Operating Systems: Concepts

Operating System

- Exploits the hardware resources of one or more processors
- Provides a set of services to system users
- Manages secondary memory and I/O devices

Basic elements

- Processor (if only one, CPU)
- Has at least two internal registers
  - Memory address register (MAR)
    - Specifies the address for the next read or write
  - Memory buffer register (MBR)
    - Contains data written into memory or receives data read from memory

Basic elements

- Main memory
  - Volatile
    - If computer shuts down, memory lost
  - Referred to as real memory, primary memory

Basic elements

- I/O modules
  - Secondary memory devices
  - Communications equipment
  - Terminals
- System bus
  - Communication among processors, main memory, I/O modules

Computer Components
Top-Level view

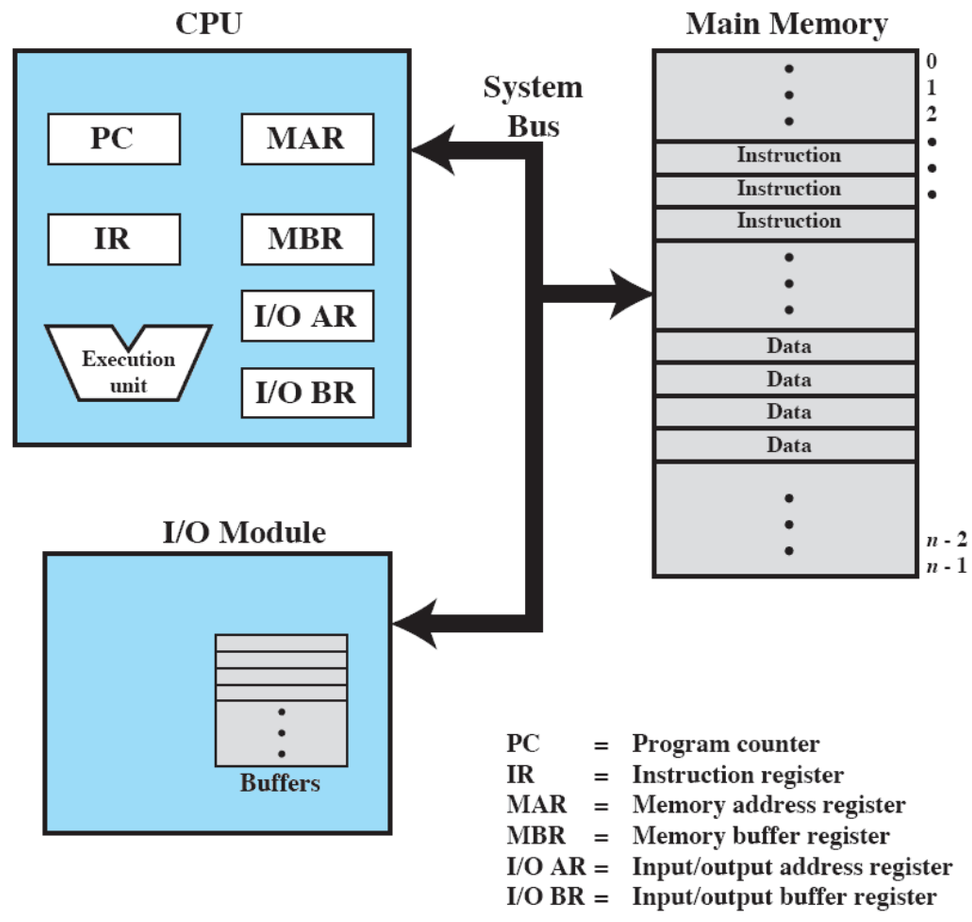


Figure 1.1 Computer Components: Top-Level View

Processor registers

- User-visible registers
  - Enables programmer to minimize main memory references by optimizing register use
- Control and status registers
  - Used by processor to control operating of the processor
  - Used by privileged OS routines to control execution of programs

User-Visible registers

- May be referenced by machine language
- Available to all programs (application, system)
- Divided into
  - Data registers
  - Address registers
    - Index register: Adding an index to a base value to get effective address
    - Segment pointer: When memory is divided into segments, memory is referenced by segment and offset
    - Stack pointer: Points to top of stack

Control and Status registers

- Program counter (PC)
  - Contains address of an instruction to be fetched
- Instruction register (IR)
  - Contains the instruction most recently fetched
- Program status word (PSW)
  - Contains status information

Control and Status registers

- Condition codes or flags
  - Bits set by processor as a result of operations
  - Examples:
    - Positive, negative, zero, or overflow result
    - Can be used for further operations, either explicitly or automatically

Instruction Execution

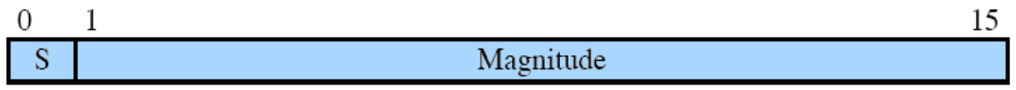
- How do machines execution instructions?
- Simplest is two-step process (instruction cycle) in a loop
    - Processor fetches instruction from memory (fetch stage)
        - Address of next instruction is contained in PC register
    - Processor executes fetched instruction (execution stage)
- When does this end?
    - Only if turned off, unrecoverable error, or an explicit instruction to halt

Instruction Execution

- Fetched instruction is loaded into instruction register (IR)
- Processor decodes instruction and performs the action
- Usually can be classified into these categories
    - Processor-memory - Data transfer from processor to memory
    - Processor-I/O - Transfer to I/O module or device (peripheral)
    - Data processing - Arithmetic or logic operation on data
    - Control - Alter the sequence of instructions
- One instruction can also combine several of these

Instruction Execution

(a) Instruction format



(b) Integer format

Program counter (PC) = Address of instruction
Instruction register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

(c) Internal CPU registers

0001 = Load AC from memory
0010 = Store AC to memory
0101 = Add to AC from memory

(d) Partial list of opcodes

Figure 1.3 Characteristics of a Hypothetical Machine

Instruction Execution

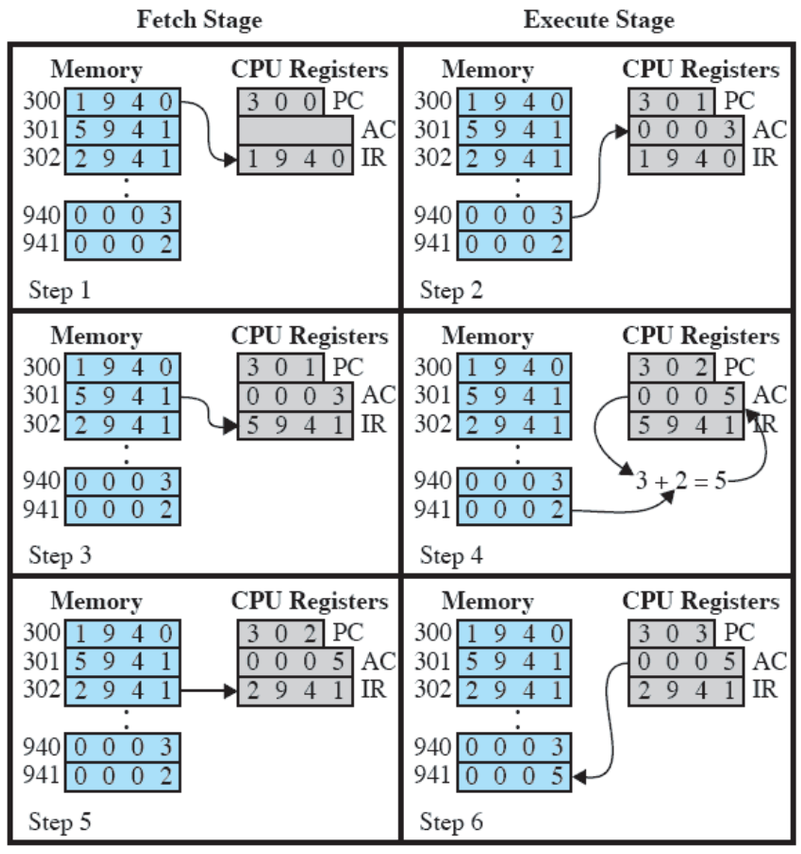


Figure 1.4 Example of Program Execution (contents of memory and registers in hexadecimal)

Interrupts

- Interrupts are a mechanism to interrupt normal sequence of a processor
  - Event that alters the sequence of instructions executed by CPU
    - Correspond with electrical signals generated by hardware circuits (in and out of the CPU)
- Help to improve CPU utilization
  - Consider typical CPU working at 3 GHZ, or 3x10^9 ips
  - Typical HDD will have rotation of 7200RPM, half a track in 4ms
  - Could have executed 12million instructions in that time

Classes of interrupts

Table 1.1 Classes of Interrupts

Program	Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, and reference outside a user's allowed memory space.
Timer	Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.
I/O	Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.
Hardware failure	Generated by a failure, such as power failure or memory parity error.

Interrupt Stage

- Processor checks for interrupts
- Does not require any extra programming
- If an interrupt is generated:
  - Suspend execution of program
  - Execute interrupt-handler routine

Types of interrupts

- Synchronous interrupts/Exceptions
  - Produced by CPU control unit
  - Control unit issues them only after terminating execution of instruction
- Asynchronous interrupts
  - Generated by other hardware devices at arbitrary times
  - No respect to CPU clock signals

Kernel tasks

- Most of the tasks performed by a kernel are in response to a process request
  - Dealing with a special file
    - Sending an `ioctl`
      - System call for a device-specific i/o operation
      - Or something that cannot be requested by regular system call
  - Issuing a system call
- Also to communicate between hardware on the machine

CPU interactions

- Two types of interactions between CPU and hardware
  - CPU giving orders
  - Hardware many tell something to CPU
    - Harder to deal with
    - Often have small buffers, if not answered immediately, could lose data
- I/O operations take awhile, stopping user process and might need to tell CPU to switch processes

Interrupts and execution cycle

- Remember we have synch or asynchronous execution of instructions and I/O
- When ext. device is ready, I/O module sends interrupt request to CPU
- CPU responds by suspending current process, branches to int. handler
    - Will return after processing request
- Interrupt could occur at any point in programs execution
    - Might not depend on any instruction
- User code will not need any special code to deal with this. CPU and OS handle it

Transfer of Control via Interrupts

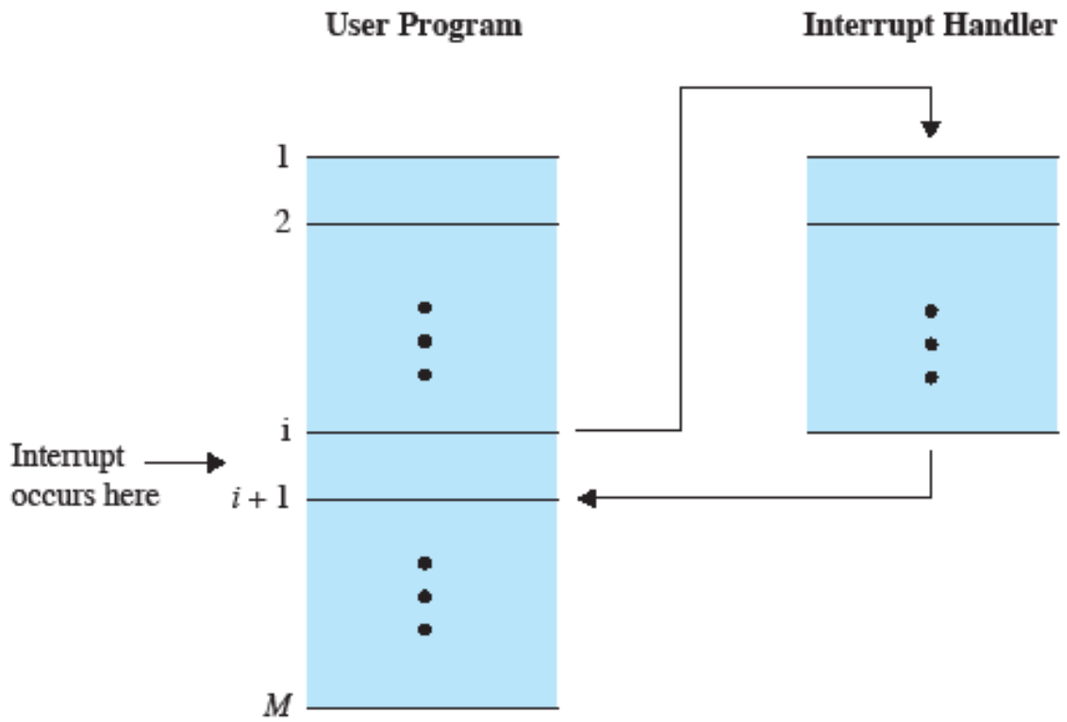
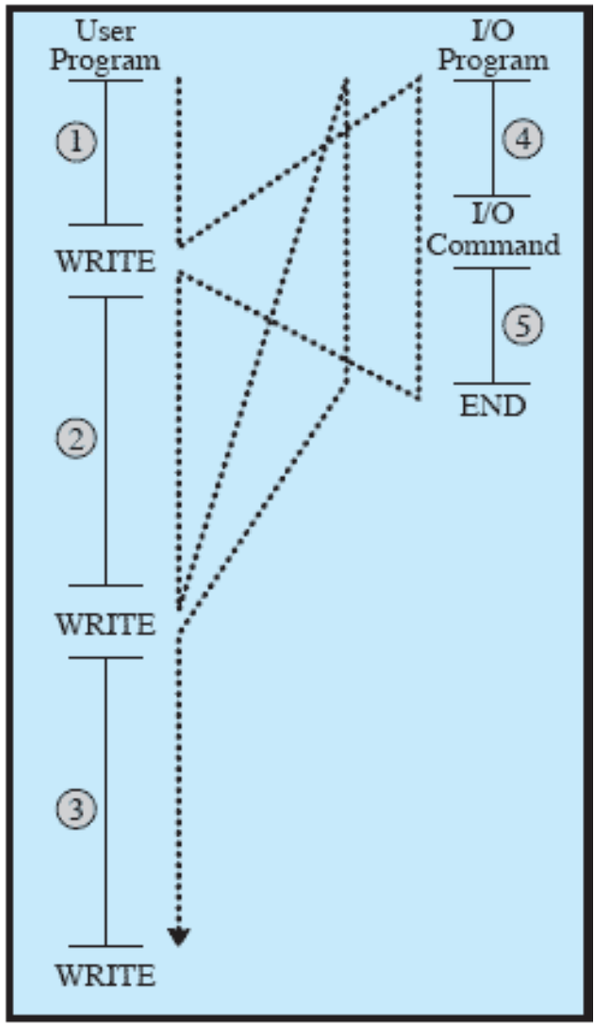
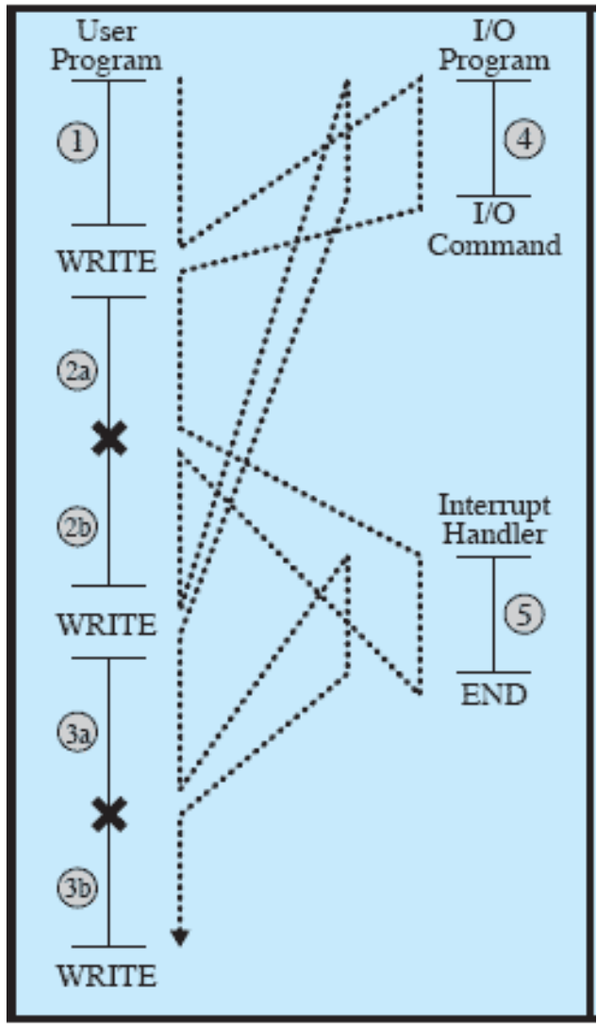


Figure 1.6 Transfer of Control via Interrupts

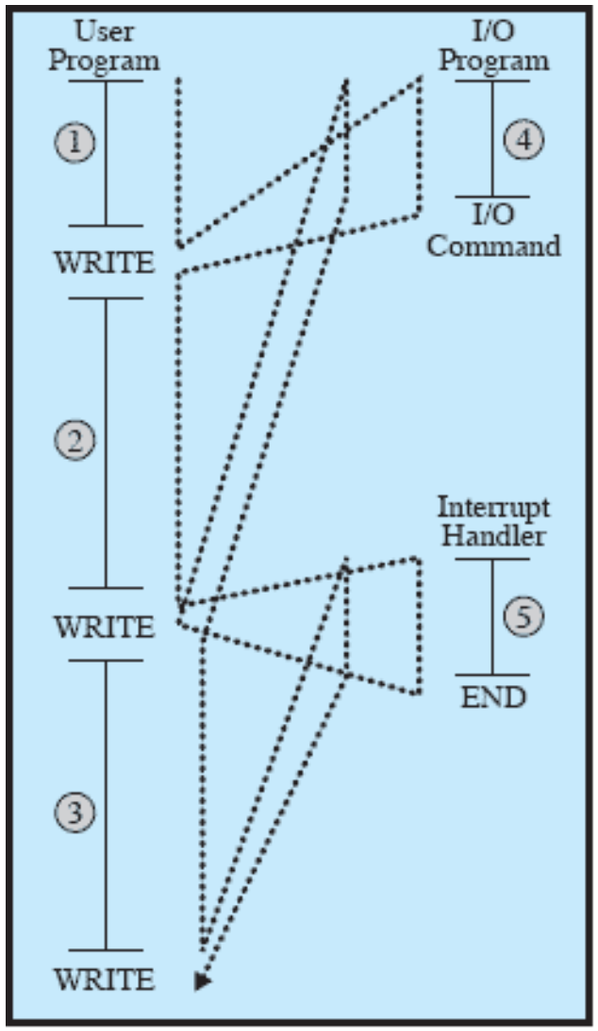
Program flow of control



(a) No interrupts



(b) Interrupts; short I/O wait



(c) Interrupts; long I/O wait

Short I/O wait

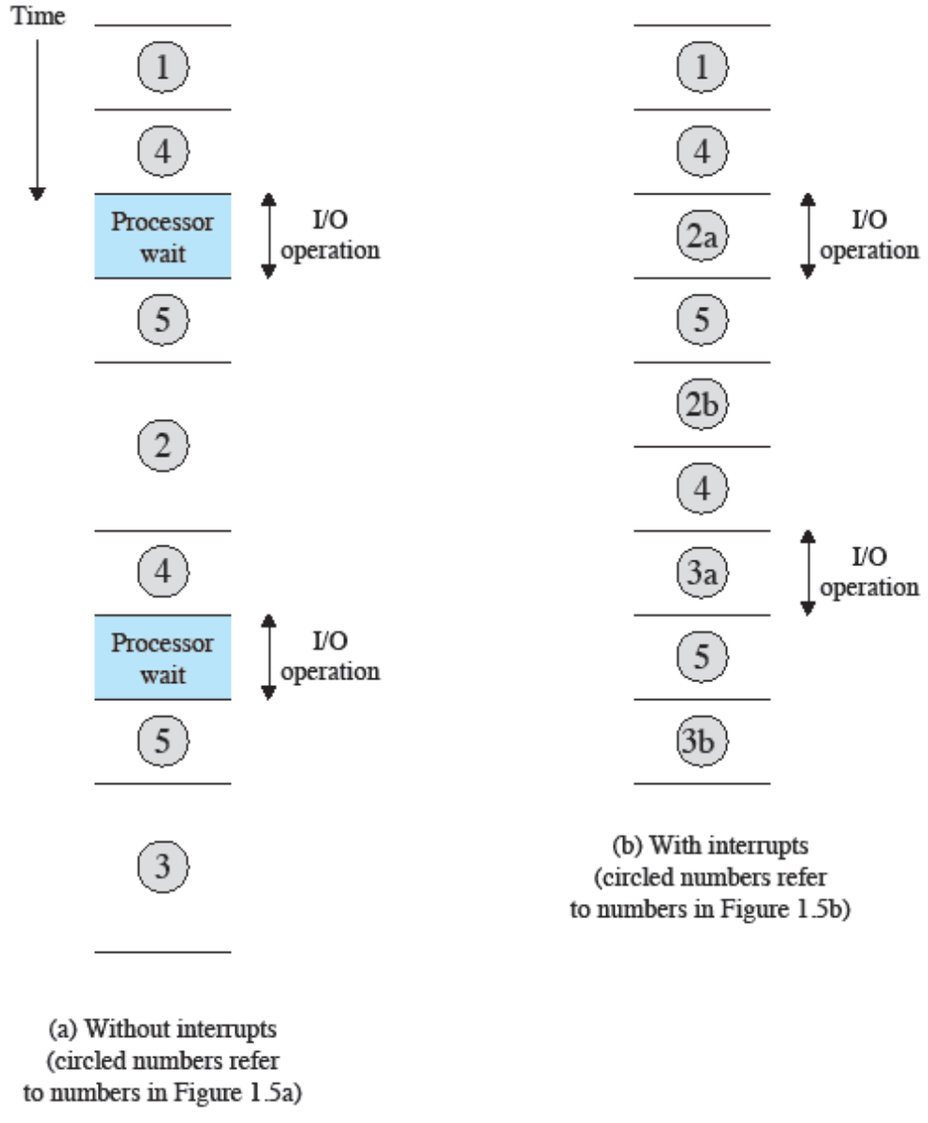


Figure 1.8 Program Timing: Short I/O Wait

Interrupts

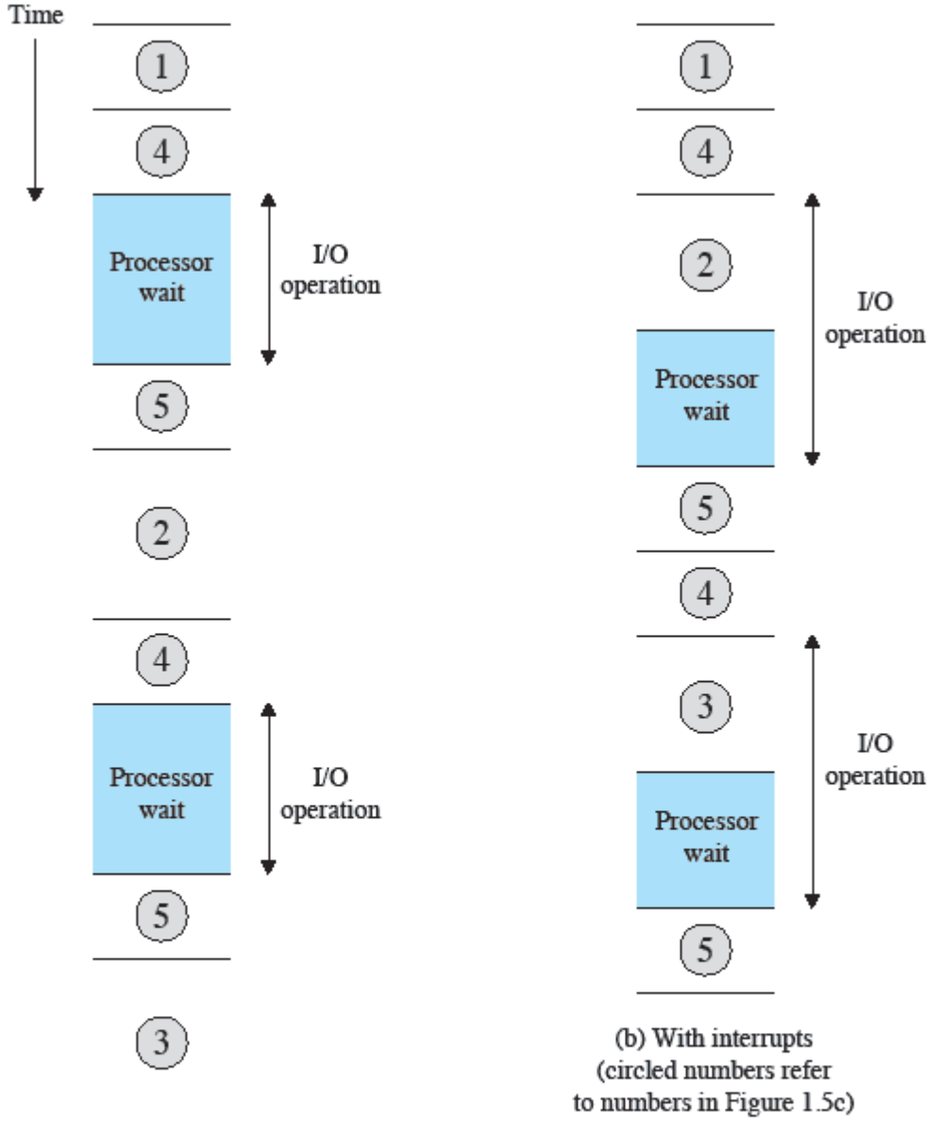


Figure 1.9 Program Timing: Long I/O Wait

Instruction Cycle with Interrupts

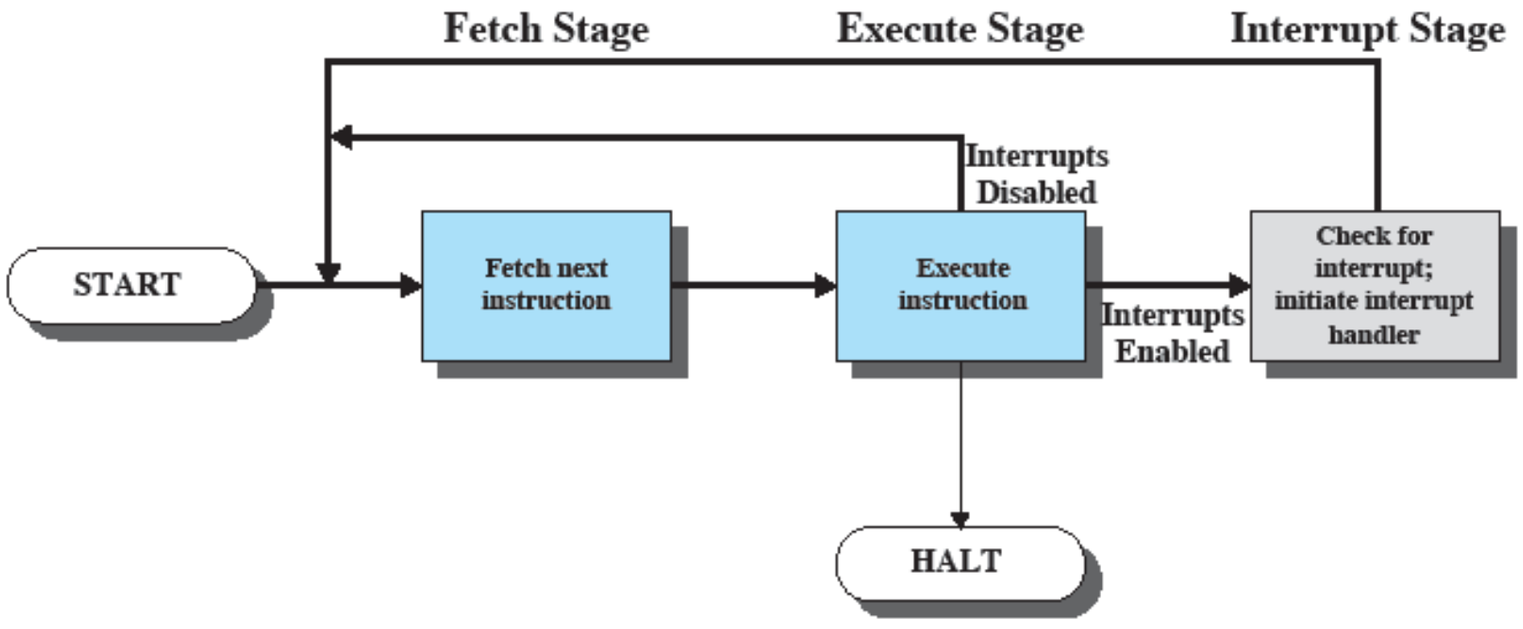


Figure 1.7 Instruction Cycle with Interrupts

Interrupt process

- If there is an interrupt (flag is set), CPU goes to interrupt handler
- Interrupt handler looks at type of interrupt and executes appropriate function
- Some overhead from doing this
  - Less than if CPU had to do everything

Interrupt constraints

- Interrupt handlers have some constraints on them
  - Must be very fast: Parts that must be executed immediately
    - Deferrable part that can be handled later
      - Block of data arrives on network line
      - Kernel marks presence of data (urgent part)
        - Gives control back to process running before on CPU
      - Rest of processing can be done later (move data to buffer)
- Interrupt handler uses interrupt vector
  - Table of pointers of addresses of interrupt service routines

Interrupt processing

- Device issues interrupt to CPU
- CPU finishes execution of current instruction
- CPU tests for pending interrupt request
  - If so, inform device, this removes interrupt signal
- CPU saves program status word onto control stack
- CPU loads the location of interrupt handler into PC register
- Save content of all registers from control stack to memory
- Find out the cause of interrupt, invoke appropriate routine
- Restore saved registers from the stack
- Restore PC to dispatch the original process

Interrupt Processing

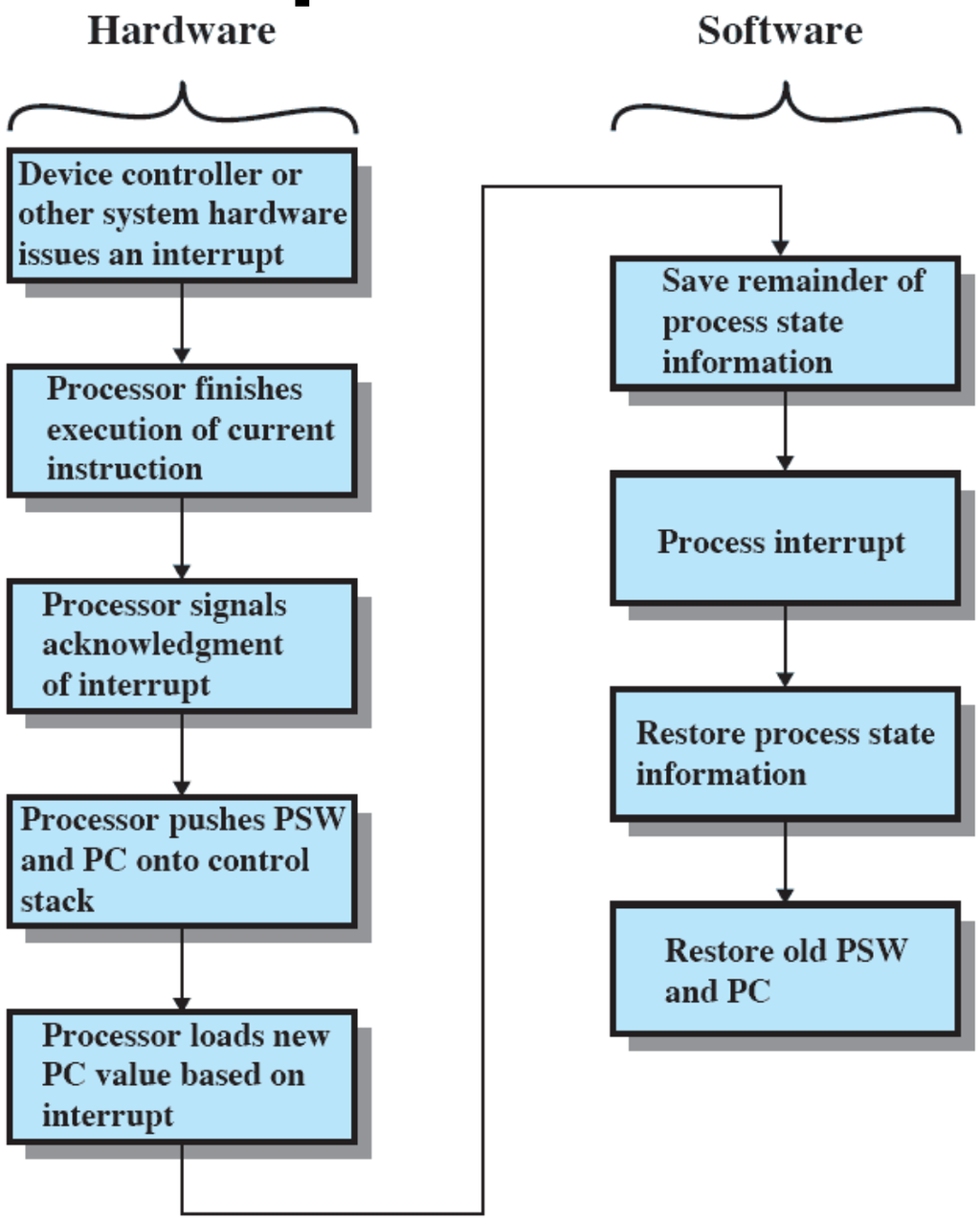
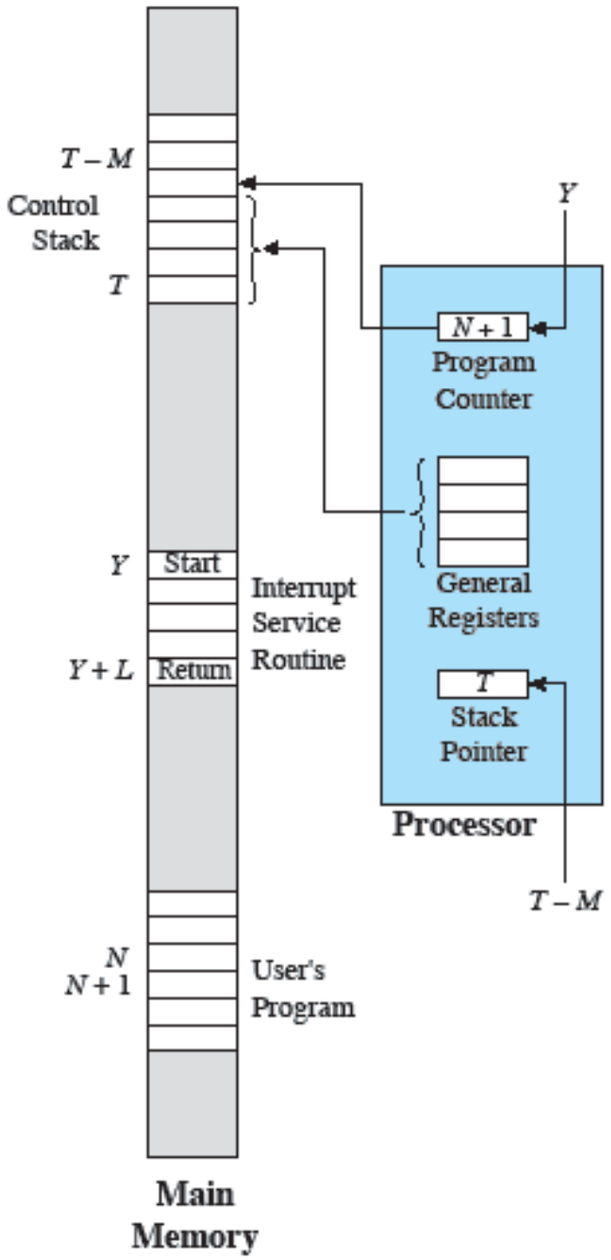
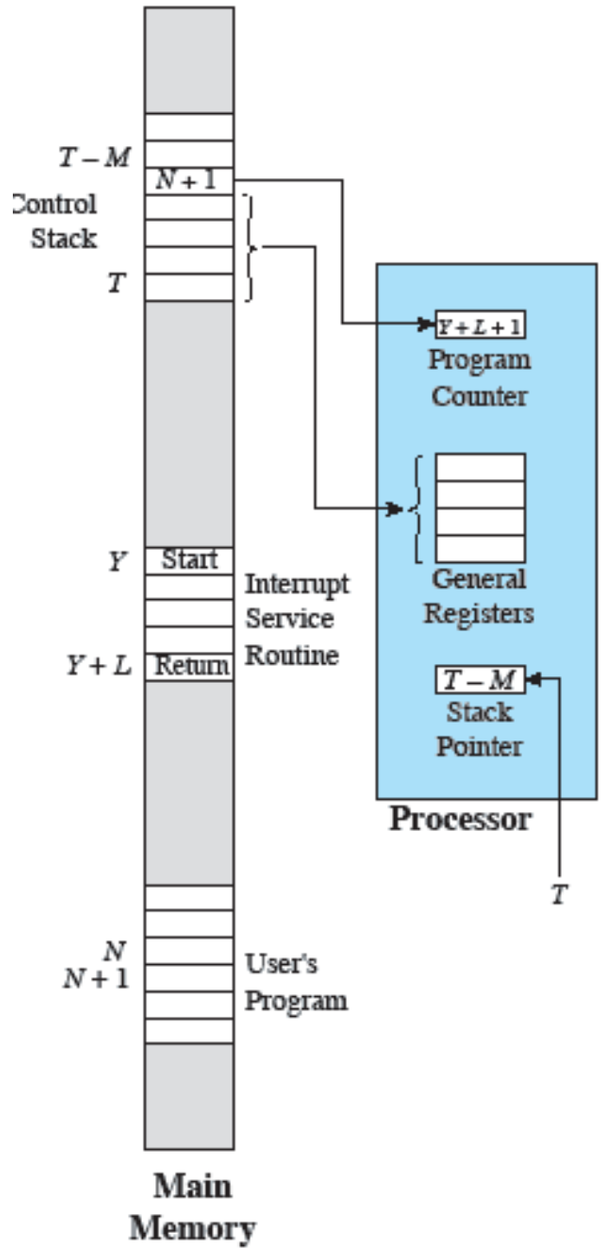


Figure 1.10 Simple Interrupt Processing

Machine state for interrupts



(a) Interrupt occurs after instruction at location N

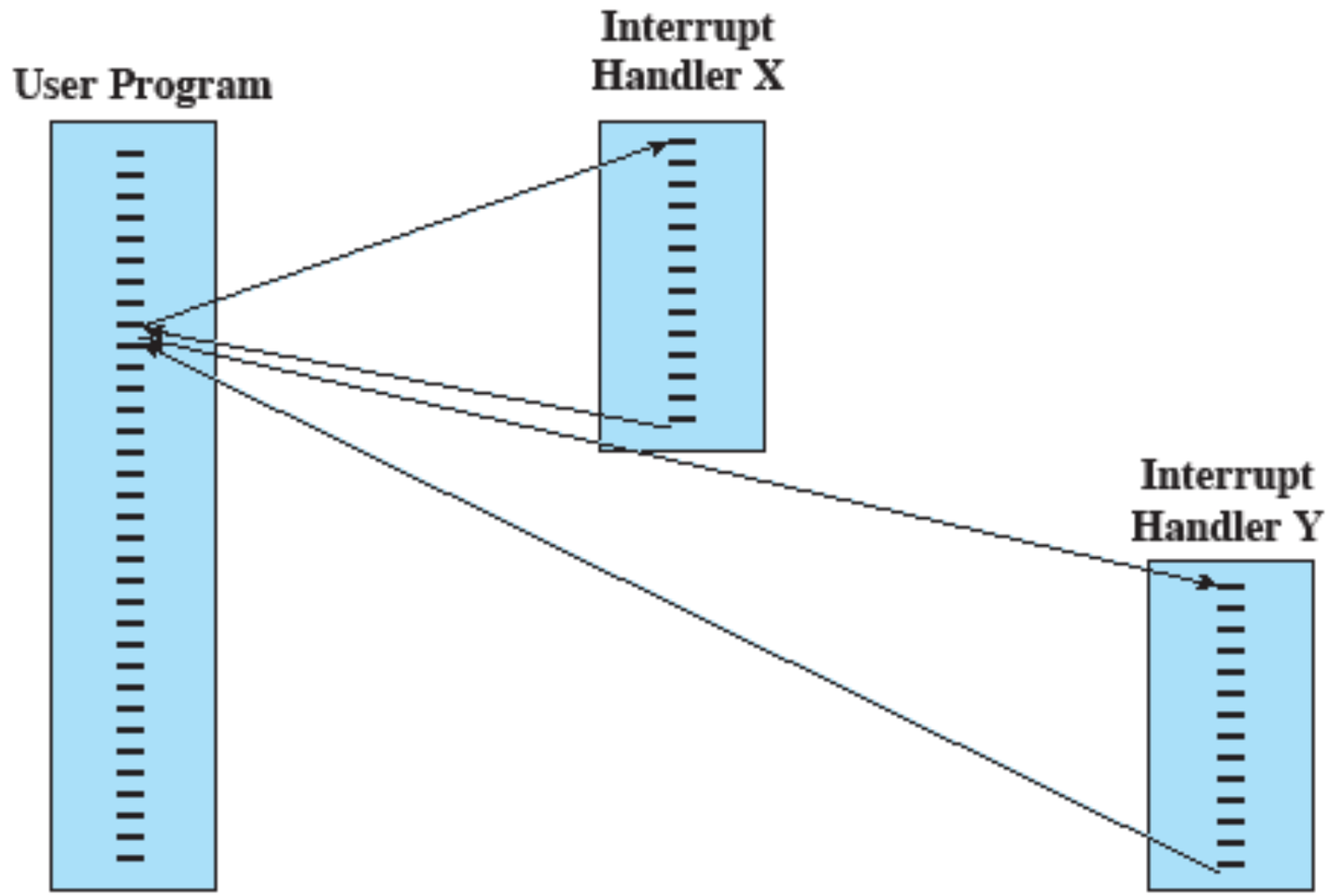


(b) Return from interrupt

Multiple interrupts

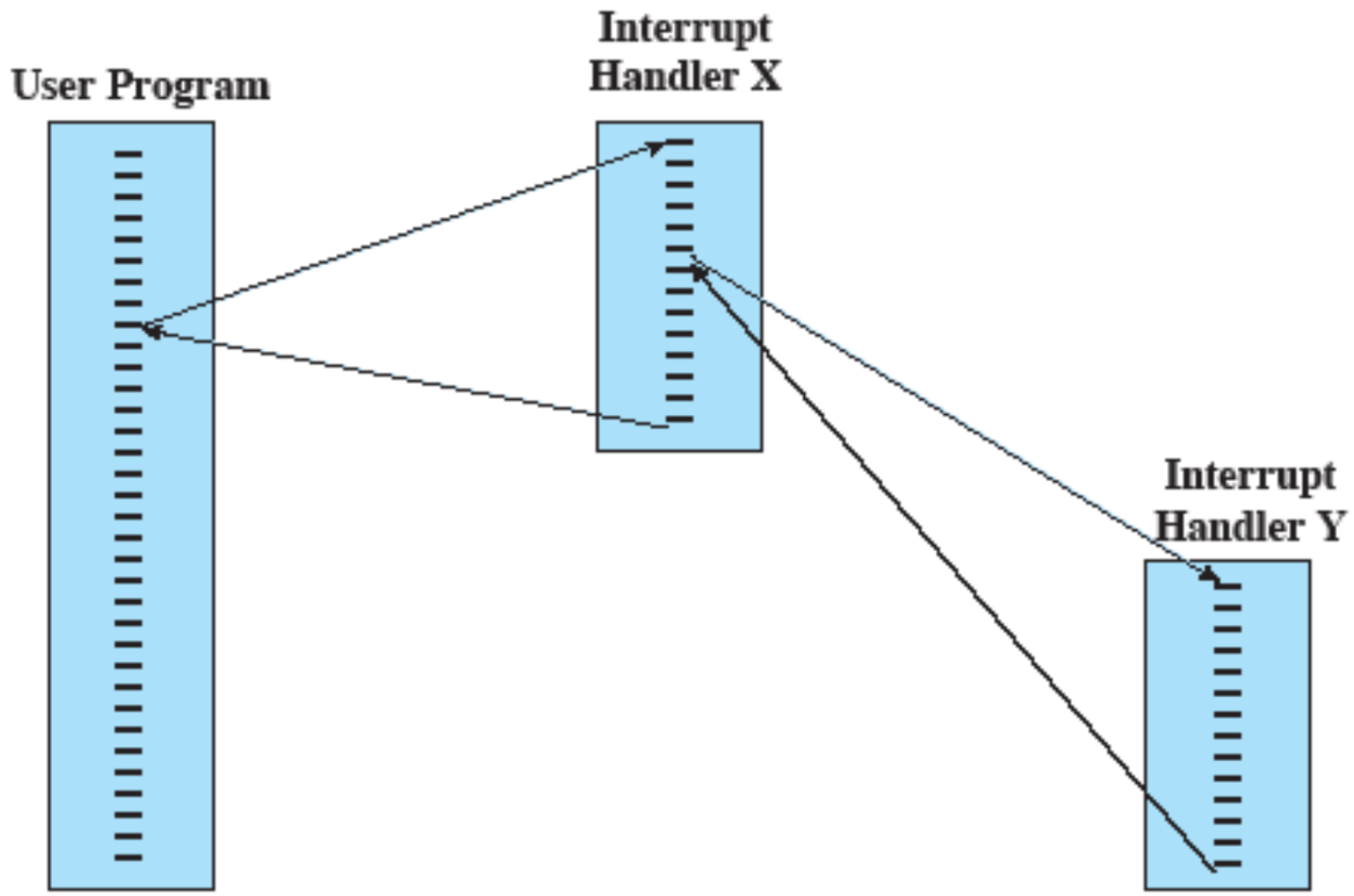
- Also need ability to handle nested interrupts
- We could be lazy and disable interrupts while one is occurring
  - Would not account for high priority or time-critical interrupts
  - Data may be lost due to buffer sizes
- We like interrupts so much we want to allow our interrupts to interrupt our interrupts!
  - How do we handle this?

Sequential Interrupt Processing



(a) Sequential interrupt processing

Nested Interrupt Processing



(b) Nested interrupt processing

Multiple interrupts

- Define interrupt priorities
  - Allow higher priority interrupts to interrupt a lower-priority interrupt handler
- Interrupts do need to be disabled inside some kernel critical regions
  - These critical regions do need to be small
- Interrupt handlers start with interrupts disabled (hardware part)
  - Allows exclusive use of a few registers

Multiple interrupts

- Some architectures allow interrupts to be interrupted by higher priority one
  - ARM processors have two interrupt levels
    - IRQ - normal interrupts (interrupt request)
    - FIQ - fast interrupt
- Each type has its own register bank for PC, SP, interrupt vector
  - IRQ handler starts with IRQ disabled but FIQ enabled
  - FIQ handler starts with both IRQ and FIQ disabled
  - FIQ can be triggered int he middle of handling an IRQ

Types of interrupts

- Maskable interrupts
  - All IRQs issued by I/O devices
  - Can be in two states: masked or unmasked
    - Masked interrupts are ignored by control unit unless unmasked
    - Examples are mouseclicks, memory read, etc
- Nonmaskable interrupts
  - Generated by critical events such as hardware failure
  - Examples would be powerfailure, softwarecorrupted, etc

Exceptions vs Interrupts

- Exception can be thought of as software-only version of an interrupt
- Only effects current process
- Two main types:
  - Process-detected exceptions
  - Programming exceptions

Process-detected exceptions

- Detected as an anomalous condition while executing an instruction
- Can be further divided into three groups
  - Faults
    - Can be corrected
    - Process restarts
  - Aborts
    - Used to report serious errors, hardware failures, inconsistent symbol tables
    - Handler usually will force affected process to terminate

Process-detected exceptions

- Traps
  - Reported by process after kernel returns control to process
  - Triggered when no need to re-execute instruction that terminated
  - Used primarily for debugging
  - Simply signals that a specific instruction has been executed
    - Breakpoints, watchpoints (debugging)
  - Execution continues from following instruction, after data is examined

Programming exceptions

- Occurs at request of programmer
- Handled by control unit as traps, that is software interrupts
- Used to implement system calls and to notify debugger of specific event

Interrupts in Linux

- Known as IRQs - interrupt requests
- Short IRQs - Expected to take a short amount of time
    - Rest of machine is blocked and no other interrupts are handled
- Long IRQs
    - Can take longer
    - Other interrupts may occur, but not from same device

Interrupts in Linux

- Handling interrupts in Linux
  - CPU receives interrupt, stops what it is doing
    - Unless it is processing higher order interrupt
  - Saves parameters on stack and calls interrupt handler
  - System is in unknown state, so not all things allowed with handler
    - Interrupt handler does what it needs to do immediately (usually read from or write hardware), schedule handling of new information (bottom half) at a later time and return back
    - Kernel is then guaranteed to call bottom half as soon as possible
      - All that is allowed in kernel modules is allowed

Interrupts in Linux

- Each interrupt/exception identified by a number in the range (0,255) called a vector
- Vectors of nonmaskable interrupts and exceptions are fixed
- Vectors of maskable interrupts can be altered by programming interrupt controller

Signals

- Mechanism to allow interactions between user mode processes
- Notify processes of system events
- A short message that may be sent to a process or a group of processes
  - Message only consists of a number to identify the signal
  - No room for argument or data

Multiprogramming concerns

- In practice, more than one core and more than one program executing
- Sequence in which programs are executed depends on relative priority and whether waiting on I/O
- An interrupted program is not guaranteed to get back control when that interrupt handler is done
    - Could go to another program/process

Memory Hierarchy

- We want as much memory as we can get
- We want it as fast as possible
- Problems:
  - Greater the capacity, the smaller cost per bit/byte
  - Greater the capacity, the slower the access speed
- More memory you have, the more applications will use
  - Applications grow in size simply based on available memory

Memory Hierarchy

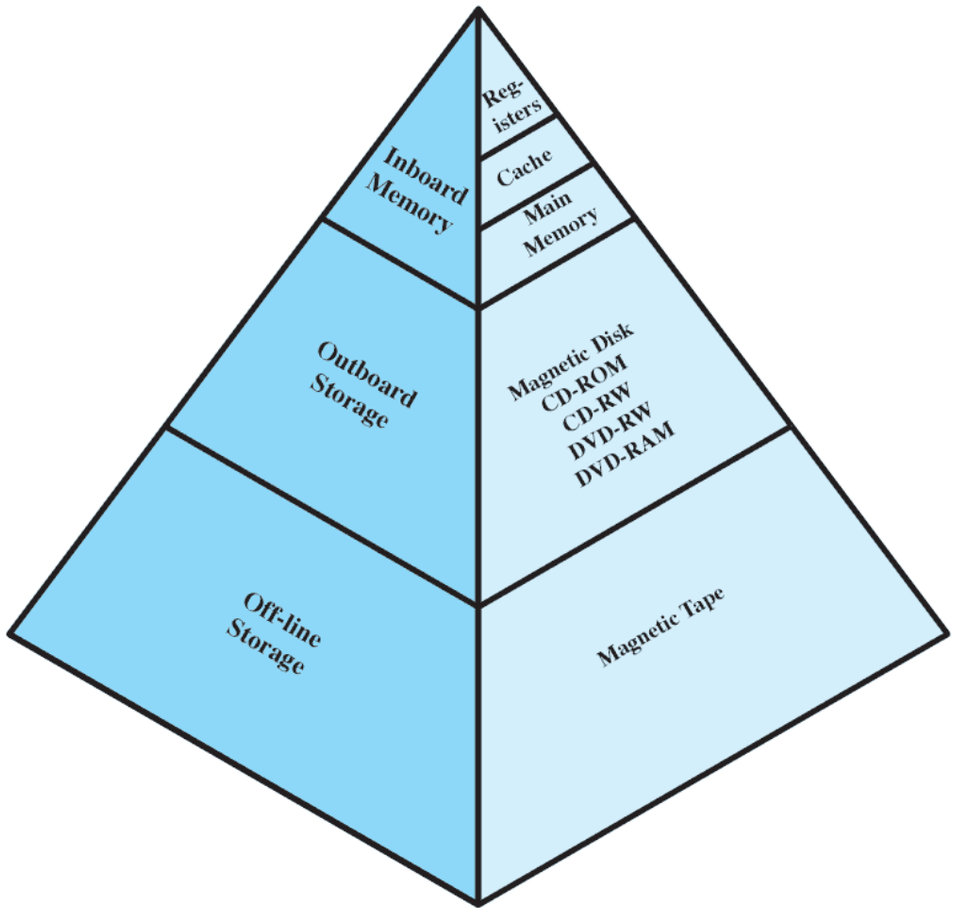


Figure 1.14 The Memory Hierarchy

Memory Hierarchy

- As we go down in hierarchy
  - Decreasing cost per bit
  - Increasing capacity
  - Increasing access time
  - Decreasing frequency of access to memory by processor

Memory Hierarchy

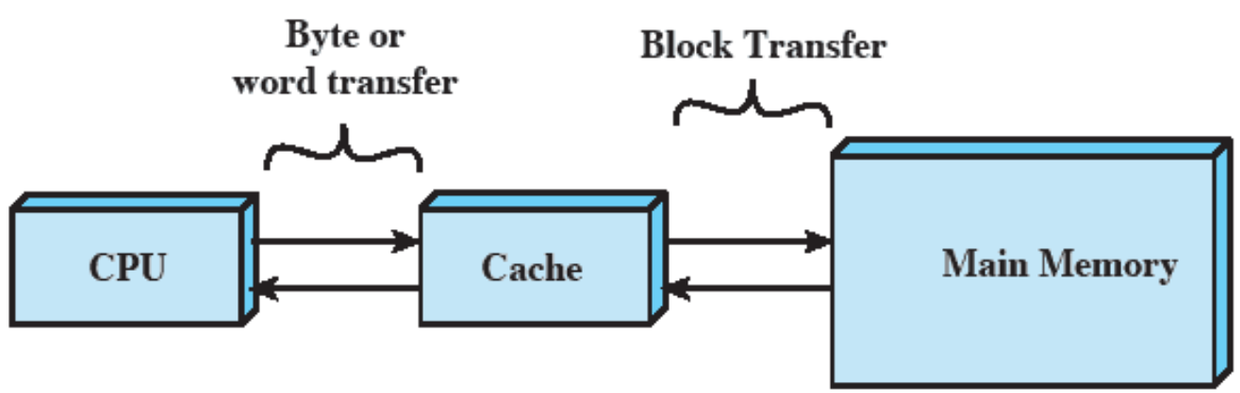
- Registers
  - A few bytes
    - Reading at almost CPU speed
- Cache memory
  - Few megabytes
  - Nanoseconds to read

Memory Hierarchy

- Main memory or RAM
  - Implemented using semiconductor technology called DRAM
  - Essentially array of bytes, index of array providing address
  - Gigabytes in total size
  - Microseconds to read
- Secondary memory
  - External
  - Non-volatile
  - Milliseconds

Memory Hierarchy

- Ideal would be to have large amounts of register or cache memory
  - Impractical
- In practice, can almost get the best of both worlds
  - Almost always use fast memory
  - Only rarely require the use of slower memory



Memory Hierarchy

- Use hierarchical memory to transfer data from lower memory M2 (away from CPU) to higher memory M1 (close to CPU) when needed
    - Smaller/expensive/faster memory supplemented by larger/cheaper/slower memory
- Only works if we can mostly use the faster memory

Hit ratio and access time

- Hit ratio H is fraction of accesses that are in faster memory
- Higher H, better the more likely we use the faster memory

```
T1 is access time in smaller/faster memory
T2 is access time of larger/slower memory

Average access time T is
T = H*T1 + (1-H)*(T1 + T2)
  = T1 + (1-H)T2
```

- Consider two levels of access with 0.1 and 1.0 microseconds respectively
  - Hit rate of 0.95

```
Average access time = 0.95*0.1µs + 0.05 * (0.1µs + 1.0µs)
                    = 0.095 + 0.055µs
                    = 0.15µs
```

Locality of reference

- This memory hierarchy only works if hit rate is high
  - Most of the references in memory are clustered
  - If we move to another cluster, most reads come from that
- Needs to work for both instructions and data
- That over any short period, CPU works with only limited set of memory locations
- Is this true in practice?
  - Yes, why?

Locality of reference

- Why clustered references happen
  - Programs execute sequentially
    - One instruction after another
    - Relatively few branches in code
    - Next instruction almost always follows previous
  - Rare to have long sequence of function calls
  - Most iterative constructs have small number of instructions
  - Data structures tend to make references to nearby locations in successive instructions

Locality of reference

- Spatial locality
  - Reference to memory locations that are clustered
  - Sequential instruction execution
  - Accessing data locations sequentially (arrays, what about LL?)
  - Exploited by pre-fetching mechanisms in pipelined architectures

Locality of reference

- Temporal locality
  - The more recently it was used, more likely it will be used in the future
  - Access locations that have been used most recently
  - Executing same instructions repeatedly in a loop (function calls)

Memory hierarchy

- How can we take advantage of this?
- Organize code and data across hierarchy
  - Percentage of accesses to lower level should be less than accesses at higher level
  - Move clusters of memory at once form higher to lower level

Cache memory

- Use of very fast memory designed to contain data for fast access by CPU
  - Often not directly visible by the system
- Used to temporarily hold data for transfer between main memory and registers
  - Exploits temporal locality
- Transfers blocks of data from memory to cache
  - Exploiting locality of reference

Cache/Main-memory Structure

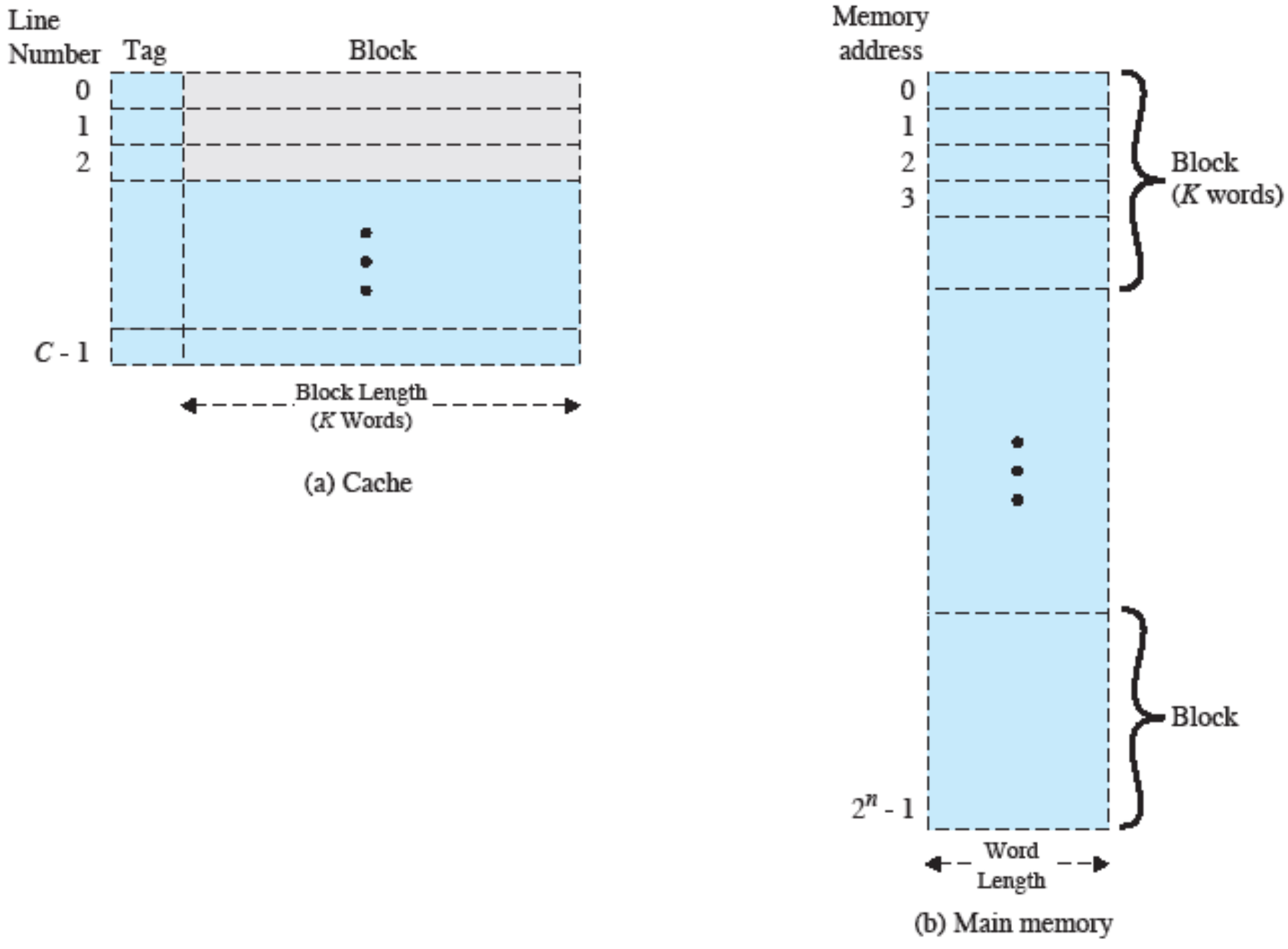


Figure 1.17 Cache/Main-Memory Structure

Cache Read Operation

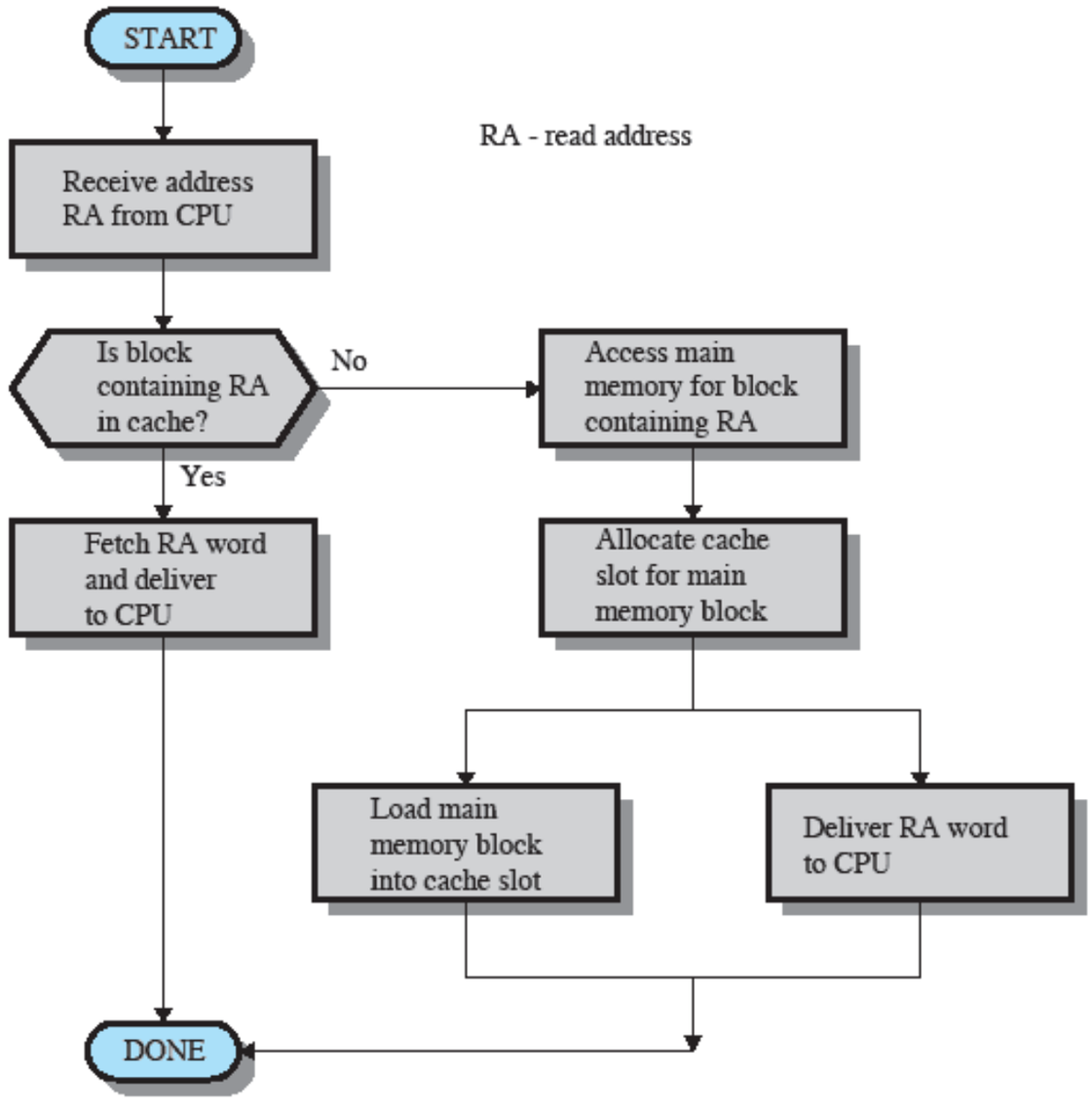


Figure 1.18 Cache Read Operation

Cache principles

- Cache size
  - Even small caches help significantly with performance
- Block size
  - Unit of data exchanged between cache and main memory
  - Larger sizes yield more hits until probability of using newly fetched data becomes less than the probability of reusing data that has to be moved out of cache

Cache principles

- Mapping function
  - Determines which cache location the block will occupy
- Replacement algorithm
  - How to choose which block to replace in cache
  - Least-recently-used (LRU) algorithm is common

Cache principles

- Write policy
  - Dictates when memory write operation takes place
    - Can occur every time the block is updated
    - Can occur when the block is replaced
      - Minimize write operations
      - Leave main memory in an obsolete state

Disk Cache

- Same ideas extended to secondary memory
  - Designate a portion of main memory for disk read/write
- Improves performance in several ways:
  - Clustering disk writes, as fewer large data transfers instead of many small ones
  - Data destined for disk can be read back before it is written

Memory management

- One of the most important services provided by an OS
- Major areas of responsibility:
  - Process isolation
    - Prevent independent processes from interfering with each others data and code segments
  - Automatic allocation and management
    - Programs should be dynamically allocated across the memory depending on availability
    - Programmer should not be able to perceive this
      - Good for the programmer, doesn't have to worry about it

Memory management

- Major areas of responsibility
  - Modular programming support
    - Programmers should be possible to define program modules
    - Should be able to dynamically create/destroy/alter size of modules
  - Protection and access control
    - Different programs should be able to co-operate in sharing memory space
    - Sharing should be controlled and processes should not be able to do this independently

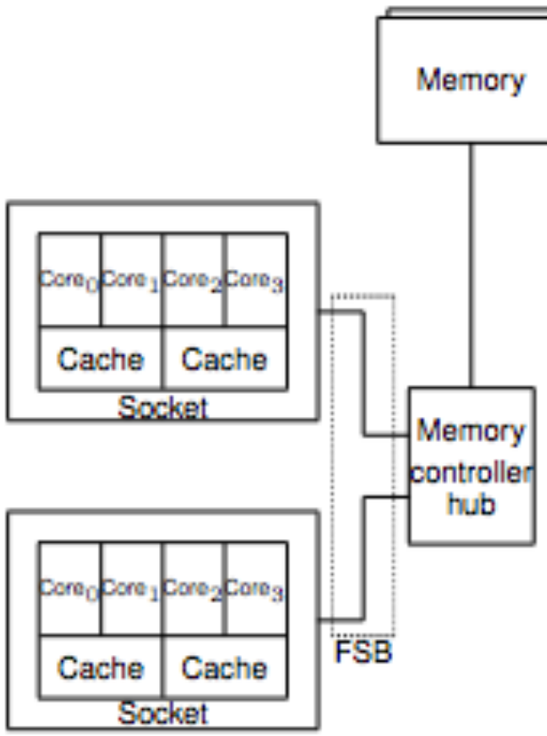
Memory management

- Major areas of responsibility
  - Long-term storage
    - Users and applications require means for storing information for long periods of time
      - Generally implemented with file system
- OS often separates memory into two distinct views
  - Basis of virtual memory:
    - Physical : How much actual physical memory
    - Logical : How much can the programs see

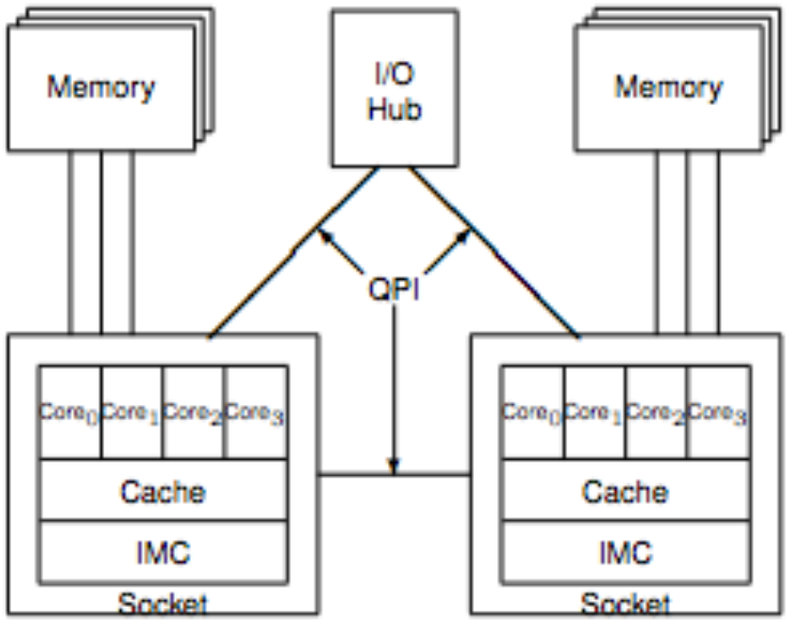
Memory bandwidth

- Keep getting more cores and larger caches
- Total processing speed going up faster than memory sub-system components
    - Can bottleneck on moving data between memory and cores
        - Negates benefit of multiple cores
- Use non-uniform memory access (NUMA) rather than FSB
    - IMC - integrated memory controller
    - QPI - Quick path Interconnect
        - point-to-point link electrical interconnect specification
        - High-speed communications among connected processor chips

Memory management



Front Side Bus



Non Uniform Memory Access

Multiprocessor organization

- Traditional view of computer as a sequential machine
  - Instructions executed in a sequence, one at a time
  - Instruction execution cycle
  - Instruction pipelines
- Multiprocessors have two or more processors in close communication
  - Share system bus, clock, memory, peripheral devices
  - Result in increased throughput, economy of scale, reliability

Multiprocessor organization

- Symmetric multiprocessing/SMP
  - Two or more processors of similar capability
    - Any process and any thread can run on any available processor
  - Processors share same memory and I/O
    - Could be through different paths
  - Shared memory used to support communication and synching
  - Very high degree of communication compared to clusters

Multiprocessor organization

- Multicore computers
  - Combine 2 or more processors on single die
  - Each core contains all components
    - Even their own caches
  - Cores share system bus, clock, memory, peripheral devices
  - Economy of scale, increased reliability, increased thoroughput

Multiprocessor organization

- Array processing / SIMD
  - Single instruction, multiple data
  - Multiple processing elements that perform the same operation on multiple pieces of data at the same time
  - Only a single instruction at any time
- General multiprocessing / MIMD
  - Processors that function asynchronously and independently

Multiprocessor organization

- Cluster computing and fault tolerance
  - Group of individual loosely-coupled computers with their own OS
    - Linked by high-speed interconnection
    - Usually in the same room
    - Virtual server accepts requests and directs them to one in a series of servers
    - High degree of availability
      - each node monitors some of the other nodes over LAN
      - One node goes down, others can take over

Multiprocessor organization

- Cluster computing and fault tolerance
  - Provides load balancing, fault tolerance, high scalability
  - Suitable for high-end, transaction processing applications
  - Often does firewalls, web servers, domain name servers, mail servers
  - Grid or Cloud computing
    - Grid is decentralized, Cloud is centralized around one owner
    - Linking a number of separate fully function computers into a large cluster on demand
    - Applications can run like a screen-saver
      - Take up unused CPU cycles

Cloud computing

- Software as a service, or Internet as a platform
- Lets us move data and programs from desktops and server rooms to the compute cloud
  - Google docs, Apple iCloud, Microsoft Office 365
- No central computer as required in time sharing topology
- No infrastructural investment in terms of OS updates
  - More mobility and collaboration
- From vendor perspective, no need to develop applications for tons of diverse platforms
  - Only need clients for each platform

Cloud computing

- Cloud OS
  - User interface resides in a single window in a web browser
  - OS inside a browser - eyeOS
  - Can bypass the web browser; more capable software can run as separate application and communicate with the servers in the cloud directly
- Application scalability
  - Coordinate information coming from multiple sources
  - Many-to-many communication - each server talks to multiple clients and each client may invoke programs on multiple servers

Classification of OS

- Low-end computer desktop
  - Assumes no intelligence on part of user
  - Focuses on hardware and software compatibility
- High-end desktop
  - User is smart but not necessarily computer savvy
  - Must provide good performance
  - Provide hardware and software compatibility

Classification of OS

- Real-time systems
  - User is professional programmer
  - Focus on performance, defined response time, extendable hardware support, programming control
- Embedded systems
  - Closed box system to do specific functions
  - No expansion slots
  - Limited applications
  - Turn it on and use it (starts, scans interfaces, runs multiple apps if required)

Operating System concepts

- Program
  - Collection of instructions and data kept in ordinary file on disk
  - Executable program, complete code output by linker/loader
    - Generated from source program and object code
    - Possible extras from library
- File is marked as executable in the i-node
- File contents are arranged according to rules established by the kernel

Operating System concepts

- Process
  - Created by kernel, environment in which a program executes
  - May be stopped and later restarted by OS
  - Process executes sequence of instructions in address space
    - Address space is set of allowable memory for this program
- Core image
  - Instruction segment
  - User data segment
  - System data segment
    - Accumulated CPU time, open files, etc

Operating System concepts

- Process cont.
  - Program initializes first two segments
    - Process may even modify instructions but more commonly data
- Process context
  - All the metadata about the process
  - Terminal, open files, register contents
  - Recorded in process table

Operating System concepts

- Process cont.
  - Process may acquire resources (more memory, files) not present in the program
  - Child and parent processes
  - Communication between processes through messages
  - Process id, UID (user), GID (group)

Operating System concepts

- Threads
  - Stream of instruction execution
    - A running instance of a process's code
  - Dispatch-able unit of work to provide intraprocess concurrency
  - Process may have multiple threads in parallel, each executing sequentially
  - Threads can execute in timesharing on single cpu, on multiple cores concurrently, or on multiple CPUs

Operating System concepts

- Files
  - Services of file management system to hide disk/tape specifics
  - System calls for file management
  - Directory to group files together
  - Organized as a hierarchical tree
  - Root directory
  - Path name
  - Path name separator

Operating System concepts

- Files
  - Working directory
  - File descriptor or handle
    - Small integer to identify a file in subsequent operations
    - Error code to indicate access denied
- I/O device treated as special file
  - Block special file
  - Character special file
- Pipe - pseudo file to connect two processes

System calls

- Interface between user program and operating system
- Set of extended instructions provided by the operating system
- Applied to various software objects like processes and files
- Invoked by user programs to communicate with kernel and request services
- Access routines in the kernel that do work for you

System calls

- Executing system calls in Linux
  - Process fills the register with appropriate values
  - Calls special instruction to jump to previously defined location
    - Known as `system_call` in kernel
- Done by interrupt 0x80 in Intel CPUs
- Procedure checks system call number to determine service requested
- Look into `sys_call_table` to see address of kernel function to call

System calls

- Library functions corresponding to each system call
  - Machine registers to hold parameters of system call
  - Trap instruction (protected procedure call) to start OS
  - Hide details from trap and make call look like ordinary procedure call
  - Return from trap instruction

System calls

- Broadly into six classes
  - filesystem (open)
  - process (fork)
  - scheduling (priocntl)
  - IPC (semop)
  - Socket/networking (bind)
  - Miscellaneous (time)

System calls

- System calls vs library functions
  - System calls run in kernel mode on user's behalf
    - Provided by kernel
- Library functions run in user space
  - Often provide interface to system calls
  - Example: printf is a library function that uses system calls

Shell

- Unix command interpreter
- User program, not part of kernel
- Supplies prompt, redirection of input/output
- Lets you launch background jobs

Shell

- For most commands, shell `forks` and the child `execs` command
  - Remaining words of command line as parameters
- Allows three types of commands:
  - Executable files
  - Shell-scripts
  - Built-in shell commands

Kernel

- Permanently resides in main memory
    - Loaded into memory by the bootstrap loader when computer powered on
        - Bootstrap loader is stored in ROM or firmware
- Controls execution of processes
    - Allows their creation, termination, suspension
    - Communication between processes

Kernel

- Schedules processes fairly for execution on the CPU
  - Processes share the CPU in a time-shared manner
    - CPU executes process
    - kernel suspends it when its time quantum elapses
    - Kernel schedules another process to execute
    - Kernel later reschedules the suspended process

Kernel

- Allocates main memory for an executing process
  - Allows processes to share portions of address space in some cases
    - Protects private address of a program from tampering
  - If system runs low on free memory, kernel frees memory by swapping some memory to secondary storage
    - swap device
  - If kernel writes entire process to a swap device, implementation is swapping system
    - If pages of memory are swapped, paging system
  - Sets up virtual to physical address map
    - Maps compiler generated addresses to physical addresses

Kernel

- File system maintenance
  - Allocates secondary memory for efficient storage and retrieval of user data
  - Allocates secondary storage for user files
  - Reclaims unused storage
  - Structures file system in a well understood manner
  - Protects user files from illegal access
- Gives processes controlled access to peripheral devices
  - Terminals, tapes, disk drives, network devices

Kernel

- Services provided by kernel transparently
  - Recognizes that a given file is a regular file or a device
    - Hides distinction from user processes
  - Formats data in a file for internal storage
    - Hides internal format from user processes
    - Returns unformatted byte stream
  - Allows shell to read terminal input, spawn processes dynamically, synchronize process execution, to create pipes and to redirect I/O

Kernel vs other processes

- In older OS's, kernel executed outside of any process
- Kernel has its own address space, own system stack to control procedure calls and returns
- In such systems, concept of process only applies to user programs
  - OS code executes in privileged mode

Micro-kernel architecture

- Smaller set of operations in a limited form assigned to kernel
  - Each OS layer is a relatively independent program
  - Interacts with other layers through well-defined/clean interfaces
  - Interprocess communication, limited process management and scheduling, some lowlvl I/O
  - Other OS services provided by processes, or servers, running in user mode

Micro-kernel architecture

- Less hardware specific because many system-specific operations are pushed into user space
    - Existing micro-kernel can be easily ported as less hardware-dependent components are encapsulated in micro-kernel
- Simplifies implementation, provides flexibility, better suited for distributed environment
- Microkernel OS makes better use of ram
    - Can swap out unneeded system processes
- Slower than monolithic ones
    - Requires explicit message passing between layers

Kernel in Unix

- Traditionally, the operating system itself
- Isolated from users and applications
    - Unix philosophy - Keep kernel as small as possible
    - Bug in any application should not crash OS
- At top level, user programs invoke OS services using system calls or library functions

Kernel in Unix

- At lowest level, kernel primitives directly interface with hardware
- Kernel itself is logically divided into two parts
  - File subsystem to transfer data between memory and external device
  - Process control subsystem to control interprocess communication, process schedule, and memory management

Unix Kernels

- Monolithic in nature
  - One large block of code
  - Runs as a single process within a single address space
  - All functional components have access to all internal data structures and functions
  - If anything is changed, all modules and functions must be recompiled, relinked and system rebooted
  - Difficult to add new device driver or file system functions

Kernel in Linux

- Kernel a set of interacting components
- Processes use system calls to request kernel service
- Structured as a collection of modules that can be automatically loaded and unloaded on demand
    - Loadable modules overcome some problems of monolithic kernel
    - Dynamically linked
    - Stackable modules
        - Arranged as a hierarchy

Kernel in Linux

- Linux kernel modules
  - Intended to achieve theoretical advantages of microkernels without incurring performance penalty
  - Designed as object files whose code can be dynamically linked/ unlinked to kernel at runtime
  - Object code is a set of functions to implement filesystem, device driver, or other features at kernel's upper layer
  - Module does not run as a specific process but is executed in kernel mode on behalf of the current process, like any other statically linked kernel function

Kernel in Linux

- Linux kernel modules
  - Advantages of using modules:
    - Modularized approach - well-defined software interfaces to access data structures handled by modules
    - Platform independence - a disk driver module relying on SCSI standard works as well on a PC as on Alpha
    - Frugal main memory use - module may be linked to a running kernel when needed and unlinked when no longer needed; useful for small embedded systems
    - No performance penalty – after linking, the object code of a module is equivalent to the object code of a statically linked kernel; no explicit message passing is required to invoke module functions

Kernel in Linux

- Runs in it own memory space (memory is divided into kernel space and user space)
- Operations include scheduling, process management, signaling, device I/O, paging, and swapping
- Being monolithic, it includes low-level interaction with the hardware, and hence is specific to a particular architecture
- Linus Torvalds chose monolithic architecture for Linux
  - Micro-kernels were experimental at the time Linux was designed
  - Micro-kernels were more complex than monolithic kernels
  - Micro-kernels executed notably slowly than monolithic kernels

Kernel in Windows

- *Executive* contains the core OS services
  - Memory management, process management, security, I/O
- *Kernel* controls execution of processes
  - Process switching, thread switching, interrupt handling
- Kernel works in cooperation with executive
  - Manages thread scheduling, process switching, exception and interrupt handling, and multiprocessor synchronization; rest in executive
  - Kernel's own code does not run in threads
  - Only part of the OS that can not be preempted or paged

Kernel in Windows

- Microsoft calls it a modified micro-kernel architecture
  - Unlike pure micro-kernel, many of the systems functions outside the micro-kernel run in kernel mode for performance reasons

Kernel in Windows

- As with Unix, it is isolated from user programs
  - User programs and applications allowed to access one of the protected subsystems
  - Each system function is managed by only one component of the OS
  - Rest of the OS and all applications access the function through the responsible component using a standardized interface
  - Key system data can be accessed only through the appropriate function
  - In principle, any module can be removed, upgraded, or replaced without rewriting the entire system or its standard application programming interface (API)

Kernel in Windows

- Two programming interfaces provided by a subsystem
  - Win32 interface – for traditional Windows users and programmers
  - POSIX interface – to make porting of Unix applications easier
- Subsystems and services access the executive using system services
- Executive contains object manager, security reference monitor, process manager, local procedure call facility, memory manager, and an I/O manager

Process execution modes in UNIX

- Two modes of process execution
- User mode
  - Normal mode of execution for a process
  - Execution of a system call changes the mode to kernel mode
  - Processes can access their own instructions and data
    - Not kernel instructions and data
  - Cannot execute certain privileged machine instructions

Process execution modes in UNIX

- Kernel mode
  - Processes can access both kernel as well as user instructions and data
  - No limit to which instructions can be executed
  - Runs on behalf of a user process and is part of the user process

Operating System Structure

- Minimal OS
  - CP/M or DOS
  - Initial program loading/ bootstrapping
  - File system

Operating System Structure

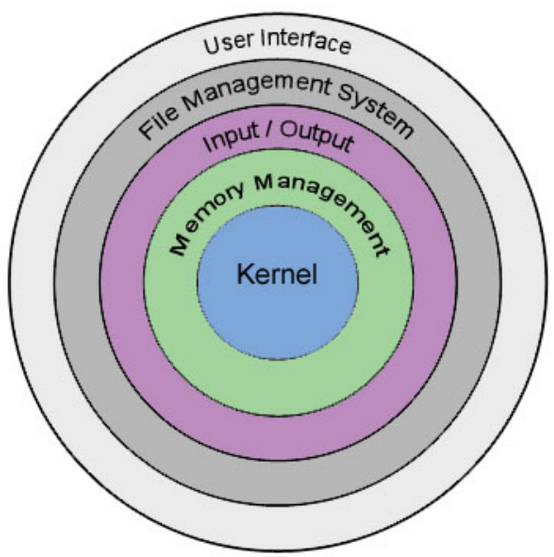
- Monolithic Structure
  - Most primitive form of operating systems
  - No structure
  - Collection of procedures that can call any other procedure
  - Well-defined interface for procedures
  - No information hiding
  - Services provided by putting parameters in well-defined places and executing a supervisor call
    - Switch machine from user mode to kernel mode

Operating System Structure

- Basic OS structure
  - Main program that invokes requested service procedures
  - Set of service procedures to carry out system calls
  - Set of utility procedures to help the service procedures
- User program executes until:
  - Program terminates
  - Time-out signal
  - Service request
  - Interrupt
- Difficult to maintain and take care of concurrency

Operating System Structure

- Layered Systems
- Hierarchy of layers to OS
  - One above the other

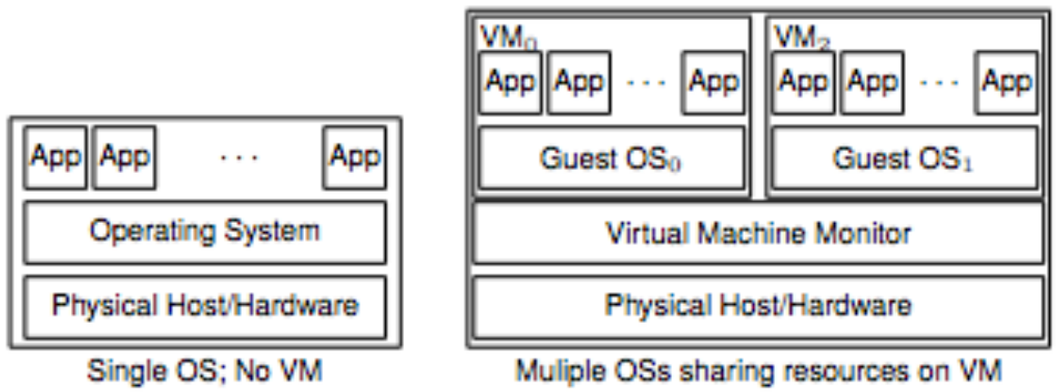


Operating System Structure

- Virtual machines
  - Basis for developing the OS in the first place
  - Provides a minimal set of operations
  - Creates a virtual CPU for every process

Operating System Structure

- IBM systems 370 - CMS, VM
  - Allowed a number of guest operating systems under control of VM
  - Useful for cross platform development
    - Lets you run Linux and Windows on same machine
    - Even concurrently



Operating System Structure

- IBM System 370 - CMS/VM
  - Has Virtual Machine monitor
    - Virtualize CPU for all processes
    - Virtual memory for all processes:
      - Single virtual memory shared by all processes
      - Separate virtual memory for each process
    - Virtual I/O devices
- Performs functions associated with CPU management and allocation
- Provides synchronization and/or communication primitives

Operating System Structure

- Process Hierarchy
  - Structured as a multi-level hierarchy
  - Parent-child model
    - Fork creates child
    - Fork returns PID from child to parent
    - Child gets 0 from the fork

Operating System Structure

- Client-Server model
  - Remove as much as possible from the OS
    - Leaving a minimal kernel
  - User process (client) sends request to server process
  - Kernel handles communications between client and server
  - Split OS into parts - file service, process service, terminal service, memory service
  - Servers run in user mode
    - Small and manageable

I/O communication

- Programmed I/O
  - Simplest and least expensive scheme
  - CPU retains control of device controller
    - Takes responsibility to transfer every bit to/from I/O devices
  - Instruction set need I/O instructions for:
    - Control: To activate and operate external devices (rewind tape)
    - Status: Test status conditions for I/O modules and peripherals
    - Transfer: Read/write data

I/O communication

- Programmed I/O
  - Bus
    - Address bus: To select a memory location or I/O device
    - Data bus: To transfer data
  - Handshaking protocal
  - Disadvantages
    - Poor resource allocation
    - Only one device active at a time
    - Gross mismatch between CPU speed and I/O device speed

I/O communication

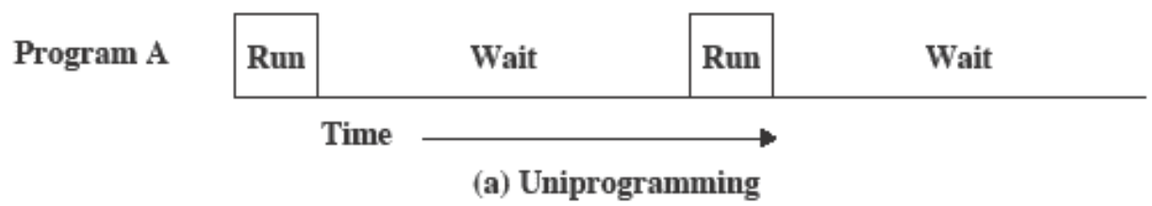
- Interrupt-driven I/O
  - CPU still retains control of the I/O process
  - Sends an I/O command to I/O module and then moves on
  - I/O module read/writes data on specified device and places it on data bus when instructed by CPU
  - I/O module interrupts CPU when it is ready to transfer data
  - Possible ot have multiple I/O modules on a machine

I/O communication

- Direct memory access
  - CPU trusts DMA module to read from/write into designated portion of memory
  - DMA module (also called I/O module) acts as slave to CPU to execute these transfers
  - DMA module takes control of bus
    - May bottleneck CPU if CPU needs bus

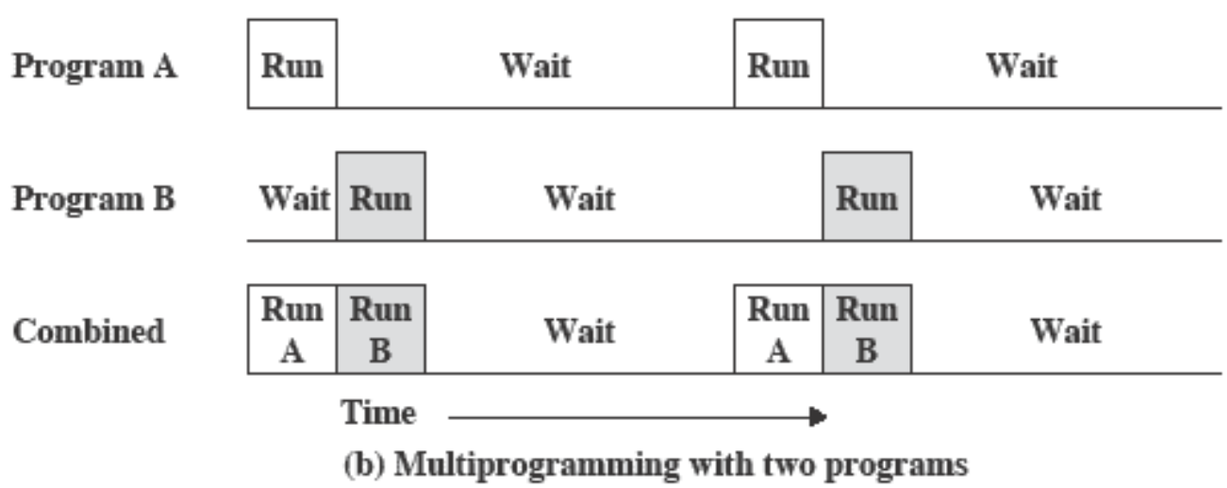
Uniprogramming

- Processor must wait for I/O instruction to complete before proceeding
- Simple hardware
- CPU and I/O bound system



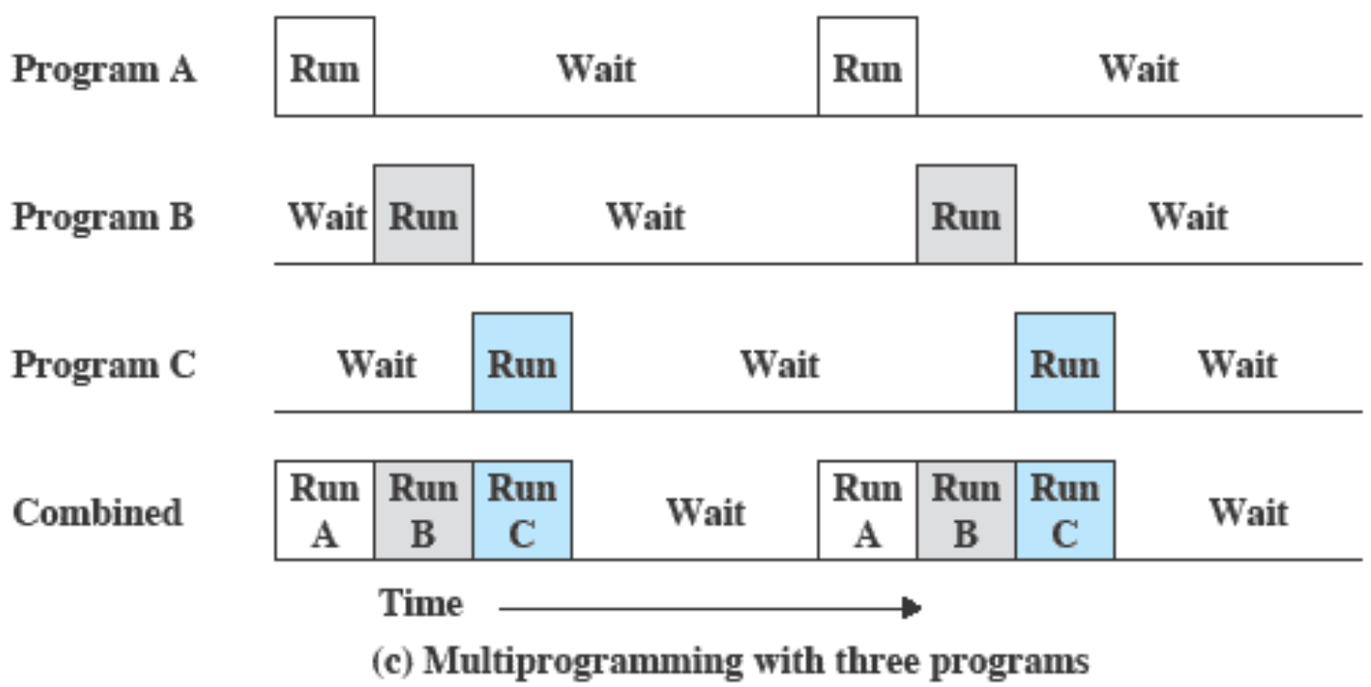
Multiprogramming

- Maintains more than one program in memory
  - Sharing of time and place
- If one job is waiting on I/O, processor can switch to other job



(b) Multiprogramming with two programs

Multiprogramming



(c) Multiprogramming with three programs

Table 2.1 Sample Program Execution Attributes

	JOB1	JOB2	JOB3
Type of job	Heavy compute	Heavy I/O	Heavy I/O
Duration	5 min	15 min	10 min
Memory required	50 M	100 M	75 M
Need disk?	No	No	Yes
Need terminal?	No	Yes	No
Need printer?	No	No	Yes

Multiprogramming

- Multiprogramming OS requires new hardware components
  - DMA hardware
  - Priority Interrupt Mechanism
  - Timer
  - Storage and Instruction protection
  - Dynamic address relocation

Multiprogramming

- Much more complex
  - Must hide the sharing of resources between different users
  - Hide details of storage and I/O devices
  - Requires more complex file system for secondary storage

Multiprogramming

- Tasks of a multiprogramming OS
  - Bridge gap between machine and user level
  - Manage the availability of resources needed by different users
  - Provide facilities for synchronization and communication

Multitasking

- Pre-emptive multitasking
  - Used by Linux
  - System preempts a process to gain control of CPU
  - After gaining control, decides which other process gets to run
- Cooperative multitasking
  - Older versions of windows
  - Process stays on CPU until it decides to give it up

Multiuser systems

- Able to concurrently and independently execute several applications, possibly belonging to two or more users
- Concurrent applications
  - Active at the same time
  - Contend for various resources like CPU, memory, secondary storage
- Independent applications
  - Each application can perform its task with no regard to what the other applications are doing
- Switching from one application to another slows both down

Multiuser systems

- Features of multiuser OS
  - Authentication mechanism to verify user identity
  - Buggy application should not block other applications
  - Malicious code should not interfere or spy on other applications
  - Accounting to limit resource usage for each individual
- Protection mechanisms need hardware features that allow privileged OS execution, implemented through system calls

Multiuser systems

- Security issues
  - Implemented through mechanism of users and groups
  - Individuals are identified through user identification (UID)
    - Helps in setting up private space for storage and email
    - Allows setting up of individual limits on resources and applications
- Selective sharing of data and resources through group id (GID)
  - Allows for controlled sharing of resources like files
- Special user called `root` is above all security mechanisms
  - Can access all files, manipulate all processes

What about the user interface

- Not a concern for the OS
- Only purpose of user interface is to shield user from the system
- Unix takes the users inside the system
    - Leads to long labyrinthine logic trails
    - Gets called “unfriendly” for doing so
- Is it GUI or CUI?
    - Creeping featurism
        - Designing for beginners vs designing for experts

What about the user interface

- GUI vs CUI
- Interface with other programs
- Effects on complexity
- Adaptability
- Scalability (adding 1 user vs 100 users)
- Who is in charge? User or computer?

Oh so much more coming!

- Any questions?