OS

CSCI 3573

[Syllabi](#h.7naaenavo0k5)

[Operating Systems](#h.1zd9ntsqv8vy)

[Historical Timeline](#h.af3gyth07cy1)

[Trends](#h.ejbdxs9sfyx1)

[What does an OS do?](#h.j02nqa32zlbs)

[ABI](#h.3njlftm1h88y)

[API](#h.9kxjuoewzjey)

[Protection](#h.2n97l4oqn71z)

[Memory Protection via Virtual Memory](#h.e77fmx1et8le)

[Interfaces](#h.35t2bspeujhm)

[Multiprocessing](#h.7m65nzl5uzxn)

[Device Failure](#h.kf21zeqkxea8)

[Asymmetric Multiprocessing](#h.cmpfxyyakn1d)

[Symmetric Multiprocessing](#h.8hra9hn2z6c2)

[Memory Access](#h.kgj58eh1bil9)

[Clustering](#h.sd8el25lqyj0)

[Asymmetric Clustering](#h.pqmgdgqqfua6)

[Symmetric Clustering](#h.qa91ccwlquyy)

[DLM](#h.3iifxvynmhs9)

[SAN](#h.mclig0hwfstn)

[Multiprogramming](#h.3mjrv493muw8)

[Traps in OS](#h.26wi5k2n4oyk)

[Kernel Mode](#h.de2l4z8tbzoe)

[Rings of Privilege](#h.9q1ebiuusfui)

[Area Networks](#h.gp9xwkti6f3v)

[aaS](#h.7fdfbjx8ukfj)

[Real Time OS](#h.tjkup3ogra2s)

[Operating system functions](#h.5cj0b8kx8kll)

[System Calls](#h.r1d3pi2jgcfk)

[Unix](#h.oni3uqo7725i)

[System call categories](#h.e2zdnu8eprqn)

[Parameters](#h.tnx4z5wi2ct2)

[Roll Your Own System Calls](#h.hu7mf1llcdbe)

[Pre-Configuration](#h.ci4ypcpvgead)

[Build Kernel](#h.67zqb7o72c24)

[Add a system call](#h.otog02m0zeal)

[Example C file](#h.hs432tno8b7c)

[Test program](#h.jfaor03kkt4f)

[Unsorted Vocab](#h.88080yfr392r)

[System programs](#h.x6s2k179vngb)

[Application programs](#h.go55nfi5gcap)

[Devices](#h.xoyh512xw45m)

[Turing/Von Machines](#h.cc21hn5csy0a)

[Turing machine](#h.5gifhxg8krfg)

[Von Neumann](#h.ggxerfoh7)

[Vocab](#h.pmeyupppskif)

[Interface differences](#h.d6b4bvn3at0h)

[Port/Mem Mapped](#h.wzn5tfw4rltj)

[Error handling](#h.w9na0e5lk31n)

[Streams](#h.pa5qgq3o4ivo)

[DMA](#h.2wazoqm4nyb0)

[DMA Modes](#h.la3n67ulqy3u)

[OS Design](#h.dx70ug94uaiu)

[How do you design an OS?](#h.n6v8tvkpg7ds)

[Structures](#h.9yzh6buoyrw3)

[Mac OS X](#h.51sr6n2rkng8)

[Processing](#h.yr86w1aqjqg9)

[Boot Process](#h.aiezldvwq02c)

[Multitasking vs. Multiprogramming](#h.d64hawu44vrw)

[Multitasking types](#h.odaq7w357o41)

[Time sharing](#h.lkzociaombkd)

[Process States](#h.g149r5lklq1x)

[Program Control Block (PCB)](#h.tckulgq27wcj)

[Scheduling](#h.ytjcf0a5wspj)

[Long-term Scheduler](#h.qlgdm1q6bcah)

[Degree of Multiprogramming](#h.f4u07mm1t6e)

[I/O-bound vs. CPU-bound](#h.t2bm31ysnlin)

[Short-term Scheduler](#h.mfgvvwk2ptj2)

[Medium-term Scheduler](#h.c392o4298c8y)

[Process Creation](#h.2wl370yxmbwg)

[Copy-On-Write](#h.1o14escz31k2)

[Unix](#h.9c2aonhowtlk)

[Zombie Process](#h.htk9vmbyw)

[Orphaned process](#h.jloqvfw5ybpg)

[Process handling](#h.ifra6rfvw7vp)

[Threads](#h.nyx7ni9vcv2b)

[Advantages](#h.jw8pp4fu3vh5)

[Benefits of multithreading](#h.pyxhyqmkevk)

[Disadvantages](#h.2pn6ezr126wj)

[Thread-safe](#h.j5zwxl3y08tr)

[Reentrant](#h.bkwp99cgt23p)

[Example](#h.loebxsolunyh)

[User vs. Kernel Threads](#h.5ilib68se65t)

[Implicit threading](#h.z79xpi27a2xj)

[Methods](#h.hqhfwjc47gxb)

[Issues](#h.6ig6um5p8021)

[Multicore](#h.d5xkq6mv8qo5)

[Challenges for multicore systems](#h.g785kg976jcd)

[Misc. Vocab](#h.7371i98lxi3u)

[IPC](#h.8qdtxnjy4y58)

[Shared memory](#h.kzs1rqdyexsg)

[Usage](#h.w0eebj2q37fo)

[Sockets (message passing)](#h.tp08dflhde0s)

[Pipes](#h.k7ss0ox7jq4c)

[Named pipes](#h.1ucxewz8o19a)

[Exceptions](#h.9jv8jeyfer11)

[Signals](#h.e395jn16qr1y)

[\*Nix signals](#h.rfg917l7v3ui)

[Problems](#h.exh5l1tv8ru8)

[Race conditions](#h.qdcosxih9g3c)

[Critical section](#h.gekpsm46oz5a)

[Critical Section Requirements](#h.ecurw2lf6o7h)

[Peterson’s solution](#h.c69ll86jx9nx)

[Mutex locks](#h.17nhv5atbf0m)

[Advantages:](#h.f7zt3i6utgos)

[Disadvantage](#h.tnmmkvi2eyoo)

[Test-and-Set](#h.koxpng1hvp3q)

[Example](#h.xlydpsrofnrd)

[Advantage](#h.tsf4rqr4ics0)

[Disadvantages](#h.qn5323neeof8)

[Semaphores](#h.rguc2i5dbl3k)

[Basic Example](#h.qo64qc4uzeeb)

[Enforcing Order Example](#h.4g28p3gu0r11)

[Monitors](#h.fq2ac8lgtikf)

[Monitor ADT](#h.lw685kuyvhwl)

[Conditionals](#h.ml689yrae9un)

[Example](#h.ix1euxvec5v)

[Issues with Monitors](#h.jipi3zizzyly)

[Problems](#h.y5x4dse9nq0m)

[Situations](#h.y0cw64ixxg2x)

[Bounded Buffer Problem](#h.70sq6nrpn0jk)

[Readers-Writers Problem](#h.k3nk8kpcskss)

[Dining-Philosophers Problem](#h.imr2j5bilqdy)

[Scheduling](#h.xmjbzjce3nu4)

[Process switching](#h.qgyfb7kckbxe)

[Context Switch](#h.7hvsq55pv59j)

[Misc. Scheduling Vocab](#h.w84c23vnqb2p)

[Scheduler’s Responsibilities](#h.lqqhi39d7dba)

[Scheduler’s goals](#h.hmj9cke7rfp)

[Analysis](#h.7n0e5elywjl3)

[Scheduling Algorithms](#h.wf5mtx251h85)

[FCFS](#h.sgdxn98mslg9)

[SJF](#h.xv1n5w20m601)

[Round Robin](#h.mng0i86g168u)

[Weighted Round Robin](#h.asb8xaxr0722)

[EDF](#h.jfrva315hrei)

[Priority scheduling](#h.x7qjatfi6drh)

[Multilevel queue scheduling](#h.f7chpj1xgthc)

[Scheduling In Linux](#h.2tf4nge8r1a5)

[CFS](#h.pe50ir5iod1j)

[Real Time](#h.lm5or143b77j)

[RMS](#h.463xbgg506z3)

[Real Time](#h.yv8szqm6sjqi)

[Soft real time systems](#h.93l3k3p4gg4c)

[Latencies](#h.o0tumhjx2600)

[Algorithm Evaluation](#h.kp5fpouq9rje)

[Analytic evaluation](#h.3uxgt85u2cbs)

[Deterministic Modeling](#h.4rvfjbpj7z33)

[Queueing](#h.8o2z5sycq4s4)

[Queueing-network analysis](#h.vqvebkpcdzbz)

[Simulation analysis](#h.vkzwnctvbs4j)

[Implementation analysis](#h.3tla3qdnkofv)

[Multi-core Scheduling](#h.8hohd56szlif)

[Load balancing](#h.ssy9lstajn7h)

[Issues](#h.xrvkafej1a0a)

[Hardware multithreading](#h.yaqmom7ft4fx)

[Little’s law](#h.m4l2h8hi1yv5)

[CPU/IO bursts](#h.5edxpt1p3wt)

[Thread scheduling](#h.146qdhk23674)

[Exam 1 Overview](#h.elrtt0qwq3qe)

[Deadlocks](#h.owtqlmkhicy0)

[System](#h.gatrqscg4l88)

[Conditions](#h.wv5qz0stx2tv)

[System Resource-Allocation Graph](#h.8hdr25gl2925)

[Deadlock Handling](#h.us5ndie8qust)

[Methods](#h.l5omnrnwh159)

[Deadlock Prevention](#h.h2lrbs13fqox)

[Deadlock Avoidance](#h.cole6ccoi5mv)

[Resource-allocation-graph algorithm](#h.m4l6w510heyd)

[Banker’s Algorithm](#h.89etbvi4zuht)

[Deadlock Detection](#h.vbwx32n8jq8i)

[Single Resource Instance](#h.4orijamxrbeo)

[Multiple Resource Instance](#h.x398zflg5h4s)

[When To Invoke](#h.fqvaft4o4rgh)

[Deadlock recovery](#h.z1opzfr50smr)

[Process Termination](#h.5jrkxqjz02bl)

[Resource Preemption](#h.hton6igi1vjj)

# Syllabi

Recitations: Can shift around, go to a different recitation if needed.

25% Midterm

25% Final

10% Weekly Reading Quizzes

40% Programming Assignments

Must avg 70% on Midterm/Final to pass the course

Midterm/final might be curved, never down

Not a math based course

Late: One week late = -20%, after that 0%

HW: 10% cleanliness, 50% functionality, 40% interview

Cite StackOverflow

5 maybe 6 HW assignments

# Operating Systems

## Historical Timeline

* 1930s/40s - Digital computers arise
* 1946 - The first general-purpose programmable digital computer
* 1950s - 1st OSs begin to emerge
* 1961 - MIT’s CTSS is the first time-sharing system
* 1966 - IBM System/360 mainframe OS
* 1969 - Unix
* 1981 - MS-DOS
* 1982 - 4.2 BSD Unix with TCP/IP networking
* 1984 - Mac OS with windowing GUI
* 1991 - Linux
* Late 1990s - First virtial machines
* 2007 - iOS

## Trends

* CPU speed is increasing, operating systems have to be more powerful as a result
* Different target environments
  + Cell phones = low power
  + Multiprocessor = large scale
  + Embedded = real time
* VMs provide more layers of abstraction
* Cloud computing on a massive scale

## What does an OS do?

* Provides an environment for other programs to do work
* Resource allocator
* Control program: Monitor execution and prevent errors
* Isolates applications from each other to prevent errors
* Sometimes contains middleware
  + In mobile, this is common, to give support for things like graphics and dbs

## ABI

“Application Binary Interface”

* Interface between a compiled executable and the OS
* size, layout, and bit alignment of data
* Calling convention (register conventions)
* System call interface

## API

“Application Programming Interface”

* Libraries, protocols, tools, etc. for building programs and applications
* Operating systems export an API to help interface with devices, memory, etc.

## Protection

One of the OS’s main goals is protection

* Protect applications from each other
* Protect the OS and the computer from the applications

It does this in a few ways

1. Preventing applications from writing into privileged memory
   * Where the kernel is stored
   * Certain devices
   * Where another app is stored
2. Preventing applications from invoking privileged functions
   * OS kernel functions, mainly

### Memory Protection via Virtual Memory

* An executable only has virtual addresses
* Virtual addresses are translated into physical addresses at run time via the page table
* OS controls the page table
* This makes it difficult for a program to write into address space that it doesn’t own
  + Any virtual address is translated into a non-conflicting physical address
  + Access to “wrong” memory causes a “page fault”
  + Hard to do with shared libraries

## Interfaces

Secondary storage = HDDs

From fastest to slowest:

registers → cache → main mem → SSD → magnetic disk → Optical disk → tapes

small computer-systems interface (SCSI) controller: has registers that the CPU can access, moves data from the device to the registers without the help of the CPU itself

## Multiprocessing

Multiprocessor systems (AKA parallel system or multicore systems) contain 2 or more processors.

### Device Failure

There are 2 types of failure when processors in multiprocessor systems start dying:

graceful degradation: service proportional to the level of surviving hardware

fault tolerant: can suffer single component failure and continue

### Asymmetric Multiprocessing

Some systems use asymmetric multiprocessing, where processors are given jobs. In this situation, one “boss” processor does the OS and device management, and the rest (known as “workers”) do computation.

### Symmetric Multiprocessing

Most systems use symmetric multiprocessing (SMP), simply enough, all processors are peers. This is much more difficult from the Operating System side, but much more efficient.

### Memory Access

There are 2 types of memory accessing:

Uniform memory access (UMA): access to any RAM from any CPU is constant time

Non “” (NUMA): some points in memory from some processors cause time penalties

### Clustering

#### Asymmetric Clustering

One machine is in “hot standby mode” while the other is running the applications. The hot standby machine only monitors the active server and becomes the active server if it fails.

#### Symmetric Clustering

Two or more hosts are monitoring each other and running applications. Can utilize “parallelization” to split up a job into many tasks that can be run in parallel

#### DLM

Data errors can occur if two machines are accessing the same data at the same time, a “distributed lock manager” (DLM) prevents this.

#### SAN

Storage-area-network (SAN): Applications and data are stored on one “drive” and all cluster machines connect to that.

## Multiprogramming

Multiprogramming is the running of multiple programs (sequences of instructions) simultaneously by a computer with more than one processor (or a cluster)

## Traps in OS

A trap is a software interrupt that can tell the CPU that the user needs an operating-system service performed.

## Kernel Mode

Processors generally have 2 different modes of operation. One, user mode, cannot perform many tasks, like accessing memory. The other kernel (or supervisor, system, or privileged) mode can do anything the processor is capable of.

### Rings of Privilege

Commonly, there are 4 rings:

* OS runs in ring 0 (kernel mode)
* rings 1-2 are unused, or sometimes house drivers
* Applications run in ring 3

Another configuration targeted towards VM’s

* Hypervisor (which runs the VMs) runs in ring 0
* VMs OS’s run in ring ½
* Applications run in ring 3

## Area Networks

There are a handful of different types of “area networks”

LAN: 1 building

W (wide) AN: the planet

M (metro) AN: buildings within a city

P (personal) AN: bluetooth/device to device that you’re using

## aaS

There are different types of services that certain companies will provide online:

SaaS: Software as a service: applications that would normally be on a desktop available online

PaaS: Platform as a service: software stack on the internet (elastic beanstalk)

IaaS: Infrastructure as a service: stuff available online (digitalocean)

## Real Time OS

Real time OS’s are OS’s that have rigid time requirements

## Operating system functions

* User interface
* Program execution
  + Must be able to load and execute something
* I/O
* Filesystem manipulation
* Communications
* Error detection
* Resource allocation
* Accounting (keep track of what users are using what resources)
* Security

## System Calls

System calls are the interface to operating system services. Usually makes use of a “system call interface” that will turn a programmed line into a system call. This is done via a trap table/branch table/jump table which knows all of the software interrupts and where to jump to when they are received. The jump to a trap table entry is called “dispatching.” The trap handler then performs the action in question.

### Unix

Unix/Linux system call library is libc. This provides interfaces/function prototypes for executing system calls.

### System call categories

* process control
  + kill, execute, create process, wait for event, allocate and free memory, and get process attributes
* File management
* Device management
* Information maintenance
  + Get time/date, get system data
* Communications
* Protection

### Parameters

3 methods to pass parameters

* Simplest: Pass in registers
* Pointers (used by Linux/Solaris) in registers
* pushed by the program and popped by the OS

### Roll Your Own System Calls

#### Pre-Configuration

1. edit /etc/default/grub
   1. Comment out “GRUB\_HIDDEN\_TIMEOUT=0”
   2. Comment out “GRUB\_HIDDEN\_TIMEOUT\_QUITE=true”
   3. Comment out “GRUB\_CMDLINE\_LINUX\_DEFAULT=”quiet splash””
   4. Add after the above “GRUB\_CMDLINE\_LINUX\_DEFAULT”””
   5. save
2. run sudo update-grub
3. in a fresh directory run “sudo apt-get source linux-image-$(uname -r)”
4. Install stuff: “sudo apt-get install cu-cs-csci-3753 ccache”
5. in your source directory run “sudo make localmodconfig”
6. run “sudo make menuconfig”
   1. Under “General Setup” change the “Local version” to something to tell your modified kernel apart from its legitimate buddies

#### Build Kernel

sudo make -j2 CC=”ccache gcc”

sudo make -j2 modules\_install

sudo make -j2 install

#### Add a system call

All file paths should be prepended with the location of your kernel directory

1. Add the C file to arch/x86/kernel/<blah>.c
2. Add to syscalls.h (/include/linux)
3. Add to Makefile (/arch/x86/kernel/Makefile) with “obj-y+=<filename>.o”
4. Add to syscall64.tbl (/arch/x86/syscalls/)

#### Example C file

#include <linux/linkage.h>

asmlinkage long sys\_helloworld (void) {

printk(KERN\_ALERT “hello world\n”);

return 0;

}

#### Test program

#include <stdio.h>

#include <linux/kernel.h>

#include <sys/syscall.h>

#include <unistd.h>

int main() {

//318 is the ID for the syscall

//from syscall64.tbl

syscall(318);

//w/ arguments

syscall(318, arg1, arg2, arg3);

return 0;

}

# Unsorted Vocab

Process can “lock” shared data, revoking access to another process.

“single step” mode allows you to step through code, breaking after every instruction

## System programs

System programs are programs that are provided by the system to perform certain tasks. Without these, an OS is basically unusable.

* file management
* status information
* file modification (text editors)
* programming-language support (compilers)
* programming loading/execution
* communications (internet)
* background services

### Application programs

On top of system programs there are “application programs” which provide everything else, like web browsers, games, and word processors.

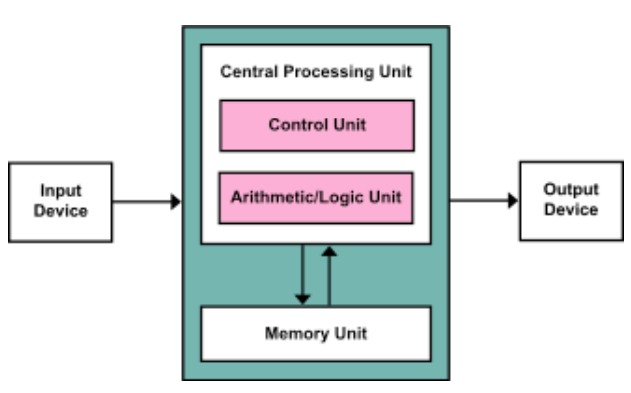
# Devices

## Turing/Von Machines

### Turing machine

Machine “head” slides along an infinite string of memory (cells), reads the contents of a cell and has a lookup table of operations on how to manipulate cells

### Von Neumann



## Vocab

device drivers: uniform device access interface between devices of the same type

Port: a connection point

PCI-Bus: Faster bus

Expansion bus: slower bus

Small Computer System Interface (SCSI) & Controller: A way for connecting slower devices

PCIe: 16GB/s Throughput

HyperTransport: 25GB/s Throughput

Serial Advanced Technology Attachment (SATA): HDD has a little processor that helps do some stuff.

nonmaskable interrupt: can’t disable

maskable interrupt: can disable

Programmed I/O (PIO): devices have little processors to communicate better with DMA controllers and the like

Escape/back door: passes commands to a device driver

raw I/O: linear array of blocks

direct I/O: allow a mode of operation on a file that disables buffering and locking

Programmable interval timer: Wait x, throw interrupt

vectored I/O: one system call performs multiple actions in multiple locations

buffering: memory area that stores data being transferred between two devices.

double buffering: decouples the producer from the consumer

copy semantics: version written to disk is the version at the time of the system call

spool: buffer that holds output for a device that cannot accept multiple jobs at once

core dump: capture of memory after an error

crash dump: core dump that happens on kernel crash

DTrace: Interface for viewing kernel info

Profiling: Looks at the IP every once in awhile to get information about what’s going on.

SYSGEN (system generation): The process of putting an OS onto a machine

EEPROM (erasable programmable ROM): BIOS ROM, so you can flash it if you need to.

## Interface differences

* Character stream vs. block
  + Character stream is one byte at a time
  + block is a bigger set
* Sequential or random access
  + Sequential: transfers data in an order determined by the device
  + Random: User can request any data section
* Synchronous vs async
  + Sync: data transfers with predictable response times
  + Async: not that. Not coordinated with other events
* Sharable vs. dedicated
  + Sharable can be used by many processes/threads at once
  + dedicated can only be used by 1
* Speed
* R/W, Read Only, Write Only

## Port/Mem Mapped

Port-mapped I/O

* special assembly instruction
* I/O address space is different/separate from main memory

Memory mapped I/O:

* I/O is mapped to the same address space as main memory
* Saves on assembly instructions
* Memory Management Unit (MMU): maps memory values and data to/from device registers

## Error handling

sense key: General descriptor (Illegal request, hardware error, etc.)

Additional sense code: category of failure (e.g. bad command parameter)

additional sense-code qualifier: more detail (e.g. which command parameter caused it?)

## Streams

Streams: Allows application to assemble pipelines of driver code dynamically

stream head: interfaces with user process

driver end: controls the device

1+ stream modules: the parts in between

flow control: buffers instructions being sent to devices

### DMA

Direct Memory Access: Bypass the CPU for large data copies

#### DMA Modes

* Burst mode (DMA transfers from start to finish)
* Interleaved mode/cycle stealing: set number of cycles
* Transparent mode: DMA transfers when CPU isn’t

# OS Design

## How do you design an OS?

1. Design goals
   1. User goals
   2. System goals
2. Mechanisms and policies
   1. Mechanism: How do we do that?
   2. Policy: What is going to be done?
3. Implementation
   1. If you write in C it’s easy to port to ARM
   2. emulators can help test other ports

## Structures

monolithic: everything’s included (Linux)

Microkernel: Mach OS and a little bit of Mac Os

* Defines communication between programs, which is done via message passing

Modules: loadable kernel modules

layered approach: everything is broken into a number of layers, layer 0 is kernel/supervisor mode, layer N is the user interface

## Mac OS X

* hybrid kernel
* Aqua user interface
* Cocoa ABI
* Kernel modules === kernel extensions

# Processing

## Boot Process

Boot sequence:

1. POST - Power On Self Test
2. BIOS
3. bit 0 of drive is loaded, which loads the main bootloader (GRUB)
4. Second stage boot loader loads the kernel

## Multitasking vs. Multiprogramming

Multiprogramming is just multiple programs in memory, multitasking is multiple programs running on the CPU at the same time.

## Multitasking types

Cooperative multitasking: Voluntarily yield the CPU before they’re done

Preemptive multitasking: Force programs to release CPU

Preemptive multitasking has 2 subsets:

Preemptive time slices: Each program is given a short interval on the CPU

Preemptive interrupts: timer interrupt suspends currently executing program and starts the next one (this is a context switch)

### Time sharing

Time sharing: “dumb terminals” are slaves to the master computer. Master multiplexes the CPU between the terminals

## Process States

Process states:

* New: being created
* Running: executing
* Waiting: waiting for some event to occur
* Ready: waiting to be executed
* Terminated: finished execution

## Program Control Block (PCB)

Process control block (PCB) contains the following:

* Process State
* Program counter
* CPU registers
* CPU-scheduling info (priority, etc.)
* Memory-management info
* Accounting information (what it’s using up)
* I/O information
* Potentially thread information

## Scheduling

The selection process for what processes are loaded and executed is done by the *scheduler*. The operation of the OS switching between execution of multiple programs is called time-multiplexing.

### Long-term Scheduler

The long-term scheduler (or job scheduler) selects processes from the pool and loads them into memory

#### Degree of Multiprogramming

The degree of multiprogramming is the number of processes in memory.

#### I/O-bound vs. CPU-bound

An I/O bound process is a process that takes more time using I/O than it does doing computations

A CPU bound process is a process that takes more time doing computations

The long-term scheduler wants a good mix of those two types.

### Short-term Scheduler

The short-term scheduler (or CPU scheduler) selects from the ready processes and decides which to execute.

### Medium-term Scheduler

The medium-term scheduler “swaps” some processes out of memory when too much is going on.

## Process Creation

There are 2 child-parent process execution possibilities:

1. Parent executes concurrently with its children
2. Parent waits until some or all of its children have terminated

As well as 2 address-space possibilities

1. Child process is a duplicate
2. Child process has a new program loaded into it

### Copy-On-Write

In copy-on-write, the child process shares parent’s code, only creates a physical copy when written to.

### Unix

New process is created by fork()

exec() replaces the process’s memory space with a new program

You can use wait() for the parent to wait for the child to exit().

### Zombie Process

The parent gets a bunch of information about the child via wait(), so if the child dies before wait() we get a zombie process because the parent has no way to know that the child has terminated.

### Orphaned process

Orphaned processes are processes whose parents are dead. In \*NIX, this is usually handled by assigning the parent to an orphaned process to init (the main \*NIX process).

## Process handling

Process Manager Responsibilities

* Creation/deletion of processes
* Synchronization of processes
* Managing process state
* Scheduling processes
* Monitoring processes

## Threads

A thread is a logical flow or unit of execution that runs within the context of a process. It is often characterized as 1 series of instructions.

### Advantages

* Reduced context switch overhead
* shared resources = less memory consumption
* inter-thread communication is easier and faster

#### Benefits of multithreading

* Responsiveness
* Resource sharing
* Economy (threads are cheaper than processes)
* Scalability

### Disadvantages

Why are processes still used?

* Some tasks are sequential
* No fault isolation between threads
* writing thread safe code is hard

### Thread-safe

A piece of code is **thread-safe** if it functions correctly during simultaneous or concurrent execution by multiple threads. Global, static, and heap variables aren’t thread safe.

### Reentrant

Code is **reentrant** if a single thread can be interrupted in the middle of the code and resume execution after the interrupt without errors.

#### Example

**Non-reentrant**

int x;

swap(int\* y, int\* z) {

x = \*y;

\*y = \*z;

//Interrupted here and swap() is run in

// the interrupt, global variable is messed

// up. Bad...

\*z = x;

}

**Reentrant e.g. 1**

int x;

swap(int\* y, int\* z) {

int a = x;

x = \*y;

\*y = \*z;

\*z = x;

//”restore” the global variable

x = a;

}

**Reentrant e.g. 2**

swap(int\* y, int\* z) {

//don’t use globals

int a = \*y;

\*y = \*z;

\*z = a;

}

### User vs. Kernel Threads

User-Space Threads: cooperatively multitasked threads. The program handles the threads not the OS.

Kernel threads: OS handles the threads and schedules them like processes

### Implicit threading

Implicit threading is when the compiler introduces threading, not the application developer.

#### Methods

Thread pool: Create a number of threads at process startup and just pick em up and use them when needed.

* Benefits
  + Servicing a request with an existing thread is faster than making a new one
  + Limits number of threads
  + Allows different strategies to execute a task

### Issues

* fork() and exec()
  + Do we have all of the threads on the new process?
* Signal handling
* Thread cancellation
  + Async cancellation: One thread kills another
  + Deferred cancellation: Thread periodically looks to see if it’s supposed to be dead

## Multicore

### Challenges for multicore systems

* Identifying tasks (what can be split up?)
* Balance
* Data splitting
* Data dependency
* Testing/debugging

## Misc. Vocab

Process: a program actively executing from main memory with its own address space

job queue: all processes in the system

ready queue: for ready processes

device queue: waiting processes for a specific I/O device

dispatched: when a process is selected for execution

An application is the sum of all of its processes

A process is the sum of all of its threads

## IPC

Interprocess Communication (IPC): Want to split an application into multiple processes, but they need to share data.

* Shared memory
  + OS creates a *shared memory buffer* between processes
  + Often implemented by having page tables of both processes point to the same section in memory
* Message passing
  + Uses send() and receive()
  + Special “ports” used to communicate
  + Advantage: doesn’t require synchronization
    - Blocking send() mitigates race conditions
* Pipe
  + Simple FIFO buffers that can be accessed like files
* Signals
  + Small numerical code sent to a process
  + Usually OS-to-process communication
  + Good for knowing when something is completed, like a read()

## Shared memory

### Usage

* shmid = shmget(key name, size, flags)
  + Creates a shared memory segment using key name
  + returns the handle to the shared memory
* shm\_ptr = shmat(shmid, NULL, 0)
  + attach a shared memory segment to the address space
  + this is called *binding*
* shmctl()
  + modify control information and permissions

## Sockets (message passing)

* sd = socket(int domain, int type, int protocol);
  + sd = “socket descriptor”
  + domain = PF\_UNIX for local sockets or PF\_INET for internet sockets
  + type = SOCK\_STREAM for reliable delivery of a byte stream
  + type = SOCK\_DGRAM for delivery of discrete messages
  + protocol = 0, default protocol to go along with type
* bind(sd, (struct sockaddr\*)&local, length);
  + The struct contains a unique unused file name
* Flow:
  + Server inits socket()
  + Client inits socket()
  + Server binds to port or file
  + Client connect()’s to that port/file (sends a connect request)
  + Server accept()’s the connect request
  + Send/recieve

## Pipes

* int piped[2] = {read end file descriptor, write end file descriptor}
* pipe(piped);
* read(piped[0], readdata, length)

### Named pipes

* basically files, but they’re not files, they’re pipes
* Operate as fifo buffers

## Exceptions

Trap: Software interrupts, always returns to next instruction, synchronous

Fault: Potentially recoverable error, might return to current instruction, synchronous

Interrupt: Hardware interrupt, always returns to next instruction, async

Abort: nonrecoverable error, never returns, synchronous

### Signals

#### \*Nix signals

|  |  |  |
| --- | --- | --- |
| Number | Name | Event |
| 2 | SIGINT | Keyboard Interrupt |
| 8 | SIGFPE | arithmetic error (Floating Point Exception) |
| 9 | SIGKILL | Kill yourself. |
| 10, 12 | SIGUSR1, SIGUSR2 | User-defined |
| 11 | SIGSEGV | Seg Fault |
| 14 | SIGALRM | Timer signal from alarm function |
| 29 | SIGIO | I/O now possible on descriptor |

#### Problems

Signals can come in at any time, so program around it to avoid race conditions. Multiple signals could lead to unpredictable results if not programmed around. One way of preventing this is masking signals.

## Race conditions

Race conditions are when two processes are (or want to be) editing the same shared information. We want to avoid those.

### Critical section

A process that is updating global info should not be running together with other processes. A critical section gives one process full control for a small amount of time. There’s also some other sections related to critical sections of code.

* entry section: portion of the code where the process asks to run a critical section
* critical section: uninterruptable section of code
* exit section: exits out of the critical section
* remainder section: everything else

#### Critical Section Requirements

* Mutual exclusion
  + If process 1 is executing its critical section, nobody else should be executing critical sections.
* Progress
  + If no one is executing a critical section and a process is waiting to execute a critical section, that process should be allowed to execute its critical section
* Bounded waiting
  + If there is a bound on the number of times other processes can enter their critical sections.

### Peterson’s solution

After one process has run its critical section, it tells the other process that it can run its critical section.

### Mutex locks

mutex, or mutual exclusion, has an acquire() function that acquires the lock and release() that releases it, as well as “available” var that will tell you if you can lock. Requires busy-waiting.

#### Advantages:

* user doesn’t have control over interrupts
* user doesn’t have to remember to reenable interrupts
* Tasks with I/O that need to process interrupts can make progress

#### Disadvantage

* User still must remember to release the lock

### Test-and-Set

* CPU provides uninterruptible hardware instruction (TS)
  + Atomically performs both the test and the set operations
  + Called by TestandSet system call

#### Example

//Acquire the lock

while(TestandSet(&lock));

//critical section

count++;

#### Advantage

* User task can do some other execution while polling TestandSet()

#### Disadvantages

* Requires user to remember putting it in while()
* while() is still busy-waiting
* requires hardware support

### Semaphores

A semaphore is a variable that can only be accessed with two atomic operations, wait() (which is the same as P()) and signal() (which is the same as V()). There are 2 types:

* counting semaphore: unrestricted
* binary semaphore: 0 or 1

used to control access to a given resource, use wait() on the semaphore, which is similar to test-and-set but also decrements the value of the semaphore, and signal() which increments the value of the semaphore.

#### Basic Example

Semaphore S = 1;

int counter = 0;

P(S); //wait

//critical section

count ++;

V(S); //signal

#### Enforcing Order Example

Process 1:

Semaphore S=0; //initial value 0

count++; //critical section 1

signal(S); //tell the other guy to execute

Process 2:

wait(S);

count--; //critical section 2

### Monitors

The general idea behind monitors is that we need to make sure locks are being used properly.

#### Monitor ADT

A monitor type is an ADT which includes a set of programmer defined operations that need mutual exclusion (within the monitor)

monitor *monitor name* {

/\* shared variables \*/

function 1() {}

function 2 () {}

inti () {}

}

#### Conditionals

Condition variables have 2 functions

x.wait(): Which suspends the calling task

x.signal(): Resumes exactly 1 task

##### Example

Task 1:

wait\_until\_empty.wait();

//proceed after queue empty

Task 2:

//queue is empty so signal

wait\_until\_empty.signal();

#### Issues with Monitors

* process might access a resource without permission
* process might refuse to release a resource
* A process might attempt to release a resource it never requested
* A process might request a resource twice (deadlocking itself)

### Problems

deadlocks: processes are waiting for an event that can be caused only by a process that is waiting

indefinite blocking/starvation: processes wait indefinitely within the semaphore

priority inversion: A medium process interrupts a low priority process while a high priority process was waiting for the low priority process to finish

## Situations

The following sections include some problems that can happen with the various types of locks presented above, leading to issues.

### Bounded Buffer Problem

How does the producer know what the consumer has consumed?

### Readers-Writers Problem

Many readers can access something at once, but if a writer and a reader or 2 writers access it, you have problems

### Dining-Philosophers Problem

Say you have a driver that always needs access to 2 devices at once in order to perform its action. If both drivers want to access both devices at the same time, but each driver has a lock on one device while waiting for the other, you get deadlocked.

# Scheduling

## Process switching

A process can be switched out by

* I/O blocking
* yielding the CPU
* being time sliced
* termination

### Context Switch

Context switching follows the following basic process:

1. save old state
2. select new process
3. load new state
4. switch to user mode and execute

## Misc. Scheduling Vocab

* Exec time E(Pi) = The time to fully exec process i
* Wait time W(Pi) = the time process i is in the reads state but not running (sum of the gaps between time slices)
* turnaround time T(Pi) = the time from the first entry of process i into the ready queue and its final exit from the system
* Response time R(Pi) = the time from the 1st entry of process i into the ready queue to its 1st scheduling on the CPU (run state)

## Scheduler’s Responsibilities

It’s the scheduler’s job to decide the next process to run, this is done via a scheduling policy.

### Scheduler’s goals

* Maximize CPU utilization
* Maximize throughput (# processes completed/sec)
* Minimize average/peak turnaround time
* “” waiting time
* “” response time
* maximize fairness
* meet deadlines
* ensure priorities are adhered to

### Analysis

* Look at wait time, turnaround time, etc.
* Simplify analysis by assuming no blocking I/O and processes are executed until completion

## Scheduling Algorithms

### FCFS

First Come First Serve: exactly as you would expect.

### SJF

Shortest Job First: exactly as you would expect

* Minimizes average wait time
* Must know execution time

### Round Robin

Preemptive time slicing, tasks are simply switched out after a certain amount of time.

#### Weighted Round Robin

Give some tasks more slices than others.

### EDF

EDF (Earliest Deadline First) scheduling: choose the task with the closest deadline. Good for real time systems

### Priority scheduling

High priority first, if there are a few that are the same priority, another policy is used

#### Multilevel queue scheduling

Each priority level has a queue

* To fix starvation, you can do a sort of weighted round robin on the priority levels
* You can also give a starved process a slightly higher priority

### Scheduling In Linux

#### CFS

Completely Fair Scheduler: Linux’s general purpose scheduler

* If there are N tasks, an ideal CPU gives each task 1/N of the CPU
* Selects the task with the maximum wait time
* Higher priority tasks get larger slices
  + this is handled by “nice” which sets the priority
* Higher priority tasks are scheduled more often
* While CFS is fair to tasks, it isn’t fair to applications
  + An app with more threads gets more time

#### Real Time

* Real time FIFO (SCHED\_FIFO): soft real time
* Real time Round Robin (SCHED\_RR): soft real time w/ priority
* Real time EDF (SCHED\_DEADLINE): hard real time

### RMS

Rate-Monotonic Scheduling:

* For periodic tasks
* A task’s priority = 1/(task period)
* If a task’s priority is higher than another task, it preempts that task.

## Real Time

Real time systems are systems with strict deadline requirements.

### Soft real time systems

* Want to hit most deadlines but some can be missed

### Latencies

* event latency
  + amount of time that elapses from when an event occurs to when it is serviced
* Interrupt latency
  + Period of time from arrival of interrupt to interrupt service
* Dispatch latency
  + time for scheduling dispatcher to stop one process and start another
  + conflict phase
    - Preemption of any process running in the kernel
    - Release by low priority processes of resources needed by high priority processes

## Algorithm Evaluation

### Analytic evaluation

Uses the given alg and the system workload to evaluate performance of that workload

#### Deterministic Modeling

Deterministic modeling is a type of analytic evaluation. Takes a predetermined workload and determines the performance of each alg for that workload

### Queueing

No static set of processes, have to go off of the distribution of CPU/IO bursts.

#### Queueing-network analysis

1. Computer system is described as a network of servers
2. Each server has a queue of waiting processes
3. After knowing arrival and service rates, we can compute a bunch of values

### Simulation analysis

Use a “trace tape” to record a series of actual events, then drive the simulation from that.

### Implementation analysis

Use a real system with a real use and try different algs on it.

### Multi-core Scheduling

Scheduling over multiple processors/cores is a new problem

* Asymmetric
  + One core handles scheduling, then dishes out processes to other cores
* Symmetric
  + All cores have their own schedulers
  + Sometimes, all cores have their own ready queues

#### Load balancing

Goal: Keep workload distributed across cores

* Central ready queue
  + Easy, processors just grab a task from the queue when idle
* Separate ready queues
  + push migration
    - A dedicated task checks load and rebalances
  + Pull migration
    - Whenever a core is idle it tries to pull a process from a neighboring core

##### Issues

* can cause cache conflict
* Can cause power management conflicts, having all cores at 100% doesn’t save power

#### Hardware multithreading

* Multiple hardware threads per core
* Threads can become blocked waiting for data
  + Instead of waiting, some processors will switch immediately to another thread
* If N hyperthreads are supported per core, it appears as if there are N logical cores per physical core
* Requires 2 levels of scheduling
  + Map threads to hardware threads
  + Which hardware thread to run

## Little’s law

N = λ\*W

* λ = arrival rate of tasks in queue
* μ = service rate of tasks
* W = average wait time in queue per process
* N = average queue length

## CPU/IO bursts

A CPU burst is one continuous string of CPU instructions, and an IO burst is the same for I/O instructions

## Thread scheduling

Lightweight process (LWP)

Process Contention scope (PCS): user level threads run on one LWP

To decide which kernel level thread to schedule on the CPU. the kernel uses a System Contention Scope (SCS)

# Exam 1 Overview

1. Intro to OS
   * Bootstrap program (MBR)
   * Secondary bootloader (GRUB)
   * OS vs. Kernel
     + Kernel: 6 pieces
       1. System calls
       2. Device management
       3. Process management
       4. Memory management (not on exam)
       5. File management (not on exam)
       6. network management (not on exam)
   * User mode vs. kernel mode
     + Switch to kernel mode via system calls
2. Devices
   * Device controller/drivers
     + Device controller is hardware within the device (the chip in a keyboard)
     + Driver interfaces with that
   * Port vs. memory mapped I/O
   * Interrupt driven I/O, polling I/O, DMA
3. Concurrency/Synchronization
   * mutex, semaphores, condition variables, monitors
   * thread safe/reentrant code
   * deadlocks
4. Processes
   * threads vs. processes
   * Process control block: how to organize code, data, heap and stack
   * scheduling

# Deadlocks

## System

The definition of a system when looking at deadlocks is a finite number of resources distributed among a number of competing processes.

Processes utilize resources in the following sequence:

1. Request
2. Use
3. Release

In a deadlock, processes never finish executing and system resources are tied up.

## Conditions

The 4 conditions for a deadlock (ALL must be met), are as follows:

1. Mutual exclusion: At least one resource must be held in a non sharable mode
2. Hold and wait: A process must be holding at least 1 resource and be waiting for others.
3. No preemption: resource can only be released voluntarily
4. Circular wait: P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P0

## System Resource-Allocation Graph

Resource allocation graphs are constructed as follows:

* Set of vertices V
  + 2 types of nodes
    - P: consisting of all active processes
    - R: consisting of all resource types
* Set of edges E
  + assignment edge goes from process to resource indicating the process has requested the resource
  + request edge goes from resource to process indicating use

## Deadlock Handling

### Methods

* We can use a protocol to prevent or avoid deadlocks
  + Deadlock prevention
  + Deadlock avoidance
* We detect deadlocks and recover
* We ignore the problem and pretend it doesn’t happen

### Deadlock Prevention

In general, here’s how to prevent each of the deadlock conditions:

* Mutual exclusion
  + Mutex must hold, i.e. one resource must be non-sharable
* Hold and wait
  + A process can’t hold one resource and request another
  + A process can only request resources when it doesn’t have any resources
* No preemption
  + If a process is holding some resources and requests another resource that cannot be immediately allocated, then all resources the process is holding are preempted, or implicitly released
* Circular wait
  + Impose total ordering of resources
  + Verification of ordering is done via a “witness” (library function)

### Deadlock Avoidance

Here’s how we go about avoiding deadlocks:

* Processes give the maximum number of resources of each type they will need
* safe state: system can allocate resources to each process
  + Only happens if there is a “safe sequence”: A sequence of processes for which each process can obtain all of its needed resources, complete its designated task, return its resources, and terminate
  + Not all unsafe states are deadlocks
  + Only a safe sequence of processes are given at any time

#### Resource-allocation-graph algorithm

* New edge: “claim edge” which indicates that process Pi might request resource Rj at some time in the future (represented by a dashed line)
* When a resource is released, an assignment edge is changed to a claim edge

#### Banker’s Algorithm

Banker’s algorithm is an algorithm for determining if a system is in a safe state at any given time. While this technically falls under “deadlock detection”, it’s more often used to avoid deadlocks by checking every incoming process

* Requires a set of data structures. n is the number of processes and m is the number of resources
  + Available: vector of length m that indicates the number of resources of each type
  + Max: n by m matrix defines the max demand of each process
    - If max[i][j] = k, then process Pi may request at most k instances of resource type Rj
  + Allocation: n by m matrix that defines the number of resources of each type currently allocated to each process
  + Need: n by m matrix indicating current resource need
* Safety algorithm: uses those data structures to define whether or not a system is safe
* Resource-request algorithm: uses those structures to define whether or not a resource request can be safely granted (usually by just reusing the safety algorithm including the new process)

### Deadlock Detection

#### Single Resource Instance

In cases that there’s only 1 instance for each resource type, we can use a “wait-for” graph, that’s just the resource allocation graph that removes the resource nodes and collapses the arrows.

#### Multiple Resource Instance

Banker’s algorithm (in the previous section) is good for this.

#### When To Invoke

One of the big questions is when should we invoke the deadlock detection algorithm? There’s 2 factors:

* How often will a deadlock occur?
* How many processes will be affected?

You probably want to invoke deadlock detection more often than the deadlocks will occur, and perhaps more if there are more processes involved.

### Deadlock recovery

#### Process Termination

There’s 2 flavors of process termination when recovering from deadlocks. Simply enough:

* Abort all processes using a deadlocked resource
* Abort one process at a time until the deadlock is unlocked

#### Resource Preemption

Resource preemption is a “nicer” method of deadlock recovery, here’s how it works:

1. Select a victim
2. Rollback that processes’ changes
3. Release the lock

Ideally, once rolled back, the deadlock will be released but the process didn’t notice that anything out of the ordinary happened.

Also, as a general rule, don’t always preempt the same process to avoid starvation