OS

CSCI 3573
By: Will Drach

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 (Intel)

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 1 or 2
- 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 (PA1)

Pre-Configuration

1. edit /etc/default/grub
    a. Comment out “GRUB_HIDDEN_TIMEOUT=0”
    b. Comment out “GRUB_HIDDEN_TIMEOUT_QUITE=true”
    c. Comment out “GRUB_CMDLINE_LINUX_DEFAULT=”quiet splash””
    d. Add after the above “GRUB_CMDLINE_LINUX_DEFAULT”””
    e. 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”
    a. 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;
}
```

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.

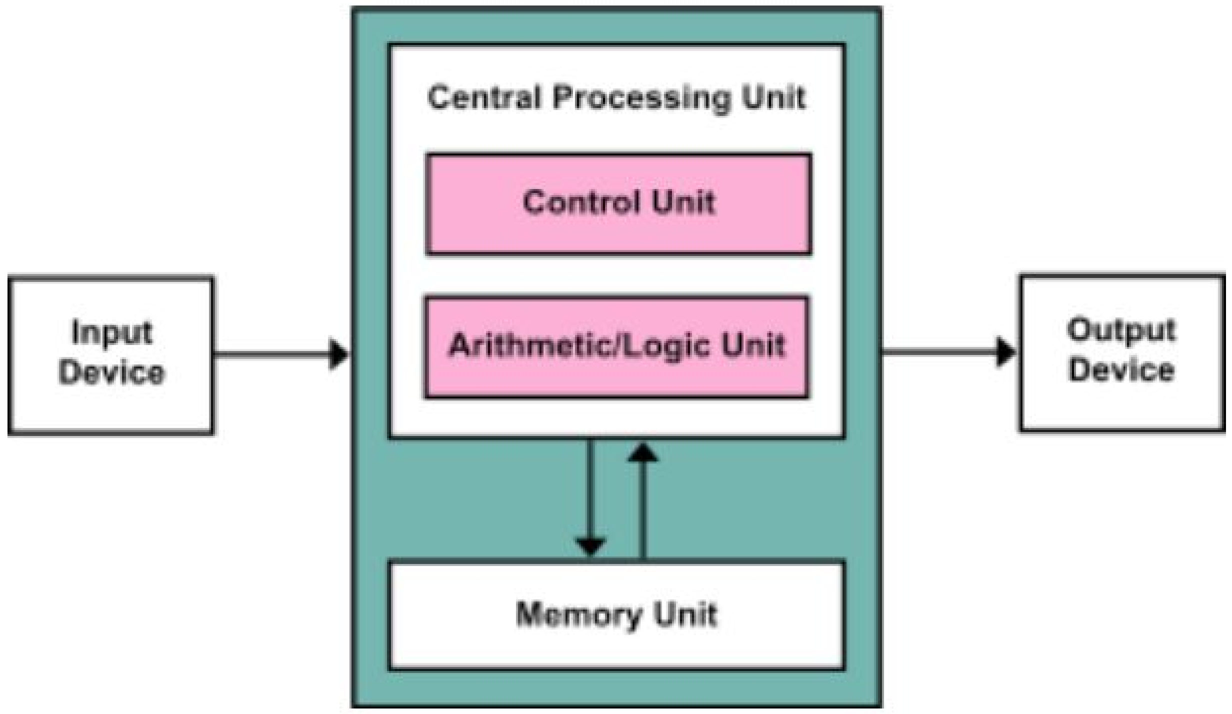
Devices

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

Drivers In Linux (PA2)

LKMs

Drivers in Linux are often implemented as a “Loadable Kernel Module” or LKM. These are sets of system functionality that can be loaded into the kernel without the need for a reboot.

Implementation

The drivers make use of a “module_init” and a “module_exit” function that allows you to tell the system where your initialization and exit functions are. These functions are run alongside `insmod` and `rmmod` respectively to make sure that your driver gets initialized properly and exited cleanly.

The rest of the implementation is all handled by Linux device driver files, which are explained in the following section.

Device Files

Linux drivers are implemented as “device files”, which can be accessed just like any other file on the filesystem.

Major/Minor Number

Device files don’t just get to be named whatever they want, however. Each device *driver* has a “major number” so the kernel knows what driver to use for which file. Each *device* under that driver has a “minor number”, which lets the kernel know which specific device you are trying to access. With these two combined, the kernel knows which device you want and which driver to use for it.

Character/Block Drivers

Simply enough, character devices are setup to read one character at a time, while block devices are setup to read blocks of data at a time. Whether a driver is for a character or block device is generally hard-coded into the device file on creation.

file_operations struct

Even though a device is treated as a file, it, well, it isn’t a file. For this reason, we have to code file operations into our device driver, and decide what exactly happens when we “read” our file. The file_operations struct contains all of your functions so that the kernel knows what to do when someone wants to “read” or “write.”

OS Design

How do you design an OS?

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

Structures

Monolithic: everything’s included (Linux)
Microkernel:Defines communication between programs, which is done via message passing, and that’s about it. (Mach OS)
Hybrid: Some mix (Mac)

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

When you have someone producing things and someone consuming things, both people need to know how many spots are open and how many spots are full. If both people try and change these values at the same time (producer increments the full spots and consumer decrements it), you get a race condition.

Bounded Buffer Solution

The producer and consumer processes share the following:
- `int n;`
- `semaphore mutex = 1;`
- `semaphore empty = n;`
- `semaphore full = 0;`

There’s a pool of n buffers, each capable of holding one item. The empty and full semaphores count the number of empty and full buffers.

The producer is implemented as follows:
```
wait(empty);
wait(mutex);
//add an item to the buffer
signal(mutex);
signal(full);
```

And the consumer is implemented as follows:
```
wait(full);
wait(mutex);
//remove an item from the buffer
wait(mutex);
signal(empty);
```

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. To avoid this, writers must have exclusive access while writing.

There are 2 varieties of the readers-writers problem:

1. No reader should be kept waiting unless a writer has already obtained permission to use the shared object. In other words: no readers should be waiting for other readers just because a writer is waiting
2. Once a writer is ready, it should perform its write immediately (no new readers accepted)

1st R-W Solution

The reader and writer processes share the following

- `semaphore rw_mutex = 1;`
- `semaphore mutex = 1;`
- `int read_count = 0;`

The writer is structured as follows:
```
wait(rw_mutex);
//write
signal(rw_mutex);
```

The reader is structured as follows:
```
wait(mutex);
read_count++;
if (read_count == 1) wait(rw_mutex);
signal(mutex);
//read
```

```
wait(mutex);
read_count--;
if (read_count == 0) signal(rw_mutex);
signal(mutex);
```

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.

Practical Synchronization In Linux (PA3)

There's nothing in PA3 that isn't described above, so this is just going to be a ton of super basic example code to remember.

pthread Thread Creation

```
pthread_t threads[NUM_THREADS]
int i;
for(i=0; i<NUM_THREADS; i++) {
    int ret = pthread_create(&threads[i], NULL, <thread routine>,
<args struct>);
    //error if ret != 0
}

for(i=0; i<NUM_THREADS; i++) {
    //pthread_join blocks until thread i is terminated
    pthread_join(threads[i], NULL);
}
```

pthread mutex

```
pthread_mutex_t test_mutex;
pthread_mutex_init(&test_mutex, NULL);
pthread_mutex_lock(&test_mutex);
//do yo thang gurl
pthread_mutex_unlock(&test_mutex);
pthread_mutex_destroy(&test_mutex);
```

semaphore.h semaphores

```
sem_t our_sem;
sem_init(&our_sem, 0, 1);
```

```
sem_wait(&our_sem);
//do whatever you want, you’re an adult.
sem_post(&our_sem);
sem_destroy(&our_sem);
```

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

O(N)

Linux's very early scheduler

- Each process gets a time slice based on priority
- If a process yields its time slice, it gets awarded "goodness"
- "goodness" ups that process's priority
- "goodness" values are unsorted, so O(N) search time

O(1)

A revised version of the Linux scheduler.

- 2 queues
    - Run queue: tasks that need to run
    - Expired queue: tasks that have run this period
- Once a task has exhausted its time slice, it is moved to the expired queue
- Expired tasks are not eligible for execution until the run queue is empty
- The scheduler iterates through a list of 140 priorities to find the highest priority task to run
- When expired tasks are moved back to the run queue, their priority can change
    - New priority = nice value (Linux's self-induced priorities) + f(interactivity)
        - The function of interactivity can change by at most +/- 5
        - This is to give tasks in I/O blocks less run time, and CPU intensive tasks more run time

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
    - When a task is given a "1" length time slice, update the other tasks' wait time by 1/N
- 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

Revised CFS

Revised CFS is very similar to CFS in concept, but handles things a little better in implementation.

- sum run times given to each task and choose the one with the minimum run time

Virtual Run Time

In Revised CFS, a new task would have a 0 run time, and would take a while to catch up in run time (and would therefore get selected a lot). To remedy this, we use “virtual run time” instead of actual run time. When a new task comes in, its virtual run time is set to the minimum of all of the other tasks’ virtual run times. Whenever a task is scheduled, its chunk of run time is added to its virtual run time, resulting in a fair metric for all tasks.

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

$N = \lambda * W$

- 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
    - Work[i] - length m (# of resources)
        - Work is initialized as Available
    - Finish[i] - length n (# of processes)
        - If Alloc[i] == 0, finish[i] = true
- 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)

Important things to point out:

- Safe sequences are not unique, there may be multiple for a given state
- Complexity is O(m*n^2)
- To avoid deadlocks, simply don't let the system enter a non-safe state
- Banker's is mainly for avoidance, and relies on knowing the available resources and max needed resources.

Safety Algorithm

Find a process such that both Finish[i] == false and Need <= Work, if it doesn't exist go to the end
Work = Work + Alloc[i]
Finish[i] = true
Goto beginning
If Finish[i] is true for all i, system is safe

Request Algorithm

If Request[i] > Need[i] the new request exceeds the maximum, exit
If Request[i] > Available Pi must wait, exit
Avail -= request, Alloc[i] += request, Need[i] -= request, run Safety algorithm and check/

Dijkstra's Banker's Algorithm

- Before granting a request, run Banker's algorithm pretending as if the request was granted
    - If it's in a safe state, grant the request
    - If it isn't, wait
- Generalize deadlock avoidance to multiple resources

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

A modification of Banker's algorithm works for this purpose.
Data structures: same, but no Max or Available
Alg

- Find a process i such that both Finish[i] == false and request[i] <= Work, if no such process exists, goto end
- Work = Work+Alloc[i]
- Finish[i] = true
- Goto start
- If finish = false for some i → deadlocked, and on top of that, every process where finish = false is deadlocked

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. On top of this, how bad the deadlock balloons after it occurs is a good consideration. Rules of thumb:

- Could check at resource request -- costly but reliable
- Could check periodically, but hard to find the right period
- Could check if utilization suddenly drops, indicating polling resource requests
- Could check if resource utilization spikes, but what's a good threshold?

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

Memory Management

Memory Hierarchy

The memory hierarchy from fastest (and least capacity) to slowest is as follows

- Registers: Instant but very low capacity
- L1 Cache: 1 clock cycle, ~16KB
- L2 Cache: 4-5 clock cycles, ~1MB
- L3 Cache: Shared between cores, ~40 clock cycles, ~8-256MB
- Main Memory: ~100 cycles
- Secondary storage
    - Disk: ~10ms
    - Flash: 10-100us
- Networked disks (very slow)

Caches

Cache Jobs

Modern CPUs have multiple caches for specific types of data

- Instruction cache
- Data cache
- TLB (page table cache)

Cache Organization

Caches are generally laid out with S sets, E lines per set, and B bytes per line. A line contains a valid bit, a tag, and the actual data, which is made up of B bytes. The cache size is SxExB data bytes.

Cache Types

There are a few ways to organize the cache past the basic organization:

- Fully associative → one row
    - The cached item can go anywhere in the cache row
- Direct mapped → one column
    - Cached item can only go in a row slot/bin corresponding to its memory address
- N-way Set associative → N columns and M rows
    - N items per bin, M bins

Cache Writes

There are generally 2 write behaviors for a cache

- Write-through: changes in caches go immediately to memory
- Write-back: changes set a “dirty bit” and are written on replacement

On top of this, we have to figure out what to do when you try and write to something that isn’t in the cache:

- Write-allocate: load into cache, update in cache write-back style
- No-write-allocate: write immediately to memory

Cache Replacement

The most common cache replacement approach is Least Recently Used (LRU), which evicts the cache entry that was used furthest in the past. This is a good approximation of the proven best algorithm, which is “evict the entry that will be next used furthest in the future.”

Address Binding

Address binding is the process of allocating memory to a process upon its entry into memory. There are 3 types of binding:

- The first is when you know at compile time where the process will be in memory
- The second is when you do not know, so the compiler must generate “relocatable code” and bind the address space on load time
- The last is when the process can be moved during execution, so you want to bind on execution

Simple Virtual Memory

- Every process has a “base address” which is stored in a base register. This is the smallest legal address.
- The “limit”, stored in the limit register, specifies the size of the chunk of memory.

Address generated by the CPU is a logical address, where the one seen by the memory (and loaded into the MAR), is a physical address.

Binding Method

For compile-time and load-time binding, the physical address is the logical address and vice versa. For execution binding, the logical address is the virtual address.

All logical addresses generated by a program represents the logical address space, and the physical addresses corresponding to them form the physical address space.

MMU

The MMU is responsible for 2 things:

- Address translation: translate logical to physical addresses

- Bounds checking: make sure the requested memory address is within the upper and lower limits of the address space.

Dynamic loading

Dynamic loading is simply that a routine is not loaded until it is called. The main benefit of this is that you are always checking for the most updated version of the routine, especially if that routine is something like an external library or a syscall.

Stub

For dynamic linked libraries, a stub indicates how to find the version of the dll that's in memory

Swapping Mechanisms

Standard swapping

Simply pop processes in and out of HDD and RAM

- very, very large overhead/context switch time.
- Need to make sure the process is actually idle, not just waiting for I/O
    - Can be solved by storing the data from I/O in an OS based buffer until the old process is context switched in
- Not used today, in general

Transient OS code

In general, we don't want to keep infrequently accessed programs (OS utilities) in memory, so we need a buffer to put them in when we do need them.

Memory allocation

Partitioning

Partitioning is simply the use of fixed size chunks.

Variable-partition

- You take a variable sized "block" from the "hole" (unallocated memory)
- Choosing where to place the block
    - First fit

        - Allocate to the first hole that is big enough
    - Best fit
        - Allocate the smallest hole that is big enough
    - Worst fit
        - Allocate to the largest hole always

Fragmentation

Variable partitioning often causes fragmentation, where holes aren't big enough, but the total memory you need is still available.

- Holes get consistently smaller and more scattered
- "External Fragmentation" happens when there is enough total memory space to satisfy a request, but not enough contiguous memory space
- Given N allocated blocks, .5N blocks will be lost to fragmentation (50% rule)
- Internal fragmentation is when the memory allocated to the process is bigger than requested (unused memory within a partition)
- Can "compact" everything, but only if the memory is allocated on execution and you can modify the page tables

Segmentation

A bit easier to digest view of memory

- Memory consists of segments that have a name and a length. The addresses specify both the segment name and the offset within the segment
- Segments are essentially variable length pages
- Logical address = [segment-number, offset]
- Segment table makes this possible in the real world. Each segment in the segment table has a segment base and a segment limit, where the base is where the segment resides in memory, and the limit specifies the length

Paging

- Break physical memory into fixed size boxes called "frames"
- Break logical memory into same sized boxes called "pages"
- Every address is split into 2 parts, page number (p) and a page offset (t)
- Page number is the index in the page table, which contains the base address of that page
- The offset points to a more specific byte.
- Page table is kept in main memory, and a page table base register (PTBR) points to its location in memory

  - This is slow (2 memory accesses are needed to look at 1 byte), but we speed that up using a “translation lookaside buffer” (TLB) which acts as a cache for the page table
  - If it isn’t in the TLB (a “TLB miss”), a memory reference to the page table must be made.
  - Most TLBs allow entries to be “wired down” which prevents them from exiting the TLB
  - TLBs sometimes add address-space identifiers (ASIDs) to the TLB entry, which allows extra security and prevents programs from accessing each other’s address spaces
- Extra protection is added to a page table in the form of a “valid-invalid” bit, which tells us whether a process is allowed to access a given page.
- A page table length register (PTLR) is sometimes included and used to double check that the correct amount of memory is allocated at a given time

Forward-mapped Page Table

Outer page table contains an inner page table which contains the base address of a given page

Inverted page table

Page table has one entry for each real page in memory, which includes information on who owns the page. Only one page table is in the system and each page only has one entry in the page table.

Hashed page table

Used for page tables where address spaces are too big, uses a linked list of elements to get the right block of memory

Clustered page table

An offset of a hashed table where each entry in the hash table refers to several pages. It is really space efficient, but can be inaccurate.

Hashed inverted page table

An inverted page table provides a value, which gets hashed and used to index a clustered page table.

Sparse page table

The sparse page tables assume that a few entries will just be 0’s or junk, so don’t store those in the page table (but you still have to allocate them in memory)

On-Demand Paging

Page tables may be large, consuming lots and lots of RAM. The key here is that not all pages in a logical address space need to be kept in memory. On demand paging swaps pages in and out of disk, and only keeps one subset (or the "working set") in memory or in the TLB.

Page-fault

On a page fault, the needed page should be loaded into RAM. The following is the process:

1. MMU detects a page is not in memory and sends a page-fault
2. OS saves process state and jumps to the page fault handler
3. Fault handler
    a. If the reference to the page is not in the logical address space, seg fault
    b. If the reference is not in RAM, load the page
    c. Free a frame/find a free frame
    d. Schedule a disk read
4. Returns with interrupt when done reading desired page, OS writes page to frame
5. OS updates page table
6. Restart interrupted instruction

Basics of Predictive Paging (PA4)

Often times, pages are swapped in and out of the TLB rapidly to ensure that the most programs can be run most efficiently in the least amount of time. This usually happens when the OS throws a page fault, indicating that a needed page is not currently in memory, and needs to load.

For predictive paging, the general idea is that you always want the currently needed and the next needed page in, and always want to be swapping out the page that will be used the furthest in the future, which LRU is an approximation of.

Basic Replacement Strategy

There is one page replacement strategy that is worth mentioning, and that is using a dirty/modify bit. Essentially, a dirty bit is set when a page is written to. If the dirty bit is set, the page needs to be written to the disk. If it isn't set, you can just evict it without writing to disk.

On top of this, the following replacement strategies are good to know

- LRU
    - "Virtual time" based
    - Queue
- FIFO
    - If frequently accessed, this is bad
    - Belady's anomaly: more pages → more page faults (contrary to logic)

- OPT
    - "OPTimal": replace the page that will not be referenced for the longest time. Hard to implement
- Reference bit
    - Set a reference bit every time a page is referenced
    - LRU approximation
        - Can record last 8 bits and the LRU is the lowest valued record
        - Second-chance/Clock: Go around in a circle, reset the reference bit to 0, then continue. If a reference bit is still 0, you have a page to evict.
        - Enhanced Clock: Use dirty bit as part of the selection, if reference=0 and dirty=0 you have the best page to replace
- Non-LRU counter replacement
    - Least frequently used
        - Lowest counter
    - Most frequently used
        - Highest counter: in theory the page with the smallest count was probably just brought in and has yet to be used.
    - Expensive to implement

Improving Replacement

A couple of improvement strategies:

- Use a dirty/modify bit to reduce disk writes
- Choose a good page replacement algorithm
- Make the search for the least important page fast
- Page buffering: select a page to evict after the page you want is already in
- Keep a pool of free frames and remember their content ("free" them, but don't erase them)

Allocation of Memory Frames

There's a fixed set of frames, so who gets them?

- Determine the minimum # of frames for a process
- Equal allocation
- Proportional allocation (bigger processes get more frames)
- Local Vs. Global
    - Local: Process gets fixed set of pages
        - Easy to manage
        - Bad: process might need more
        - Bad: "Fake isolation," P1 causes page faults → P2 suffers in disk I/O
    - Global: Pool all pages together

Thrashing

Repeated page faulting, usually occurs when a process' working set is bigger than the memory it has.

- Under local replacement, P1 thrashes, P2-PN only suffer in disk I/O
- Under global: Bad for everyone, but it can be resolved quickly by adding more stuff

Solution
Working set: Find a set of recently accessed pages that is repeated often, allocate to the process the size of the working set. You can set a reference bit and use a timer to evict pages out of the working set.

OR

Page fault measuring: If the page fault frequency is higher than some threshold for a process, allocate it more memory.

Linux Replacement

- Global page replacement
- Clock-like LRU (global)
- Thrashing can only occur if the sum of all working sets exceed the size of physical memory
    - If this happens, terminate a process

Status Bits

Page tables often contain a handful of status bits:

- Valid/invalid bit: is the page valid?
- Dirty/modify bit: has the page been modified?
- R/W or read only: can you write to this page?
- Reference bit: For some replacement algorithms.

Paging and L1/L2 Caching

In order to use paging and L1/L2 caches together, we first have to determine whether the cache uses physical or virtual addresses

- If physical, the MMU must first convert virtual to physical before accessing the cache, which is slow but reliable
- If virtual, the cache can be consulted without even consulting the MMU

Homonym Problem

The homonym problem is simply that when a new process is switched in, it might use the same virtual address as the previous process

Solutions:

- Add an address space/process id to each entry of the cache
- Flush the cache on each context switch
- Give each process it's own non-overlapping virtual address (not very good, screws up allocation)

In Practice

Most L1 caches are virtually indexed to be fast, and most L2 caches are physically indexed to be reliable.

Memory-Mapped Files

The general idea behind memory-mapped files is that it can be fast to read/write to memory. If you load the file into memory all at once, and then use close to push the changes.

Steps

1. Obtain a handle to a file by creating or opening it
2. Reserve virtual addresses for the file in the logical address space
3. Declare a portion of the file to be memory mapped (void* mmap(void* start, size_t length, int prot, int flags, int fd, off_t offset)
4. When the file is first accessed, it's demand paged into memory
5. Subsequent read/writes are served by physical memory

File Systems

What is a file?

Human perspective: logical storage unit that is useful to humans
OS perspective: Sequence of bytes that exists in a physical storage device

A file is a collection of data and attributes.

File Metadata

Includes (usually no more than 1KB)

- Name
- Unique ID
- Size

- Protection info/permissions
- Timing info (access time, create time, modify time)
- Location
- Type (some OS's)
- Creator (what application created it, some OS's)

This is stored in a file header or a file control block. In UNIX this is called an inode.

System Calls

The OS has to provide basic system calls to manipulate files

- create()
- read()/write()
    - read/write can be either
        - Sequential access (remember where the last read/write was and start there)
        - Direct access (given an offset)
- delete()
- open()/close()
- seek()/reposition()
- truncate()
- append()
- rename()
- copy(), move(), etc.

Directories

Files aren't enough, humans need directories to organize.

Operations

- List
- Create
- Delete
    - 2 behaviors, force delete or delete if empty
- Move, copy, rename
- Search?

Directory Structure

There are 3 types of directory structure:

1. Single level (flat)
    - Hard to organize, multiple users have files jumbled together

2. Two level
    - Each user gets a flat directory
3. Tree-structured
    - :fire:

Sharing

Symbolic Links

Simply a pointer to another file. Deleting the file won't delete the links, so you have to go clean those up.

Hard Links

Essentially a copy of the file

Duplicate Directories

Really hard to maintain consistency

Set Permissions

Users can access each other's files, very good option.

Mounting

We need a way to translate physical devices to virtual directories. Mounting does this.

Unix allows you to mount anywhere within the current directory tree.

Virtual File Systems

In lots of cases, the mounted file system could be of a different type than the current OS file system. You can fix this by adding a "virtual file system layer" in between the file systems and the file system interface

File System Implementation

- File system elements stored on both Disk/Flash and RAM
- On disk/flash, the entire file system is stored, which includes
    - Its entire directory tree
    - Each file's file header/inode
    - Each file's data
    - Boot block

        - Volume control block (partition details)
- In memory, the file manager maintains only a subset of open files
    - Memory is essentially a cache
    - The four main components in memory are:
        - Recently accessed parts of the directory structure
        - System-wide “open file table” (OFT) that knows which files are open
        - Per-process OFT
        - Mount table of mounted devices

Open Process

1. The directory structure is searched for the file
2. Copy the file control block to the OFT
3. Add an entry to the per-process OFT that points to the global OFT entry
4. Return a file descriptor

Close Process

1. Remove the entry from the per-process OFT
2. Decrement the open file counter in the system OFT
3. If counter = 0, write back to disk any metadata changes

Directory Implementation

There are a few options here:
- Implement directory as a linear list
    - Could be very slow, unless it is sorted
- Implement as a hash table
    - ext3/ext4 use a hashed B-Tree variant

File Allocation

How do we lay files out on the disk? Similar to memory, there are a few ways to do this:
- Contiguous
    - If a file is N blocks long, it is allocated N contiguous blocks
    - Disadvantages
        - External fragmentation
        - File sizes change
        - Over-allocation

- Splitting into blocks
    - we could “page” it similar to memory. Divide the disk into fixed size blocks, and allow the file to be spread across the whole device
    - We need a data structure to keep track of where each block of the file is, though.

- Linked Allocation
    - Each file is a linked list of disk blocks
    - Good at solving problems caused by other allocation schemes, but random access is slow.
- File Allocation Table (FAT)
    - Similar to linked list, but instead of the pointers being in the current block's data, it's in a table
    - The linked list is terminated by an EOF
- Indexed allocation
    - Collect all pointers into a list/table called an index block
        - Index j → j'th block on the disk
    - The index block can be anywhere, unlike FAT, which has to be at the beginning
    - Problem: Size of index is unknown, can fix by linking index blocks
- Multilevel indexed
    - First level index block points to a second level index block
    - You don't have to allocate unused second-level blocks
- UNIX multilevel
    - Assume N entries in the index block
        - N-3 points to a singly indirect block (index block pointing to disk blocks)
        - N-2 points to a doubly indirect block (2 levels of index blocks)
        - N-1 points to a triply indirect block

Free Space Management

Just like you have to keep track of what is allocated and where, you have to keep track of what isn't allocated. Options are as follows:

- Bit vector
    - Each block is represented by a bit
    - Have a vector of all of the blocks. If bit = 1: free, else allocated
- Linked List
    - Link together all free blocks
    - Only keeps track of free blocks
    - You can find your 1st free block immediately
- Grouping
    - Linked list, but with multiple pointers to free blocks in each node
- Counting
    - Same as grouping, but you store the number of free blocks with the pointer
    - If a pointer has "3" free blocks, that means the 2 blocks immediately following the first block are also free

Performance

Improve performance using memory

- Cache the FCB
- Cache directory entries
- Memory mapping files

On disk

- Indexed allocation
- Counting, grouped, or linked list free block lists

Other

- Disk controller provides its own cache
- Cache file data in memory
- Smarter layout on disk
- Read ahead
    - If the OS knows the access is sequential, read the requested page and several subsequent pages into memory
- Async writes
    - Delay writing of file data

Fault Recovery

In general, the OS should be able to recover from as much as possible. However, the file system is quite fragile, particularly because any in-memory data is lost on power loss.

Problems

- Async writes produce inconsistency between memory and disk
    - UNIX gets around this by caching read-only data, but any writes that change metadata or free space allocation are immediate
- Even if all writes are synchronous, they can fail halfway through
- Complex operations can create inconsistency while waiting for them to complete
    - Using create() for example has many operations to fully create a file

Solutions

There are a few ways to deal with these problems

- Run a consistency checker
    - Fsck in UNIX
    - Chkdsk in MSDOS
    - They don't like vowels
    -
    - Vowels are the source of hard drive inconsistencies
    -
    - You heard it here first
    - Anyways, this is reliable, but slow
- Log-based recovery

    - The log on disk is consulted after a failure to reconstruct the file system
    - Each operation is written as a record to the log before the operation is performed
        - Write-ahead logging
    - Operations are grouped in sets called transactions
        - create() has many steps, but would be grouped into 1 transaction
        - Transactions are atomic, they either all succeed or they fail
        - Transactions “start”, then all of the functions are logged, and then the transaction is committed, which finishes the operation
        - The file system then performs the actions and “completes” the transaction
        - A “checkpoint” is used to tell the fs what is done executing. Everything older than the checkpoint has been completed
        - To recover, perform all of the actions that have been committed and undo actions that are not fully committed
    - Journaling vs. Log-structured
        - Journaling has a separate circular buffer that records the most recent actions
        - Log-structured is where the filesystem IS that buffer

Journaling

Some filesystems only write changes of the metadata of the fs to the log. Ext4 is a bit different, in that it has 3 modes:

- Journal: both metadata and file data are logged (good but high latency)
- Ordered: Only metadata is logged, but the file contents are written to disk before the metadata is committed
- Write back: only metadata is logged, no other safety.

Disk Scheduling

The disk is usually made up in 5 sequential parts, the boot area, the free space manager, the directory structure, the FCBs, and the file data blocks. With magnetic (spinning) disks, we have to be careful of how we access these to try and improve performance. Each platter of the disk is subdivided into tracks, and each track is subdivided into sectors. The collection of the same track (same radius, different platters) forms a logical cylinder. In order to write/read, you have to move the arm to the correct track (seek time), rotate the disk under the R/W head (rotational latency), and then read/write the bit. This means the total access time is seek time + rotational latency + transfer time, but is mainly dominated by seek time and rotational latency.

Disk Scheduling Algorithms

- FCFS: First Come First Serve: Decent, no starvation
- SSTF: Shortest Seek Time First: Scheduling selects the next request with the minimum seek time: Faster, but can cause starvation

- OPT: minimize movement of the disk head: move disk head to innermost track, sweep out
    - Have to recalculate every time there's a new request
    - Can just move in and out sequentially
- SCAN: is just that, move from innermost to outermost and then back to innermost
    - Approximates OPT
    - Starvation free
    - Not perfect, unnecessarily goes to extreme ends of disks even if there's no requests, and is unfair to the tracks on the edges of the disk
- C-SCAN: Circular scan: when reaching one edge, move back to the other edge, but don't read on your way back, only read on your way out
- LOOK: SCAN but don't travel to the edge if there's no request out there
- C-LOOK: C-SCAN for LOOK

SSTF and LOOK are the most reasonable choices in most cases. These algorithms are usually part of the disk controller, along with algorithms to reduce rotational latency.

File Layout

- Linked or indexed allocation spreads files out, increasing seek times
    - Opening a file for the first time needs directory info, file header, and file data, which can be slow if they are spread out
    - Store directories in the middle tracks, so at most half of the disk is traversed to find the file header and data
    - Can store the file header near the data
- Which disk block is chosen to allocate out of the free blocks?
    - Ext3fs tries to cluster the location and layout of disk blocks for file data around the inode
    - It also locates the inode near the parent directory
    - Spreads high-level parent directories around disk to minimize fragmentation in 1 area of the disk

Flash

Flash memory is a form of EEPROM (electrically erasable programmable read-only memory).There are generally 2 types:

- NAND
    - Word accessible
    - Slower random access (have to read a word)
    - Short erasing and programming times
    - Long write life time (for flash)
- NOR
    - Byte accessible
    - Faster random byte access

    - Much longer erasing and programming times

Flash is typically divided into blocks, each containing many pages, which each have 12-16 bytes of error correction and bookkeeping (which the fs can use/access). Reads are pretty typical page + offset, but writes can be tricky. You absolutely have to erase everything that was currently where you want to write before you can write there. So rewrites require an erase and then a write. This erase can only happen on an entire **sector**, so if you have to write one byte, you have to delete and rewrite the whole sector. For this reason, writing to flash is about the same speed as writing to magnetic disks.

Flash also has "memory wear," where basically there is an upper cap on how many writes and erases you can do. To keep this manageable, fs' or disk managers usually try and level the wear out over all of flash memory. This can cause problems, however, because frequently used data (directories, metadata) are generally in one place.

Filesystems On Flash

Must be designed with the 2 major problems in mind:
- Rewrites require erases
- Limited writes

For this reason, Log-structured file systems are well-suited to flash (everything is appended to the end of the log, and the log is the file system). Another system is JFFS (Journaling Flash File System), which is essentially a circular log-structured file system. You can clear up free space by combining log entries (garbage collecting).

RAID

Simply enough, magnetic disks are cheap, so how can we arrange them to get the most out of them?

RAID0

Data striping with no redundancy

RAID1

Mirror each disk

RAID0+1

Data striping for performance + mirroring the stripes for redundancy. The stripes are made first (RAID0) and then the mirroring is added on top (RAID1)

RAID1+0/RAID10

Same thing, opposite order. The mirroring is first (RAID1) and then the stripes are later. E.g. if you have 4 disks, 2 of them will contain groups of stripes and then the other 2 will be an exact mirror of those 2.

RAID2

Put error correction code (ECC) bits on other disks

RAID3

Bit-interleaved parity: For each bit at the same location on different disks, compute a parity bit and store it on the parity disk. (This is after striping)

RAID4

Block-interleaved parity: For each block, compute a parity block and store it on the parity disk

RAID5

RAID4 but instead of a parity disk, the parity blocks are distributed among the disks. So if you have 3 disks, sector 0 of disk one and two will contain data, and sector 0 of disk three will contain a parity block. However, in sector 1, disk one might have the parity block.

Implementations

- Operating system supported
- RAID controller (OS just sees 1 disk, but all disks are on the same bus)
- RAID array (OS just sees 1 device)

Networking

Networking is typically achieved via network sockets using a network stack.

- The bottom layer is the “physical layer” (fiber optic, wire, etc.) (1)
- Then the “Data Link/MAC layer” (2)
- Then the network layer (3)
- Then the transport layer (4)
- Then the application layer (5)

To send a packet of data, each layer passes a packet of data to the next lowest layer.

Packetization

To send, you simply call the socket API’s send(). However, if a file is too large, you have to segment it into smaller packets. The application layer prepends a layer 5 header and sends it on. Typically at each layer a new header is prepended to the data and then sent on. When receiving, this goes in reverse, where headers get stripped at each point.

Layers

Application Layer

The application layer sender communicates application-specific information with its peer receiver

Transport Layer

The internet is likely to lose the application's message, so it's the transport layer's job to do error detection or correction.

TCP/UDP

TCP (Transmission Control Protocol) does a few things to help packet loss
- It is the receiver's job to send an ACK packet to tell the sender it got the packet
- If the ACK isn't received before a timeout, the sender retransmits
- TCP also ensures in-order delivery

TCP essentially can be used like an internet based pipe, everything that goes in is going to come out in the same order. However, TCP can be slow, so we need something that can transmit more! This something is UDP (user datagram protocol), which broadcasts packets and doesn't care about retransmission or ordering.

Network Layer

The internet consists of many routers that connect together and route all of the packets. The Internet Protocol (IP) is responsible for this. It tries to find the shortest route and send the packet along that.

Data Link Layer

Each link between 2 routers must be able to transmit packets. The data link layer handles this.

MAC Sublayer

In a way, inside of the data link layer, exists the Medium Access Control (MAC) sublayer. This makes an attempt at tagging all of the incoming and outgoing packets with a MAC address to make sure they are getting to the right place, as there is usually a many-to-many relationship in the data link layer. The MAC sublayer also decides who gets to transmit to avoid collisions

MAC Protocols

There are a few ways of doing this:

- TDMA: time division multiple access: each user is assigned a time slot where they can transmit
- FDMS: frequency division multiple access: each user is assigned a separate frequency
- CSMA: Carrier sense multiple access: check if the medium is free and transmit if it is

Physical layer

Simply enough, we need a way to transmit raw digital bits.

Distributed FS

The general idea of a distributed FS is to be able to share remote files. This is usually built off of a TCP/IP stack.

NFS

NFS (Network File System) is a virtual file system that runs off of an RPC based protocol.

- Files are abstracted as vnodes
- The VFS hides the details of the underlying file system
- Multiple file systems can be supported simultaneously
- Operations are done on the vnodes, which is sent to the server and then translated to the local fs.
- NFS v3 & earlier is stateless (no knowledge of previous requests)
- NFS can cache, and v4 keeps track of open files
- NFS client can read-ahead, as well as write-behind (write async)

SMB

Samba (SMB or CIFS) allows file sharing between UNIX and Windows machines.

File Hosting

Google Drive, Dropbox, etc. are all examples of file hosting services. They are usually accessed via HTTP protocols making them easy to use everywhere.

RPC

RPC (Remote Procedure Call) essentially allows remote (via TCP) execution of a function/procedure call.

Virtual Machines

A process is already given the illusion that it has its own memory and own CPU (via virtual memory and time slicing), so why not draw that out and let a process believe it has its own full hardware stack (memory, CPU, I/O, etc.).

Benefits

There are many benefits to VMs:

- Can run multiple OS's
- Fault isolation, OS's are completely independent
- Easier to deploy applications, and customize to certain applications

How it works

We want close to native CPU speeds, so interpreting every instruction is bad

- Guest OS executes normally
- If a trap is thrown (needs to get kernel mode) it goes to the hypervisor, not the CPU
- The hypervisor deals with this trap the way the virtual OS would want
- The most ordinary instructions are just as fast as native, but a trap is slightly slower
- Intel uses their ring levels to do this (guest OS = ring 1, user process = rind 3)

Paging is still a pain, so basically, you have to allocate all of the memory to the VM itself, not to the processes. The VM then controls its own memory.

- Create shadow page tables (run by the virtual memory manager - VMM)
    - Virtual address → virtualized physical address (guest OS does this)
    - Virtualized physical address → machine addresses (real physical addresses)
    - Can be slower
- When there are multiple OS's, the VMM round-robins between them

Full Virtualization

Full virtualization is when the hypervisor provides the illusion of identical hardware as the real system.

- Guest OS doesn't even realize it's a VM
- Virtualized applications can run slower
- Demoting the privilege of the guest OS can cause excessive faulting when trying to run privileged instructions
    - Uses a binary translator
    - Turns an instruction into a series of instructions that will perform the action on the virtual device instead of the actual device.
    - Can get around this with hardware support

        - Intel's VT-x is a set of virtual machine extensions that support virtualization of processor hardware
            - Introduced 2 new modes
                - Root mode: for the hypervisor
                - Non-root mode: guest OS
                - Both in ring 0
                - Allows guest OS to run in the same ring it's used to
            - Guest OS faults less and can execute more instructions, which means faster performance
            - Hardware extended page tables
                - Each guest OS gets its own hardware support for page tables, translates to machine addresses direct from virtual addresses
                - Guest OS can directly handle page faults

Paravirtualization

Dissimilar from full virtualization, for paravirtualization the OS no longer sees full virtualization of the hardware. Guest OS is modified to make special system calls.

Type II Hypervisors

Type II hypervisors run virtual machines similar to an application process on top of the host OS. They can be insecure, but are generally good enough. The VMM is simply a patched linux kernel, and then a kernel module is added to talk to the user process.

Java VMS

For Java VMs, the goal is not to run multiple OS's, but to be able to run the same code on every OS. For this reason, Java byte code can be run on any Java VM. This is compiled at run time and run.

Containers

Run a VM under the same kernel as the parent OS.