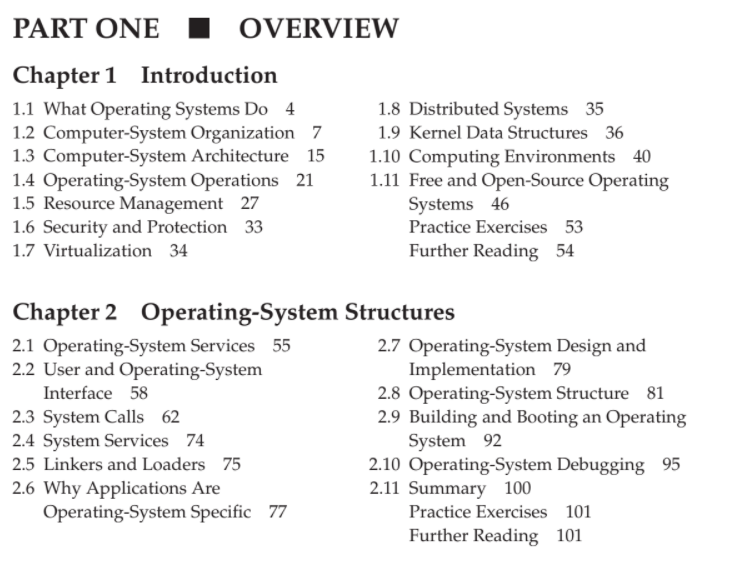
Umar Kagzi

340 Svadlenka Notes

2/3/22

* Need to rewatch lecture for notes on this section

2/7/22



Abstract view of Components of Computer Page

* Basicly goes over that slide in depth.

I/O

* How do we get IO? From a file, from your keyboard input.
* How is that data transferred? From that file, the translation from data to result is done. This is called IO Burst. CPU Burst is when data is translated CPU Burst
* Why is this important and how does it affect what we do? – check recording

Von Neumann Arch

* There willl be multiple Von neumann architectures that happen.
* 1) Instructions 2) Load Instruction from RAM 3) Load Operand from RAM 3) Execute CP istructions 5) Results back outside to RAM
* CPU Burst and IO Burst – read about them and how they work.

Storage Device Hierarchy

* Fastest thing to access is REGISTERS (smallest amount of storage, thus it is fast). Then comes the RAM.
* Later, a secondary memory storage is second fastest, which are hard drives, storage devices etc. Nonvolatile memory is ike NVM storage, such as flash drives. These are known as block devices.
* Lastly at the tertiary level, we see Optical drives

Summary

* There is a disparity between the I/O speed and Processing Speed.
* I/O -> CPU -> I/O

1. First IO is your input/ then it is the CPU working on that / 3) then an IO is given again
2. When the bottleneck happens, we do an interrupt – this checks if the process is complete.

CPU sees an interrupt from Disk controller to PCU – When it sees the interrupt at this point, it will send a special signal on the INTERRUPT line.

When the interrupt starts, interrupt handler starts and will go to the RAM. It will then update the state of the person that used the handler. At this point, an interrupt scheduler/ state switching occurs in the operating system.

* The part we are focused on is the KERNEL. Kernel has a lookup/interrupt vector look up table which has an array of the different interrupt lines.

Polling – Another type of Operating Systems, but in the particular example he showed in class, it wouldn’t be that great.

Multiprogramming – Having mult user processes that we can switch amongst. A step beyond this is called multitasking.

Multistasking – Gives another situation where a process is also switched off the CPU.

Interrupt-driven processes – The description he gave today is exactly interrupt driven processes.

Software Interrupts – Later covered in chap 2-3

* Check out the interrupt handler timeline on the sheets and the interrupt drive I/O cycle.

Kernel restores the process of the task that process of IO bursts.

Non-Blot

Multithreading – comes later.

Multitasking – CPU burst only, not IO burst.

Next day – we go over in-depth of system calls and OS calls.

Round Robin Scheduling

Processes that don’t fit in memory, swapping move thmein and cut out to run.

Virtual Memory allows execution of processess not completely in memory.

DualMode Operation – Alllows OS to protect itself and other system Components.

* User mode and kernel Mode
* Hardware capability that in one mode, runs a simple set of instructions
* Another mode runs Kernel mode which allows it to run with all power of CPU;
* Usually everything runs on DUAL Mode.

HW – Research on topics, do HWKS.

2/9/22

Keywords on Board

* Dual Mode
* Finish Chapter 1, start chap 2. Read from 2.1 to 2.8.4
* OS Services, more on sys calls, HW source code.

DMA – Direct Memory Access Structure

1. Setup data transfer (xfer)
2. Data transfer is initiated and completed
3. Disk controller sends interrupt to the CPU
4. CPU is interrupted and executes intrrupt handler
5. ….

Multithreading

* Usually a program executes one function at a time – however in multithreading many thread/programs can execute/run at the same time (all of them)
* T = 0 to T= 8 class example – threads slowly process each thread, and are able to do this simultaeneously
* Squiggly line = thread of execution.

Dual Mode

* Hardware feature of CPUs, is important as it ties into the Sys call interface. (Our way of being able to take advantage of services and get access to the hardware (ex reading files))
* Dual mode = 2 modes. CPUs have more than 2 modes but for this course 2 suffice.
* User mode and Kernel mode are the two mode.
* Mode bit is provided by the hardware, this allows to distinguish when sysstem is running user code or kernel code.
* How does this play out with system calls?
* Page 1.34.
* Thin Top lauyer above kernel – this is called Syscall interface. This is the medium
* Listen to recording at 29 MINS in to see the description of what he does. (Transition from User to Kernel Mode page)

User Design Goals

1. (FAST PROCESS) Initialize the data first. (I/o or cpu burst,
2. In big programs that you dot know much about, simply try it in the test eviroment
3. System calls have positives and iniu

Timers – prevent infinite resource usages and loop

* It is set to interrupt the computer after set
* Is classified in “tics”
* OS sets the counter time, counter zero sets up an interrupt. Scheduled.
* \Interrupt defniition – change in the flow of controll.

User process

0 User process executes first, then does sys call and gets

Bootstrap program – simple code to initialize the system and load kernel

\Kernel loads

System Daymon

Interrupt driven kernel does a Software Interrupt via a catch

Migration of Data A from Diskto register.

Multitasking environments

Multitasker environs must use most recent values

CPU Burst

* Intrc 1 (opz)
* Instruction 1 Operations
* 2) Instruction 2 oerations
* 3) Instruction 3 Ops

Application

* Process 1-3 are cooomunicating via LAN, all using the same ram \*one stick
* Cache helps processes run at the same time (read/ write data)>

System Processes (Daemons)

* They are not created by or have user control of them. They are just us
* Print Score – tells you what the score is between people’ OS>
* Kernel is always there to run. If kernep

Operating Systen picture (

* OS has a top part which is called the system caller interface.
* The kernel bottom part is kernel interface to the Device printer

Next Lecture

* Readings must be complete
* HW needs to be done as well, code will be given to run..

2/16/2022

Todays Topics

* IPC Intro
* Linking/Loading (&compiling)

InterProcess Communication

* Has two parts - Shared memory and Message Passing.

Scenario forSHARED MEMORY

* Student 1 wrote data on a sticky pad on a whiteboard – everyone can read this
* Student 2,3,4 also does the same thing – this clogs up the whiteboard.
* At this point youll have to remove the sticky pads that are old. Stu1 will prolly communicate with stu 5 to talk about removing the old sticky.
* This is the same thing with RAM. Data is put on it, once full it needs to unload

Scenario for Message Passing

* User 1 sends a message to user 2. After this transfer, the message is handed to user 2 and user 1 never sees this again.
* This is easier to program as there is no synhronization step (the removal of old msgs when full in “shared memory model”
* There still is an initialization step in this
* It is a bit wasteful as new memory gets used up for each new data content, or message.

Threads

* Define a global variable if you want to do shared memory on a thread.
* Sync techique called- LOCK. To stop threads from overlapping eachother when commmucnating, we put a LOCK on the thread.

Linkers and Loaders

* When you compile code, you can’t just create it right away
* From C file -> .exp doesn’t work
* C file ->.o file -> .exp file
* Cpfile.c - > cpfile.o (object files, contain machine instructions)
* Func1.c turns into func 1.o -🡪 mycp.exe
* All these three files (cpfile.o, func1.o) turn into an .exe file
* The process from .c file to a .o file is called COMPILING
* When you combine all the o files in a program, it is called linking
* Compiling, linking, loading.
* The last part, loading is when you turn the exe file into the running part. That is called loading.
* Relocatable Object Files

Below is a diagram

RAM

Kernel

User process 1

User proc 2

Daemon

* Cant use physical addresses for an .exe file. The purpose of it being compiled already is that it is easier to use.

Executable File

* This .exe contains both instrauctions and data
* Logical – is the adressing for our executable
* Logical address 0 will map to Physical addr 4000.
* Logical 1 will map to 4001 – there was an increase of the address # of ogical to physical.
* RAM only deals with Physcial. Cpu knows nothing about physical. When you map, this is called RELOCATION – the translation that allows for multiprogramming.
* Token – anything separated by whitespace. Ex int c = 4; int, c, = , 4 , ; are all tokens.
* After tokens are analyzed, this is tested & analyzed & optimized, then this is finally loaded into ram memory after we go thru the process. Linking can be done two ways – static or dyncamically.
* Dynamically linking – linking the pathname where the code exists. First user of adding this pathname/object file it costs a bit of memory/usage, but for others it does not.
* Every process used the copy of the object code.
* Chek Page 2.33 “The role of the linker and loader”

Dynamically Loading vs Dynamically

* We don’t need to have a whole exe at once – how does this play out?\
* We have a main function from cp.file.cc
* Main() has the object file inside of it. The point when we really need the object code, we use other provides.

Application Binary Interface (ABI)

Top layer – platform independent

Bot layer – platform dependent t the lower level

Only code neeeded to modify is the bottom layer

When you write kernel code in C, there are no librabries, it is a different animal that is very difficult.

Types of Structure

1. Monolithic
2. Layering
3. Microkernel
4. Loadable Modules
5. (another word , Modules) (general word, like modularizing or loadable modules
6. Monolithic (check page slides 2.40)

Traditional UNIX is not fully organized and not fully layered. Every process works out its own process space. Efficient performance (everything is doen in functions)

1. Layered Approach

Modules – our kernel is structured in a circular fashion, like rings. Each layer allows for a different function and an interface to get to the lower layer. Better code maitennace as we only work with one layer at a time. At runtime, this is slower as it has to traverse thru all different layers. This is the trde off.

1. Microkernels

* Make everythig into smaller parts and only have a few features in kernel . Everything else like device drivers, application programs are in user mode. Kernel has interprocess communication, mem management, cpu scheduling in it. Basically this one breaks down eveyrthing into kernel and user mode

1. Modules

* We want easily maintable code & be able to enhance our code.
* Uses object oriented approach. Uses dynamically loading, example Linux and solaris.
* LKM – loadable kernel modules – dyncamically loading kernels as you need them.

Linux System Structure

* Monolithic plus MODULAR DESIGN
* Syscall interface, device drivers, hardware, c library are all interconnected but in different modules- this makes to load/unload things in this.

Chapter 1 Notes

* An operating system acts as an intermediary between the user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient and efficient manner. An operating system is software that manages the computer hardware. The hardware must provide appropriate mechanisms to ensure the correct operation of the computer system and to prevent programs from interfering with the proper operation of the system. Internally, operating systems vary greatly in their makeup, since they are organized along many different lines. The design of a new operating system is a major task, and it is important that the goals of the system be well defined before the design begins. Because an operating system is large and complex, it must be created piece by piece. Each of these pieces should be a well-delineated portion of the system, with carefully defined inputs, outputs, and functions.
* A computer system can be divided roughly into four components: the hardware, the operating system, the application programs, and a user

Hardware

* Includes CPU, memory, IO devices

Application Programs

* Software like processors, spreadsheets, web brownsers

Resource Utilization

* How hardware and softwware resources are shared

Kernel

* The program running at all times on the computer
* Has two other types of programs not necessarily part of it: System programs and application programs
* Sys programs associated with the OS/CPU.
* Apps are programs not associated with runnign the system

1.2 – Computer System Organization

Bus

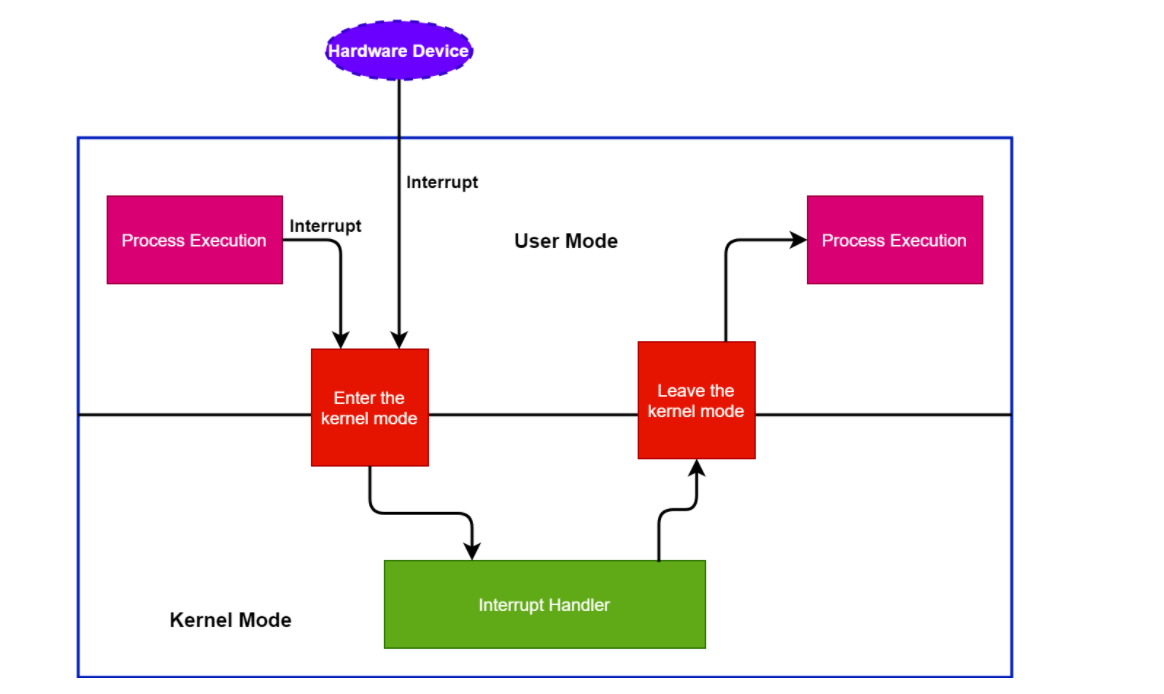
* Provides common access between components and shared mmory

Device Driver

* Understands device controller & provides interface to the device for the OS

<https://www.baeldung.com/cs/os-trap-vs-interrupt>

Interrupts



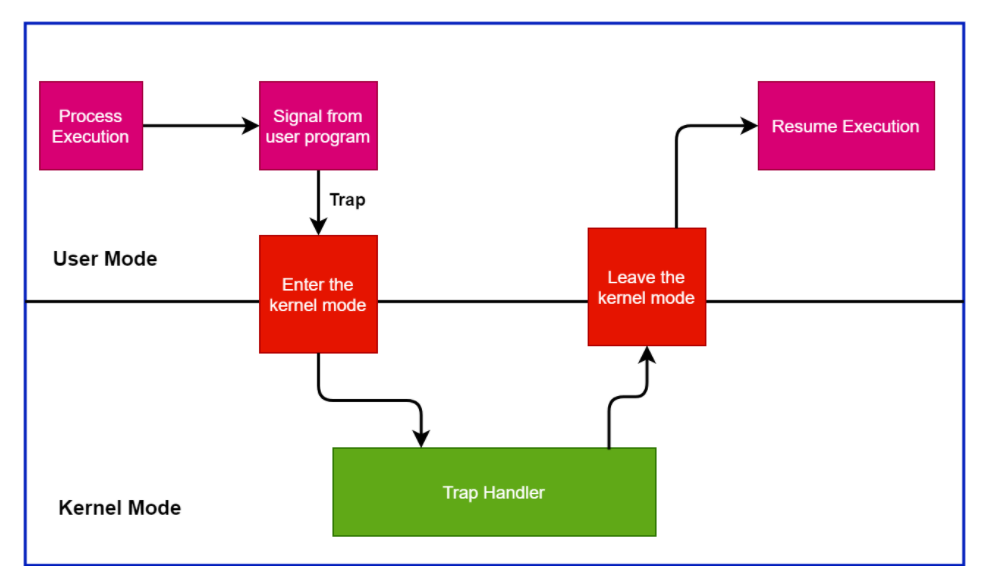
* Alert the CPU to events that require its attention
* OS is event driven, executes only when there is an interrupt

Events are categorized into 3 Types:

Hardware Interrupts (called Interrupts0 –

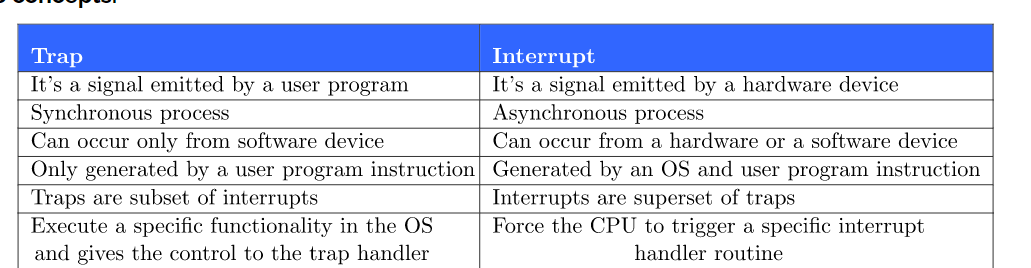
* Raised by IO devices
* May happen any time

Traps

* Software Interrupts, raised by user software programs, invokes OS functionality
* Example, div by zero
* 

Exceptions

* Raised automatically by the processor itself
* Has faults and aborts, two types of exceptions
* Faults is recoverable errors
* Aborts – difficult to recover



Interrupt Controller –

* Job is to ensure that the single pin of the interrupt is shared between different IO devices.

What happens with an Interrupt?

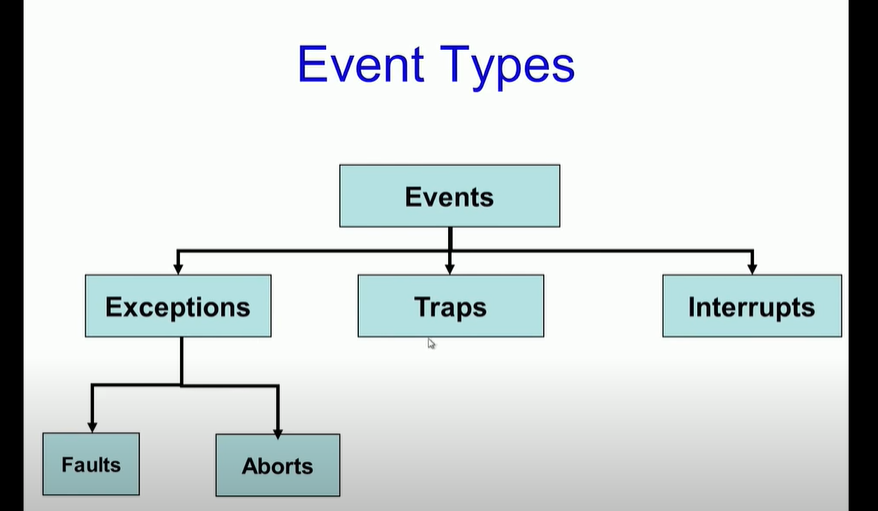
* When interrupted, CPU stops what it is doing and transfers execution to a location. Once the execution is complete, the process continues running and CPU works normally.

Interrupt Vector

* Holds addresses of the interrupt routine for devices

Interrupt Request Line

* Wire that CPU senses after executing an intruction. When it asserts a signal on the line, it reads the interrupt number and jumps to the interrupt-handler routine



Nonmaskable Interrupt

* Interrupt request line Reserved for events such as unrecoverable memory errors

Maskable Interrupt

* Interrupt Request line that can be turned off by CPU

Interrupt chaining

* in which each element in the interrupt vector points to the head of a list of interrupt handlers.

Interrupt Priority Levels

* Levels that show and rank the interrupt requests in a level of priority, starting with 0.

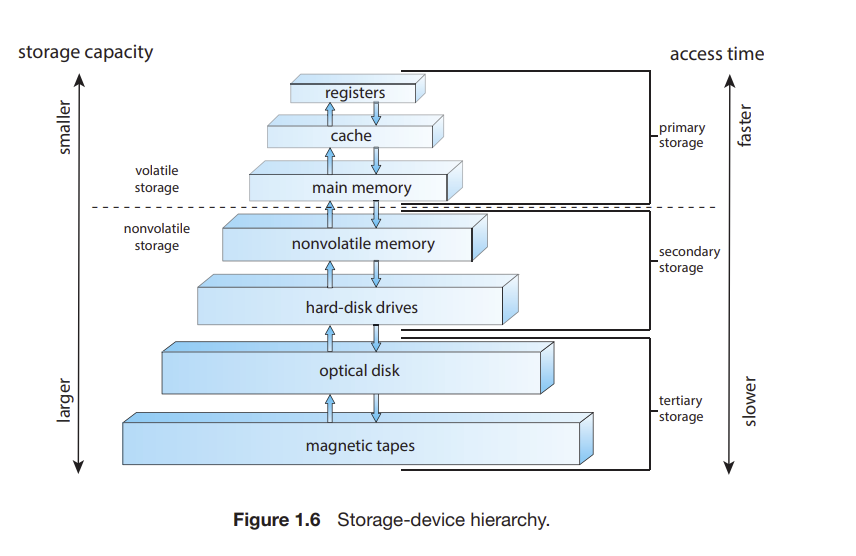
Memory

* CPU Loads intructions from memory, so anything must be loaded into it first
* Most computers run it in RAM, or main memory in DRAM.
* RAM is volatile, loses its content when powered off

Instruction-Execution cycle

* Examples are von Neumann architecture, it first fetches intrsuction from memory and stores that intruction in the intruction register.

Secondary Storage

* B/c ram does not have all the space, we use secondary storage such as Hard disk drives or NVME drives. They will keep data inside of it until it is loaded on to memory
* Secondary storage is much slower than Main memory.
* 
* Registers are fastest, magnetic tapes slowest

I/O structure

* Basically, it is exchanging common data between a bus.
* DMA – direct memory access – the direct transfer of an entire block of data directly to or from the device and main memory with no CPU intervention. CPU is available during this time.

Core of computer

* Core is the component that executes intructions and registers for storing data locally

Multiprocessor systems

* Use two or more process each with a single core CPU.
* Main advantage is increased number of throughput with the number of processors.

SMP – Symmertric Multiprocessing

* Each peer CPU performs all tasks including OS functions and User processes.
* Benefit – can run many processors simultaenously

Multicore

* Multiple cores on a single chip (\*more efficient)

Cache

* Local cache is claled L1 or level 1 cache

CPU—The hardware that executes instructions.

• Processor—A physical chip that contains one or more CPUs.

• Core—The basic computation unit of the CPU.

• Multicore— Including multiple computing cores on the same CPU.

• Multiprocessor— Including multiple processors.

The CPUs are connected by a shared system interconnect,  
- so that all CPUs share one physical address space. Thisapproach—known as non-uniform memory access, or NUMA

**blade servers**

* are systems in which multiple processor boards, I/O boards, and networking boards are placed in the same chassis

Clusters

* Another type of multiprocessor system is a **clustered system**, which gathers together multiple CPUs
* Clustering is usually used to provide high-availability service— that is, service that will continue even if one or more systems in the cluster fail

Parallelization

* divides a program into separate components that run in parallel on individual cores in a computer or computers in a cluster.

SAN

* storage-area networks (SANs), as described in Section 11.7.4, which allow many systems to attach to a pool of storage

1.4 OS Operations

Systems Daemons

* system daemons, they are programs that are loaded into memory at boot time which run the entire time the kernel is running
* A daemon is a long-running background process that answers requests for services.

Interrupts

* If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt
* Another form of interrupt is a **trap** (or an exception), which is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed by executing a special operation called a system call

Multiprogramming/Multitasking

* One of the most important aspects of operating systems is the ability to run multiple programs, as a single program cannot, in general, keep either the CPU or the I/O devices busy at all times
* Multiprogramming – increases CPU utilization by organizing programs so that CPU always has one to execute.
* A program in execution is called “process”
* Multitasking is a logical extension of multiprogramming. In multitasking systems, the CPU executes multiple processes by switching among them, but the switches occur frequently, providing the user with a fast response time

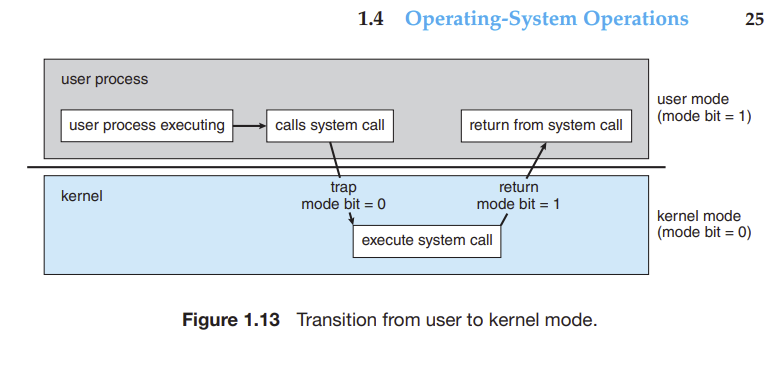
CPU Scheduling

* In addition, if several processes are ready to run at the same time, the system must choose which process will run next. Making this decision is CPU scheduling

Virtual Memory

* In a multitasking system, the operating system must ensure reasonable response time. A common method for doing so is virtual memory, a technique that allows the execution of a process that is not completely in memory (Chapter 10). The main advantage of this scheme is that it enables users to run programs that are larger than actual physical memory.

Dual-Mode and Multimode Operation

* In order to ensure the proper execution of the system, we must be able to distinguish between the execution of operating-system code and user-defined code
* At the very least, we need two separate modes of operation: user mode and kernel mode (also called supervisor mode, system mode, or privileged mode).
* A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1).
* 
* We can now better understand the life cycle of instruction execution in a computer system. Initial control resides in the operating system, where instructions are executed in kernel mode. When control is given to a user application, the mode is set to user mode.
* Eventually, control is switched back to the operating system via an interrupt, a trap, or a system call.

Sys calls

* System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program’s behalf. A system call is invoked in a variety of ways, depending on the functionality provided by the underlying processor. In all forms, it is the method used by a process to request action by the operating system. A system call usually takes the form of a trap to a specific location in the interrupt vector. This trap can be executed by a generic trap instruction, although some systems have a specific syscall instruction to invoke a system call
* When a system call is executed, it is typically treated by the hardware as a software interrupt. Control passes through the interrupt vector to a service routine in the operating system, and the mode bit is set to kernel mode

Timer

* We must ensure that the operating system maintains control over the CPU. We cannot allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system. To accomplish this goal, we can use a timer.

Neumann

* The CPU reads instructions from main memory during the instruction-fetch cycle and both reads and writes data from main memory during the data-fetch cycle (on a von Neumann architecture).

Cache

* Information is normally kept in some storage system (such as main memory). As it is used, it is copied into a faster storage system— the cache—on a temporary basis
* When we need a particular piece of information, we first check whether it is in the cache. If it is, we use the information directly from the cache.
* If it is not, we use the information from the source, putting a copy in the cache under the assumption that we will need it again soon

Virtualization

* Allows us to abstract the hardware of a single computer into different execution environments
* This creates an illusion that each part runs on its own.
* Emulation, which involves simulating computer hardware in software, is typically used when the source CPU type is different from the target CPU type.
* Example, emulators that make you run PS4 games on Ps5

• An operating system is software that manages the computer hardware, as

well as providing an environment for application programs to run.

• Interrupts are a key way in which hardware interacts with the operating

system. A hardware device triggers an interrupt by sending a signal to the

CPU to alert the CPU that some event requires attention. The interrupt is

managed by the interrupt handler.

• For a computer to do its job of executing programs, the programs must be

in main memory, which is the only large storage area that the processor

can access directly.

• The main memory is usually a volatile storage device that loses its contents

when power is turned off or lost.

52 Chapter 1 Introduction

• Nonvolatile storage is an extension of main memory and is capable of

holding large quantities of data permanently.

• The most common nonvolatile storage device is a hard disk, which can

provide storage of both programs and data.

• The wide variety of storage systems in a computer system can be organized

in a hierarchy according to speed and cost. The higher levels are expensive,

but they are fast. As we move down the hierarchy, the cost per bit generally

decreases, whereas the access time generally increases.

• Modern computer architectures are multiprocessor systems in which each

CPU contains several computing cores.

• To best utilize the CPU, modern operating systems employ multiprogramming, which allows several jobs to be in memory at the same time, thus

ensuring that the CPU always has a job to execute.

• Multitasking is an extension of multiprogramming wherein CPU scheduling algorithms rapidly switch between processes, providing users with a

fast response time.

• To prevent user programs from interfering with the proper operation of

the system, the system hardware has two modes: user mode and kernel

mode.

• Various instructions are privileged and can be executed only in kernel

mode. Examples include the instruction to switch to kernel mode, I/O

control, timer management, and interrupt management.

• A process is the fundamental unit of work in an operating system. Process management includes creating and deleting processes and providing

mechanisms for processes to communicate and synchronize with each

other

* An operating system manages memory by keeping track of what parts of

memory are being used and by whom. It is also responsible for dynamically allocating and freeing memory space.

• Storage space is managed by the operating system; this includes providing

file systems for representing files and directories and managing space on

mass-storage devices.

• Operating systems provide mechanisms for protecting and securing the

operating system and users. Protection measures control the access of

processes or users to the resources made available by the computer system.

• Virtualization involves abstracting a computer’s hardware into several

different execution environments.

2/23 Notes

Midterm – End march/ First wk April

Programming Project – Starting Next week

Quizzes – End of march

Between Midterm to Final – One more projet

Chapter 3 – read all of chapter 3 except 3.7.2, .3, 3.8.2.1

Types of Operating Systems

1. Monolithic
2. Layering
3. Microkernel
4. Loadable Kernel Modules
5. Linux (Hybrid model)

Issue with Layering

* Needs to traverse thru many layers to interact with user/CPU

Microkernel

* Breaking down the kernel into smaller kernels, yet having its own system.
* Interprocess communication – many programs running & need to communicate with eachother.
* No need to worry about a gigantic kernel
* Interprocess communication used to communicate with Programs

Modular + Monolithic (Linux(

* Benefits are that everything is best of both worlds
* Has the speed & no code bloat (as it can dynamically load/unload kernel functionality
* Avoids downsides of each system.
* Don’t get all the benefits but neither do we get all the bad parts
* Basically tradeoffs.

File Descriptor

* Reserved
* Write(stdout-filestream #, NAME of file)
* Printf calls write in an easier format by not having you to do the system calls.

Int I = 52; // how to convert into char stream?

i/10 = 5;

I % 10 = 2;

--- You may have to do this instead of doing a printf, this is systems programming.

HW 2 Answers

1. Services and functions –

Services

1. General 3 things

Trap (sys call # is called)

1. Parameters can be passed in registers
2. Registers can pass starting address of blocks of parameters
3. Parameters can be placed or pushed onto the stack by the program and popped off the stack by the operating system.
4. What are adv/disadvantages of using same sys call interface for manipulating both files and devices?

Since most of kernel deals with devices in this interface it is easier to add new device driver by implimitngting proper code. Code can be written in the same manner which can be helpful in writing a well defined API that works with it.

Disadvanatage – it might be difficult to capture the functionality of certain devices with in the context of the file access api which causes loss of functionality or performance.

1. Two models of interprocess communication? Strengths and weaknesees?

Message passing and shared memory. Shared is faster but more difficult to program, message passing is easier thru the APIS

1. Adv of using loadable kernel mmodules?

Difficult to say, but functionality can be added and removed from the kernel while it is running, there is no need to recompile or reboot the kernel. Helps keep high uptimes for users.

2.9) The services and functions provided by an operating system can be divided into two main categories.

Briefly describe the two categories, and discuss how they differ.

Answer: Services and functions provided by an operating system may be provided to either a user or

towards proper funtioning of the system. Services provided to the user reflect the ease and functionality

the system must provide for the user to interact with the system. Services that ensure efficient operation

of the system involve allocating system resources and well as providing the necessary protection and

security of the system. (See Section 2.7.2)

2.10) Describe three general methods for passing parameters to the operating system.

Answer: a. Parameters can be passed in registers.

b. Registers can pass starting addresses of blocks of parameters.

c. Parameters can be placed, or pushed, onto the stack by the program and popped off the stack

by the operating system.

(See Section 2.3.2)

2.12) What are the advantages and disadvantages of using the same system-call interface for manipulating

both files and devices?

Answer: An advantage of using the same system-call interface for manipulating both files and devices is that,

since most of the kernel deals with devices through this interface, it is relatively easy to add a

new device driver by implementing the appropriate hardware-specific code. This benefits the development

of both user program code, which can be written to access devices and files in the same manner, and

device-driver code, which can be written to support a well-defined API. The disadvantage of using the

same interface is that it might be difficult to capture the functionality of certain devices within the

context of the file-access API, resulting in either a loss of functionality or a loss of performance.

2.15) What are the two models of interprocess communication? What are the strengths and weaknesses of the

two approaches?

Answer: The two models of interprocess communication (IPC) are the message-passing model and the shared-memory model.

Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided.

It is also easier to implement than is shared memory for intercomputer communication. Shared memory

allows maximum speed and convenience of communication, since it can be done at memory transfer speeds

when it takes place within a computer. However, this method involves problems in the areas of protection

and synchronization between the processes sharing memory. More on this topic in Chapter 3.

(See Section 2.3.3.5)

2.20) What are the advantages of using loadable kernel modules?

Answer: It is difficult to predict what features an operating system will need when it is being designed. The

advantage of using loadable kernel modules is that functionality can be added to and removed from the

kernel while it is running. There is no need to either recompile or reboot the kernel. (See Section 2.8.4)

# Process Memory Layout

* Has 4 major sections ----- Stack, Heap, Data, Text
* Used for global variables that occur only once.

Data and text section in executable and RAM

* Located in .exe file, it will only be a text value and a separate section for Initialized global variable and uninitialized global variables.
* In RAM however, data and text are just stored as they are.
* Once exe is moved to RAM, they collapse into the two sections of RAM – Data and text

From the time till function is invoke till the end, the stack keeps record of whatever runs through it.

Things we need on the stack / invocation record

1. Input parameters
2. Expression Evaluation
3. Stack Management Pointers (on a per function basis)
4. Local Variables

Invocation Record – As a function returns, stack starts shrinking in size. The record keeps every past command remembered

Why do we need a heap?

* DMA at runtime, the heap gives us this.
* How to dynamically do this? Sys call or CSTD library
* While storage for variable is on the stack, the \*5 dynamically allocated memory are on the heap .
* Heap grows and shrinks – used when working with lots of memory & memory that we want to use as we go, or we intialize it.

2/28/22

What we are coering today:

* Quiz 1 answers
* Finish Process Memory Layout
* ON the project (due on Midterm time)
* Process cration/termination
* Producer/ consumer model
* Shared memory/ posix
* Cpfile.c

Q1 Quiz) Multiprogramming, B Correct

#2 – process - correct

#3 – A & B correct

4) B – kernel sys call interface

#5) Alternative to hardware interrupt mechanism for comuncation – POLLING

Process Memory Layout

* Refer to the diagram images in the respective chapter
* In the data section, there are two parts – one is uninitialized and other is for intialized calues.
* The process Layout – has a stack, heap,
* The stacj controls the function calls – starts from the main function, then has the values
* It has 1) Input paramter values 2) local variables 3) expression evaluation 4) Stack maagement pointers
* Every func call gets an invocation

Heap – whats its use?

* If you don’t have space in the stack, you instead will use the heap’s storage at runtime using DMA. Dynamic Memory Allocation
* Use DMA anytime you don’t have much space. In C language we use Malloc to do this.
* Malloc is a great example to get more memroy (it includes a system call in it whnever it is clled.) to get arround this, just do one single malloc with the the ability to cover all future needs
* You don’t want to call malloc too much as it will cause the system too much lag & will slow you down.

Process Creation/ Termination

* Every new process has a different ID. New processes are created out of existing processes\
* Diagram of a parent child process.
* Every process created comes out of a parent process. Parents go all the way up to the original, first process.
* Code on the page – page 3.23 Slide

C program forking separate process.

* When a parent creates a child, the parent creates a clone of itself. This now has a process ID.
* A clone is created because of the fork call to keep a copy of it in memory. As a result this causes different process IDs, but besdies that it is all normal.

Execlp – sys call to run another program.

* First part, the fork runs, but first time it was just a copy. But if we want child to do something we need to run execlp to allow the child to do something.
* Alternatively, we can tell the child to do else if (pid ==0). By doing this, we can tell the child to directly run the code. This is what we’ll need to do on the project, shared memory.
* Parent and child will work togethr to work out the cpfile.c
* We will breka the while loop and the read will be done in parent, write will be done to a child process. But parent and child will communicate.
* Parent reads source file and will share to the child so that they can complete the process. They communicate via IPC.
* Parent producing for consumption by the child.
* Parent producer – write
* Child consumer – read and write to a target file.
* Does the parent/child creation process slow down the process? It depends on the CPU process scheduler.
* Put together the 4 different parent child, forking, producer consumer model into shared memory, posix system call, cp file code acorss two processes parent and child.

Process Creation Contd:

* Fork() system calll
* Every process should have a return code.

Every process has 1 parent (system daemon is the main one, doesn’t have a parent. It is part of the kernel.)

Abort() is not a sys call. GZives parent the ability to abort a process from child/ abort child process.

Orphan process- parent goes away, child process is still alive

Zombie Process –

Wait() – blocks

Waitpid() no blocking (like polling)

Asynchronous – using Unix signals.

If child is an orphan, the daemon (PID = 0 ) gets control of the termination on that child.

Aynschronous way -

Signal( SIGCHILD, myhandlerfunction)

Myhandler() { wait() };

By doing this, you wouldn’t have to do waitpid() in the above approach. No polling in this approach, and there is no blocking by doing a non-blocking PID.

Don’t miss next lecture (weds).

Chapter 2 Notes from Textbook

We can view an operating system from several vantage points. One view focuses on the services that the system provides; another, on the interface that it makes available to users and programmers; a third, on its components and their interconnections. In this chapter, we explore all three aspects of operating systems, showing the viewpoints of users, programmers, and operating system designers. We consider what services an operating system provides, how they are provided, how they are debugged, and what the various methodologies are for designing such systems. Finally, we describe how operating systems are created and how a computer starts its operating system.

CHAPTER OBJECTIVES

• Identify services provided by an operating system.

• Illustrate how system calls are used to provide operating system services.

• Compare and contrast monolithic, layered, microkernel, modular, and

hybrid strategies for designing operating systems.

• Illustrate the process for booting an operating system.

• Apply tools for monitoring operating system performance.

• Design and implement kernel modules for interacting with a Linux kernel

2.1 Operating-System Services

Some operating system services are:

* User interface UI – Mostly a GUI is used, although CLI and other methods can also be used to interface with the OS. Example Microsoft windows OS
* Program execution – Must be able to load a program into memory and run the program. Must help it execute properly or improperly (error occuring)
* IO operations – allows reading in from a network device or other IO device like a mouse.
* File system manipulation – Able to manipulate files, read and write directories, create and delete files, rename/search
* Communication – should be able to do shared memory or message passing. Communications may be implemented via shared memory, in which two or more processes read and write to a shared section of memory, or message passing, in which packets of information in predefined formats are moved between processes by the operating system.
* Error detection. The operating system needs to be detecting and correcting errors constantly. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on disk, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow or an attempt to access an illegal memory location).
* Resource Allocation - When there are multiple processes running at the same time, resources must be allocated to each of them. The operating system manages many different types of resources. Some (such as CPU cycles, main memory, and file storage) may have special allocation code, whereas others (such as I/O devices) may have much more general request and release code
* Logging. We want to keep track of which programs use how much and what kinds of computer resources.
* Protection and security. The owners of information stored in a multiuser or networked computer system may want to control use of that information.

2.2 User and Operating-System Interface

- We mentioned earlier that there are several ways for users to interface with the operating system. Here, we discuss three fundamental approaches. One provides a command-line interface, or command interpreter, that allows users to directly enter commands to be performed by the operating system. The other two allow users to interface with the operating system via a graphical user interface, or GUI.

Shells –

* On systems with multiple command interpreters to choose from, the interpreters are known as shells. For example, on UNIX and Linux systems, a user may choose among several different shells, including the C shell, Bourne-Again shell, Korn shell, and others.
* The main function of the command interpreter is to get and execute the next user-specified command.
* This includes create, delete, print, copy, execute, etc
* You can use GUI, CLI, your choice of interface to work with the shell

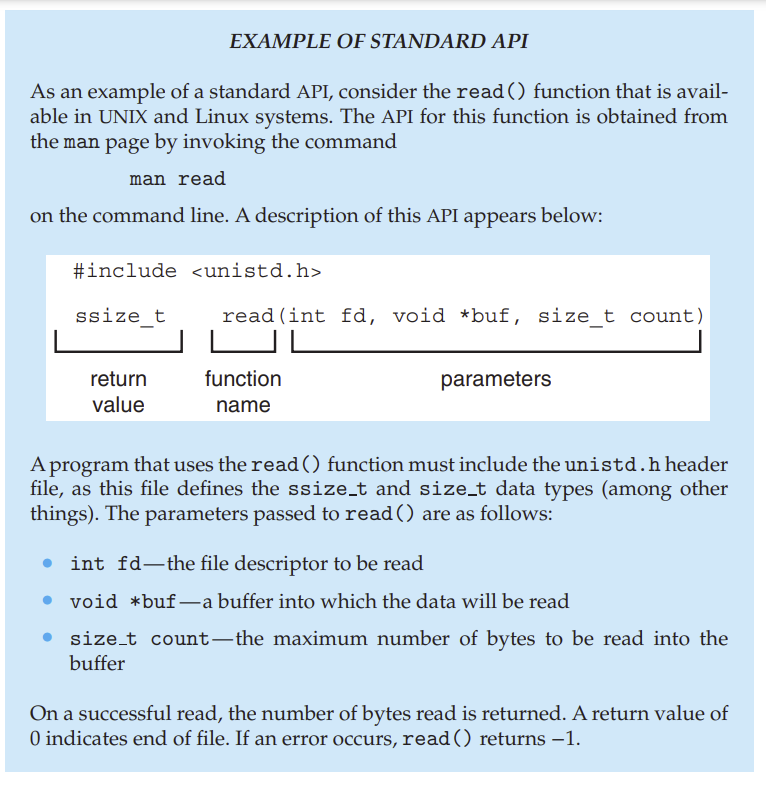
2.3 System Calls

System calls provide an interface to the services made available by an operating system. These calls are generally available as functions written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly) may have to be written using assembly-language instructions. MUST READ UP ON SYS CALLS

2.3.2 APIs

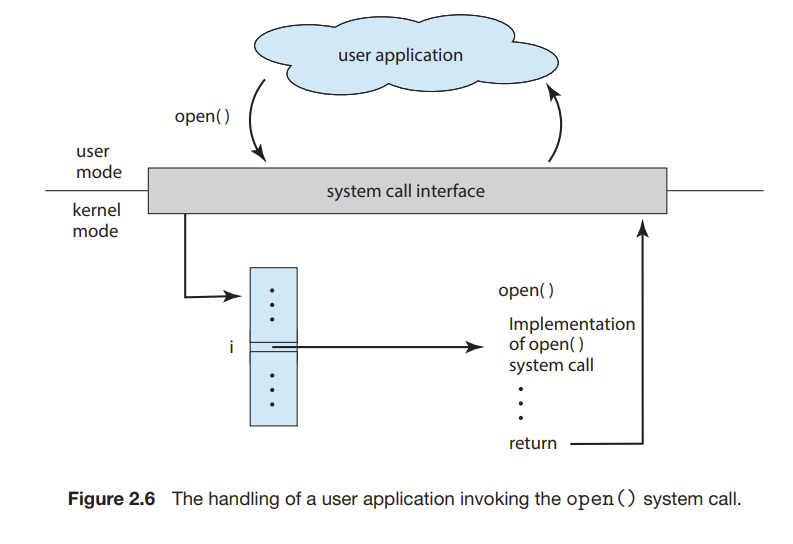
The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect. Three of the most common APIs available to application programmers are the Windows API for Windows systems, the POSIX API for POSIX-based systems (which include virtually all versions of UNIX, Linux, and macOS), and the Java API for programs that run on the Java virtual machine. A programmer accesses an API via a library of code provided by the operating system. In the case of UNIX and Linux for programs written in the C language, the library is called libc.

Behind the scenes, the functions that make up an API typically invoke the actual system calls on behalf of the application programmer. For example, the Windows function CreateProcess() (which, unsurprisingly, is used to create a new process) actually invokes the NTCreateProcess() system call in the Windows kernel



RTE – Run Time environment

run-time environment (RTE)— the full suite of software needed to execute applications written in a given programming language, including its compilers or interpreters as well as other software, such as libraries and loaders.



Sys-Call Interface - system-call interface that serves as the link to system calls made available by the operating system. The system-call interface intercepts function calls in the API and invokes the necessary system calls within the operating system. Typically, a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers. The systemcall interface then invokes the intended system call in the operating-system kernel and returns the status of the system call.

2.3.3 Types of System Calls

System calls can be grouped roughly into six major categories: process control, fil management, device management, information maintenance, communications, and protection

PAGE 97 on slides

• Process control

◦ create process, terminate process

◦ load, execute

◦ get process attributes, set process attributes

◦ wait event, signal event

◦ allocate and free memory

• File management

◦ create file, delete file

◦ open, close

◦ read, write, reposition

◦ get file attributes, set file attributes

• Device management

◦ request device, release device

◦ read, write, reposition

◦ get device attributes, set device attributes

◦ logically attach or detach devices

• Information maintenance

◦ get time or date, set time or date

◦ get system data, set system data

◦ get process, file, or device attributes

◦ set process, file, or device attributes

• Communications

◦ create, delete communication connection

◦ send, receive messages

◦ transfer status information

◦ attach or detach remote devices

• Protection

◦ get file permissions

◦ set file permissions

Lock – Shared data is sometimes locked in order to protect the data to not be accessed by anyone else.

Start a new process – fork()

File Management

* Use syscalls such as create(), delete (), write(), open(), reposition() close() to do different things for files. Some OS provide more calls such as move() or copy(). Others even provide APIs using sys calls.
* You can also use time() date() to find the time. Strace() lists calls being run. Dump() dump memory.
* CPU Mode Single Step - Even microprocessors provide a CPU mode, known as single step, in which a trap is executed by the CPU after every instruction. The trap is usually caught by a debugger.

Communication 2.3

* Two Models of Communication: 1) Message passing 2) Shared Memory

Message Passing - the communicating processes exchange messages with one another to trans- 2.3 System Calls 73 fer information. Messages can be exchanged between the processes either directly or indirectly through a common mailbox. Uses process\_id to find out which is a process

Shared Memory - In the shared-memory model, processes use shared memory create() and shared memory attach() system calls to create and gain access to regions of memory owned by other processes. Recall that, normally, the operating system tries to prevent one process from accessing another process’s memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas.

* Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. It is also easier to implement than is shared memory for intercomputer communication. Shared memory allows maximum speed and convenience of communication, since it can be done at memory transfer speeds when it takes place within a computer. Problems exist, however, in the areas of protection and synchronization between the processes sharing memory

2.4 System Services

System services, also known as system utilities, provide a convenient environment for program development and execution.

• File management. These programs create, delete, copy, rename, print, list,

and generally access and manipulate files and directories.

• Status information. Some programs simply ask the system for the date,

time, amount of available memory or disk space, number of users, or

similar status information. Others are more complex, providing detailed

performance, logging, and debugging information. Typically, these programs format and print the output to the terminal or other output devices

or files or display it in a window of the GUI. Some systems also support a

registry, which is used to store and retrieve configuration information.

• File modificatio . Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may

also be special commands to search contents of files or perform transformations of the text.

• Programming-language support. Compilers, assemblers, debuggers, and

interpreters for common programming languages (such as C, C++, Java,

and Python) are often provided with the operating system or available as

a separate download.

• Program loading and execution. Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may

provide absolute loaders, relocatable loaders, linkage editors, and overlay

loaders. Debugging systems for either higher-level languages or machine

language are needed as well.

• Communications. These programs provide the mechanism for creating

virtual connections among processes, users, and computer systems. They

allow users to send messages to one another’s screens, to browse web

pages, to send e-mail messages, to log in remotely, or to transfer files from

one machine to another.

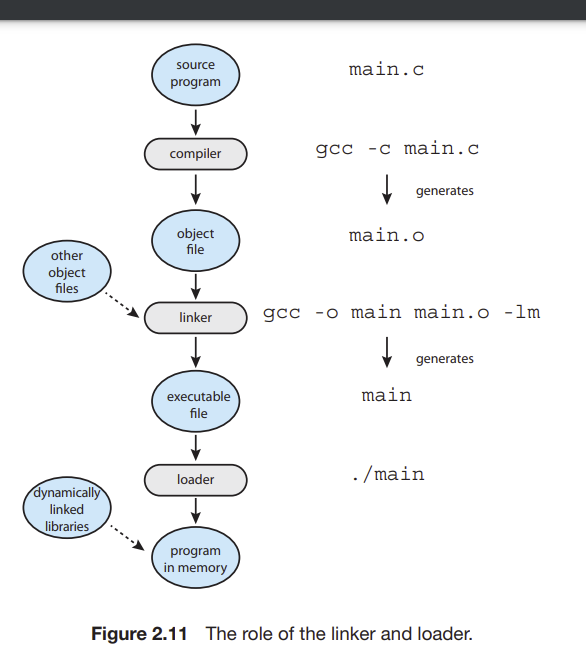
• Background services. All general-purpose systems have methods for

launching certain system-program processes at boot time. Some of these

processes terminate after completing their tasks, while others continue to

run until the system is halted.

Linkers and Loaders 2.5

* Linker – combines relocatable object files into a single binary exe file. Other libraries are added in this time.
* Loader – Used to load the binary exe file into memory, where it is eligible to run on a CPU core. Activity with linking and locading is called relovation.
* 

2.6 Different OS Programs

- POSIX API is used to have a standard API for different platforms

ABI – Appplication binary interface – used to define how different compenents of binary code can interface for a given OS on a given architecture.

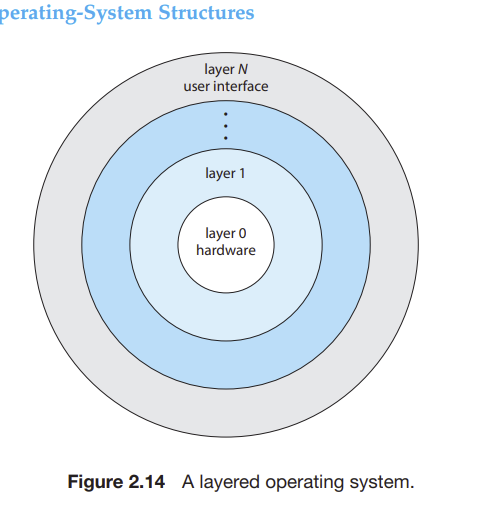
2.7 OS Design and implementation

Design Goals

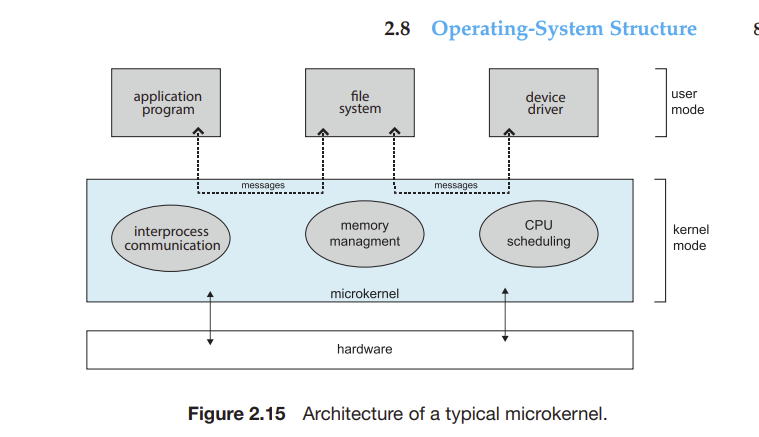
* Two requiremenets – User goals and System Goals.
* Users want processes in a system, like fast loading, safe. System should be efficient.
* One important principle is the separation of policy from mechanism. Mechanisms determine how to do something; policies determine what will be done. For example, the timer construct (see Section 1.4.3) is a mechanism for ensuring CPU protection, but deciding how long the timer is to be set for a particular user is a policy decision

2.8 Different Structures

- Monolithic – no structure at all.

- Layered Approach – OS is broken into a number of layers. 

- Microkernels – easy to add new services to the OS



* Modules – Best method is using Loadable Kernel Modules (LKM’s)
* the kernel has a set of core components and can link in additional services via modules, either at boot time or during run time. This type of design is common in modern implementations of UNIX, such as Linux, macOS, and Solaris, as well as Windows
* The idea of the design is for the kernel to provide core services, while other services are implemented dynamically, as the kernel is running.

## 2.11 Summary

• An operating system provides an environment for the execution of programs by providing services to users and programs.

• The three primary approaches for interacting with an operating system

are (1) command interpreters, (2) graphical user interfaces, and (3) touchscreen interfaces.

• System calls provide an interface to the services made available by an operating system. Programmers use a system call’s application programming

interface (API) for accessing system-call services.

• System calls can be divided into six major categories: (1) process control,

(2) file management, (3) device management, (4) information maintenance,

(5) communications, and (6) protection.

• The standard C library provides the system-call interface for UNIX and

Linux systems.

• Operating systems also include a collection of system programs that provide utilities to users.

• A linker combines several relocatable object modules into a single binary

executable file. A loader loads the executable file into memory, where it

becomes eligible to run on an available CPU.

• There are several reasons why applications are operating-system specific.

These include different binary formats for program executables, different

instruction sets for different CPUs, and system calls that vary from one

operating system to another.

• An operating system is designed with specific goals in mind. These goals

ultimately determine the operating system’s policies. An operating system

implements these policies through specific mechanisms.

• A monolithic operating system has no structure; all functionality is provided in a single, static binary file that runs in a single address space.

Although such systems are difficult to modify, their primary benefit is

efficiency.

• A layered operating system is divided into a number of discrete layers,

where the bottom layer is the hardware interface and the highest layer is

the user interface. Although layered software systems have had some suc-

Further Reading 101

cess, this approach is generally not ideal for designing operating systems

due to performance problems.

• The microkernel approach for designing operating systems uses a minimal

kernel; most services run as user-level applications. Communication takes

place via message passing.

• A modular approach for designing operating systems provides operatingsystem services through modules that can be loaded and removed during

run time. Many contemporary operating systems are constructed as hybrid

systems using a combination of a monolithic kernel and modules.

• A boot loader loads an operating system into memory, performs initialization, and begins system execution.

• The performance of an operating system can be monitored using either

counters or tracing. Counters are a collection of system-wide or perprocess statistics, while tracing follows the execution of a program through

the operating system.

3/2/2022

Today’s topics

* Finish process creation/termination table thing (Slide of “C program forking separate process ) I missed this in the class.
* Producer/consumer model for IPC

POSIX memory shared

Project Program

* Pointer comes before the fork.
* Need to ask Hassan & Wahab notes on this

Interprocess Communication

* Processes can be independent or cooperating

Reasons for cooperating

* Info sharing, compute speedup, modularity convenience,
* Cooperating processes need interprocess communication.
* Two models of IPC – Shared memory and message passing.

Central Computer Diagram

* Client server model – this allowed a server to just hold the business logic whereas clients would have the GUI logic.
* This is a prime example of modularization

Dumb Terminal - no CPU

Shared memory

* This will be on the project, so he is covering extensively.
* Shared memory model has – Process, shared memory, process B, and the kernel.
* Shared Memory is something that would be available to process A and Process B>

Producer – processes info that is consumed by a consumer process.

Unbounded – buffer – Places no practical limit on size of buffer. Limitless memory

Bounded buffer – assumes there is a fixed buffer size.

Buffer

* In the diagram, the buffer is designed as an array
* Consumer(child) consumes the write of the Parent (producer)
* Consumer reads the produced shared memory and converts into its own local variable. After this, the shared memory is no longer necessary and will be trashed.
* Sometimes the consumer will eat up ( read) data quicker than the producer can produce theknowledge. In this case, a buffer will slow it down
* Synchronization will help in this case by

Project - Tying together 4 pieces of code and see if they work in harmony. Will have to do pointers in C language. Will need to do indirection with Pointers (dereference)

Message Passing

* Shared Memory Posix API.
* Need to learn some sys calls to do this.
* How would we know that a file ended? We need to make up our own way of how to break out of a file (either Boolean, or something else)

Set up shared memory

* Create share memory segment.

POSIX Shared Memory – slide 3.48

* First create shared memory segment
* Set size of object using ftruncate(shm\_fd, 4096) #truncate segment into this size.
* Buffer \* chunk size.
* Will also need sizeof(int)
* 3rd call will be nmap(). To make shared mem available in own programming space to a pointer so that it maps it properly.

Producer Code

* First thing – include files must be correct. Add in shared memory, sys stat.h
* Producer produces content (like hello world printing.)
* Page 3.49
* Create shared mem segment, then make a file (4;18 time in class, check the notes). Do error checking = 0 on this.
* Configure size using ftruncate.
* Typecasting to char\*
* All this producer code on page 3.49 of slide
* Fd = shm\_open()
* Ftruncate()
* Ptr = nmap(producer)
* Retpid = fork();
* Child

Consumer Code

* Has same include files and the unique name.’
* Read only when you open the shared meory object. In our project we don’t need to do

Srcfile

tgtfile

Buffer-SIZE

CHUNK-SIZE

* Code needs to be generalized. He will input these commands into your code to see if it works. Need to

Bounded buffer problem for Shared Memory

* pPseudo code is on page 3.35 of Producer process- shared memory

shm open, ftruncate, nmap, call for the integer in and out. These are synchronization variables. When it comes to that make sure

Put all that before the fork. Use in and out variables for pseudocode logic. He wants 4 parameters with different chunk size. Read write buffer size and chunck size. Values like 10 to 20. Typically smaller chunk size causes producer to slow down and stop as they will catch up to

Own input source file, he will tell us what

3/7/22

Todays Topics

* Pipes
* Process States/transitions
* HW 3.11, 3.12
* Process control block (PCB)
* Process Scheduling Queues.

HW will have a part 2 to it.

Pipes

* Acts as a conduit allowing two processes to communicate.
* Message sharing mechanism
* Very limited in the way you can utilize it.
* Cannot exist without a parent child relationship.
* Pipes are manifested as a set of file descriptors, they act as a show of telling you the read and write of a file
* Producer writes(), consumer read()

Code for files

#define maxline 128 // defines max buffer space

* Declare variables and file descriptors, fd[2]
* // The pipe is created and its file descriptors returned
* If pipe(fd < 0 ) { fprint --- “pipe failed”
* // do the above before the fork so that parent and child have info before the fork is done to it.
* After pid = fork();
* If pid < 0, fork failed
* If pid ==0, chlose the write end of the pipe and c
* Code is on Blackboard

HW Problems 3.11 and 3.12

11) Might be on quiz or exam

- See how focused you are. Can take different takes on this question

10-15 mins into lecture

* What happens in thi code? Including the intial parent process, how many processes are created by the program shown in the figure
* Getpid() shows us a parent.
* Process management sys calls – fork(), wait(), waitpid(), exit(), execlp(). There are a few others.
* Fork() was called 15 times, 16
* How many processes created? Assume I > 5, how many processes
* Fork creates two children in this sense, assuming that the I limit has not been reached.

12)

- If execlp runs till execution, executable will overlay what the child process currently has

Process states and transitions

* Diagram of process state, 3.9
* Process State Definition – in the lifetime of a process, it will go thru different states while it is in memory. There are 5 different states.
* CPU = running states
* Process spends most of the itme in ready, running, and waiting stages.
* Ready State – you are hoping that you are the process that will be in the run state. After program is loaded you don’t immediately get into CPU. You have to be scheduled to get into the PCU, that scheduling happenings from the schedulers.
* Ready Queue – where the CPU process scheduler looks to see which process should be run next..
* Queue does not mean that it is FIFO, but not necessarily. This algorithm can be different, but we adopt a general language that a queue is here.
* When you request invocation of a sys call, there is a wait to be executed. There needs to be a data structure to show whos in the waiting state.
* Wait has a “wait queue” which is a container that keeps the processes running in them.
* Running State – kernel runs the code that has been waiting/ready to run.
* New state and terminate only happens once.

Definining Turnaround Time Of A Process

* Time to load up and till the time the process terminates finally.
* The formula is:

Time(new state) + ∑ T (ready state) + ∑ T (running state) ∑ T (waiting state)+ T (Temrinate State)

Running State (CPU)

* A designer cannot affect this much, it depends on the programmer’s code

Waiting State

* A designer can’t affect this much (more than the running state tho). If program is an IO intensive program it will impact the wait times.

Ready Queue

* Also the same as above. Check recordings, 54 mins in.

Two Other States not mentioned but you need to know

* Halted Ready (subset of ready state if we suspend it, then we can resume the process once again.)
* Halted waiting State
* Both of these take the resources off to balance load of other processes
* Process state will be swapped out so that load will be less and resources can be used for computing. (Swapped out/ swapped in)
* Know the diagram by heart.

Process Control Block (PCB) Includes this and other section

* Parent Process
* Child Process
* Signal Disposition – you can ignore or take default behavior for it.
* Current, Root Directories
* Credentials (RWX)
* Pointer to Memory Map (Stack Heap Data Text)
* Kernel Stack Space
* Pointers are used as less memory intensive
* Page 3.12 on the slides

3/9/22

Todays Tasks

* Finish Process Controll Block
* Process Scheduling Queues
* Process context switches
* IPC message passing
* HW CH 3 # 8, 13
* Project 1 – Part 1 Update
* Start CH4 – Multithreading – Read Ch4 all except
* Quiz on Chapter 2 next Wednesday

Process Scheduling

* Types of scheduling queues – ready queue, wait queue

Wait Q includes;

* Hardware IO, Child termination, Software interrupts
* Only 1 wait queue at a time

Ready Q includes:

* One queue at a time

Run State

* Youre not even in a queue

Ready and Wait Queues

* You can add another field to process control block.
* Every process has only 1 PCB, no duplicates

Process Scheduling

1. First, everything is new and the process (state) is new and you go into the ready queue. Once you’re in there, you’re in the ready state.
2. From ready state, you get chosen to go into the CPU. You’ve been chosen by process scheduler and dispatcher and will not go into the CPU, and wil go into the running state.
3. Once the disk deice control runs, the IO is requested and goes into the wait queue. Once it process the IO it is now in the ready state.
4. After IO, a child process is created and you would go into the child termination wait queue (the wait state). You are here till child terminates.
5. After Child terminates you are back into ready state
6. After this, you wait for an interrupt as it is a sys call. You will then go into the interrupt wait queue and an interrupt occuers. After this interrupt you would go into the ready state
7. There is an additional process scheduling fork which is called interrupt. After the interrupt you would go into the ready state.
8. After this a program needs to terminate when finished. So, there is another fork that writes that the program, at its end, will terminae.

Process/Multithreading

* A thread is similar to a sub-process.
* Task\_struct – linux command, serch it

|  |
| --- |
| Stack |
| Heap |
| Data |
| Text |

Input params, expressions, stak management pointer

Single thread vs Multithread

* Single
* Text section, data section, heap,but there are multiple stack.

Multithread

* Each thread has its own private stack. Implications – single threaded has the entire pcb to itself, multithreaded share heap data and text section]
* Single thread is heavy, multithread is lightweight as everyone shares heap and text. Only stack space if free.
* Pointer is used in muthreadded process to alert
* Every thread has own stack, needs 2 be trailed

Process Context switching

Listen in, time at 3.17 – 40 mins in

CPU Diagram – Process Context Switches – IPC Messaging

* Old CPU only has 1 bank of registers to use
* New CPU has 4 banks of registers – having multiple banks would speed things up
* Have a prefix strategy for the save and restore…

Message Passing

Interprocess Communication – Message Passing

* Mechanism for processes to communicate and to sync their actions
* Message system – processes communicate with eachother without resorting to shared variables
* IPC Facility provides two operations – send (msg), receive (msg)
* The message size is either fixed or variable.
* Sender send into the queue (producer), consumer receives it
* If processes wish to communicate, they need to establish a communication link and exchange message via send recv
* You can have multiple sneders/multiple receivers

Sychronization – different behaviors of message passing

* Message passing may be either blocking or non blocking
* Blocking is considered synchronous
* Blocking send – the sending is blocked till msg recvd
* Blocking receive – The receiver is blocked till msg available
* Non-blocking is considered Asynch.
* Non blocking send – sender sends message and continue
* Non-blocking receive – the receiver receives a valid message, or null message
* Different combos are possible – if both send and recv are blocking, we have a rendezvous

Producer Shared memory Code – page 3.45

Project 1 Part 1 Update

* Very end of instruction file, there’s an error. See on the page
* Memory efficiency is important – he will grade based on that

Fork

* Fork uses a lot of memory so careful. Use one process with mult threads, and create more threads as it is cheaper

3/16/22

Topics:

* Chap 4 hw questions
* Project Part 2
* Multithreaded Server Architecture
* Sample M.T. code
* Boss-workers design pattern
* Thread tools
* Quiz

Design a way that child recognizes that the file has ended. Design needs that. (Lecture describes this). Both part 1 and 2 are due 9 am. Use a pipe, and used shared memory. Follow his instructions, must do that.

# of characters needs to be sent thru the pipe, not shared memory. 1st is shared memory, second is the pipe.

Project Pt 2 – cant use printf

You count total # of characters you put into shared memory.

Chunk size as a parameter, 20 \* chunk\_size + characters.

Count all the chunk size stuff.

Use divide/modulo arithmetic to do the final converting calculations.

Multithreaded Server Architecture

* Client requests server to have a process. Server then creates a thread for that process and constantly begins to listen for any new client requests.
* Worker Threads - threads that are called to do the processing work
* Thread – we need to supply it a function that will make it work

Network Listener’s Job

1. Listen for a request
2. Check what request type (INSERT, delete, etc)
3. Create the worker handler functions, and continue listening. This is called Async operation.

The BOSS thread drives everything in the model. It drives the processes, the worker threads, to do their work & goes back to work.

1. Data vs Task Partitioning (Hybrid)
2. Concurrency vs Parallelism
3. Design Pattern
4. (7 or 8 things to think about multithreading before applying it)

POSIX Stuff

* Runner Function – sums up from i to the #, and will come up with the result, then it will have a sum global variables. Main thread will then go into the pthread and run it, which will print a value. It will then use the block() function call from posix.

Chapter 3 Notes – Processes

Process - which is a program in execution. A process is the unit of work in a modern computing system.

# CHAPTER OBJECTIVES

• Identify the separate components of a process and illustrate how they are

represented and scheduled in an operating system.

• Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls

that perform these operations.

• Describe and contrast interprocess communication using shared memory

and message passing.

• Design programs that use pipes and POSIX shared memory to perform

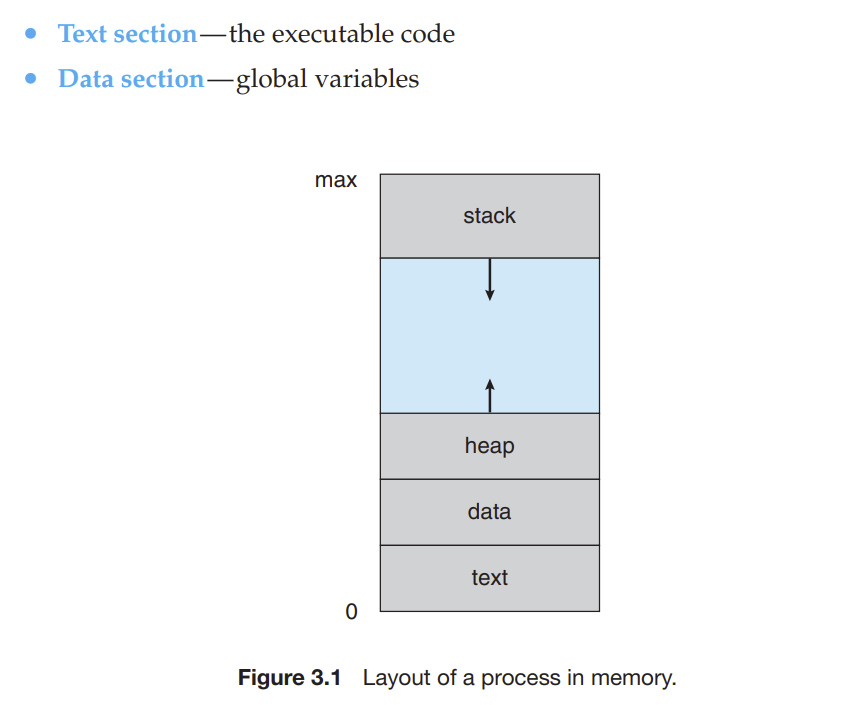
interprocess communication.

• Describe client – server communication using sockets and remote procedure calls.

• Design kernel modules that interact with the Linux operating system.

# 3.1 – Process Concept

Process- a program in execution

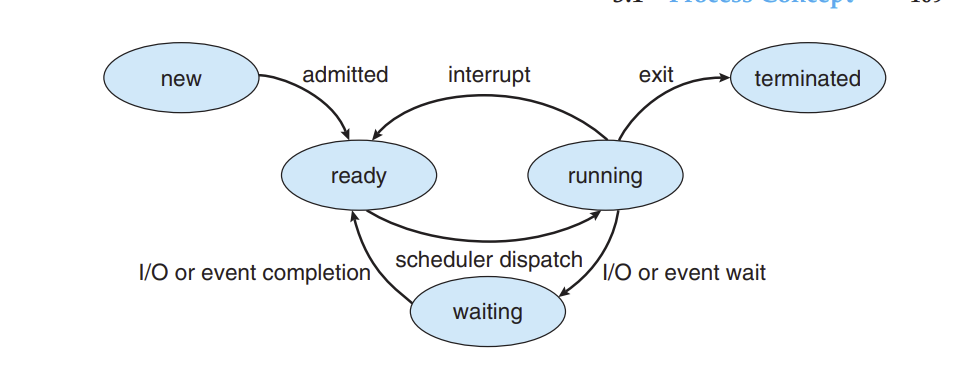
* The status of current activity of a process is represented by a value called program counter.
* The memory layour of a process is divided into multiple sections, as visualized below: 
* • Text section— the executable code
* • Data section—global variables
* Heap section—memory that is dynamically allocated during program run

time

* • Stack section— temporary data storage when invoking functions (such as

function parameters, return addresses, and local variables)

* The sizes of text and data sections are fixed at runtime, however stack and heap shrink/grow during program execution.
* Each time a function is called, an activation record containing function parameters, local variables, and the return address is pushed onto the stack; when control is returned from the function, the activation record is popped from the stack. Similarly, the heap will grow as memory is dynamically allocated, and will shrink when memory is returned to the system. Although the stack and heap sections grow toward one another, the operating system must ensure they do not overlap one another.
* A program by itself is not a process. Programs are passive entities. Process is active entity, where a program counter specifies the next instruction for it to execute.
* Programs turn into process when exe file is loaded into memory.
* As process executes, it changes it state, or current activity of the process. States are:
* • New. The process is being created.
* • Running. Instructions are being executed.
* • Waiting. The process is waiting for some event to occur (such as an I/O
* completion or reception of a signal).
* • Ready. The process is waiting to be assigned to a processor.



Process Control Block

* Each process is represented in the OS by a PCB. Includes info such as:
* • Process state. The state may be new, ready, running, waiting, halted, and
* so on.
* • Program counter. The counter indicates the address of the next instruction
* to be executed for this process.
* • CPU registers. The registers vary in number and type, depending on the
* computer architecture. They include accumulators, index registers, stack
* pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved
* when an interrupt occurs, to allow the process to be continued correctly
* afterward when it is rescheduled to run.
* • CPU-scheduling information. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
* (Chapter 5 describes process scheduling.)
* • Memory-management information. This information may include such
* items as the value of the base and limit registers and the page tables, or the
* segment tables, depending on the memory system used by the operating
* system (Chapter 9).
* • Accounting information. This information includes the amount of CPU
* and real time used, time limits, account numbers, job or process numbers,
* and so on.
* • I/O status information. This information includes the list of I/O devices
* allocated to the process, a list of open files, and so on

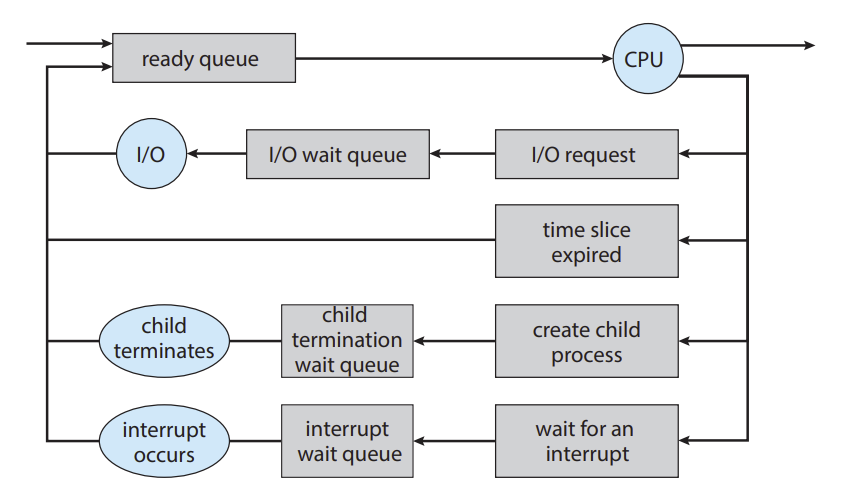
Basically, a PCB contains all necessary info about the process, is the repository for all the data needed to start or restart a process alllong with accompanying data.

Threads

* Allows multiple processes to be executed at the same time.
* Used on multiple core systems as they can handle the load.

# 3.2 Process Scheduling

**process scheduler** selects an available process (possibly from a set of several available processes) for program execution on a core. Each CPU core can run one process at a time.

* For a system with a single CPU core, there will never be more than one process running at a time, whereas a multicore system can run multiple processes at one time.
* In general, most processes can be described as either I/O bound or CPU bound
* An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations.
* A CPU-bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations.
* As processes enter system, they are put into ready queue, ready and waiting to execute on a CPUs core. This is generally storedas a linkedlist.
* Processes that are waiting for a certain event to occur such as completion of IO, are placed in a wait queue.
* A new process is initially put in ready queue, waiting there till it is selected for execution (dispatched). Till then, the following could occur:
* The process could issue an I/O request and then be placed in an I/O wait
* queue.
* • The process could create a new child process and then be placed in a wait
* queue while it awaits the child’s termination.
* • The process could be removed forcibly from the core, as a result of an
* interrupt or having its time slice expire, and be put back in the ready queue.
* 
* CPU Scheduler - The role of the CPU scheduler is to select from among the processes that are in the ready queue and allocate a CPU core to one of them. The CPU scheduler must select a new process for the CPU frequently. An I/O-bound process may execute for only a few milliseconds before waiting for an I/O request
* Some OS have a form of scheduling that is called SWAPPING. This makes the CPU think that removing a process from memory might be advantageous and speed the system up.
* Context switch – A context switch allows system to save the process running on the COU core so that it can process something, and allow it to retain to the same state after a process is done computing. It has a state save and a state restore. Context is saved in PCB.

# 3.3 Operations on Processes

A process may create several new process, such as children processes of the main parent process. Many new parents/childs can come, so it can be called a TREE of processes.

PID – Processes are identified with process identifier (unique) integer number. Used to index to access attrivutes of a process within a kernel

* Ps –el command – Lists complete info for all processes active in the system.
* In general, when a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent’s resources prevents any process from overloading the system by creating too many child processes.

- When a process creates a new process, two possibilities for execution exist:

1. The parent continues to execute concurrently with its children.

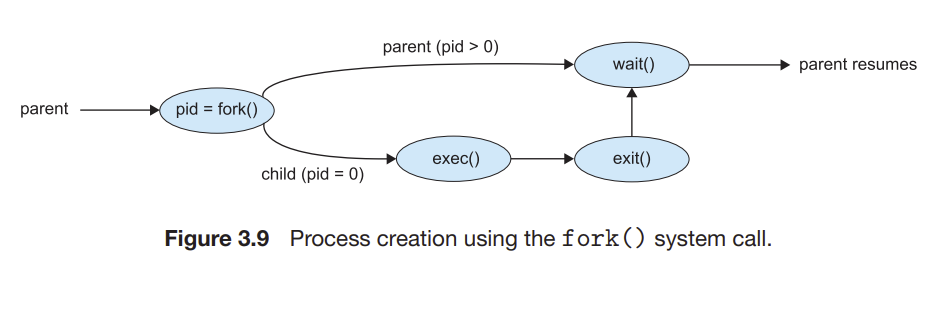
2. The parent waits until some or all of its children have terminated.

- There are also two address-space possibilities for the new process:

1. The child process is a duplicate of the parent process (it has the same

program and data as the parent).

2. The child process has a new program loaded into it.

* New processes are created by the fork() sys call.
* Both processes (parent and child) continue execution at the instruction after the fork(), with one difference. The return code for the fork() is zero for the new child process. Whereas the nonzero PID of the child is returned to the parent.
* After fork(), one of the two processes use exec() syscall to replace memory with a new program.
* 

Process Termination

* A process terminated when it finishes executing its final statement and ask the OS to delete it by using the exit() sys call.
* At that point, the process may return a status value (typically an integer) to its waiting parent process (via the wait() system call). All the resources of the process —including physical and virtual memory, open files, and I/O buffers—are deallocated and reclaimed by the operating system.

A parent may terminate the execution of one of its children for a variety of

reasons, such as these:

• The child has exceeded its usage of some of the resources that it has been

allocated. (To determine whether this has occurred, the parent must have

a mechanism to inspect the state of its children.)

• The task assigned to the child is no longer required.

• The parent is exiting, and the operating system does not allow a child to

continue if its parent terminates.

Cascading Termination – when some systems don’t allow a child to exist if its parent has been terminates, the process terminates the childs in the process called “Cascading Termination”

Zombie Process - A process that has terminated, but whose parent has not yet called wait(), is known as a zombie process

Orphan Process - Now consider what would happen if a parent did not invoke wait() and instead terminated, thereby leaving its child processes as orphans

# 3.4 Interprocess Communication

A process is independent if it does not share data with any other processes executing in the system. A process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

• Information sharing. Since several applications may be interested in the

same piece of information (for instance, copying and pasting), we must

provide an environment to allow concurrent access to such information.

• Computation speedup. If we want a particular task to run faster, we must

break it into subtasks, each of which will be executing in parallel with the

others. Notice that such a speedup can be achieved only if the computer

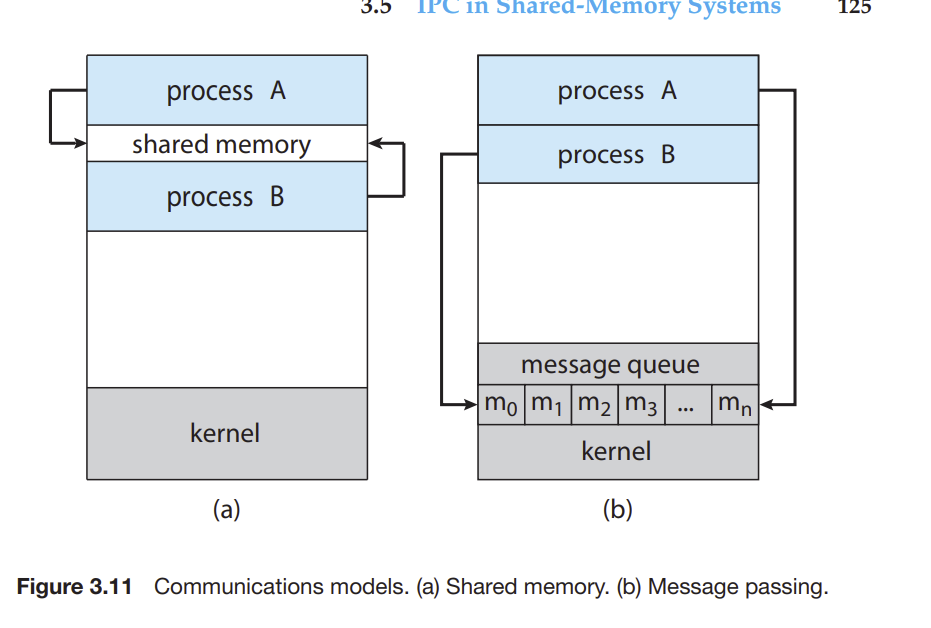
has multiple processing cores.

• Modularity. We may want to construct the system in a modular fashion,

dividing the system functions into separate processes or threads, as we

discussed in Chapter 2.

Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data— that is, send data to and receive data from each other. There are two fundamental models of interprocess communication: shared memory and message passing.

* In the shared-memory model, a region of memory that is shared by the cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.
* In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Figure 3.11.
* 
* Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement in a distributed system than shared memory.
* Shared memory can be faster than message passing, since message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention. In shared-memory systems, system calls are required only to establish sharedmemory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

# 3.5 IPC in Shared-Memory Systems

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space

* Recall that, normally, the operating system tries to prevent one process from accessing another process’s memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by these processes and are not under the operating system’s control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously

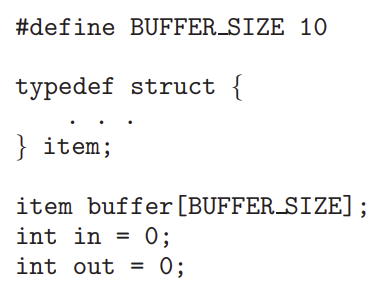
To illustrate the concept of cooperating processes, let’s consider the producer–consumer problem, which is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process.

## Solution to Project 1 Part 1

One solution to the producer–consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Two types of buffers can be used. The unbounded buffer places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The bounded buffer assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

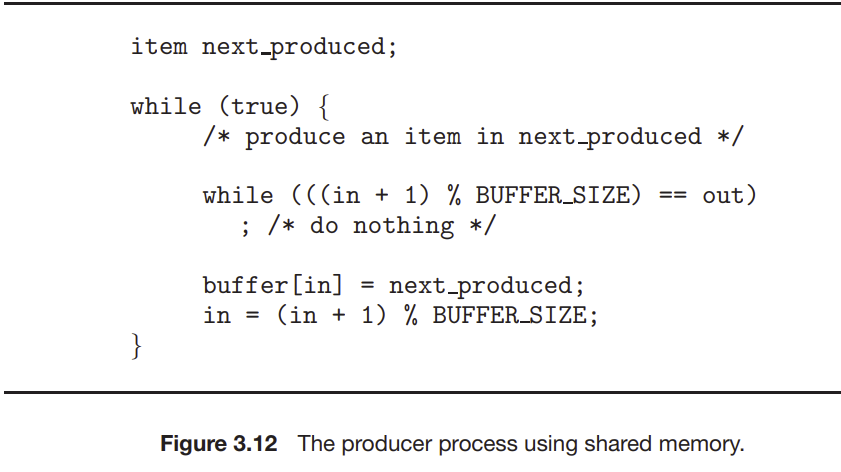
Let’s look more closely at how the bounded buffer illustrates interprocess communication using shared memory. The following variables reside in a region of memory shared by the producer and consumer processes:

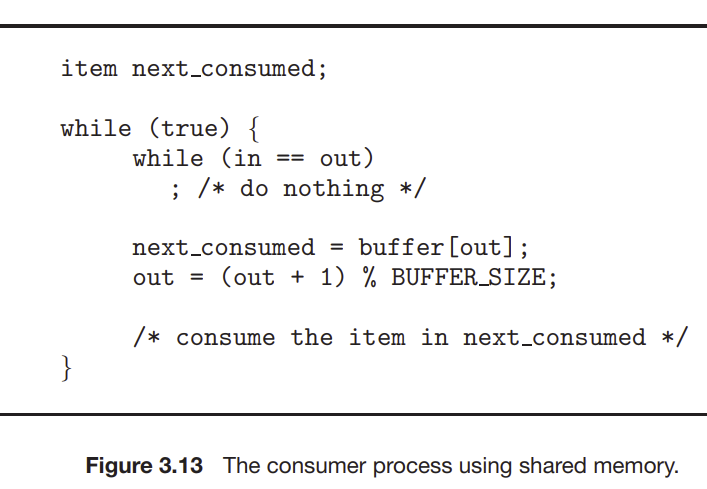


The shared buffer is implemented as a circular array with two logical pointers: in and out. The variable in points to the next free position in the buffer; out points to the first full position in the buffer. The buffer is empty when in == out; the buffer is full when ((in + 1) % BUFFER SIZE) == out.

The code for the producer process is shown in Figure 3.12, and the code for the consumer process is shown in Figure 3.13. The producer process has a local variable next produced in which the new item to be produced is stored. The consumer process has a local variable next consumed in which the item to be consumed is stored.

This scheme allows at most BUFFER SIZE − 1 items in the buffer at the same time. We leave it as an exercise for you to provide a solution in which BUFFER SIZE items can be in the buffer at the same time. In Section 3.7.1, we illustrate the POSIX API for shared memory





# 3.6 IPC in Message-Passing Systems

- In Section 3.5, we showed how cooperating processes can communicate in a shared-memory environment. The scheme requires that these processes share a region of memory and that the code for accessing and manipulating the shared memory be written explicitly by the application programmer. Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via a message-passing facility.

* Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.
* Messages sent by a process can be either fixed or variable in size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult
* If processes P and Q want to communicate, they must send messages to and receive messages from each other: a communication link must exist between them.
* Here are several methods for logically implementing a link
* and the send()/receive() operations:
* • Direct or indirect communication
* • Synchronous or asynchronous communication
* • Automatic or explicit buffering

## Naming Issues for Processes

* Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication
* direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In thisscheme, the send() and receive() primitives are defined as:
* • send(P, message)—Send a message to process P.
* • receive(Q, message)—Receive a message from process Q

In direct communication, a link is established automatically, only needing to know each others identity to communicate. A link is associated with exactly TWO processes, and between them there is ONLY one link.

Another scheme of the above is called assymetric connection in which :

only the sender names

the recipient; the recipient is not required to name the sender. In this scheme,

* the send() and receive() primitives are defined as follows:

• send(P, message)—Send a message to process P.

• receive(id, message)—Receive a message from any process. The variable id is set to the name of the process with which communication has taken place.

* The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions

Indirect Communcation, messages are sent to and received from mailboxes, or ports. Mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messaged can be removed.

* Mailboxes have unique IDs. POSIX msg queues use INT values to identify a mailbox. Process can communicate with another process via a number of different mailboxes, BUT two process can only communicate if they share a mailbox.
* Send and receive calls are defined as follows for INDIRECT communication.

• send(A, message)—Send a message to mailbox A.

• receive(A, message)—Receive a message from mailbox A.

In this scheme, a communication link has the following properties:

• A link is established between a pair of processes only if both members of

the pair have a shared mailbox.

• A link may be associated with more than two processes.

• Between each pair of communicating processes, a number of different links

may exist, with each link corresponding to one mailbox.

**With indirect, we have a choice when 3 processes share a mailbox. It depends on what we wish to choose.**

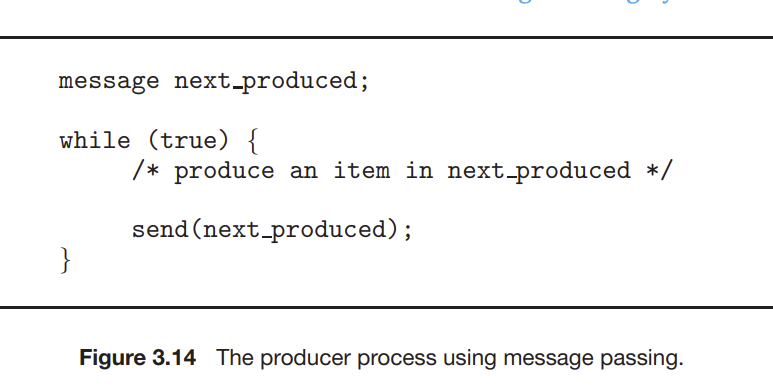
• Allow a link to be associated with two processes at most.

• Allow at most one process at a time to execute a receive() operation.

• Allow the system to select arbitrarily which process will receive the message (that is, either P2 or P3, but not both, will receive the message). The system may define an algorithm for selecting which process will receive the message (for example, round robin, where processes take turns receiving messages). The system may identify the receiver to the sender.

* Round Robin – Processes take turns for receiving messages.
* A mailbox may be owned either by a process or by the operating system
* If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (which can only receive messages through this mailbox) and the user (which can only send messages to the mailbox)
* When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.

IMPORTANT

* The operating system then must provide a mechanism that allows a process to do the following: • Create a new mailbox. • Send and receive messages through the mailbox. • Delete a mailbox
* Synchronization - Message passing may be either blocking or nonblocking— also known as synchronous and asynchronous.
* • Blocking send. The sending process is blocked until the message is
* received by the receiving process or by the mailbox.
* • Nonblocking send. The sending process sends the message and resumes
* operation.
* • Blocking receive. The receiver blocks until a message is available.
* • Nonblocking receive. The receiver retrieves either a valid message or a
* null.
* 

Buffering

Messages reside in a temporary queue and can be implemented in three ways:

1. Zero Capacity. No messages can be waiting, queue length is 0
2. Bounded Capacity – finite length of the queue, so only certain number of msgs can reside in it.
3. Unbounded Capacity – unltd messages can wait in it, sender never blocks.

# 3.7 Examples of IPC Systems

POSIX Shared Memory

* POSIX shared memory is organized using memory-mapped files, which associate the region of shared memory with a file. A process must first create a shared-memory object using the shm open() system call, as follows:
* 
* The first parameter specifies the name of the shared-memory object. Processes that wish to access this shared memory must refer to the object by this name. The subsequent parameters specify that the shared-memory object is to be created if it does not yet exist (O CREAT) and that the object is open for reading and writing (O RDWR). The last parameter establishes the file-access permissions of the shared-memory object. A successful call to shm open() returns an integer file descriptor for the shared-memory object.
* Once the object is established, the ftruncate() function is used to configure the size of the object in bytes. The call ftruncate(fd, 4096); sets the size of the object to 4,096 bytes
* Finally, the mmap() function establishes a memory-mapped file containing the shared-memory object. It also returns a pointer to the memory-mapped file that is used for accessing the shared-memory object

The programs shown in Figure 3.16 and Figure 3.17 use the producer– consumer model in implementing shared memory. The producer establishes a shared-memory object and writes to shared memory, and the consumer reads from shared memory.

Page 172 for Notes for project

3.7.4 Pipes

A pipe acts as a conduit allowing two processes to communicate. Pipes were one of the first IPC mechanisms in early UNIX systems. They typically provide one of the simpler ways for processes to communicate with one another, although they also have some limitations. In implementing a pipe, four issues must be considered:

1. Does the pipe allow bidirectional communication, or is communication

unidirectional?

2. If two-way communication is allowed, is it half duplex (data can travel

only one way at a time) or full duplex (data can travel in both directions

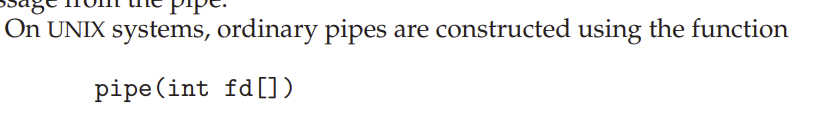
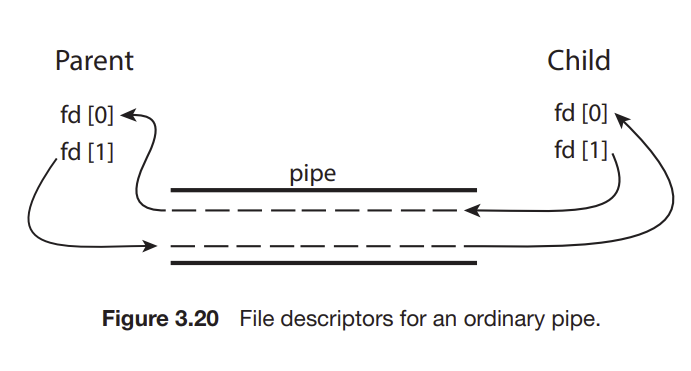
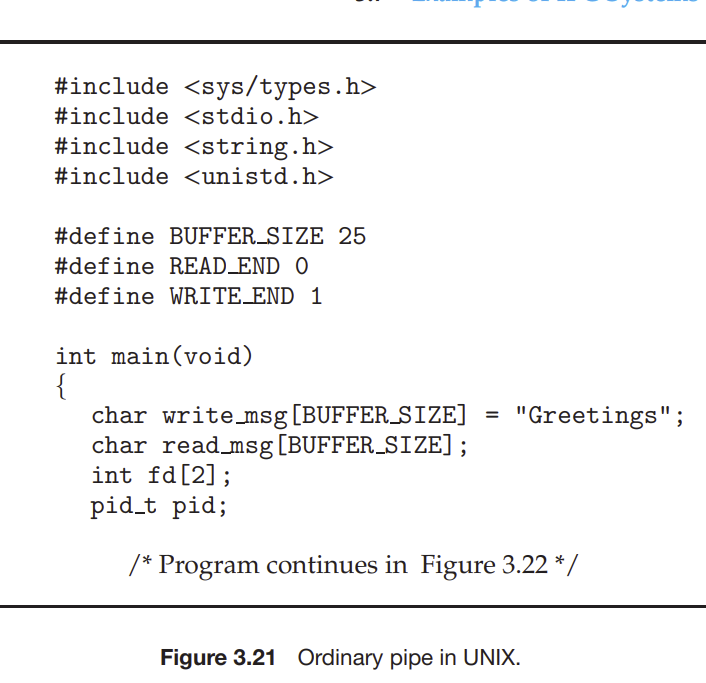
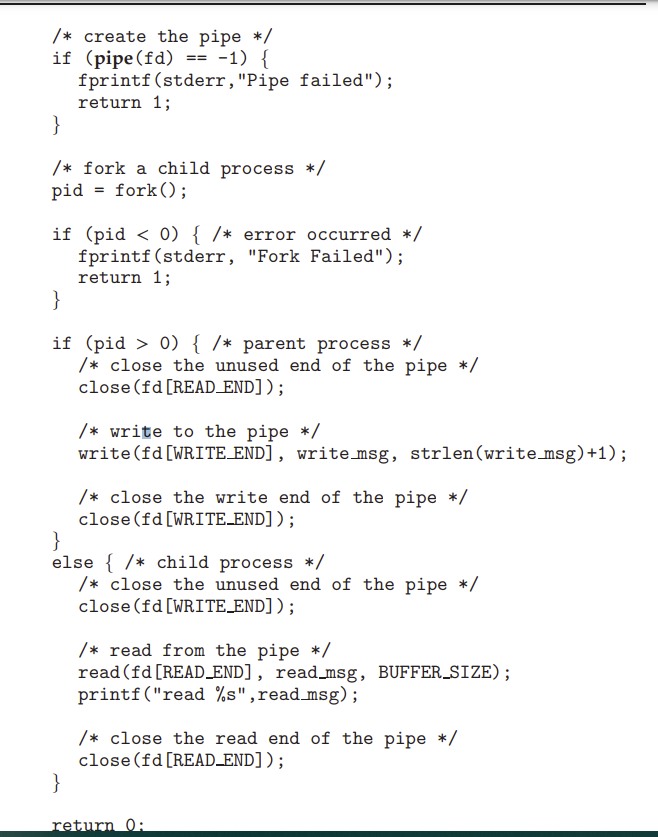
at the same time)?

3. Must a relationship (such as parent–child) exist between the communicating processes?

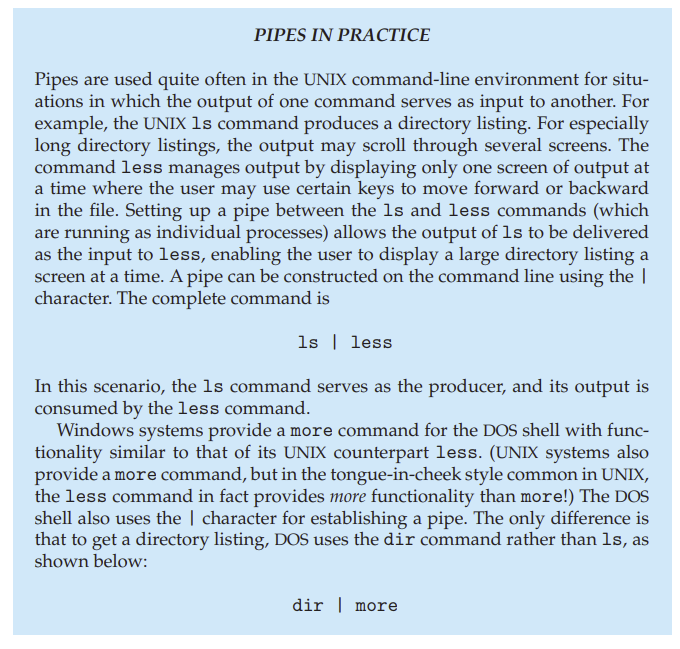
4. Can the pipes communicate over a network, or must the communicating

processes reside on the same machine?

Ordinary Pipes

* Ordinary pipes allow two processes to communicate in standard producer– consumer fashion: the producer writes to one end of the pipe (the write end) and the consumer reads from the other end (the read end). As a result, ordinary pipes are unidirectional, allowing only one-way communication
* 
* This function creates a pipe that is accessed through the int fd[] file descriptors: fd[0] is the read end of the pipe, and fd[1] is the write end. UNIX treats a pipe as a special type of file. Thus, pipes can be accessed using ordinary read() and write() system calls.
* 
* 
* Ordinary pipes on Windows systems are termed anonymous pipes, and they behave similarly to their UNIX counterparts: they are unidirectional and employ parent–child relationships between the communicating processes
* In addition, reading and writing to the pipe can be accomplished with the ordinary ReadFile() and WriteFile() functions.­
* 

Named Pipes

* Ordinary pipes provide a simple mechanism for allowing a pair of processes to communicate. However, ordinary pipes exist only while the processes are communicating with one another. On both UNIX and Windows systems, once the processes have finished communicating and have terminated, the ordinary pipe ceases to exist.
* Named pipes provide a much more powerful communication tool. Communication can be bidirectional, and no parent–child relationship is required. Once a named pipe is established, several processes can use it for communication
* Page 180 of the book
* 

# 3.9 Summary

• A process is a program in execution, and the status of the current activity of

a process is represented by the program counter, as well as other registers.

• The layout of a process in memory is represented by four different sections:

(1) text, (2) data, (3) heap, and (4) stack.

• As a process executes, it changes state. There are four general states of a

process: (1) ready, (2) running, (3) waiting, and (4) terminated.

• A process control block (PCB) is the kernel data structure that represents a

process in an operating system.

• The role of the process scheduler is to select an available process to run on

a CPU.

• An operating system performs a context switch when it switches from

running one process to running another.

154 Chapter 3 Processes

• The fork() and CreateProcess() system calls are used to create processes on UNIX and Windows systems, respectively.

• When shared memory is used for communication between processes, two

(or more) processes share the same region of memory. POSIX provides an

API for shared memory.

• Two processes may communicate by exchanging messages with one

another using message passing. The Mach operating system uses message

passing as its primary form of interprocess communication. Windows

provides a form of message passing as well.

• A pipe provides a conduit for two processes to communicate. There are

two forms of pipes, ordinary and named. Ordinary pipes are designed for

communication between processes that have a parent–child relationship.

Named pipes are more general and allow several processes to communicate.

• UNIX systems provide ordinary pipes through the pipe() system call.

Ordinary pipes have a read end and a write end. A parent process can, for

example, send data to the pipe using its write end, and the child process

can read it from its read end. Named pipes in UNIX are termed FIFOs.

• Windows systems also provide two forms of pipes—anonymous and

named pipes. Anonymous pipes are similar to UNIX ordinary pipes. They

are unidirectional and employ parent–child relationships between the

communicating processes. Named pipes offer a richer form of interprocess

communication than the UNIX counterpart, FIFOs.

• Two common forms of client–server communication are sockets and

remote procedure calls (RPCs). Sockets allow two processes on different

machines to communicate over a network. RPCs abstract the concept of

function (procedure) calls in such a way that a function can be invoked on

another process that may reside on a separate computer.

• The Android operating system uses RPCs as a form of interprocess communication using its binder framework

3/21 Notes

Todays Topics

* Midterm on Monday April 4th (Mult choice, Multiple Answer, Short Answer. All material up to March 30
* Quiz 2 Grades are up
* Project Part 2 – Instructions on CUNY BB

Todays Stuff

* Quiz 2 Answers
* Thread Pools and Local Storage
* M-T Fork(), exec(), signal handling
* Thread-safe APIs
* Other Design Patterns
* HW Chap 4 Solutions: 6, 13
* Andahls Law
* M-T Models

Thread Pools and Local Storage

* What do we want when having thread pools?
* Basically, we just want the function to execute (request processing)
* Create the thread for the pool as messages come in

For(int I =0; I < pool\_size;i++)

Pthread\_init();

Pthread\_create(ThdPoolFunc);

* We want another function after threads are created:

ThdPoolFunc() {

Pthread\_setspecific (key, Ptr);

While (true) {

Msg = recv(SYNC) // receive from message queue, done in sync value

\*fptr w/ data

}}

Ptr = pthread\_getspecific(key) // If you want a specific, relevant memory block

**Thread Local Storage**

When message comes in, we will run either one of the 4 functions: intert, delete, update, retrieve

Each of these functions will need its own working space that might not be allocated in the workspace itself.

* Thereforre, we will give each thread some blocks of data dynamically to give it space to do functions. We do DYNAMIC MEMORY ALLOCATION for the Threads. Each thread in the pool gets a pointer to its own dynamic Memory allocation, whether or not they do any of the fucnntions given to them.

Metric/Data on stuff in the queue – what do they include?

* Xfptr
* Data
* Time stamp Network listener
* Pool thread info, including timestamp, taken from msg queue
* Timestamp when the pool thread finished request processing

How to know pool size?

* Use a “config.ii” file. Pool size =100;
* Restart the DBMS and it will read the new change in the field

Thread Pools – Create a number of threads in a pool where they await work

* Advantages

Slightly faster tp service a request than creating new thread

Allows # of threads in the applocations to be bound to the size of the pool

Separating task to be performed from mechanics of creating task allows different starts for running tasks. – tasks could be scjeduled to run periodically.

Thread-local Storage (TLS) – Allows each tjread to have its own copy of data.

* Useful when you don’t have control over the thread creation process (when using a thread pool)
* Different from local variables – local variables visible only during single fnction invocation. TLS visible across function invocation
* Similar to static data – TLS is unique to each thread.

Cooperating vs Indeoendent Threads

* There is a ix of cooperating and hybrid threads (just like hybrid parititioning of data vs Tasks)
* Cooperating -> Shared Variables -> Global variables (blocks of dynamic memory
* Message Queue (we are sharing info with this)
* All of this is part of INTERTHREAD COMMUNICATION
* Orderly Shutdown of threads/ Orderly Startup - this is called Thread Synchronization.
* There is synchronization in the threads process/thread synchronization
* Next project will have Pthreads in it. Check pthread file on the notes and Multithreading.
* Pthread Examples

We will put a for loop on the attr\_init and pthread create FIRST. Then separate for loop for PTHREAD join.

* Important to do 2 loops as every time you create a thread, you will be blocked by pthreadJoin, and only 1 pthread will be running at the same time.
* 2 FOR LOOPS

Sfemantics of Fork() and exec()

* Does fork duplicate only the calling thread or all threads? Some unixes have two versions of fork
* Exec() usually works as normal – replace the running process including all threads.

Signal Handling Semantics

Slide 4.15 – WILL BE ON EXAM.

Thread Safe APIs

* Instead of write, we use pread
* Instead of write, we use pwrite.
* Strtok() not good.

Producer Consumer model – Producer is thread, consumer is thread.

You can also make a chain of producer/consumers.

First thread is the producer, 2nd thread plays 2 roles, producer for thread 3, consumer of thread 1. Thread 3 is theconsumer.

This is called task partitioning. Where initially code was 2 parts, you break it into 3 parts, with three algos that cover the models.

Benefit – after first request omes in, thread 1 does part of the work, the remaining parts of work move on to the other thread. This is called pipeline, a chain of producers and consumers.

Sh mem using API calls

Shm.open, ftruncate, nmap

* Do this for IN and OUT variables.
* Make these 3 calls again – use between producer and consumer. Make these calls before getting to fork. – if you don’t do this they wont share anything. Keeps track of where the producer and consumers are at.
* Then convert into C code (nmap call gives a ptr)
* Know when producer consumer are done. When it hits end of file, end the process (see the while true loop, break out of it). When end of file, break out
* How does producer indivate to consumer when it hits end of file. What can you put in there so that consumer can see end of file. Create a third shared mem variable that tells you eof indivator.
* After
* Cygwin gcc compile or Venus
* Pipe – counter , use producer

3/23/22

Today – Chap 4

Wait sys call – BLOCKS or POLLING approach where you call wait\_pid() or ASYNCH approach with linux. True multithread is where each thread does its own work, concurrency and parallelism.

Todays Stuff

* Amdahls Law
* Multithreading Models (thread scheduling)
* Ch4 HW Solutions
* CH5 – CPU Process Scheduling
* 1) Process Characteristics
* 2) Scheduling Objectives
* 3) Types of Scheduling ALgos
* 4) Process State Diagram

Amdahls Law

* Identifies Performance Gains from adding addtl cores to an application that has both serial and parralell components.
* S is serial portion, N processing cores. Formula in book
* Serial components – Main thread summation.
* Worker threads = parallel components (they can do work independently, while working in parallel)
* If app is 75% parallel/25% serial, moving from 1 to 2 cores results buildup of 1.6 times.
* As N approaches infinity, speedup approaches 1/S, where S = serial portion
* Moral – in real world, if a program is slow look for improvement in algorithm
* Have to know it for the exam.

Multithread Models

One to one model, many to one model

User and Kernel level Thread

* User threads – think in the program, where you have main and posix p threads. The sequence of instructions appearing in source code.
* Kernel threads – actually called kernel scheduled entities.
* CPU Process Scheduler – takes kernel scheduled entities and processes them to the CPU cores.

Kernels are multithreaded. User threads are also called kernel threads, and will get kernel scheduled entities.

We get parallelism by having a core each for each kernel thread. If the #’s are not equal to each other, it would be limited parallelism.

Many to Many Model

* Allows many user level threads to be mappened to many kernel threads.
* Software is more complicated however
* Works in Windows ThreadFiber package.
* Multiplexing

Scheduler Activiations

* Both MM and 2level models require communication to maintain appropriate communication of kernel threads
* - Uses an intermediate data structure – Lightweight Thread Processesor

One-to one is the modern approach.

Third approach – older style Many to One

* Many user threads mapped to single kernel thread
* One thread blocking causes all to block. Multiple threads may not run in parallel on multore system because one one may be in kernel at a time.

Chapter 5

CPU Process Scheduling

Basic Concepts

* Max CPU util is obtained with multiprogramming
* CPU/IO burst cycle – Process execution consists of a cycle of CPU execution and IO wait
* Non-premeptive and Preimptive Algorithms. Easy algos use non-preiptive, difficult algos use Preemptive (more popular and sophisticated to maintain)
* CPU Burst followed by IO burst
* CPU burst distributuion is of MAIN concern.

Processes are characterized into 1 of two types:

1. CPU bound Processes

* Program runs from time =0 to time = (certain end time)
* We will have rather long cpu burst and occasionally in between some sort of IO burst.

1. IO Bound processes

* We see smaller CPU burst but more bursts segmented throughout. Why is this? It is b/c there are multiple inputs of IO throughout IO processes, whereas in CPU bound you are not taking in much data, rather you are crunching on data in one single time and not taking IO at that time.

Scheduling Criteria

* CPU Utilization – keep PCU as busy as possible
* Throughput – amount of time for process to go in and exit
* Turnaround time – amount of time to execute a particular process
* Waiting Time – amount of time a process has been waiting in ready queue.
* Response Time – Amount of time it takes when a request was submitted till the first response is produced, not its output.

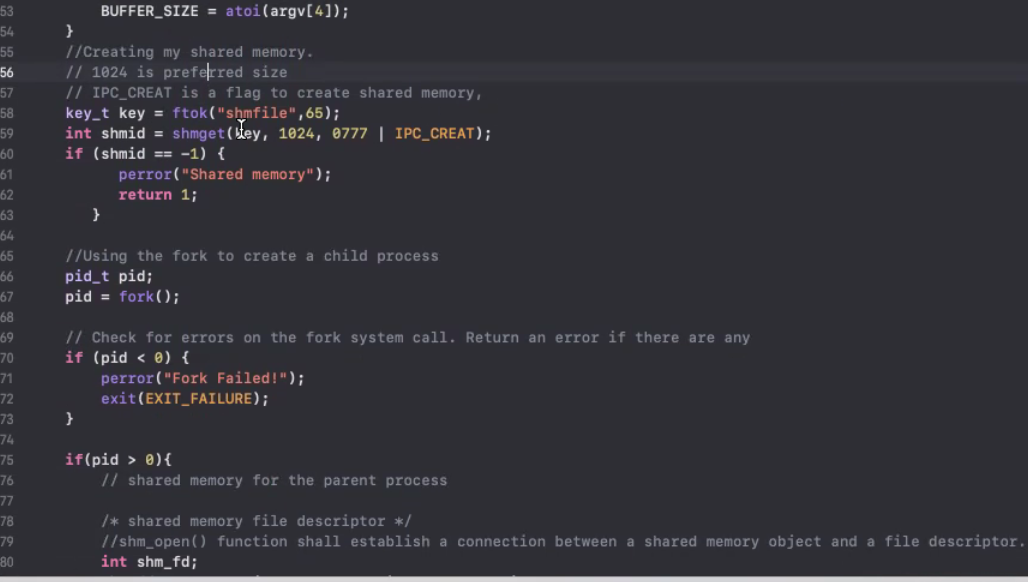
Scheduling is mainly broken down into the first two parts above. Waiting time is what we’ll focus on the most in these chapters

CPU Scheduler

* We will revisit this later

Diagram of Process State

* Memorize this diagram.



Design and implement 40pts

Execution and output 15pts

Inline code comments 15pts

Adherence to instructions 15 pts

Writeup 15pts

TOTAL – 100pts

Design and implementation Issues – if you didn’t implement the algos, you lost a lot of points.

Project Details;

Multithread – 2 threads are running ONLY then scheduler takes over. Lousy thread & 1 other thread

Implement FIFOs for ready queue logger and scheduler dispatcher.

Agenda –

Project 2 due next week

Quiz 3 on Wednesday

Locks

* We need locks for simple amounts of data. So add a lock inside your programs.
* Before you make changes add a lock so that your program will run first. Then remember to unlock. (Same thing in project (lock mutex and unlock mutex)
* Look at underlying algorithms – lock)mutex (pthread)
* Make sure that one thread at a time gets the lock. Peterson Solution handles this, remember to say that.

Questions on Quiz

* Chapter 5

Prod Consumer – has in, out, and counter variables.

The turn variable

* This variable operates as a locking strategy. It is a synchronization algorithm (integer) for syncing. The limitation is that the threads will go in alternating fashion.
* Flagging the threads – this is also important.
* In project 2 – if you go into fifo for logger thread or scheduler dispatcher, you need to have implemented a flag variable that acts like a guard before getting into the thread.
* It is called Boolean basic algorithm that will fail.
* Another one is called Boolean modified that will make code changes to the algorithm m

This all becomes into - Petersons Solution – We will still have the two flag variables, but we will put back the int turn variable back inside so that the function will work fine.

Algorithmcly this works fine, but when programmed it still may fail. But we cant rely on it on runtime.

4th - Boolean enhanced Algorithm – has the two flag variables and we will solve the deadlock issue now (that both are trying to grab the lock). We introduce a randomness value using sleep(). They will sleep for different amount of times and that will cause them to break the deadlock.

Second - Bool Basic Algo (use flag [1] and flag[2]) It fails due to mutual Exclusions

Next – Use bool modified algorithm (will not fail for mutual exclusion, but fail due to deadlock – both processes are waiting for the other to do something, wont go into the code until the other guy to do something and viceversa)

First Part – My turn your turn algo

Myturn Yourturn Algo

Main() {

Int turn;//declare this global variable first

threadOne() {

while (true) { //loop that will go thru the shared and private data alternating it)

//entry section (get the lock) while (turn ==2);

// C.S. Shared Data

// Exit section (release the lock) turn =2;

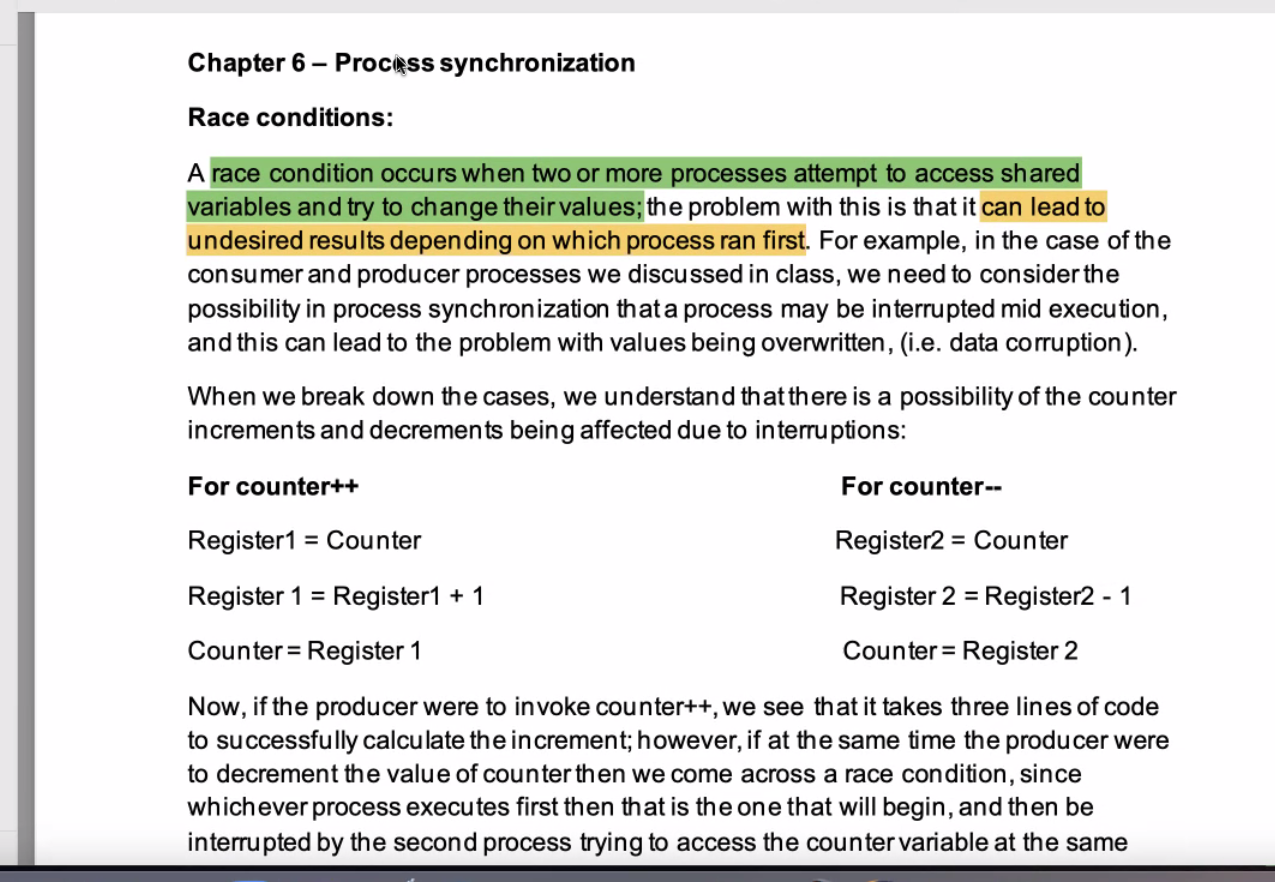
// Reminader Section – accessing my provate data

threadTwo() {

while (turn ==1); //

//Critical Section

Chap 6



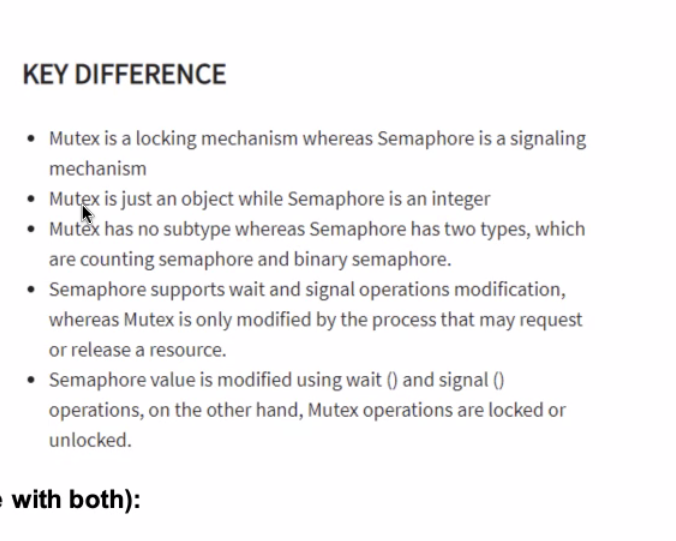
* Race condition gives us undesiered results
* Shared memory the variables are affected if the two processes call the vairbale a the same time. Give incorrect values to both processes. can cause issues

Crit Section

* Tries to solve the issue that shared memory can be used at the same time, only 1 process can use the shared memory at the time. Will call issues in shared mem.
* Critical section is shared (prod cons algorithm)
* 1st condition – only 1 process is allowed at a time (only1 can use at a time) DEADLOCK
* 2nd – Progression – Process starvation
* 3rd – Must be bounded waiting. There must be num of times/limitation that other processes are allowed to enter critical section.

Peterson’s – a solution but it does not guarantee the optimal solution for mutual exclusion.

Mutex locks –When a process accesses the crticial section, it locks it so that no other resources can utilize it when used, in exit section the lock is released. ISSUE – if preempted in between (you get called midway between running state). If preempted two processes will be in the shared mem at the same time.



Semaphore – sync tool, uses wwait() and signal(). Allows you to sync.

Boolean question

Spinlock – check code samples of different one (short duration, long duration)

Semahphores

Temporal Localaity – storage locations referenced recently are likely to be referenced in the near future

Code examples

* Looping
* Methods functions etc
* LIFO structs
* Variables for couting and summing

Spacial Locality – once a storage location is referenced, it is highly likely that nearby loctions will be referenced.

Code examples

* Array accesses
* Sequential code
* Related variables defined near each other.

Swapping