

Chapter 1 - Introduction

Luis Tarrataca

`luis.tarrataca@gmail.com`

CEFET-RJ

1 Motivation

2 What is an operating system?

OS as an Extended Machine

OS as a Resource Manager

3 Computer Hardware Review

Von Neumann architecture

Central Processing Unit

Central Processing Unit

Parallel Processing

Memory

I/O Devices

I/O Devices

Programmed I/O

I/O Commands

Interrupt-Driven I/O

Interrupt Processing

4 DMA Module

Bus Structure

Data Lines

Address Lines

Control Lines

Booting the computer

Booting the computer

5 Operating System Concepts

Processes

Process States

Address Spaces

Paging

Virtual Memory

Demand paging

Files

6 System Calls

System calls for process management

System calls for file management

System calls for directory management

Miscellaneous system calls

7 Operating System Structure

Monolithic Systems

Layered Systems

Microkernels

Client-Server Model

Virtual Machines

Exokernels

Motivation

Today's class is about introducing **Operating Systems**:

But what is an operating system? Any ideas?

Motivation

Today's class is about introducing **Operating Systems**:

But what is an operating system? Any ideas?

What does an operating system do? Any ideas?

Motivation

Today's class is about introducing **Operating Systems**:

But what is an operating system? Any ideas?

What does an operating system do? Any ideas?

Why do we need an operating system? Any ideas?

Numerous reasons why an OS is important:

- OS are cool!

Numerous reasons why an OS is important:

- OS are cool!
- OS will help give meaning to your life!

Numerous reasons why an OS is important:

- OS are cool!
- OS will help give meaning to your life!
- If you are alone:

Numerous reasons why an OS is important:

- OS are cool!
- OS will help give meaning to your life!
- If you are alone:
 - An OS will help you find a boyfriend / girlfriend

Numerous reasons why an OS is important:

- OS are cool!
- OS will help give meaning to your life!
- If you are alone:
 - An OS will help you find a boyfriend / girlfriend
- If you already have a boyfriend / girlfriend:

Numerous reasons why an OS is important:

- OS are cool!
- OS will help give meaning to your life!
- If you are alone:
 - An OS will help you find a boyfriend / girlfriend
- If you already have a boyfriend / girlfriend:
 - An OS will help you find a better boyfriend / girlfriend

Numerous reasons why an OS is important:

- OS are cool!
- OS will help give meaning to your life!
- If you are alone:
 - An OS will help you find a boyfriend / girlfriend
- If you already have a boyfriend / girlfriend:
 - An OS will help you find a better boyfriend / girlfriend
- #NOT...

Now a little bit more serious. A computer consists of

- One or more processors;
- Main memory
- I/O devices, e.g.:
 - Disks, printers, a keyboard, a mouse, a display, network interfaces

What would happen if a programmer had to manage all of these?

What would happen if a programmer had to manage all of these?

For those of you who did Computer Architecture with me:



What would happen if a programmer had to manage all of these?

- Too many things to manage:
 - No useful code would ever get written;
- Furthermore, incredibly difficult to manage optimally these resources;

Idea:

- Create a program called an **Operating System**;
- OS job is to provide user programs with:
 - A better, simpler, cleaner, model of the computer;
 - Manage all the computer resources;

Simple overview of the main components:

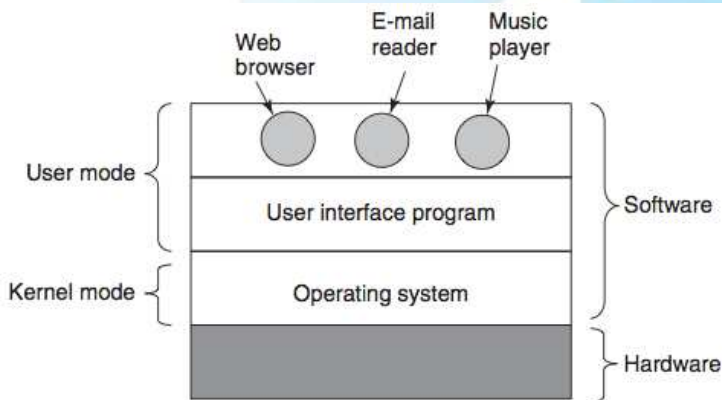


Figure: Where the operating system fits in. (Source: (Tanenbaum and Bos, 2015))

- First layer represents the **hardware**:
 - Consisting of chips, memory, disks, etc...
- On top of the hardware is the **software**:
 - **Kernel mode**:
 - Complete access to all the hardware;
 - Can execute any instruction the machine is capable of executing;
 - OS runs in kernel mode;
 - **User mode**:
 - Only a subset of the machine instructions is available;
 - Prohibited: Instructions that affect control of the machine or do I/O;

Everything running in kernel mode:

- Part of the OS;

However, some programs running outside the kernel mode:

- Are also part of the OS;
- Examples:
 - `chmod`;
 - `passwd`;

Sometimes it is difficult to draw a boundary.

Other important things about OS:

- OS are huge, complex, and long-lived:
 - Windows 10 consists of around 50 - 60 million lines of codes;
 - Excluding things like Windows Explorer, Windows Media Player, etc...
- Accordingly: OS are very hard to write and pieces are **shared** between OS;

What is an Operating System?

Essentially, two perspectives exist:

- OS as an **Extended Machine**
- OS as a **Resource Manager**

Lets have a quick look at each one of these

OS as an Extended Machine

- Computer Architecture at the machine-language level is primitive:
 - Remember Pong? Arkanoid? Snake? Space Invaders?
- No sane programmer would want to deal with this **nightmare**
- Idea: OS **abstracts** devices and hides the complexity, e.g.:
 - Read file;
 - Write file;
 - Program timer;
 - Process interruptions;

OS as a Resource Manager (1/2)

- OS job is to manage:
 - Processors;
 - Memories;
 - I/O devices and the various programs competing for them;
- Keep track of:
 - Which programs are using which resource
 - Grant resource requests
 - Account for usage
 - Mediate conflicting requests from different programs and users

OS as a Resource Manager (2/2)

Resource management includes **multiplexing** (sharing) resources (1/2):

- **Time multiplexing:**

- Different programs / users take turns using resource;
- Example: Single CPU multiplexing:
 - OS allocates the CPU to one program;
 - After a certain time another program gets to use the CPU;
 - Then another and then eventually the first one again.
- OS responsible for managing the multiplexing;

Resource management includes **multiplexing** (sharing) resources (2/2):

- **Space multiplexing:**

- Each program / user gets part of the resource;
- Example: main memory:
 - Normally divided among several running programs;
 - Assuming there is enough memory to hold multiple programs:
 - More efficient to hold several programs in memory;
 - Rather than give one of them all of it;
 - Remember principle of locality from computer architecture?
- OS responsible for managing this multiplexing;

Computer Hardware Review

Remember that semester you spend learning Computer Architecture?

Computer Hardware Review

Remember that semester you spend learning Computer Architecture?

- Lets review the entire's course syllabus in a single class ;)

Von Neuman architecture

The main components of the von Neumann architecture:

- **Memory module**
- **I/O module**
- **CPU**

Lets have a look at each one of these

Von Neumann Architecture:

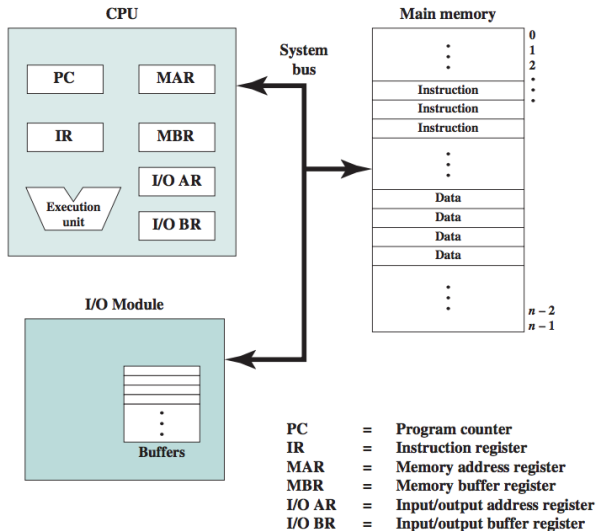


Figure: A top level view of the main computer components (Source: (Stallings, 2015))

Central Processing Unit

- “Brain” of the computer;
- Instruction cycle:

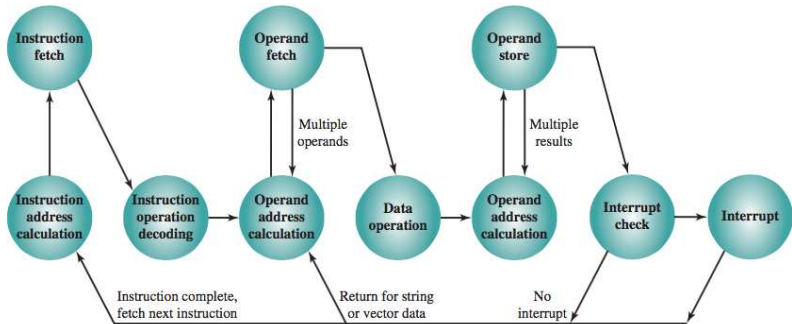


Figure: Instruction Cycle State Diagram, with Interrupts (Source: (Stallings, 2015))

CPU has a set of internal registers (1/4):

- **Program Counter (PC):**
 - Specifies the memory address of the next instruction to be executed.
- **Instruction Register (IR):**
 - Holds the instruction currently being executed or decoded.

CPU has a set of internal registers (2/4):

- **Stack Pointer (SP):**

- Points to the top of the current stack in memory
- Stores inputs, local and temporary variables that are not kept in registers;

- **Program Status Word (PSW):**

- Holds the state of the processor (e.g: Z, C, O, N, etc)
- Additional bit for kernel mode and user mode.

CPU has a set of internal registers (3/4):

- **Memory address register (MAR):**
 - Specifies memory address to be read/written;
- **Memory buffer register (MBR):**
 - Contains the data to be written into memory or...
 - Receives the data read from memory;
 - Used for interruption handling;

CPU has a set of internal registers (4/4):

- **I/O address register (I/OAR):**
 - Specifies a particular I/O device;
- **I/O buffer (I/OBR) register:**
 - Used for the exchange of data between an I/O module and the CPU;

Modern CPUs execute more than one instruction at the same time:

Do you remember how these mechanisms these are called? Any ideas?

Do you remember how these mechanisms these are called? Any ideas?

- Pipelines =)

Time →

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO

Pipelines are complex structures, remember the following?

- RAW
- WAR
- WAW
- NOP

Pipelines are responsible for great headaches:

- They expose the complexities of the underlying machine;

Remember these ``headaches``?

	Time →							← Branch penalty						
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO							
Instruction 5					FI	DI	CO							
Instruction 6						FI	DI							
Instruction 7							FI							
Instruction 15								FI	DI	CO	FO	EI	WO	
Instruction 16									FI	DI	CO	FO	EI	WO

Figure: Effect of a Conditional Branch on Instruction Pipeline Operation. Instruction 3 is a conditional branch to instruction 15 (Source: (Stallings, 2015))

Do you know a structure more advanced than pipelines? Any ideas?

Do you know a structure more advanced than pipelines? Any ideas?

- Superscalar CPU;
- Simple idea: increase number of pipelines;
- Multiple execution units are present, e.g.:
 - One for integer arithmetic;
 - One for floating-point arithmetic;
 - One for Boolean operations.

- Increase the number of “headaches”:
 - Instructions are often executed out of order, remember?
 - In-order issue out-of-order completion?
 - Out-of-order issue out-of-order completion?
 - Up to the hardware to make sure the result produced is the same.

Simple idea: increase number of pipelines

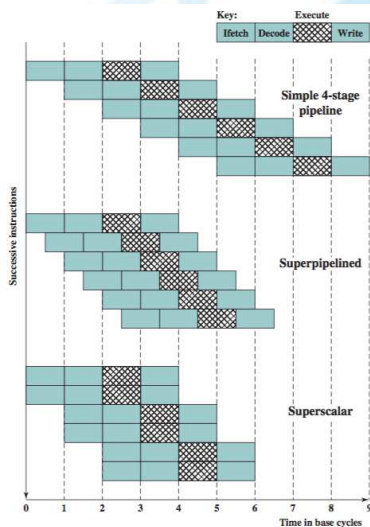


Figure: Timing Diagram for a 6-stage instruction Pipeline Operation (Source: (Stallings, 2015))

Do you know of a structure similar to superscalar processors? Any ideas?

Do you know of a structure similar to superscalar processors? Any ideas?

- Superpipelining is an alternative performance method to superscalar:
 - Many pipeline stages require less than half a clock cycle;
 - A pipeline clock is used instead of the overall system clock:
 - To advance between the different pipeline stages;

Simple idea: pipeline clock is used instead of the overall system clock

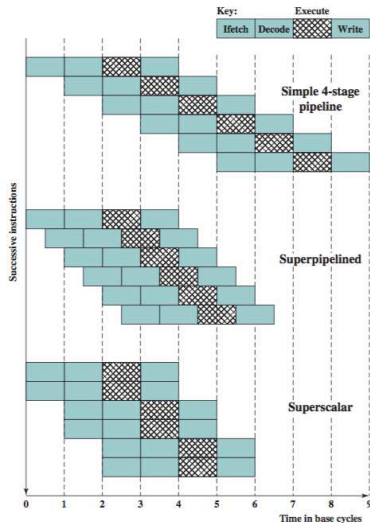


Figure: Timing Diagram for a 6-stage instruction Pipeline Operation (Source: (Stallings, 2015))

What else can be done to improve performance?

What else can be done to improve performance?

- Parallel Processing

Parallel Processing

Remember this?

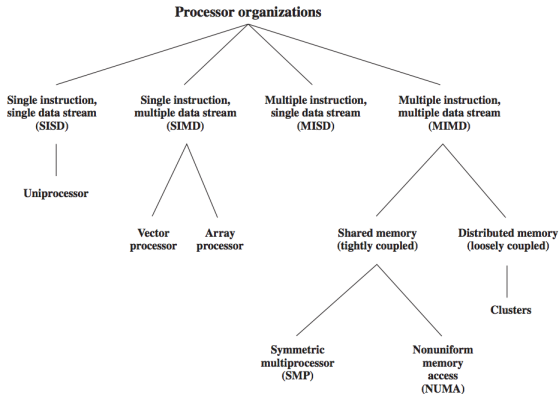


Figure: A Taxonomy of Parallel Processor Architectures (Source: (Stallings, 2015))

Fun, right?

Lets look at some architectures for multicore systems (1/2):

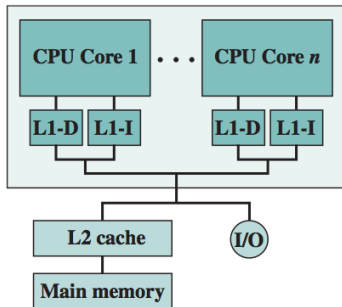


Figure: Dedicated L1 cache - Ex: ARM11 MPCore (Source: (Stallings, 2015))

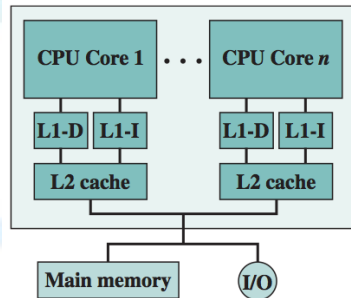


Figure: Dedicated L2 cache - Ex: AMD Opteron (Source: (Stallings, 2015))

- L1-D data cache;
- L1-I instruction cache;

Lets look at some architectures for multicore systems (2/2):

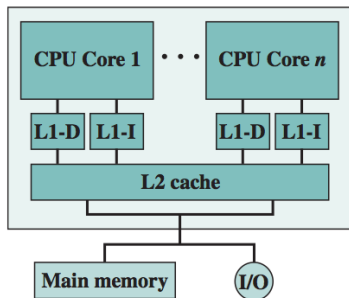


Figure: Shared L2 cache - Ex: Intel Core Duo
(Source: (Stallings, 2015))

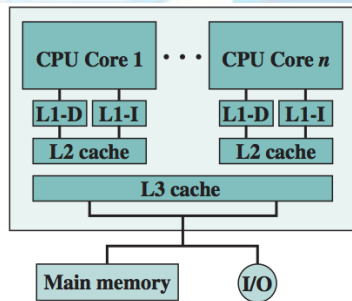


Figure: Shared L3 cache - Ex: Intel Core i7
(Source: (Stallings, 2015))

- L1-D data cache;
- L1-I instruction cache;

Multicore systems introduce a specific problem, remember what it is?

Multicore systems introduce a specific problem, remember what it is?

- Cache coherence problem;

What is the Cache coherence problem? Any ideas?

Multicore systems introduce a specific problem, remember what it is?

- Cache coherence problem;

What is the Cache coherence problem? Any ideas?

- Changing a word in a cache may invalidate other copies;

Multicore systems introduce a specific problem, remember what it is?

- Cache coherence problem;

What is the Cache coherence problem? Any ideas?

- Changing a word in a cache may invalidate other copies;

How can we solve the cache coherence problem? Any ideas?

Multicore systems introduce a specific problem, remember what it is?

- Cache coherence problem;

What is the Cache coherence problem? Any ideas?

- Changing a word in a cache may invalidate other copies;

How can we solve the cache coherence problem? Any ideas?

- *E.g.:* MESI protocol;

Memory

Ideally, memory should be:

- Extremely fast
- Abundantly large;
- Dirt cheap.

Unfortunately: No current technology satisfies all of these goals.

What can be done to tackle this issue? Any ideas?

What can be done to tackle this issue? Any ideas?

- We can have a memory hierarchy:

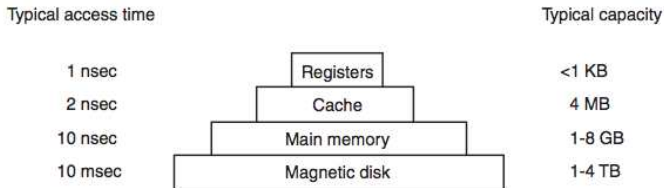


Figure: (Source: (Tanenbaum and Bos, 2015))

- Top layers have higher speed, smaller capacity, and greater cost per bit;
- Bottom layer have slower speed, higher capacity and lower cost per bit;

Registers:

- Just as fast as the CPU;
- Extremely small amount of memory;

Cache memory:

- Constituted by cache lines;
- Each line contains a block;
- Each block contains K words;
- Cache hit:
 - When a word is **searched** in a cache and **found**;
- Cache Miss:
 - When a word is **searched** in a cache and **not found**;
- Requires a mapping mechanism. Remember these?
 - Direct, Associative, Set-Associative;

Main memory:

- Usually called RAM;
 - Set of sequentially numbered addresses:
 - Each location contains binary information
 - Data;
 - Or instructions.

Magnetic Disks:

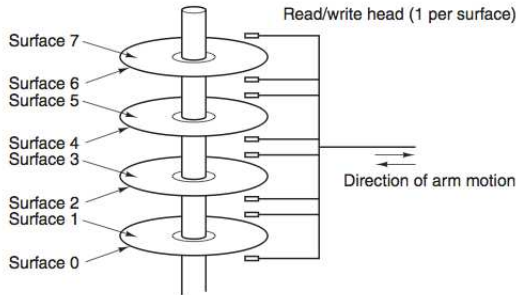


Figure: (Source: (Tanenbaum and Bos, 2015))

- Plates;
- Surfaces;
- Tracks;
- Sectors;

I/O Devices (1/2)

I/O module is responsible for:

- Transferring data from external devices to CPU and memory;
 - And vice versa;
- Containing internal buffers for temporarily holding data;

I/O Devices (2/2)

Device Controller is responsible for:

- Presenting simple interface to the OS;
 - Control of the device is complicated and detailed:
- A device driver is the software that interacts with a controller.

I/O Module structure

Lets take a closer look at a generic I/O module.

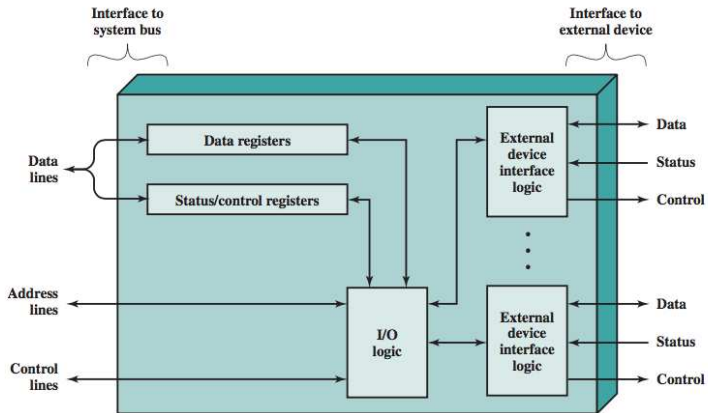


Figure: Block Diagram of an of an I/O Module. (Source: (Stallings, 2015))

Organization of a generic I/O module (1/2):

- Module connects to computer through a set of signal lines;
- Data transferred to and from the module are buffered in data registers.
- Status registers provide status information:
 - Also function as control registers, to accept processor control info;

Organization of a generic I/O module (2/2):

- Logic within the module interacts with the processor via control lines:
 - Processor uses the control lines to issue commands to the I/O module;
 - Some of the control lines may be used by the I/O module
 - *E.g.* arbitration and status signals;
- Module must also be able to recognize and generate addresses:
 - For each device it controls
- I/O module contains logic specific for a set of interfaces;

I/O module allows processor to view peripherals in a simple way:

- Presents a high-level interface to the processor;
- Taking most of the I/O processing burden away from the processor;
- Also called an I/O processor =)

Now that we have an idea of the main components...

How can we manage the communication between the μP and the I/O module?

- Any ideas?

Essentially, there are three techniques are possible for I/O operations:

	No Interrupts	Use of Interrupts
I/O-to-memory transfer through processor	Programmed I/O	Interrupt-driven I/O
Direct I/O-to-memory transfer		Direct memory access (DMA)

Figure: I/O Techniques (Source: (Stallings, 2015))

Lets have a look at each one of these =>

Programmed I/O

Data are exchanged between the processor and the I/O module:

- 1 Processor executes program controlling I/O operation;
 - *E.g.*: sensing device status, read/write command, data transfer.
- 2 Once the processor issues a command to the I/O module:
 - Processor must wait until the I/O operation is complete;
- 3 If the processor is faster than the I/O module:
 - Wasteful of processor time = '(

I/O Commands

To execute an I/O-related instruction:

- Processor issues an address:
 - Specifying the particular I/O module and external device;
- An I/O command which can be of the following type:
 - Control, Test, Read and Write

I/O commands the processor can issue are of the following type (1/2):

- **Control:** Used to activate a peripheral and tell it what to do:
 - *E.g.:* Rewind magnetic-tape; move to HD track;
- **Test:** test I/O module and its peripherals. *E.g.:*
 - Is the peripheral powered on?
 - Is the peripheral available for use?
 - Has the most recent I/O operation completed? Did any errors occur?

I/O commands the processor can issue are of the following type (2/2):

- **Read:** I/O module obtains a data item from the peripheral:
 - Placing the data in an internal buffer;
 - Processor requests I/O module to place data on bus;
- **Write:** I/O module writes a data item to the peripheral:
 - I/O module reads data from bus;
 - I/O module transmits data to peripheral;

In flowchart form:

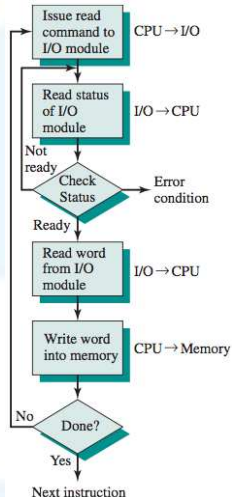


Figure: Programmed I/O technique for input of a block of data (Source: (Stallings, 2015))

Can you characterize the system from an efficiency perspective?

Very wasteful, recall that:

- Processor issues a command to the I/O module:
 - then waits for I/O operation to complete.
 - While waiting, processor repeatedly interrogates status of I/O module.
- If processor is faster than I/O module: wasteful of processor time.

Is it possible to do any better?

Interrupt-Driven I/O

What is the ideal scenario for processor performance?

- Do not wait for I/O module;
- Instead, continue processing other tasks;
- And be notified when I/O module has something for processor;

This is the concept of **interruption**, *i.e.*:

- 1 Ask for something from the I/O module;
- 2 Continue processing without waiting for the I/O module;
- 3 Be interrupted when the I/O module has something ready.

From the point of view of the **I/O module**:

- 4 Module waits for processor to request data:
- 5 When request is made:
 - When possible: module interacts with peripheral;
 - Once the data is completely buffered;
 - data are place on data bus;
- 6 An interrupt signal is sent to the processor over a control line;
- 7 Module becomes available for another I/O operation.

From the point of view of the processor (1/2):

- 1 A READ command is issued to I/O module;
- 2 Processor goes off to do something else;
- 3 Processor checks for interrupts at the end of each instruction cycle;

From the point of view of the processor (2/2):

- 4 When the interrupt from the I/O module occurs:
 - Processor saves program context;
 - Processor proceeds to read data from I/O module
 - Processor stores data in memory;
- 5 Processor then restores previous program context;
- 6 Processes resumes execution of previous program

In flowchart form:

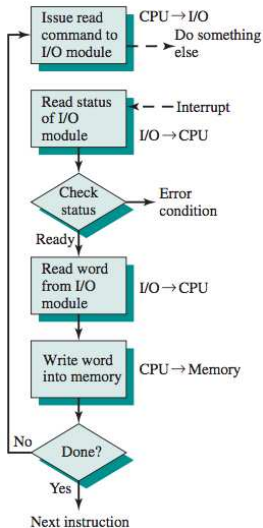


Figure: Interrupt-driven I/O (Source: (Stallings, 2015))

Interrupt Processing

Lets take a closer look at the interruption-based strategy.

- Interruption triggers a number of events
 - Processor, hardware and software.
- Automatically, we can pose a series of questions:
 - What happens to the program that is executing?
 - What happens to the processor?
 - How is the interruption processed?

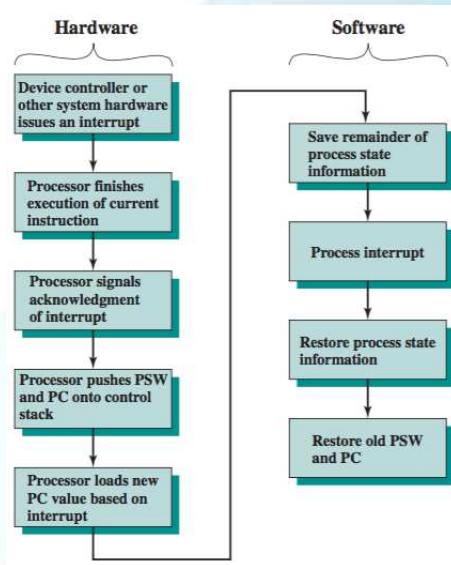


Figure: Simple Interrupt Processing (Source: (Stallings, 2015))

When an I/O device completes an I/O operation (1/4):

- 1 Device issues an interrupt signal to the processor.
- 2 Processor finishes execution of the current instruction
 - Before responding to the interrupt;
- 3 Processor tests for an interrupt:
 - Determines if there is one;
 - If one exists, sends an acknowledgement signal to peripheral;
 - Acknowledgment allows the device to remove its interrupt signal.

When an I/O device completes an I/O operation (2/4):

- 4 Processor needs to transfer control to the interrupt routine;
 - This is done by saving the program context:
 - Processor status word;
 - Program counter;
- 5 Processor then loads the program counter associated with the interrupt-handling routine.

When an I/O device completes an I/O operation (3/4):

- 6 Interruption routine may use the registers:
 - This means that these registers need to be saved;
 - This happened when you were developing your CA project;
- 7 Typically, the interrupt handler will begin by saving all registers on the stack;
- 8 Interrupt handler then processes the interrupt

When an I/O device completes an I/O operation (4/4):

- 9 When interrupt processing is complete:
 - Saved registers are retrieved from stack and restored;
- 10 Final act is to restore the PSW and program counter
 - Next instruction to be executed will be from the previously interrupted program.

Interrupt I/O is more efficient than programmed I/O:

- Eliminates needless waiting...

Despite the improvement, can you see any potential upgrade that can be performed with interrupt I/O?

Despite the improvement, can you see any potential upgrade that can be performed with interrupt I/O?

Interrupt I/O still consumes a lot of processor time:

- Data is exchanged between memory and I/O module...
- But this exchange still needs to go through the processor....
- **Processor spends time transferring data**
 - While it could be doing something more useful..

DMA module

Idea: Copy data directly to memory, bypassing processor:

- Memory accesses are performed by DMA module;
- Unburdens the processor;
- Combine with interruption scheme for optimum efficiency.

This strategy is called **Direct Memory Access (DMA)**

DMA involves an additional module on the system bus:

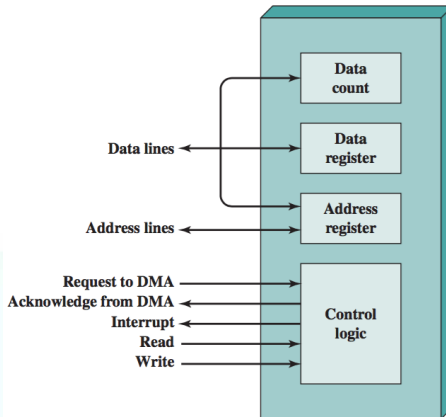


Figure: Typical DMA block diagram (Source: (Stallings, 2015))

- Uses the bus only when the processor does not need to;
- Forces the processor to suspend bus operations temporarily;

Processor issues a command to the DMA module:

- The command contains (1/2):
 - Whether a read or write is requested:
 - Transmitted over the bus control lines;
 - Address of the I/O device involved
 - Transmitted over the bus data lines;
 - Stored in the data register;

Processor issues a command to the DMA module:

- The command contains (2/2):
 - Starting location in memory to read from or write to:
 - Communicated on the data lines and...
 - Stored by the DMA module in its address register;
 - Number of words to be read or written:
 - Communicated via the data lines and stored in the data count register;

Processor then continues with other work, *i.e.*:

- I/O operation delegated to DMA module;
- DMA module transfers block of data:
 - Bypassing the processor;
- When the transfer is complete:
 - DMA module sends interrupt signal;
- Processor is involved only at:
 - Beginning of transfer;
 - End of transfer;

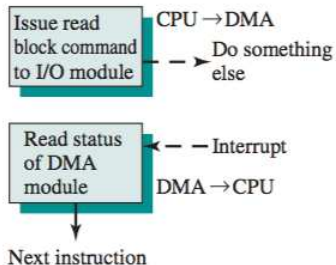


Figure: DMA-driven I/O (Source: (Stallings, 2015))

Bus Structure

Bus lines can be classified into three functional groups:

- **Data:**
 - for moving data among system modules
- **Address**
 - for specifying the source or destination of the data:
- **Control**
 - for transmitting command information among the modules.

Lets have a quick look into each one of these...

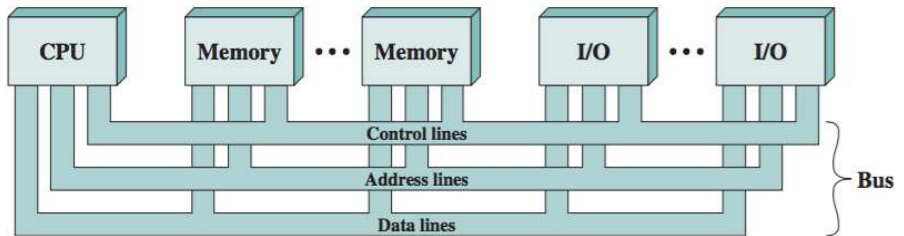


Figure: Bus Interconnection Scheme (Source: (Stallings, 2015))

Data Lines

The data bus may consist of 32, 64, 128, or even more separate lines:

- A.k.a. width of the data bus;

Each line can carry only 1 bit at a time:

- Number of lines determines how many bits can be transferred at a time.

Data bus width is key to system performance, *e.g.*:

- If the data bus is 32 bits wide and each instruction is 64 bits long;
- Each instruction requires two memory accesses.

Address Lines

Used to designate the source or destination of the data on the data bus:

- The width of the address bus determines the maximum system memory;
- The address lines are generally also used to address I/O addresses;
 - Higher-order bits are used to select a particular module on the bus;
 - Lower-order bits select a memory location or I/O port within the module.

Control Lines

Command signals specify operations to be performed, e.g.:

- Memory write: write bus data to a memory address;
- Memory read: read memory at memory address;
- I/O write: write bus data to an I/O address;
- I/O read: read data from an I/O address;
- Bus request: a module needs to gain control of the bus;
- Bus grant: a requesting module has been granted bus control;
- Many more control signals...

However, as processors and memories got faster:

- Ability of a single bus to handle all the traffic was strained;
- As a result, additional buses were added:

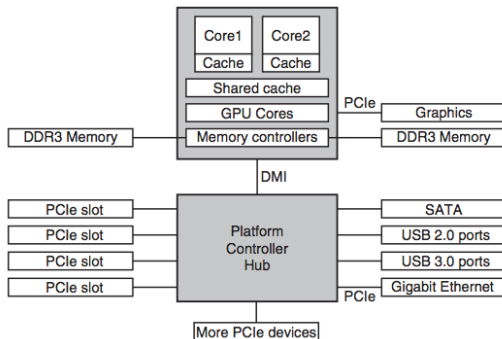


Figure: (Source: (Tanenbaum and Bos, 2015))

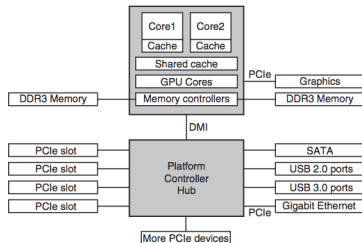


Figure: (Source: (Tanenbaum and Bos, 2015))

This system has many buses, *e.g.*:

- Cache
- Memory
- PCIe (main bus)
- PCI, USB, SATA...

Each with a different transfer rate and function.

Booting the computer

Now that we remembered a little bit of computer architecture:

How does the boot procedure of a computer works? Any ideas?

The boot process is as follows.

- ① Every PC has a motherboard containing the system BIOS:
 - Containing low-level I/O software, e.g.:
 - read the keyboard;
 - write to the screen;
 - do disk I/O

2 BIOS is started and checks :

- How much RAM is installed;
- Keyboard and other basic devices;
- Buses to detect all the devices attached to them;

3 BIOS then determines the boot device:

- List of devices stored in the CMOS memory

4 First sector from the boot device is read into memory and executed:

- Sector contains a program;
- Program examines the partition table at the end of the boot sector:
 - in order to determine which partition is active;

- 5 Secondary boot loader is read in from that partition:
 - Reads in the operating system from the active partition and starts it.
- 6 OS queries the BIOS to get the configuration information:
 - For each device, it checks to see if it has the device driver;
 - If not the driver needs to be supplied (CD, internet,...)
 - Once it has all the device drivers, OS loads them into the kernel.
- 7 OS initializes data structures and starts up a login program or GUI.

Operating System Concepts

Do you know any basic concepts from OS? Any ideas?

Operating System Concepts

OS provide basic concepts such as:

- **Processes**
- **System Calls**
- **Address Spaces**
- **Files**

Lets look at these concepts briefly.

Processes

A **process** is basically a program in execution (1/2):

- Associated with each process is its **address space**
 - List of memory locations from 0 to some maximum:
 - Which the process can read and write
 - Address space contains:
 - Executable program (Instructions);
 - Program data;
 - Function call stack

Processes

A **process** is basically a program in execution (2/2):

- Associated with each process is a set of **resources**:
 - Registers (PC, SP, etc...);
 - List of open files;
 - Outstanding alarms;
 - Lists of related processes;
 - Other information needed to run the program;

So, in your opinion what is a process? Any ideas?

So, in your opinion what is a process? Any ideas?

Process is fundamentally a container that holds all the information needed to run a program.

Example

Consider a single-core multiprogramming system executing:

- Program converting a one-hour video to a certain format;
- Web browser (Not Microsoft Edge =P);
- Email client (Not Microsoft Outlook =P);

How do you think the OS manages these processes? Any ideas?

How do you think the OS manages these processes? Any ideas?

Periodically the OS:

- 1 Decides to stop running one process;
 - A.k.a. suspending the process;
- 2 Start running another process;

But if we suspend a process and later need to run it again what needs to happen? Any ideas?

But if we suspend a process and later need to run it again what needs to happen? Any ideas?

Process must be restarted in exactly the same state before stopping:

- Process information must be saved somewhere before the suspension;
 - Registers (PC, SP, etc...);
 - List of open files;
 - For each file the number of bytes read;
 - For each file the number of bytes written;
 - Outstanding alarms;
 - Lists of related processes;
 - Other information needed to run the program;

All the information about each process:

- Is stored in an OS table called the **process table**;
 - Array of structures, one for each process currently in existence.

Processes change over time...

What are some possible changes that you can envision? Any ideas?

Processes change over time...

What are some possible changes that you can envision? Any ideas?

Some possible states:

- New
- Ready
- Running
- Blocked
- Terminated

Processes change over time...

What are the set of transitions from the previous states? Any ideas?

Process States (1/2)

During the lifetime of a process, its state will change a number of times:

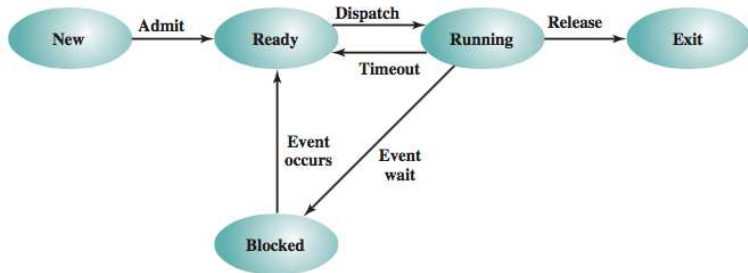


Figure: Five state process model (Source: (Stallings, 2015))

Process States (2/2)

During the lifetime of a process, its state will change a number of times:

- **New:** Process is created but not yet ready to execute.
- **Ready:** Process is ready to execute, awaiting processor availability;
- **Running:** Process is being executed by the processor;
- **Waiting:** Process is suspended from execution waiting a system resource;
- **Halted:** Process has terminated and will be destroyed by the OS.

Example

Key process-management system calls are those dealing with:

- creation and termination of processes.

Example:

- A process called the shell reads commands from a terminal;
- User compiles a program;
- Shell must create a new process for the compiler;
- When the compiler process has finished:
 - Executes a command to terminate the process;

Conclusion: Processes can create other processes;

- Parent processes;
- Child processes;

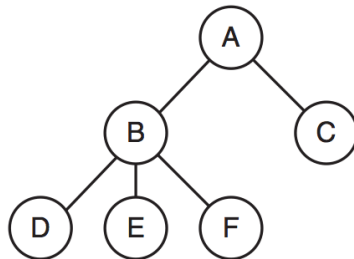


Figure: A process tree. Process A created two child processes, B and C. Process B created three child processes, D, E, and F. (Source: (Tanenbaum and Bos, 2015))

How can a program request the OS to do something? Any ideas?

How can a program request the OS to do something? Any ideas?

- **System Calls**

How can a program request the OS to do something? Any ideas?

- **System Calls**

What is a system call then? Any ideas?

- Programmatic way in which a program requests a service from the kernel;
- Provide an interface between a process and the OS;

Can you think of any examples of system calls? Any ideas?

Can you think of any examples of system calls? Any ideas?

Examples of system calls (1/2):

- Open / Read / Write file;
- Request more memory (malloc, calloc, etc..);
- Release unused memory (free);
- Wait for a child process to terminate (wait);

Can you think of any examples of system calls? Any ideas?

Examples of system calls (2/2):

- Set an alarm signal (e.g.: timer):
 - Process an interruption (just like in Computer Architecture);
 - The context needs to be saved;
 - An interruption routine is then executed;
 - The context is then restored;
- Many others exist...

Other important OS concepts:

- **UID (User IDentification):**

- Each system user is assigned one by the administrator;
- Every process started has the UID of the user who started it;
- A child process has the same UID as its parent.

- Users can be members of groups:

- Each of which has a **GID (Group IDentification)**.

Do you know any OS users? Any ideas?

Do you know any OS users? Any ideas?

- Superuser in Unix / Linux
- Administrator in Windows

Address Spaces

Lets talk about another important OS concept

What is an address space? Any ideas?

Address Spaces

Computers have main memory used to store:

- Instructions;
- Data;

Sophisticated OS allow multiple processes to be in memory:

- At the same time...

How can we stop programs from interfering with one another? Any ideas?

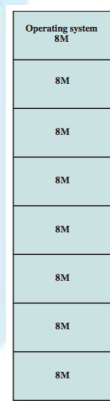
How can we stop programs from interfering with one another? Any ideas?

- Some kind of protection mechanism is needed;
- This mechanism is hardware-based:
 - But controlled by the OS;
- We will study this later on.

Memory Partitioning (1/2)

How should the OS partition the memory?

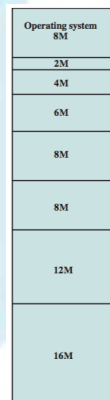
- Should every process have the same amount of memory?
- But what if we need less/more space?



Memory Partitioning (2/2)

How should the OS partition the memory?

- Or should different processes have different amounts of memory?
- When a process is brought into memory, it is placed in the smallest available partition that will hold it.



Can you see any problem with this type of partitioning?

Can you see any problem with this type of partitioning?

- Wasted memory: even with the use of unequal fixed-size partitions;
- In most cases:
 - A process will not require as much memory as provided by the partition;
 - *E.g.* a process that requires 3M bytes of memory would be placed in the 4M partition, wasting 1M that could be used by another process...

Can you think of an alternative method for partitioning memory?

Can you think of an alternative method for partitioning memory?

- What about **variable-size partitions**:
 - When a process is brought into memory:
 - Allocate exactly as much memory as it requires and no more.

- What about **variable-size partitions**:
 - When a process is brought into memory:
 - Allocate exactly as much memory as it requires and no more.

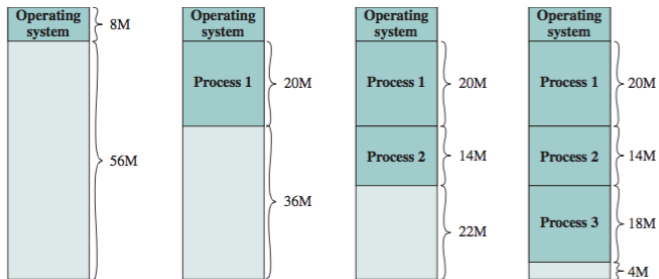


Figure: Variable-size partitions (Source: (Stallings, 2015))

Can you see any problems with this type of partitioning scheme?

Can you see any problems with this type of partitioning scheme?

- This method starts out well:
 - However, eventually the memory will be full of holes. The process either:
 - Terminates;
 - Is removed from main to secondary memory.
 - From time to time:
 - The OS **compacts** the processes in memory;
 - This results in all the free memory being placed together in one block;
 - This is a time-consuming procedure, wasteful of processor time.

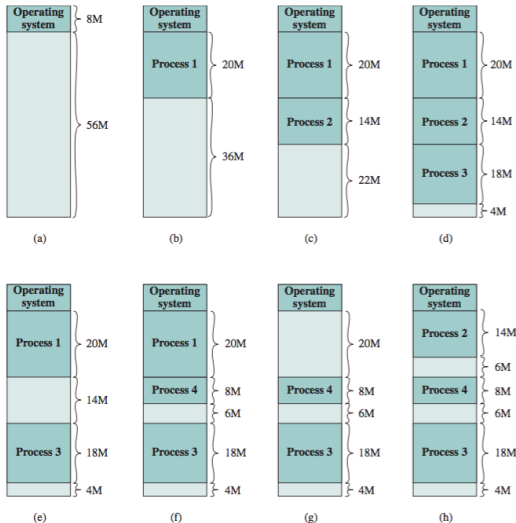


Figure: The effects of dynamic partitioning (Source: (Stallings, 2015))

Overall conclusion:

- Fixed-size and variable-size partitions are inefficient in the use of memory.

Can we do any better than these types of partitioning schemes?

Paging

Lets consider an alternative partitioning scheme:

- Allow memory to be partitioned into equal fixed-size small chunks:
 - Known as **page frames**
- Each process is also divided into small fixed-size chunks of some size:
 - Known as **pages**
- Typically: frames have the same size as pages
- Each **page** can be assigned to a **page frame**, then:
 - At most, wasted space for a process will be a fraction of the last page.

Example

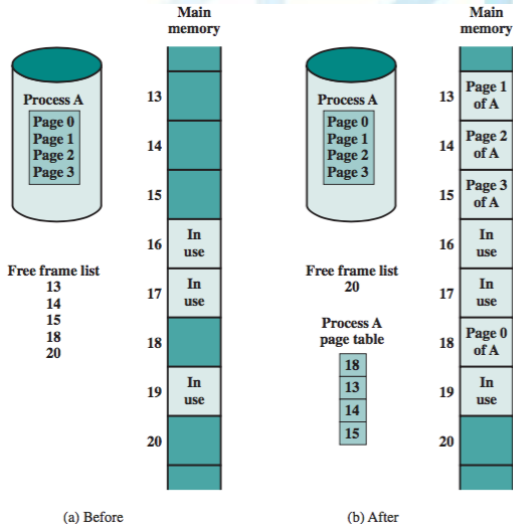


Figure: Allocation of free frames (Source: (Stallings, 2015))

At a given point in time:

- Some of the frames in memory are in use and some are free;
- The list of free frames is maintained by the OS;
- Process **A**, stored on disk, consists of four pages.
- When it comes time to load this process the OS:
 - Finds four free frames;
 - Loads the four pages of the process A into the four frames.

Do the frames need to be contiguous (1/2)?

- No! We can use the concept of **logical address**.
- OS maintains a **page table** for each process:
 - Showing the frame location for each page of the process;
- Within the program each logical address consists of:
 - a page number and a relative address within the page;
- Logical- to-physical address translation is done by processor.

Do the frames need to be contiguous (2/2)?

- Processor must know how to access the process's page table:
 - Input is a logical address:
 - **(page number, relative address)**
 - Output is a physical address obtained through the process page table:
 - **(frame number, relative address)**

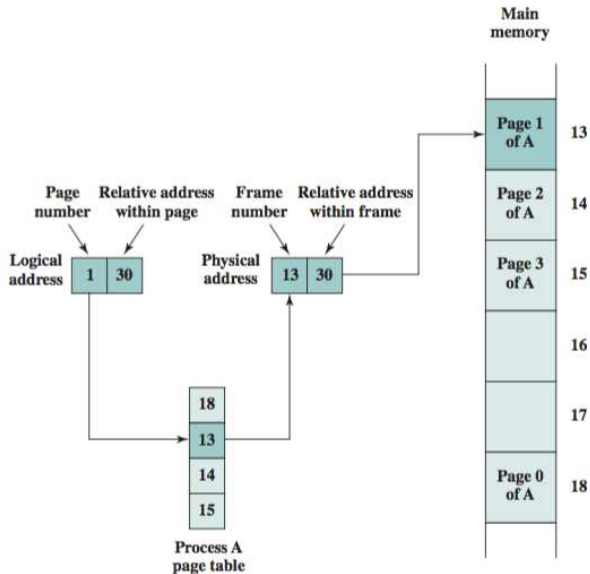


Figure: Logical and physical addresses (Source: (Stallings, 2015))

Can you see any other improvement that can be done to memory management?

Can you see any other improvement that can be done to memory management?

HINT: Space-time locality principle

Can you see any other improvement that can be done to memory management?

- The OS always loads all the memory of a process;
- IDEA: What if we only load those pages that are required at a single moment?
 - This is the concept of virtual memory...

Virtual Memory

Each process page is brought in only when it is needed (1/2):

- 1 Procedure is known as **demand paging**;
- 2 Locality principle: the same values, or related storage locations, are frequently accessed.
 - Why then would we need to load every page? Wasteful...

Each process page is brought in only when it is needed (2/2):

- 3 We can make better use of memory by loading in just a few pages
- 4 If the program attempts to access a page not in main memory:
 - a page fault is triggered, and the OS brings in the desired page;
 - These pages reside in secondary memory;
- 5 **Virtual Memory** refers to this much larger memory usable by the program.

At any one time, only a few pages of a process are in memory:

- Therefore more processes can be maintained in memory.
- Time is saved because:
 - Unused pages are not swapped in and out of memory;
 - Less RAM/HD accesses;
- Consequence: Possible for a process to be larger than all of main memory.

OS must be clever about how it manages this scheme:

- When it brings one page in, it must throw another page out;
 - This is known as page replacement.
- OS might throw out a page just before it is about to be used:
 - OS will just have to get that page again almost immediately;
 - Too much of this leads to a condition known as **thrashing**:
 - Processor spends most of its time swapping pages...
 - ...rather than executing instructions
 - extremely slowwww computerrrr...

Do you have any idea how to solve this problem? Any ideas?

Do you have any idea how to solve this problem? Any ideas?

- OS needs to guess which pages are least likely to be used:
 - *E.g.* based on recent history.
 - We have seen some when we studied Cache systems...

Files

Another key concept of modern OS:

- Hides peculiarities of the disks and other I/O devices;
- Present the programmer with a transparent model;
- System calls exist for:
 - Creating files;
 - Removing files;
 - Reading files;
 - Writing files;

Most OS have the concept of a directory (1/2):

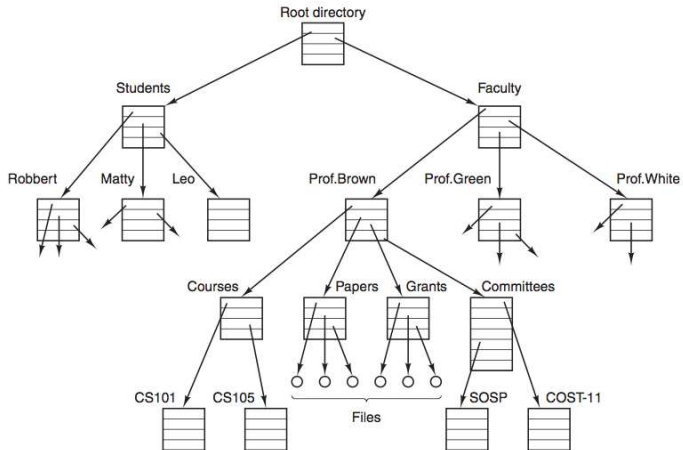


Figure: A file system for a university department. (Source: (Tanenbaum and Bos, 2015))

Most OS have the concept of a directory (2/2):

- Directory entries may be either files or other directories;
- System calls are then needed to:
 - Create directories.
 - Remove directories.
- Each process has a current working directory;

Before a file can be read or written:

- File must be opened, at which time the permissions are checked:
 - If the access is **permitted**:
 - OS returns a small integer called a file descriptor;
 - If the access is **prohibited**:
 - error code is returned.

Lets consider an additional requirement:

But what if two processes need to communicate with one another?

Lets consider an additional requirement:

But what if two processes need to communicate with one another?

This is done through the **pipe** concept:

- A sort of file that can be used to connect two processes

This is done through the **pipe** concept:

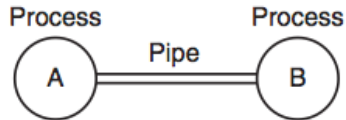


Figure: Two processes connected by a pipe. (Source: (Tanenbaum and Bos, 2015))

- Processes A and B must configure a pipe in advance;
- When process A wants to send data to process B:
 - it writes on the pipe as though it were an output file.
- Process B can read the data by:
 - reading from the pipe as though it were an input file.
- Pipe implementation is very much like that of a file;

System Calls

Ok, based on what we have seen until now:

What are the main functions of an OS? Any ideas?

System Calls

Ok, based on what we have seen until now:

What are the main functions of an OS? Any ideas?

Two main functions:

- Provide **abstractions** to programs;
- Manage computer **resources**;

Two main functions:

- Provide **abstractions** to programs:
 - *E.g.:* create, write, read and delete files.
- Manage computer **resources**:
 - Largely transparent, since computers need not worry about:
 - CPU, I/O, etc;
 - Done Automatically;
- **Conclusion:**
 - Interface between OS / Programs primarily deals with abstractions:

- **Conclusion:**

- Interface between OS / Programs primarily deals with abstractions:
- To understand what an OS is we must examine this interface:
- This interface can be seen through the available **system calls**
 - These vary from OS to OS:
 - Although the underlying concepts are the same;
 - We will focus on POSIX:
 - A.k.a. International Standard 9945-1;
 - Unix / Linux / BSD;

Important:

- Implementation of systems calls is highly machine-dependent:
 - They are often implemented in assembly;
- OS makes available a library:
 - Effectively working as an **interface**;
 - Allows for system calls to be made from programs:
 - Program invokes system call (**user mode**);
 - OS performs system call (**kernel mode**);
 - Program continues execution (**user mode**);

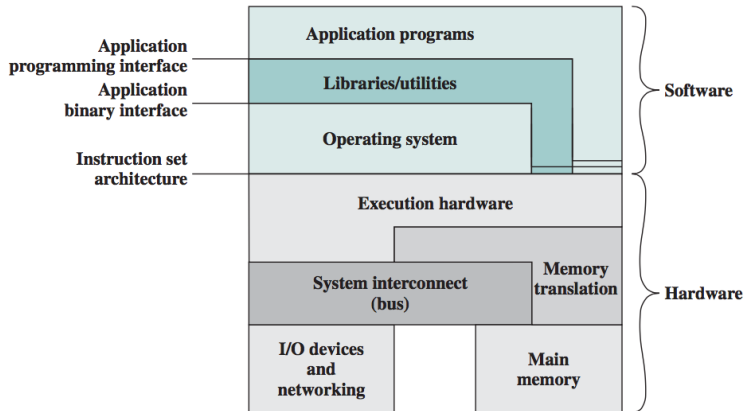


Figure: Computer Hardware and software structure (Source: (Stallings, 2015))

Guess what we will be doing next? Any ideas?

Guess what we will be doing next? Any ideas?

- We are going to see some of the available system calls ;)

Example

Lets take a look at the **read** specific system call:

```
count = read( fd , buffer , nbytes )
```

- 1st parameter specifies the file to be read;
- 2nd parameter specifies the buffer where the content is read to;
- 3rd parameter specifies the number of bytes to read;
- This system call returns:
 - Number of bytes actually read in count;
 - -1 if an error occurred;

Example

System calls are performed in a series of steps:

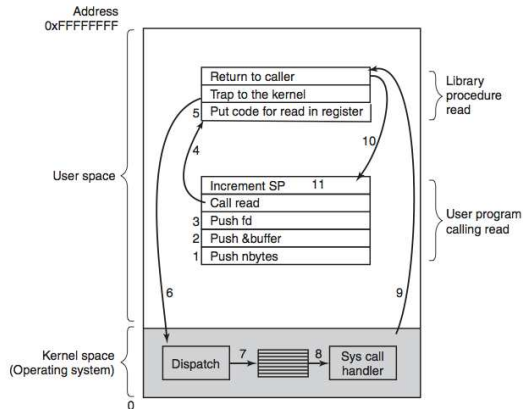


Figure: (Source: (Tanenbaum and Bos, 2015))

The previous picture in textual form (1/3):

- 1 Push nbytes:
- 2 Push &buffer:
- 3 Push fd:
- 4 The actual call to the library procedure;
- 5 System call identification is placed in a register;

The previous picture in textual form (2/3):

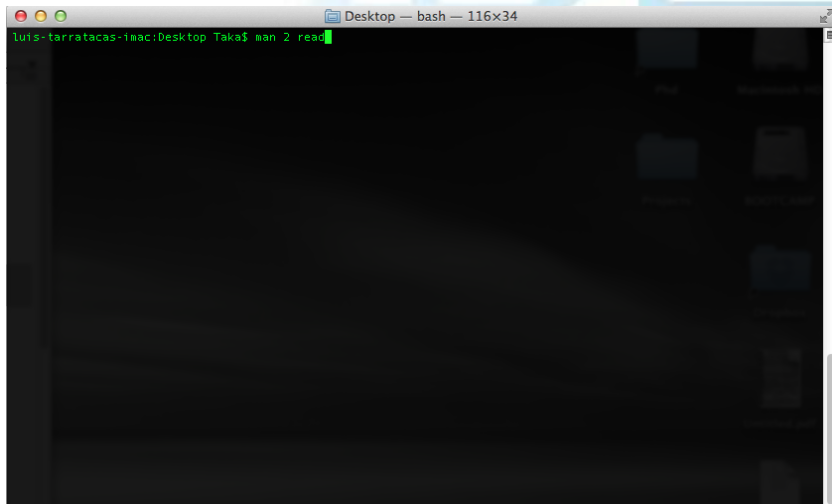
- 6 OS switches from **user** to **kernel** mode;
 - This is a special instruction called **trap**;
- 7 System call is dispatched to the appropriate handler;
- 8 System call is executed;
- 9 Once execution finishes:
 - control is returned to the user-space library;
- 10 Once library finishes:
 - control is returned to the original program;

The previous picture in textual form (3/3):

- 11 User program cleans up the stack:
 - Removes from stack the system call arguments;

The program is now free to do whatever it wants!

Linux makes all this information available through the terminal (1/2):



```
Desktop — bash — 116x34
luis-tarratacas-imac:Desktop Taka$ man 2 read
```

Linux makes all this information available through the terminal (2/2):

```

Desktop — less — 116x34

READ(2)                                BSD System Calls Manual                                READ(2)

NAME
    pread, read, readv -- read input

LIBRARY
    Standard C Library (libc, -lc)

SYNOPSIS
    #include <sys/types.h>
    #include <sys/uio.h>
    #include <unistd.h>

    ssize_t
    pread(int d, void *buf, size_t nbyte, off_t offset);

    ssize_t
    read(int fildes, void *buf, size_t nbyte);

    ssize_t
    readv(int d, const struct iovec *iov, int iovcnt);

DESCRIPTION
    Read() attempts to read nbyte bytes of data from the object referenced by the descriptor fildes
    into the buffer pointed to by buf. Readv() performs the same action, but scatters the input data
    into the iovcnt buffers specified by the members of the iov array: iov[0], iov[1], ...,
    iov[iovcnt-1]. Pread() performs the same function, but reads from the specified position in the
    file without modifying the file pointer.

    For readv(), the iovec structure is defined as:

        struct iovec {
;

```

The **read** system call is just one example:

- POSIX has about **100 procedure calls**
- All of which you should know for your exam

The **read** system call is just one example:

- POSIX has about **100 procedure calls**;
- All of which you should know for your exam ;)

System calls for process management

Process management

Call	Description
<code>pid = fork()</code>	Create a child process identical to the parent
<code>pid = waitpid(pid, &statloc, options)</code>	Wait for a child to terminate
<code>s = execve(name, argv, environp)</code>	Replace a process' core image
<code>exit(status)</code>	Terminate process execution and return status

Figure: Some of the major POSIX system calls. The return code `s` is `-1` if an error has occurred. The return codes are as follows: **pid** is a process id, **fd** is a file descriptor, **n** is a byte count, **position** is an offset within the file, and **seconds** is the elapsed time. (Source: (Tanenbaum and Bos, 2015))

System call **fork** (1/4):

- Only way to create a new process in POSIX;
- Creates an exact duplicate of the original process, including:
 - File descriptors;
 - Registers;
 - Everything!
- After the **fork**:
 - The original process and copy go their separate ways;
 - All the variables have identical values at the time of the fork
 - However, since the parent's data are copied to create the child:
 - Subsequent changes do not affect the other one;

System call **fork** (2/4):

- **Fork** returns a value:
 - Value zero in the child;
 - The child's **Process Identifier (PID)** in the parent;
- In most cases:
 - Child will execute different code;

Consider the case of the shell in Linux:

How do you think the Linux shell works? Any ideas?

Consider the case of the shell:

- 1 Reads a command;
- 2 Forks off a child process (**fork**);
- 3 Waits for the child to execute (**wait**);
 - input parameter indicates PID to wait for;
- 4 Childs executes different code (**execve**):
 - 1st parameter: name of the file to be executed;
 - 2nd parameter: pointer to argument array;
 - 3rd parameter: pointer to environment array;
- 5 Reads the next command

```
#define TRUE 1
```

```
while(TRUE){ /* repeat forever */
```

```
    type_prompt( ); /* display prompt on the screen */
```

```
    read_command(command, parameters); /* read input from terminal */
```

```
    if ( (pid = fork( )) == 0){ /* fork off child process */
```

```
        /* Parent code. */
```

```
        wait( pid ); /* wait for child to exit */
```

```
    }else{
```

```
        /* Child code. */
```

```
        execve(command, parameters, 0); /* execute command */
```

```
    }
```

```
}
```

System calls for file management

File management

Call	Description
<code>fd = open(file, how, ...)</code>	Open a file for reading, writing, or both
<code>s = close(fd)</code>	Close an open file
<code>n = read(fd, buffer, nbytes)</code>	Read data from a file into a buffer
<code>n = write(fd, buffer, nbytes)</code>	Write data from a buffer into a file
<code>position = lseek(fd, offset, whence)</code>	Move the file pointer
<code>s = stat(name, &buf)</code>	Get a file's status information

Figure: Some of the major POSIX system calls. The return code `s` is `-1` if an error has occurred. The return codes are as follows: **pid** is a process id, **fd** is a file descriptor, **n** is a byte count, **position** is an offset within the file, and **seconds** is the elapsed time. (Source: (Tanenbaum and Bos, 2015))

System calls for directory management

Directory- and file-system management

Call	Description
<code>s = mkdir(name, mode)</code>	Create a new directory
<code>s = rmdir(name)</code>	Remove an empty directory
<code>s = link(name1, name2)</code>	Create a new entry, <code>name2</code> , pointing to <code>name1</code>
<code>s = unlink(name)</code>	Remove a directory entry
<code>s = mount(special, name, flag)</code>	Mount a file system
<code>s = umount(special)</code>	Unmount a file system

Figure: Some of the major POSIX system calls. The return code `s` is `-1` if an error has occurred. The return codes are as follows: **pid** is a process id, **fd** is a file descriptor, **n** is a byte count, **position** is an offset within the file, and **seconds** is the elapsed time. (Source: (Tanenbaum and Bos, 2015))

Miscellaneous system calls

Miscellaneous

Call	Description
<code>s = chdir(dirname)</code>	Change the working directory
<code>s = chmod(name, mode)</code>	Change a file's protection bits
<code>s = kill(pid, signal)</code>	Send a signal to a process
<code>seconds = time(&seconds)</code>	Get the elapsed time since Jan. 1, 1970

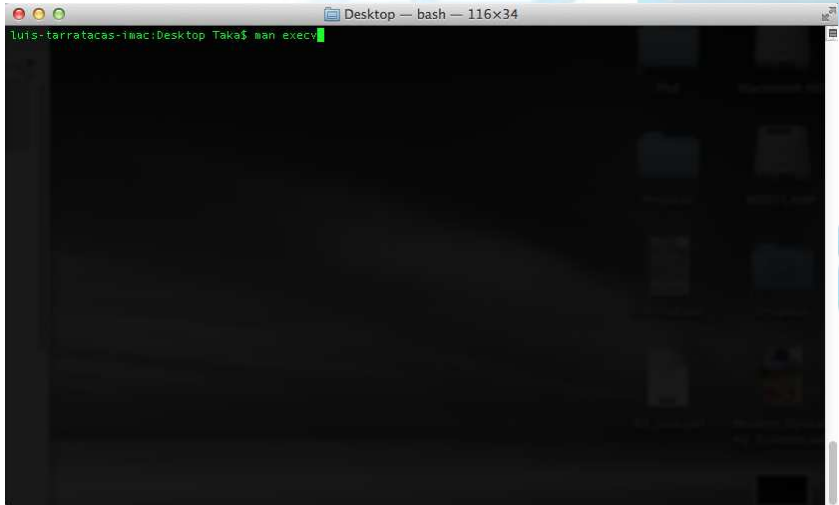
Figure: Some of the major POSIX system calls. The return code `s` is `-1` if an error has occurred. The return codes are as follows: **pid** is a process id, **fd** is a file descriptor, **n** is a byte count, **position** is an offset within the file, and **seconds** is the elapsed time. (Source: (Tanenbaum and Bos, 2015))

Does this seems complicated?

Does this seems complicated?

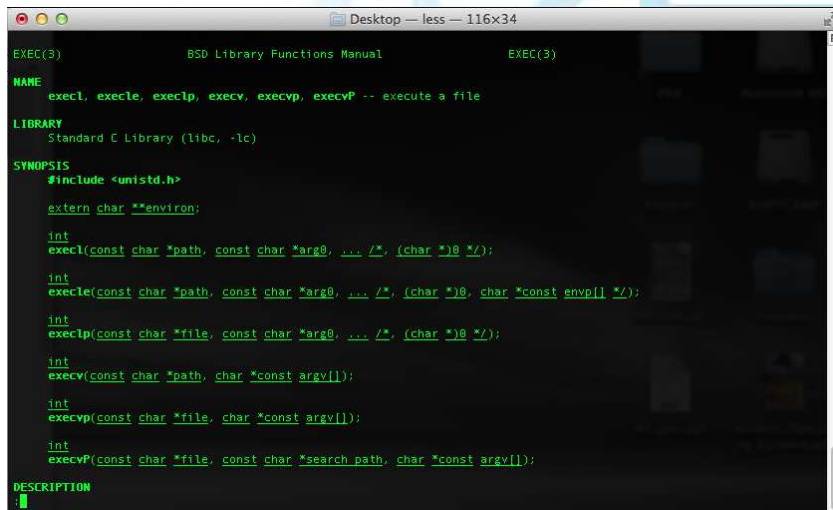
- Completely normal:
 - First time you are seeing it;
- But fear not...
 - Linux provides all the manuals that you need!

Linux makes all this information available through the terminal (1/2):



```
Desktop — bash — 116x34
luis-tarratacas-imac:Desktop Taka$ man execv
```

Linux makes all this information available through the terminal (2/2):



```
Desktop — less — 116x34

EXEC(3)                BSD Library Functions Manual                EXEC(3)

NAME
    execl, execlx, execlp, execv, execvp, execvp -- execute a file

LIBRARY
    Standard C Library (libc, -lc)

SYNOPSIS
    #include <unistd.h>

    extern char **environ;

    int
    execl(const char *path, const char *arg0, ... /*, (char *)0 */);

    int
    execlx(const char *path, const char *arg0, ... /*, (char *)0, char *const envp[] */);

    int
    execlp(const char *file, const char *arg0, ... /*, (char *)0 */);

    int
    execv(const char *path, char *const argv[]);

    int
    execvp(const char *file, char *const argv[]);

    int
    execvp(const char *file, const char *search_path, char *const argv[]);

DESCRIPTION
:|
```

But how can I discover the manual's number?

But how can I discover the manual's number?

Multiple solutions exist:

- Command **apropos**
- Command **man -wK**

Command **apropos**:

```
takaw@pi~$ Takaf apropos malloc
malloc(3), calloc(3), free(3), realloc(3), mmap(3), realloc(3) - general memory allocation operations
calloc(3), free(3), malloc(3), realloc(3), realloc(3), valloc(3) - memory allocation
filtercalltree(1) - Filter or prune a call tree file generated by sample or malloc_history
heap(1) - List all the malloc-allocated buffers in the process's heap
lber-memory(3), ber_memalloc(3), ber_memcalloc(3), ber_memrealloc(3), ber_memfree(3), ber_memvfree(3) - OpenLDAP LBER memory allocators
ldap_memfree(3), ldap_memvfree(3), ldap_memalloc(3), ldap_memcalloc(3), ldap_memrealloc(3), ldap_strdup(3) - LDAP memory allocation routines
leaks(1) - Search a process's memory for unreferenced malloc buffers
libgmalloc(3) - (Guard Malloc), an aggressive debugging malloc library
malloc_create_zone(3), malloc_destroy_zone(3), malloc_default_zone(3), malloc_zone_free_ptr(3), malloc_zone_malloc(3), malloc_zone_calloc(3), malloc_zone_valloc(3), malloc_zone_rea
loc(3), malloc_zone_memalign(3), malloc_zone_free(3) - zone-based memory allocation
malloc_good_size(3), malloc_size(3) - memory allocation information
malloc_history(1) - Show the malloc allocations that the process has performed
stringDups(1) - Identify duplicate strings or other objects in malloc blocks of a target process
takaw@pi~$ Takaf
```

Command **man -wk**:

```
takambp:~ Takaj$ man -kv malloc
/usr/local/share/man/man3/gdbm.3
/usr/local/share/man/man3/libpng.3
/usr/local/share/man/man3/readline.3
/opt/local/share/man/man1/i686-apple-darwin13-llvm-g++-4.2.1.gz
/opt/local/share/man/man1/i686-apple-darwin13-llvm-gcc-4.2.1.gz
/opt/local/share/man/man1/ld.1.gz
/opt/local/share/man/man1/llvm-c++-4.2.1.gz
/opt/local/share/man/man1/llvm-g++-4.2.1.gz
/opt/local/share/man/man1/llvm-gcc-4.2.1.gz
^Ctakambp:~ Takaj$
```


As a curiosity lets look at Windows API:

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount, so no umount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Figure: (Source: (Tanenbaum and Bos, 2015))

Win32API also contains **hundreds** of system calls:

- All of which you should know for the exam

Win32API also contains **hundreds** of system calls:

- All of which you should know for the exam ;)

Win32API also contains **hundreds** of system calls:

- All of which you should know for the exam ;)
- Who cares about Windows =P

Operating System Structure

Now we have a better understanding of the different OS components:

How should the OS be organized?

What are the different OS design possibilities?

Operating System Structure

Now we have a better understanding of the different OS components:

How should the OS be organized?

What are the different OS design possibilities?

Essentially there are six designs:

- Monolithic Systems;
- Layered Systems;
- Microkernels;
- Client-Server systems;
- Virtual Machines;

Monolithic Systems

First: What does monolithic means? Any ideas?

Monolithic Systems

First: What does monolithic means? Any ideas?

- Large, indivisible and slow to change;

Monolithic Systems

The entire OS runs as a program in kernel mode:

- Collection of procedures linked into a single executable;
- Any procedure can call any other procedure:
 - **Pros:**
 - Very efficient since there are no restrictions;
 - **Cons:**
 - Very difficult to understand;
 - If a procedure crashes the entire OS crashes;

For those of you who have seen **Object Programming**:

- No information is hidden;
- There is very little structure:
 - All functions are accessible to every other function;
- As opposed to using modules or packages:
 - Information is hidden away in modules or packages;
 - Only official designated entry points can be called;

Basic organization for monolithic OS (1/2):

- A main program invokes a service procedure;
- A set of service procedures that carry out the system calls:
 - Each service call manages the system calls;
- A set of utility procedures that help the service procedures:
 - *E.g.*: Fetch data from user programs;

Basic organization for monolithic OS (2/2):

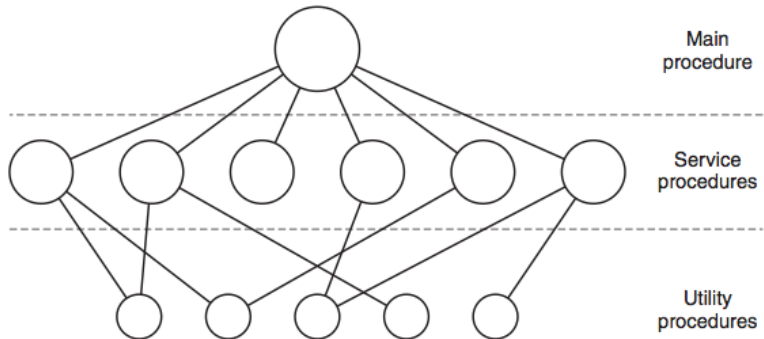


Figure: Simple structure model for a monolithic system(Source: (Tanenbaum and Bos, 2015))

Layered Systems

Idea: Organize OS as a hierarchy of layers:

- Each layer is constructed upon the one below it, e.g.:

5	Computer user
4	User programs
3	I/O Management
2	Process Communication
1	Memory management
0	Processor management

Microkernels

Traditionally: all layers are in the kernel

- Not necessary!
- **Idea:** put as little as possible in kernel model:
 - Less code less probability of bugs in kernel mode;
 - Less bugs less probability of bringing OS down;
- **Idea:** set user mode processes to do non-critical tasks:
 - Bug in user mode may not be fatal;
 - *E.g.:* bug in audio driver:
 - Stops or ruins sound;
 - But will not crash the computer;

Common desktop OS do not use microkernels:

- With the exception of Mac OS ;)

However, microkernels are dominant in:

- real-time OS;
- industrial avionics;
- military applications;

An example of a microkernel organization:

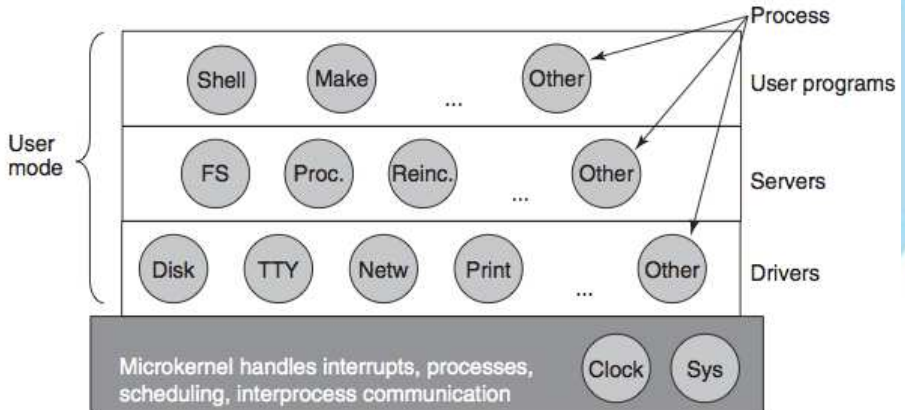


Figure: Simplified structure of the MINIX system (Source: (Tanenbaum and Bos, 2015))

Client-Server Model

Idea: Distinguish between two processes:

- **Servers:**
 - Provide some services;
- **Clients**
 - Use the services provided;
- Slight variation of the microkernel:

Communication between clients and servers:

- Often done by **message passing**, e.g.:
 - **Client** constructs message and sends to **service**;
 - **Server** executes and sends back result;

Because of this structure:

- Clients and servers can run on different computers;
- Requests are sent and replies come back;

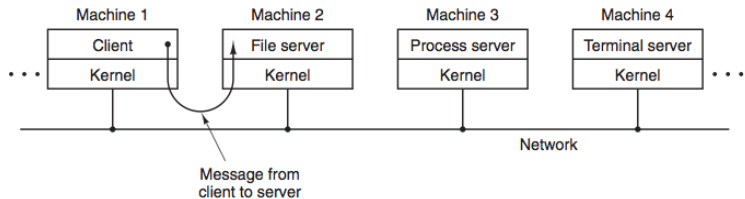


Figure: The client-server model over a network. (Source: (Tanenbaum and Bos, 2015))

Virtual Machines

Have a single computer run different OS:

- Each virtual machine is given time to run;
- If one virtual machine crashes other virtual machines can continue;
- Allows for efficient use of machine:
 - Hardware is never idle, always executing some VM;
 - Brings costs down;
 - Run Windows/Linux/Unix in the same machine

But what are the different virtualization techniques? Any ideas?

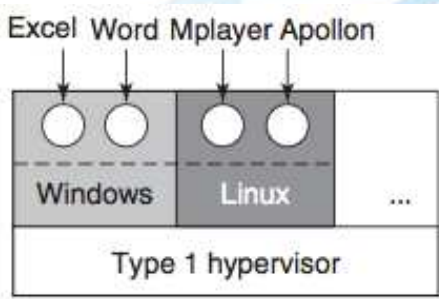


Figure: Execute directly over hardware (Source: (Tanenbaum and Bos, 2015))

- Efficient but had some peculiarities:
 - Virtual OS needs to execute in kernel mode...
 - But Virtual OS is running in user mode...
 - This was impossible to execute in some processors;

What can be done to circumvent this problem? Any ideas?

What can be done to circumvent this problem? Any ideas?

- Provide a processor simulator;
- Execute privileged instructions in simulator;
- Solves problem but is inefficient;

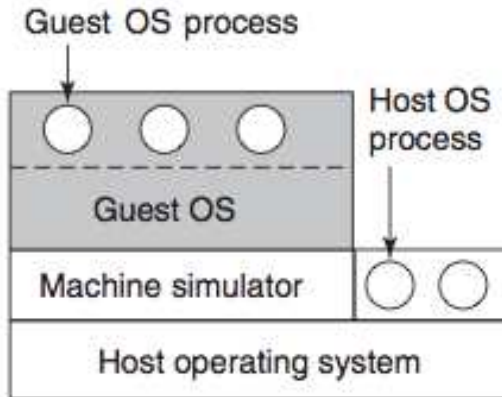


Figure: Execute an host OS and a machine simulator (Source: (Tanenbaum and Bos, 2015))

What can be done to improve performance? Any ideas?

What can be done to improve performance? Any ideas?

- Add a kernel module to original OS for virtualization;
- This allows for better virtualization performance

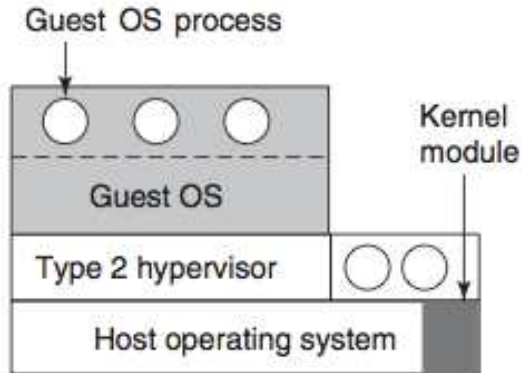


Figure: Execute directly over hardware (Source: (Tanenbaum and Bos, 2015))

Exokernels

Rather than cloning a machine:

- Give each VM a subset of the resources;
- At the bottom layer, running in **kernel mode**:
 - Is a program called the **Exokernel** whose function is:
 - Allocate resources to virtual machine;
 - Check that each VM uses its own resources;
- **Advantage:** Saves a layer of mapping:
 - Exorkernel keeps track of the resources allocated to each machine;
 - No need to remap any resources which makes it simpler;

References I



Stallings, W. (2015).

Computer Organization and Architecture.

Pearson Education.



Tanenbaum, A. and Bos, H. (2015).

Modern Operating Systems.

Pearson Education Limited.