaris.
16. Distinguish between write-invalidate and write-update cache coherence policies.

17. Given the following details about an SMP (symmetric multiprocessor):

Cache coherence protocol: **write-invalidate**
Cache to memory policy: **write-back**

Initially:

The caches are empty

Memory locations:

C contains 31

D contains 42

Consider the following timeline of memory accesses from processors P1, P2, and P3.

T1		Load C	Store #50, D
T2	Load D	Load D	Load C
T3			
T4		Store #40, C	
T5	Store #55, D		

Fill the table below, showing the contents of the cache after each timestep. We have started it off for you by showing the contents after time T1.

(I indicates the cache location is invalid. NP indicates not present)

T1	C D	NP NP	31 NP	NP 50	31 42
T2	C D				
T3	C D				
T4	C D				
T5	C D				

18. Why is it considered a good programming practice to do a `pthread_cond_signal` while still holding the mutex lock associated with the corresponding `pthread_cond_wait`?

19. Why is it considered a good programming practice to re-test the predicate upon resuming from a `pthread_cond_wait` call?
20. Implement the monitor shown in Example 17 in C using pthreads library.
21. Implement the solution shown in Example 17 using Java.
22. Write a solution to the readers-writers problem using mutual exclusion locks that gives priority to writers.
23. Write a solution to the readers-writers problem using mutual exclusion locks that is fair to both the readers and writers (Hint: use an FCFS discipline in the solution).
24. Repeat Exercises 18 and 19 using monitors.
25. Repeat Exercises 18 and 19 using Java.
26. Implement Example 17 using counting semaphores, allowing at most n simultaneous readers or 1 writer into the database at any time.
27. Figure 12.34 gives a monitor solution for the Dining Philosophers problem. Re-implement the solution using mutual exclusion locks.