CSE 120 Review

Kevin Jacob and Taanish Reja

Spring 2025

Table of Contents

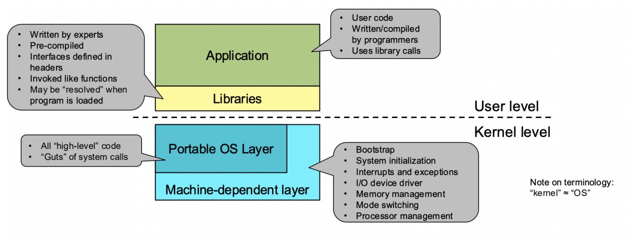


Figure 1: User and Kernel Level of operating systems. The unclosed ”gap” represents syscalls, which can be invoked at user level but are defined at kernel leve

1 Interaction with Apps and Hardware

1.1 User Level and Kernel Level

1.2 Dual Mode Operation

Dual Mode Operation is one of the ways in which operation systems perform special tasks and protect applications from themselves and others

- **Kernel Mode**: Can run all instructions
- **User Mode:** can run all non-privileged instructions

The mode is indicated by a mode bit in a protected CPU control register

1.2.1 Privileged Instructions

- Special class of instruction that can only run in kernel mode
- Can access I/O, manipulate memory management state (i.e. page tables), manipulate protected control registers (mode bit)

1.3 Memory Protection

- Prevent applications from manipulating memory management state (which could allow applications to hoard memory or take control of memory)
- Uses priviliged instructions

1.4 Events

- An event is a unnatural change in control flow
- Immediately stops current execution and changes from user to kernel mode
- The OS defines a handler for each event type
- Event handlers are executed in kernel mode
- After the system is booted, all entry into the kernel is the result of an event
- Exceptions and interrupts are the two main types of events

1.5 Exceptions

Exceptions are caused by the program executing instructions.

- Executing a privileged instruction (fault)
- Requesting services from the OS (system calls)
- Also called traps

1.5.1 Faults

- Hardware detects and reports exceptional conditions (divide by zero, page faults, ...)
- Hardware is responsible for saving state (PC,registers,mode,etc)
- Each exception type has an associated number and CPU finds handler for that number
- Switch to kernel mode and execute exception handler

Handling Faults

Recovery

- Some faults are handled by ”fixing” the exceptional condition such as the OS bringing missing pages into memory during a page fault. The fault handler can then return the program and re-execute the instruction that caused the exception.
- Some faults are handled by notifying the process so application can register a fault handler with the OS and the OS fault handler will return to the user-mode handler. Some examples of signals are SIGFPE, SIGTERM, and SIGSEGV.

```c
void signal_handler(int signum,siginfo_t *si, void *context){
    /* was this an integer divide-by-zero exception? */
    if(si->si_signo == SIGFPRE && si->si_code == FPE_INTDIV) {
        print("oh no, we tried to divide by zero!\n");
    } else {
        print("unexpected signal %d\n", si->si_signo);
    }
    /* let's exit */
    exit(-1)
}
```

```c
int main(int argc, char* argv[]) {
    struct sigaction sigact;
    int x =0;

    /* let's register our signal handler with the OS */
    sigact.sa_sigaction = signal_handler;
    sigact.sa_flags = SA_SIGINFO | SA_RESTART;
    if (sigaction(SIGFPE, &sigact, NULL) != 0) {
        printf("sigaction returned error: %d\n", errno):
    }


    /* now let's try to divide by zero */
    let y = 1/x;
    printf("divided by 0!\n"0;
    return 0;
}
```

Termination

- OS may handle unrecoverable faults by killing the user process which can happen when the program faults with no registered handler. OS can also halt the process, write process state to file, and then destroy the process.
- If a fault occurs in the kernel (divide by 0, etc) these faults are considered fatal and will cause the operating system to crash.

1.5.2 System Calls

- Operating system API
- CPUs provide syscall instruction that causes an exception (handled by kernel handler), provides a parameter for the syscall number to call, and stores the caller's state so that it can be restored
- Hardware support is crucial for saving and restoring state, switching execution modes, and resuming execution (CPU provides instruction, while Kernel provides code, memory addr of code, sys call number)

Syscall categories

- Process management
- Memory management
- Device management
- File Management
- Communication

1.6 Interrupts

Interrupts are caused by an external event

- Device finished I/O, timer interrupt, etc

Steps

- Disable interrupts at lower priorities
- Save state
- Transfer control to interrupt handler
- When done, re-enable interrupts and resume user-level program on next instruction

Example: Timer Interrupt

- Important for allowing OS to regain control of system
- Ensures fairness and helps timer-based functions keep track of time (i.e. sleep)

Example: Async I/O

- OS requests resource
- Device operates independently, acquriing resource
- Device sends interrupt to CPU when ready
- CPU context switches to interrupt handler and resumes process

2 Processes

A **process** is the OS abstraction for a running program. It is used to manage execution, scheduling, and other resources.

2.1 Components

2.1.1 Process State

- Memory address space for a process
- Code and data for the executing program
- An execution stack encapsulating the state of procedure calls
- Program counter
- Set of registers
- Set of OS resources
- Execution state
  - **Running:** executing instructions on the CPU
  - **Ready:** waiting to be assigned to the CPU
  - **Waiting (blocked):** waiting for an event such as I/O completion

2.1.2 PID

A process is names using its process ID (PID). The top command in Linux can be used to list the processes and their PIDs.

2.1.3 Process Control Block

- Contains process state
- Memory management state
- Scheduling and executing information
- I/O and file resources

2.2 Process API

2.2.1 Creation

- Every process is created from another process
- Child inherits some properties from parent (e.g. process user ID - child executes with your privileges)
- After creating child, parent may either continue execution or wait for child
- OS creates the first process (Linux: init/systemd)

We can create processes two ways

- From scratch
  Windows CreateProcess, UNIX spawn
  CreateProcess:

  - Creates and initializes new PCB
  - Creates and initializes new addr. space
  - Loads program specified in prog (prog)
  - Initializes saved hardware context (fresh context)
  - Set process state as ready

- From an existing process UNIX fork
  Fork:
  - Creates and initializes new PCB
  - Creates and initializes new addr. space. Addr. space is initialized with copy of entire parent's addr. space
  - Initializes resources pointers to point to same resources held by parent
  - Initializes hardware context to be a copy of parents
  - Set process state as ready

2.2.2 Overwrite

We can overwrite a process using exec()

- Stops the current process
- Loads the program prog into the process' address space
- Initializes hardware context and args for the new program
- Files remain open
- Sets the process state as ready
- exec can return if there is an error (file not found, permission denied, etc)

If run exec bash in a terminal it will replace bash with new bash and if you run exec ls in a terminal it will replace shell with ls and shell goes away once ls completes.

2.2.3 Termination

Unix: exit(int status)   Windows: ExitProcess(int status)
The OS frees resources and terminates the process

- Close open files/network connections
- Releases allocated memory
- Terminates all threads
- Removes PCB from kernel data structures, delete

A process doesn't have to clean itself up since the OS should take the responsibility and not trust the process.

2.2.4 Wait

We often need to wait until child process has finished before continuing (i.e. executing commands in a shell)
Wait

- Suspends execution until child process has finished
- waitpid() suspends until the specified process has finished

In Unix, every process must be "reaped" (killed and cleaned by its parent)

- Process waiting to be cleaned up: zombie process
- Unix: If parent exits before child, child becomes root process' child (other OS' may not do this)

2.3 Real World Example: Shell

```c
while (1) {
    char *cmd = read_command();
    int child_pid = fork();
    if (child_pid == 0) {
        Manipulate STDIN/OUT/ERR file descriptors for pipes, redirection, etc.
        exec(cmd);
        panic("exec failed");
    } else {
        waitpid(child_pid);
    }
}
```

s

2.4 Communication between threads

OS provides mechanisms for communication between processes

- IPC
- pipes
- shared memory
- Files

3 Threads

3.1 Components

We can rethink processes by separating them from execution state. Threads are execution state
Process
Address privileges, resources address space
Thread
Program counter, stack pointer, registers

3.1.1 Process Address Space (in terms of threads)

In our revised model, the address space is shared among threads and each thread has its own stack pointer and program counter (we have multiple of each now)

3.1.2 Thread Control Block

Per-Thread information is stored in Thread Control Blocks

- State (ready,running, or waiting)
- PC, registers
- Execution stack

When a thread is running, its hardware state (PC,SP,regs) are in the CPU. When the OS stops running a thread, it saves the registers in the thread's control block.
Context Switch: process of changing CPU hardware state from one thread to another (very frequent)

3.2 Scheduling

3.2.1 Thread Queues

The OS contains a collection of threads

- Ready queue
- Waiting queues (may be many wait queues depending on each type of wait, i.e. disk, network, timer, network, synchronization)

Each TCB is put on a queue according to its current state, and when its state changes, the OS unlinks it from one queue ot another

3.2.2 Preemptive vs. Non-preemptive

Non-preemptive
Threads manually give up access via **yield()**
Example:

```
while (1) {
 print("ping\n");
 yield();
}
```

```
while (1) {
 printf( pong \ n );
 yield();
}
```

(They alternate in order depending on what starts first)
How does yield work?

- Context switches to another thread
- Return from yield means another thread called yield()

Implementation

```
void yield() {
    thread_t old_thread = current_thread; // old thread execution
    current_thread = get_next_thread();
    append_to_queue(ready_queue, old_thread);
    context_switch(old_thread, current_thread);     // starting now, from new thread
    return;
}
```

Preemptive

- Non-preemptive threads must voluntarily give up CPU (Only voluntary calls to yield, sleep, or exit cause a context switch)
- Preemptive switching uses involuntary context switches via timer interrupts (forces current thread to yield) (OS default; can't trust threads to cooperate)

Note On Concurrency For a web sever, instead of spawning processes to handle requests, we can just fork new threads.

3.3 Threading Types

3.3.1 Kernel Level Threads

The OS manages threads and processes, so all thread operations are implemented in the kernel/the OS schedules all the threads in the system.
Benefits

- Multiple kernel thread (OS manages/schedules)
- Physical parallelism (can run on multiple cores)
- Multiple separate system calls/events

Limitations

- Make concurrency much cheaper than processes
- For fine-grained concurrency, kernel level threads suffer from overhead
  - thread operations require system calls
  - Kernel-level threads need to be general to support needs of all programmers, anguages, runtimes, etc
  - Want "cheaper" threads for such fine-grained concurrency

3.3.2 User Level Threads

Hide threads from the kernel by using user-level threads which are managed entirely by a runtime system (user-level library)
Benefits

- Small and fast since a thread is simply represented by a PC, registers, stack, and small TCB
- Creating a new thread, switching between threads, and synchronizing threads are done via procedure calls
- User level thread operations are 10-100x faster than kernel level

Limitations

- No physical parallelism since multiple user level threads are multiplexed onto one kernel level thread
- Only one system call/event at a time
- User level threads are invisible to OS which can lead to more integration
  - Block a process that initiated I/O request even though it has other threads that can be run
  - Schedule a process with no runnable threads

3.4 Multithreading Models

3.4.1 Many to One

- Many user-level threads per kernel thread
- Used in user-level threads

3.4.2 One to One

- One user-level thread per kernel thread
- Used in in kernel level threads
- e.g. c++ std::thread, C posix threads

3.4.3 Many to Many

- Allows many user-level threads to be mapped to many kernel threads
- Used in user-level threads
- e.g. Go, Java

4 Synchronization

4.1 Bank Withdrawals

```
withdraw (account, amount) {
    balance = get_balance(account);
    balanace = balance - amount;
    put_balance(account, balance);
    return balance;
}
```

- With one thread, we have deterministic results and scheduling order doesn't matter

- What about multiple threads?
  Execution can be extremely interleaved; race conditions can easily occur

4.2 Source of Concurrency Problems

- Race conditions
- Interleaved executions
  - Threads interleave executions arbitrarily and at different rate
  - Scheduling is not under program control
- Shared resources
  - Threads can share resources (i.e. variables)
  - Which resources are shared? Heap variables, static objects, global variables ,
  - Which resources are not shared? Local variables

4.3 Synchronization

Goal: restrict interleavings of thread operations
Ways to provide synchronization

- **Mechanisms** to control access to shared resources (locks,mutexes,semaphores,monitors,condition variables,etc)
- **Patterns** for coordinating access to shared resources (producer-consumer,reader-writer,etc)

4.4 Mutual Exclusion and Critical Sections

Critical Sections: section of code in which only one thread may be executing at a given time. Threads are forced to wait on entry and once a thread leaves a critical section, another one can enter. Utilize mutual exclusion to create critical section.
Critical Section Goals

- Mutual Exclusion
- Progress (A thread in a critical section will eventually leave it and it can't prevent other threads from entering critical section if it is not in the critical section)
- Bounded Waiting (No Starvation)
- Performance (Overhead of entering/exiting critical section is small relative to the work inside of it)

4.5 Locks

4.5.1 API

- acquire()/lock() (enter critical section)
- release()/unlock() (release critical section)

4.5.2 Spinlock Implementation

Spinlock

```
struct lock {
    bool held = False;
}

void acquire(lock* lock) {
    // disable interrupts
    while (lock->held);
    lock->held = True;
    //enable interrupts
}

void release(lock* lock) {
    // disable interrupts
    lock->held = False;
    //enable interrupts
}
```

By disabling interrupts, we prevent race conditions on a single core by preventing involuntary context switches due to timer interrupts (making the operation atomic), but it still has problems:

- Will not work on multi-core systems (disabling interrupts is per-core)
- Inefficient; spinlocks are wasteful of cpu cycles

4.5.3 Atomic instructions

There are hardware instructions that execute atomically and can be utilized to create mechanisms for critical sections
test-and-set

- Record the old value
- Set the value to true
- return the old value

```
bool test_and_set(bool *flag){
    bool old = *flag;
    *flag = True;
    return old;
}
```

4.5.4 Atomic Spinlock

```c
struct lock {
    bool held = False;
}
void acquire(lock) {
    while(test_and_set(&lock->held));
}
void release(lock) {
    lock->held = False;
}
```

4.5.5 Guard Implementation

```c
struct Lock {
    bool held = False;
    bool guard = False;
    queue Q;
}

void acquire(Lock* lock) {
    // disable interrupts
    while (test_and_set(&lock->guard));
    if (lock->held) {
        // put current thread on lock->Q
        lock->guard = False;
        // block current thread;
    }
    lock->held = True
    lock->guard = False
    // enable interrupts
}

void acquire(Lock* lock) {
    // disable interrupts
    while (test_and_set(&lock->guard));
    if (lock->Q is Empty) {
        lock->held = False;
    }else {
        // dequeue thread from lock->Q and put it into ready queue
    }
    lock->guard = False
    // enable interrupts
}
```

5 Semaphores

5.1 API and Components

Contains a counter that is initialized by some value (based on how many resources we are keeping track of) that is hidden away from the user

- wait()/P() atomically waits for semaphore to become greater than 0, then decrements by one
- signal()/V() atomically increments semaphore by one

5.2 Implementations

5.2.1 Spinning (executed atomically)

```
wait(s) {
    while (s <= 0);
    --s;
}
signal(s) {
    ++s;
}
```

5.2.2 Blocking (executed atomically)

```
wait(s) {
    if (s <= 0)
        sleep();
    --s;
}
signal(s) {
    if (queued thread)
        wakeup();
    ++s;
}
```

Blocking semaphore explained

- Each semaphore is associated with a queue of waiting threads
- When wait() is called by a thread
  - If semaphore is open (positive) thread continues
  - If semaphore is closed (negative) thread is blocked
- Then signal() opens the semaphore
  - If the queue is waiting on a thread, the thread is unblocked
  - If no threads are waiting, the signal is remembered by the counter (no lost wakeups as the counter is a history)

5.3 Semaphore Types

5.3.1 Binary Semaphore

- Behaves like a lock
- Represents access to a single resource
- Guarantees mutual exclusion in ciritcal sections

5.3.2 Counting Semaphore

- Represents a resource with many units available
- Multiple threads can pass the semaphore at once
- Number of threads determined by semaphore ”count”

5.4 Benefits

- Semaphores have a value enabling more semantics
- Two use cases: mutual exclusion and event sequencing

5.5 Producer-Consumer/Bounded Buffer

We want to make a limited buffer where consumers take items and producers add items
Problems:

- No serialization (things can happen simultaneously)
- Tasks are independent
- Multiple consumers/producers
- Buffer allows each to run without explicit handoff
- How do we ensure synchronized, safe access to the buffer?

5.5.1 Locks (problematic)

```
while (1) {
    produce an item
    if (count == N)
        sleep();
    acquire(lock);
    insert item in buffer
    count++;
    release(lock);
    if (count == 1)
        wakeup(consumer)
}
```

```
while (1) {
    if (count == 0)
        sleep();
    acquire(lock);
    remove item from buffer
    count--;
    release(lock);
    if (count == N-1)
        wakeup(producer)
    consume an item
}
```

Problems

- What if we context switch before sleeping in the consumer? Then the producers may fill the buffer until full, try waking up other consumers, etc.... But once we context switch back, the wakeups weren't "recorded," so the consumer's wake up is lost.

5.5.2 Semaphores

Lets examine our constraints:

- Consumer must wait for the producer to produce items
- Producer must wait for consumer to consume items
- Only one thread can manipulate the buffer at once

If we initialize full_count with 0 and empty_count with N, we satisfy the first two conditions. A binary semaphore or lock allow us to satisfy the third condition.

```
while (1) {
    produce an item
    wait(empty_count)
    acquire(lock);
    insert item in buffer
    count++;
    release(lock);
    signal(full_count)
}
```

```
while (1) {
    wait(full_count)
    acquire(lock);
    remove item from buffer
    count--;
    release(lock);
    signal(empty_count)
    consume an item
}
```

This implementation works. We don't lose wakeups because of the semaphore keeping track of them!

5.6 Reader-Writers

Overview

- An object is shared among several threads
- Some threads only read the object, others only write it
- We can allow multiple readers but only one writer
- Writer can proceed if there are no readers or writers/Reader can proceed if there are no writers

5.6.1 Semaphores

Initializtion

```
int read_count = 0;
semaphore mutex = 1;
semaphore block_write = 1;
```

Writier

```
write() {
    wait(block_write);
    do writing;
    signal(block_write);
}
```

Reader

```
read() {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(block_write);
    signal(mutex);
    do reading;
    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(block_write);
    signal(mutex);
}
```

Note: this approach is not fair since a bunch of read calls can potentially starve the writers.

5.7 Implementing Semaphores

Use a **queue** to block waiters, **guard** on lock, and a **count** of waiters

```
struct semaphore {
    int count = 1;
    bool guard = False;
    queue Q;
}
```

```
void wait(s) {
    disable interrupts;
    while (test_and_set(&s->guard));
    if (s->count <= 0) {
        put current thread on s->Q;
        s->guard = False;
        block current thread;
        void signal(s) {
        disable interrupts;
        while (test_and_set(&s->
            guard));
        if (s->Q is empty)
        s->count++;
    }
    s->count--;
    s->guard = False;
    enable interrupts;
}
```

```
void signal(s) {
    disable interrupts;
    while (test_and_set(&s->guard));
    if (s->Q is empty)
        s->count++;
    else
        move a waiting thread to the
             ready queue;
    s->guard = False;
    enable interrupts;
}
```

6 Condition Variables

6.1 API and Components

Makes it possible to go to sleep inside a critical section, by atomically releasing the look at the same time a thread goes to sleep

- wait() or sleep() in Nachos: release the lock, go to sleep, wake up and re-acquire the lock when signaled
- signal()/wake() or notify() in Nachos: wake up a waiting thread if any
- broadcast()/wakeAll() or notifyAll() in Nachos: wake up all waiting threads, if any

Condition variables are used in conjunction with locks, so a lock must be specified upon creation. Condition variables can be used to implement semaphores and vice verse.

6.2 Signal Semantics

6.2.1 Mesa semantics

- Signaler keeps the lock and continues running
- waiter is put on the ready queue
- Used by Nachos, most real operating systems

6.2.2 Hoare semantics

- Signaler passes the lock to the waiter
- Waiter runs immediately

6.3 Produce-Consumer/Bounded Buffer

Below is a solution to producer-consume problem using condition variable. Under the conditional where we check count, we use a while loop for Mesa semantics and could use an if statement for Hoare semantics.

```
while (1) {
    produce an item
    acquire(lock);
    while (count == N)
        wait(not_full);
    insert item in buffer
    count++;
    if (count == 1)
        signal(not_empty);
    release(lock);
}
```

```
while (1) {
    acquire(lock);
    while (count == 0)
        wait(not_empty);
    remove item form buffer
    count--;
    if (count == N-1)
        signal(not_full);
    release(lock);
    consume item
}
```

6.4 Common Pitfalls with Condition Variables

- CVs can't be tested, so you can't do cond_var!=true (need to maintain seperate flag)
- Do not release the lock before using the CV
- Need to hold the lock when testing the condition since involves shared variable

7 Monitors

7.1 Details

- A monitor is a programming language construct that controls access to shared data (synchronization code added by compiler)
- A monitor protects its data from unstructured access
- Guarantees mutual exclusion
- Threads can use condition variables within a monitor (if thread blocks within monitor another one can enter)

7.2 Producer/Consumer with a Monitor

```
Monitor produce_consumer {
    Condition not_full;
    Condition not_empty;

    void put_resource() {
        ...
        wait(not_full);
        ...
        signal(not_empty);
    }

    void get_resource() {
        ...
        wait(not_empty);
        ...
        signal(not_full);
    }
}
```

8 Deadlock

8.1 Conditions for Deadlock

A deadlock can exist if and only if the following conditions hold simultaneously:

- **Mutual Exclusion:** a resource is assigned to at most one thread at once
- **Hold and wait:** threads holding a resource can request new resources while continuing to hold old resources
- **No preemption:** resources cannot be taken away once obtained
- **Circular wait:** one thread waits for another in a circular fashion

8.2 Prevention

8.2.1 Ignore

The Ostrich Algorithm
If there is a deadlock, we can reboot/remove device(driver deadlock)/terminate application.

8.2.2 Prevent

Ensure that at least one of the conditions for deadlock cannot occur. We can make resources sharable (not always possible), don't hold and wait (lock all resources at beginning), OS can preempt resources (costly), or impose order on all resources to prevent circular wait.

8.2.3 Avoidance

Specify in advance what resources will be needed by threads so system only grants resources if it knows the process can obtain all resources it needs. (Hard to determine all the resources needed in advance)

8.2.4 Detection and Recovery

We can detect deadlocks by looking for cycles in resource graph but detection algorithm can be costly with many threads. Once a deadlock is detected, we can recover by aborting threads and preempting resources.

8.3 Dining Philosophers

Group of philosophers are in a circle and have a fork in between all of them. Eating requires two forks, and we can enter a deadlock situation if the philosophers all pick a fork on the same side (left or right).
Solution: Assign a number to each fork and have each philosopher take the highest number fork first

8.4 Resource Allocating Graph

- Thread A holds resource R

$$\begin{tikzcd} A & R \arrow[l] \end{tikzcd}$$

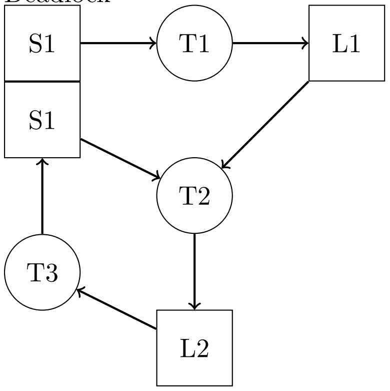
- Thread B requests resource S

$$\begin{tikzcd} B \arrow[r] & S \end{tikzcd}$$

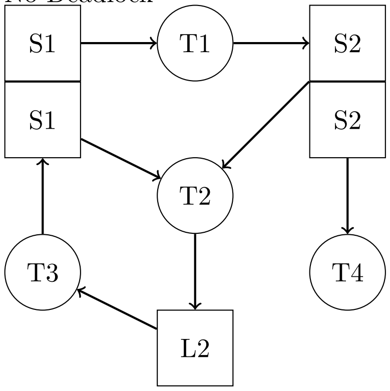
- If the graph has a cycle: deadlock **may** exist
- No cycles: no deadlock
- represent multiple resources with multiple boxes (semaphores)

8.4.1 Example

Deadlock



No Deadlock



9 CPU Scheduling

9.1 Basics

CPU Scheduling aims to effectively share the CPU by time slicing the CPU. The scheduler moves threads between queues and states.
Scheduling mechanisms

- Context switching
- Thread queues
- Timer interrupts

Scheduling policies

- What thread should run next and for how long?

When does the scheduler run?

- Interrupts/exceptions
- when a thread terminates
- when a thread switches from running to waiting or ready

Scheduler Goals

- Turnaround time $T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}}$
- Throughput
  - Jobs per second
  - Minimize context switches
  - Use system resources (disk, network, CPU, etc..) efficiently)
- Fairness (no starvation or deadlock, fair access to CPU)

Different applications have different goals

- Interactive apps - minimize response time
- Batch applications - high throughput, low turnaround time

Preemptive vs Non-Preemptive

- Preemptive: timer interrupts; OS takes control of CPU from time to time and can interrupt a running job
- Non-preemptive: scheduler waits for a block/yield

9.2 First-Come First-Severed/FIFO

- Schedule jobs in order they arrive
- Non-preemptive

Pros:

- Simple
- jobs treated equally
- no starvation (assuming finite runtime)

Cons:

- Short jobs can get stalled behind long jobs

9.3 Shortest Job First

- Run job with shortest remaining time first
- Preemptive

Pros:

- If jobs all come in the beginning, turnaround time is minimized

Cons:

- Cant preempt long jobs
- Long jobs may be starved
- Difficult to predict runtime

9.4 Shortest Remaining Time To Completion First

- Run job with shortest time first
- Non-preemptive

Pros:

- optimal for reducing average turnaround time

Cons:

- Difficult to predict runtime
- Can starve long jobs

9.5 Round Robin

- Preemptive
- Each job runs for a time slice (quantum)
- Ready queue is a circular queue

Pros:

- Short response time
- Fair
- No starvation

Cons:

- Lots of overhead (context switches)

9.6 Examples

9.6.1 Round Robin vs. FCFS

If jobs are all equal in runtime, FCFS will have a lower turnaround time than round robin **If long jobs are queued before short jobs**, FCFS will have a higher turnaround time than round robin

9.7 Priority queue

- Assign each job a priority (Use FIFO for jobs of equal priority)
- Can be preemptive or non-preemptive

Pros:

- Flexible

Cons:

- Low priority jobs can be starved
- Who sets priorities?

9.8 Multilevel feedback queue

- Multiple queues w/ a different priority
- Jobs start at highest priority
- If quantum expires without voluntary yield/block from thread, then drop one priority level
- If thread yields/blocks before quantum expires, then stay in priority level or go up

Pros:

- No starvation, flexibility

Cons:

- Complexity
- Requires parameter tuning for optimal performance

9.9 Scheduling Overhead

OS aims to minimize overhead
Useful work: CPU doing something
Useless work: context switching (time for scheduling decision + contxt switch)
Typical scheduling quantum time: 1 ms
Typical context switch time: 1 μs

9.9.1 CPU Utilization Metric

Fraction of time spent doing useful work:
$\frac{\text{Time spent doing work}}{\text{Time spent doing work + Time Making Scheduling Decision + Time Context Switching}}$

10 Memory Management

10.1 Overview

10.1.1 Memory Management Challenges

- Finite memory capacity
- Locating data in memory
- Protection
- Efficiency

10.1.2 Memory Management Goals

- Multitasking: allow multiple processes to be in memory at once
- Transparency: abstraction for programming, processes should not know memory is shared, processes should run regardless of the number/locations of other processes
- Isolation/Protection: processes shouldn't be able to corrupt each other
- Efficiency: CPU and memory utilization shouldn't be significantly degraded with sharing memory

10.1.3 Multi-tasking with static relocation

- highest memory holds the OS and when a process is loaded a region of memory is allocated for it
- Cons: No protection, low memory utilization, no sharing, and entire address space needs to fit in memory

10.1.4 Dynamic Memory Relocation

- processes use virtual addresses to refer to memory locations
- translated to physical addresses during memory translation
- Benefits
  - Flexible: OS can move processes around in memory as they execute
  - Transparent: hardware handles address translation
  - Protection: check for isolation during address translation
  - Efficiency: use memory efficiently

10.1.5 Address Translation

Base and Bound

- 2 hardware registers (base and bounds) set for each process
- Process can only access memory in [base,base+bound)
- Benefits
  - Simple fast translation
  - Can relocate segment at execution time
  - Protection through bound register
- Limitations
  - External fragmentation: wasted memory between segments
  - Internal fragmentation: wasted memory within a segment
  - Cannot share memory bewteen processes

Segmentation

- Split each virtual address space into multiple segments
- Each process has a segment map that holds base and bounds/permissions for each segment
- On context switch, save/restore pointer to table
- Benefits
  - Process memory can be split among several segments
  - Flexibility: segments can be assigned, moved, grown, or shrunk independently
- Limitations
  - Still have fragmentation
  - Large segment tables can be complex to manage
- See section below about address translation with paging

10.2 Paging

Divide physical and virtual memory into fixed sized pages (both same size)

10.2.1 Advantages

- No external fragmentation
- Easy to allocate memory (allocation a page is just removing it from the list of free pages)
- Easy to swap out chunks of a program (all chunks are same size)
- Use valid bit to detect references to swapped pages

10.2.2 Limitations

- Internal fragmentation still possible
- Massive page tables require lots of memory. Can address this with hierarchical page tables
- Overhead in memory references (2 references per address lookup). Can be solved with a cache

10.2.3 Address translation

Have two parts: Virtual page number (VPN) and offset
Page Tables:

- Map VPN to PFN (physical frame number)
- Include permissions
- Use VPN as index into page table
- Use concatenation instead of addition (possible due to fixed power-of-2 sized pages)
- Per process (context switch causes page tables to be swapped out)

Example Problem setup

- Pages are 4KB
  - Offset is 12 bits
  - Assume 32 bit system
  - Leaves 20 bits for VPN

Virtual address: 0x00007468

- Offset: 0x468
- VPN: 0x7
- Suppose page table entry 0x7 contains 0x2. Then the PFN is 0x2 and the physical address is 0x0002468

10.2.4 Page Table Entry

- Modify bit (has this page been written to?)
- Reference bit (has page been accessed?)
- Valid bit (is the PTE valid i.e. not swapped out)
- Protection bits (which operations are allowed on list)
- Page frame number (physical mem location)

Example: Memory Requirements for Page Tables
Problem setup

- Assume linear page tables
- Assume 4kb pages (12 bits for offset within each page)
- Assume 4 bytes per PTE

With 32-bit address space, how many pages do we get:

- 20 bits for page number, so 2^{20} PTES per process, which is 4mb

With 64-bit address space, how many pages do we get:

- 20 bits for page number, so 2^{52} PTEs per process, which is 16 petabytes ($2^{52} \times 4$ bytes)

Solutions for reducing memory overhead of page tables

- Bigger pages (more internal fragmentation, doesn't entirely solve problem with 64 bit addresses)
- Multi-level page tables (solution most often used)

10.2.5 Multi-level Page Tables

Split virtual address into multiple parts
Two level example

- Directory
- Secondary Page Number
- Offset

Address translation

- Locate root directory from first bits of virtual address and find the right page table from the directory table
- Locate the VPN from the second part of the virtual address and match it with its PFN pair
- concatenate the offset and PFN to get the physical address

Example

- 4kb page - 12 bits for offset
- We want the directory in one page: 4kb / 4 bytes = 1024 entries in directory; 10 bits for directory index
- 10 bits left for secondary page index

Can omit empty pages, generalize further, etc... Example: x86-64 utilizes a 4 level page table with the first 16 bits unused, 12 bit offset, and 9 bits for each directory

10.2.6 Storing page tables

- Too large to store in MMU
- Store in memory
- Special register stores address of highest level page table

OS has its own virtual address space, extending the user level virtual memory address space (its on top of the user level virtual address space)
In user mode, only the user address space can be accessed. In kernel mode, can access entire address space

10.3 TLBs and Swapping

10.3.1 Translation Lookaside Buffer (TLB)

- Takes advantage of locality since in a short period of time programs tend to access the same page repeatedly
- A small hardware cache of recently sued translations where each cache entry stores a VPN and corresponding PTE
- For address translation, check for hit in TLB otherwise need to traverse page table
- Random, Least Recently Used (LRU) are example policies for replacing TLB entries

Handling Misses

- Hardware (MMU) [x86, ARM, RISC-V]
  - Knows where page tables are in main memory
  - Hardware parses page tables and loads PTE into TLB
  - Page tables need to match a hardware defined format (inflexible)
- Software (OS) [MIPS, Alpha, Sparc, PowerPC]
  - TLB faults to OS, OS finds PTE and loads it into TLB
  - CPU ISA has instructions for manipulating the TLB
  - Tables can be any format (flexible)

10.3.2 Manging TLBs

- Context Switches
  - invalidate all TLB entries
  - Tag each entry with an Address Space Identifier (ASID)
- Consistency
  - When OS modifies a PTE it needs to invalidate the PTE if in the TLB
  - On multi-core CPUs must invalidate PTE entries on all cores since there is a TLB on each core

10.3.3 Demand Paging

- Takes advantage of locality
- Each virtual page is either in memory or on the disk
- Allocate page in memory when first accessed
- Swap pages in from disk when they are accessed
- Page Faults
  - When a process references a page that is in the swap file, a page fault is triggered and trap to OS
  - In page fault handler, OS find free page in physical memory, reads page in from swap file, updates valid bit, and returns to process
  - MMU stores faulting address
  - If no available page frames evicta page to disk
  - Store information of position in swap file in PTE
- Can use prefetching to predict which pages

10.3.4 Shared Memory

- Shared memory allows processes to access the same memory
- There are PTEs in both processes that map to the same physical memory
- Can be shared at the same or different virtual adresses

10.3.5 Copy on Write

- Avoid copying until needed
- Share pages as read only initially using shared mappings
- Only copy when a process tries to write to the page (protection fault handler)

10.3.6 Mapped Files

- Map a file into the virtual address space
- Process can access file in the same way as memory

10.4 Page Replacement and Memory Allocation

Goal: Maximize hit rate and minimize page faults
Challenges:

- OS cannot predict pages accessed in the future
- Minimize overhead (make common case, a page hit, fast, minimize overhead in rare case of page miss)

Optimal Algorithm: Belady's algorithm
Evict page that will be accessed farthest in the future
Pros

- Provably optimal

- Can be used as a benchmark for other algorithms

Cons

- Requires you know all future page accesses
- No real implementation

Random Randomly choose the page to evict **Pros**

- Does not require knowledge of future accesses
- Easy to implement

Cons

- Could be slow

FIFO Maintain a list of pages in order they were brought in
On replacement, evict oldest page
Pros

- Low overhead

Cons

- May replace heavily used pages

Belady's anomaly Does more page frames result in fewer page faults?
No, with some replacement policies (e.g. FIFO), more page frames does not result in fewer page faults
Locality

- Temporal locality - Locations referenced recently are likely to be referenced again
- Spatial locality - Locations near referenced memory locations are likely to be referenced

Goal: leverage temporal locality in replacement algorithm to improve page replacement policy **Least Recently Used** Replace page that hasn't been used for the longest time (order pages by time of reference)
Pros:

- Good approximation of min

Cons:

- Need to time stamp every mem access; high overhead

But LRU can be approximated using the reference bit
Clock Algorithm

- Arrange PFNs in circle
- Clock hand points to oldest page
- Sweep through pages in a circular order. If reference bit is 0, evict the page. Else, set the bit to 0 and advance the hand

Prons:

- Simple to implement
- Considers how recently a page was used

Cons:

- Worst case may take a while to evict a page

Asynchronous vs synchronous page eviction
Synchronous execution

- in page fault handler, run algorithm to evict a page
- Might require writing changes to disk to first

Asynchronous execution

- A background thread maintains a pool of clean, unused pages
- Occasionally evict more pages to keep pool large enough
- Allows batching of page evictions for efficiency

Multiprocessing
Global Replacement (Most common nowadays)

- All pages are in a single replacement pool
- Pros: process' set of pages grows and shrinks dynamically
- Cons: processes compete for page frames

Local Replacement

- Each process has a separate pool of pages
- A page fault in one process can only replace one of its own frames
- Pros: eliminates interference between processes
- Cons: how many pages should be in each pool

Working Set Model
Used to model dynamic locality of memory usage (defined by Peter Denning in 60s)
Working set: set of pages used over last T time units
Working set size: number of unique pages in working set
Working set changes as program executes
Challenges:

- How do we pick T?
- How do we efficiently track this information?
- How do we know when the working set changes?

Because of these challenges, its not used in practice
Thrashing
Memory overcommitment: Pages used by process don't all fit into physical memory
Thrashing: Swapping in and out all the time; maxing out I/O usage. Causes processes to be blocked as processes wait for pages to be fetched from disk
Solutions:

- Buy more ram
- Kill processes

Virtual Memory Allocation At any given time, some of a processes' virtual address space is mapped and some is not (OS needs to know this so it can raise a segmentation fault)
How can a process allocate more virtual memory?

- Grow the stack
- Grow the heap

Growing the stack Grows linearly downwards
To grow the stack:

- Process tries to grow the stack
- Accesses unmapped memory
- Triggers page fault
- Page fault handler allocates a new page and zeros it out
- Process resumes

Growing the heap Grows and shrinks with malloc/free
Allocation and freeing are unpredictable
Memory has allocated areas and free areas (which can lead to fragmentation)
Heap management:

- Maintain a free list of holes
- Managed by a library (e.g. libc)

Growing the heap

- Use syscall like brk or mmap
- brk: grow heap by certain size
- mmap: allocate chunk of virtual memory, starting virtual address can be anywhere

11 Files

11.1 Storage Devices and File System API

Hard Disk Drives

- Consists of multiple platters with a series of circular tracks on each
- An actuator controls and arm with a head that reads data off the disk

- A sector is a small chunk in a track, and sectors that are alignes with each other over multiple platters consist of a cylinder

Disk Performance

- Seek: move the disk are to the correct cylinder
- Rotations: waiting for the sector to rotate to the head
- Transfer: transferring data from surface into disk controllers, sending it back to host

Disk Latency = seek + rotation + transfer

API between disk and OS (old)

- specifying disk requests requires a lot of info such as surface, track, sector, track size
- Older disks required OS to specify all of this
- Con: OS needs to know all disk parameters and modern disks are more complicated

API between disk and OS (modern)

- Modern disk provide higher level interface
- Disk exports its data as a logical array of blocks (block interface)
- Pro: simpler API
- Con: disk parameters are hidden from OS and harder for OS schedule

Disk Scheduling Policies

- First in First Out
  - Pro: fair, no out of order requests
  - Con: long seeks/low throughput
- Shortest seek time first
  - Pick the closest request on disk, minimize arm movement
  - Pro: tries to minimize seeks time
  - Con: starvation, favors middle blocks
- Elevator (SCAN)
  - Pick the closest requests in direction of travel
  - Pro: bounded time for each request
  - Con: requests at the other end will take a while

Flash Bases Solid State Drives

- No physical moving parts, 100x speed, and more reliable
- cost 5-10x more per bit
- Flash sufferes from where out
- Block interfaces and file systems typically remained unchanged

Non-Volatile Memory

- Phase change (PCM), spin-torquetransfer (STTM), Intel Optane

- performance close to DRAM but persistent
- Byte addressable
- Require both OS and applications to adapt

Redundant Array of Inexpensive Disks (RAID)

- Use redundancy to increase throughput and reliability
- Raid 0
  - Striping: split data across multiple disks and read/write in parallel
  - Benefit: improves throughput
- Raid 1
  - Mirroring: Store redundant information on a mirror/shadow disks (write to every disk read from 1)
  - Benefit: improves read throughput and reliability
- Raid 4
  - Parity Disk: Store XOR of the data disks, and data can be recovered using other disks and parity disk
  - Benefits: improves throughput and reliability

File System Components

- Naming: how to refer to data with files and directories
- File access: read, write, and other operations
- Disk Management: how to allocate and arrange data on the storage device/map data to blocks
- Protection and permissions: protect data from different users
- Reliability and durability:

Files

- a named collection of bytes stored on durable storage such as a disk
- Properties: size,owner, last modified time, permissions, etc
- Types
  - Understood by file system: link, character, block, etc
  - Understood by parts of OS/runtime libraries: text, source, object, executable, application-specific, unspecified
  - Encoded in file name: .zip/.jpg/.exe/.txt/.pptx/.o/.sh
  - Encoded in contents: magic numbers, initial characters(#! for shell scripts)

File Access Patterns

- Sequential: read bytes one at a time in order
- Random Access: data accessed in random order
- Indexed Access: file system contains in index to blocks with paticular contents

Directories

- A structured way to organize files
- Convenient naming interface
- file systems support multi-level directories
- OS supports notion of current directory (relative to current directory)
- Contains a list of entries (name,location)
- For path translation start at / directory and walk path (seperate open from read/write so we don't need to traverse path each time)

Protection

- UNIX Access Rights
  - Mode of access: read, write, execute
  - Classes of users: owner, group, public
  - Can create a group and add users to group to specify access policies for that group
  - ls-al to list out files with their permissions
- Root and Administrator
  - root (Unix) administrator (Windows) bypasses all protection checks in kernel
  - Always running in root can be risky

11.2 File System Disk Layout

File System Challenges

- Files grow and shrink
- Disk access is not uniform, efficiency is challenging
- Failures

Workloads influence design

- Most files are small (8kb)
- Most of disk is allocated to large files

Access patterns

- Sequential vs. random
- Access files in same directory together (spatial locality)
- Access metadata at same time as file (need metadata to find file)

Block Layout Partition disk into fixed file system blocks

- Typically 4kb in size
- Independent of disk physical sector size (if a sector is 512 bytes, file system will use 8 sectors/block)
- One file can span multiple disk blocks
- A small file still uses an entire block

Contiguous Layout Allocate a contiguous set of blocks to each file
File metadata

- Location of first block on disk
- Number of blocks

Pros:

- Simple
- Easy access, both sequential and random
- Few seeks for I/o

Cons:

- Difficult to grow files
- As files are created and deleted, fragmentation can occur

Linked Layout
Allocate a linked list of blocks
File metadata

- Location of first block on disk
- Each block contains pointer to next

Pros:

- Can grow files dynamically
- No fragmentation

Cons:

- Random access is slow
- Sequential bandwidth may not be good
- Unreliable: If we lose on block, we lose the rest

Indexed layout Use index block to store pointers to data blocks
File metadata:

- Location of index blocks on disk
- Index block contains pointer to data blocks

Pros:

- Can grow files dynamically
- No fragmentation
- Easy random access (after reading the index block)

Cons:

- What if one index block isn't big enough?
- Sequential bandwidth may still not be good

Multi-level Indexed Layout Use special index block to store pointers to data blocks and indirect blocks to store more pointers to datablocks
File metadata:

- Location of index block on disk
- Index block contains pointers to data blocks and indirect blocks

Pros:

- Can support much larger files

Cons:

- There is still a limit on maximum file size
- Sequential bandwidth may still not be good

Unix inode
Each file is associated with an inode on disk, and each inode has a unique inode number
An inode stores all the metadata for a file

- file size
- data blocks
- user and group of file owner
- Protection bits (user/group/other, read/write/execute)
- Link count (how many directory entires point to this inode)
- Timestamps (created, modified, last accessed, any change)

Use an unbalanced indexed structure

- Each inode contians 15 block pointers
- First 12 are datablocks
- 1 points to a single indirect block
- 1 points to a double indirect block
- 1 points to a triple indirect block

inodes are small (256 bytes each) and one inode block may contain many inodes
Superblock Stores pointer to inode of root directory, which is the basis for translating all path names. Because of this, it is placed in a predetermined location on disk
Free Map Blocks Store a bitmap, one bit per block which indicate which blocks are allocated. There is one bitmap for datablocks and one bitmap for inode blocks **Bitmap vs. Linked List**

- Bitmap pro: Bitmaps make it easy to find contiguous blocks
- Bitmap con: Need extra space to store a bitmap
- Linked list pro: no wasted space for bitmap (use free blocks to store free block list)

- Linked list con: harder to find contiguous blocks

Hard Links Directory entry that associates a name to a file
Uses ln command or link syscall
Deleting a link may or may not delete a file (if reference count is 0, file is deleted)
Limitations:

- Users cannot create hard links to directories
- Cannot create hard links to other file systems

Soft Links A file whose contents contain another file path (additionally, the inode is marked as a soft link)
Created via ln -s or symlink
Enables aliasing
Limitations:

- Slower than hard links

Optimizing file operations Create Allocate an inode (initialize metadata and update inode bitmap)
Allocate a directory entry
Only allocate data blocks when process starts writing (update inode to point to data blocks and update data block bitmap) **Rename** Rather than creating a file with a new name, copying contents of an old file, and deleting the old file, we can create a directory entry with the new name and delete the old one (rename syscall)
Delete Decrement reference count of file (hence unlink syscall) and if there are no remaining links, updating the inode and data block bitmaps (but don't erase data)

11.3 File Caching and Reliability

11.3.1 Original UNIX File System Disk Layout

- Simple Disk Layout
  - Inodes are in an array in outermost tracks
  - Data blocks are on inner tracks
- Con: fixed number of files

11.3.2 BSD Fast File System

- Increased block size of 4KB
- Bitmap instead of a free list

Disk Cylinders: data in a given track number across surfaces of all platters (can access data in cylinder without seek)
Cylinder Groups: a group of consecutive cylinders
FFS Disk Layout

- Divide the disk into cylinder groups
- Include all file system data structures within each group (replicate superblock, include bitmaps, inodes, etc)
- Group file data together with its directory
  - Place file inodes in the same group as their directory
  - Place file data blocks in the same group as their inode
- Balance directories across cylinder groups (find group with few directories and many free inodes)
- Modern file systems use block groups instead of cylinder groups

11.3.3 File Buffer Cache

- Applications exhibit significant locality for reading and writing files
- file buffer cache caches file blocks into memory to capture locality
  - cache all kinds of blocks (super blocks, inode blocks, data blocks, etc)
  - reading from cache makes disk behave like memory
  - shared by all processes
- Unified page cache with file buffer cache (shared replacement policy)
- Reads
  - If in cache, copy from buffer cache to user buffer
  - If not in cache, evict if necessary, read into buffer cache, copy into user buffer
- Writes
  - If in cache, write data from suer buffer to buffer cache
  - If not in cache, evict if necessary, read file block into buffer cache, write data from user buffer
  - Write-through caching: write to storage immediately (simple to implement, but application has to wait with increased number of writes)
  - Write-back caching: write in memory then write buffered data back to device in intervals (fast writes with batching and reduced number of writes, but can potentially lose data)

11.3.4 File System Reliability

- accidental or malicious deletion of data/disk failure/system crash or loss of power
- Crash Consistency Problem
  - File system can be left in inconsistent state as file system operations involve writing to multiple blocks
  - Only write data: as if we never tried to append block
  - Only write inode of file: could read garbage data/file system is inconsistent as bitmap and inode disagree
  - Only write the bitmap: space leak as block isn't being used
- Check and repair with fsck which is the file system checker in Unix
  - Scan file system for inconsistencies
  - Repair/notify admin of inconsistencies
  - Cons: very slow especially for large disk volumes, may not store data to valid state, and can pose security issues
- Ordered writes
  - Prevent inconsistencies by writing blocks to disk in a safe order
  - initialize blocks before writing pointers to them
  - Nullify existing pointers to a block before reusing it
  - Set a new pointer to a resource before clearing the last one
  - Pro: no need to wait for fsck
  - Con: can leak space so run fsck in background, must wait for writes to compelte, difficult to find safe sequence

  - Safe Orders: data-¿bitmap-¿inode/itmap-¿data-¿inode

- Journaling
  - Write down what you are going to do before you do it
  - append-only file containing log records
  - start transaction/blocks/commit transaction
  - Transaction is only final after commit block is written
  - Checkpointing: record current position in log, copy all modified blocks to final destinations on disk, can clear log before recorded position
  - Crash recovery: replay all transactions that have committed but not check pointed and discard uncommitted transactions
  - Pro: much faster recovery, eliminate inconsistencies, log is written sequentially (no seeks), can delay writes to improve performance
  - Con: have to write data twice

12 Virtual Machines

Abstractions for Processes

- Virtual memory
- System calls
- Most instructions in ISA
- Most Registers

Abstractions for Virtual Machines

- Physical memory
- Interrupts
- All instructions in ISA
- All registers
- I/O devices

12.1 Virtual Machine Monitor/Hypervisor

- Virtualizes entire physical machine
- Presents a HW interface to the guest OSes above
- Provides illusion to each guest OS that it has full control of the HW (VMM controls hardware)
- Isolates VMs from each other

12.2 Goals

- Isolation
- Efficiency
- Fidelity (everything works the same with little modification
- Manageability (ease of creation, provisioning, deletion)
- Performance (minimize overhead)
- Scalability (support many VMs at once)

12.3 Techniques

Simulation: Simulate instruction execution (memory, I/O, etc...). This is super slow **VMM:** Run VMM in kernel mode and guest OS in user mode, allowing most insturctions to run at regular speed. If the guest OS needs to perform a priviliged instruction, it must trap to the VMM, which simulates the appropriate behavior (trap and emulate)

12.4 Type 1 and Type 2 Hypervisor

Type 1 hypervisor: Runs directly above hardware. All VMs run on top of it
Type 2 hypervisor: Runs above a host OS. All VMs sitll run on top of it, but the hypervisor no longer has direct access to the hardware

12.5 Challenges with X86 virtualization

Not fully virtualization initially, because some instructions run differently in user/kernel mode and a VMM can not easily manage a TLB miss (since the TLB is hardware managed)

12.6 Ways to virtualize X86

- Para-virtualization: Directly modify guest OS to better cooperate with VMM, sacrificing transparency for better performance
- Binary translation: translate privileged instructions at runtime (incurs overhead)
- Rely on hardware support (i.e. Intel/AMD added a new root mode; Guest OS has kernel/user mode in non-root mode, and hypervisor has kernel/user mode in root mode)

12.7 Virtualizing Privileged Instructions

For privileged instructions that cause a trap, we trap to the VMM, handle the instruction and return to the guest OS. For privileged insturctions that do not trap:

- With para-virtualization: modify guest OS to call into VMM
- With binary translation: rewrite instruction to call into VMM at runtime
- With HW support: add new CPU mode/instructions to support trap-and-emulate

12.8 Virtualizing Events

VMM recieves interrupts and faults (faults/syscalls), and we need to vector events to the right syscall

- With para-virtualization: VMM notifies guest OS using an event queue
- Binary translation: call into guest OS from VMM
- Hardware support: HW delivers event directly to guest OS

Example: read() syscall with binary translation

- Instruction is translated at runtime to call into VMM
- VMM calls guest OS trap handler, which leads to execution of a syscall and causes a trap to the VMM (VMM needs to have hardware read from the file)
- The VMM directly returns to the process

12.9 Virtualizing CPU

Simply reuse scheduling techniques. Here, we typically timeslice VMs and use a scheduler like round-robin

12.10 Virtualizing memory

Challenges:

- VMM needs to assign hardware pages to VMM
- Hardware-managed TLBs: hardware will walk page tables with no opportunity for VMM to run

Shadow page tables:

- VMM maintains shadow page table for each VM
- Shadow page tables map from virtual pages in VM directly to physical pages allocated by VMM
- In essence, the guest OS page table is ignored by the hardware but is important for maintaining transparency and allowing the guest OS to operate without significantly changing how it manages memory

MMU points to the shadow page tables. When a VM tries to change the MMU to point to a different page table, it traps to the VMM which updates the MMU to point to the shadow page table. To keep shadow page tables in sync with guest OS page tables, we mark pages of the guest OS page table as read only so when the VM tries to write to the page table, it traps to the VMM which updates the shadow page table

12.11 Virtualizing I/O

Challenges

- Lots of I/O devices
- Do not want virtualized device drivers for all possible I/O devices

Approach 1

- Run real device drivers in VMM
- VMM provides simple virtual I/O devices to guest VM
- Can optimize using paravirtualization or specialized hardware

12.12 HW support for virtualization

- Root/non root mode (VMM runs in root mode, guest OS in non-root mode)
- Interrupts are delivered directly to the right guest OS
- Virtualize IO devices with SR-IOV (allows a hardware device to appear as multiple devices)
- Intel extended page tables support virtualization of page tables (removes need for shadow page tables. Guest OS maintains map of guest virtual to guest physical memory mappings. EPT add another MMU from guest physical memory addresses to the true physical memory addresses, which are set up by the VMM. Instead of trapping, we then just do direct HW translation (adds another translation but still less expensive than trapping))

12.13 Containers

Containers virtualize OS abstractions

- Group of processes that appear to have an entire OS to themselves
- Manage isolated sets of OS objects (using namespaces)
- Manage isolated sets of resources (using resource management)

Lightweight virtualization
Containers on the same machine share an OS
OS implements a collection of namespaces

- PID: processes
- User IDs: users and groups
- Network: IP addresses/network ports/routing
- File system: files and directories

Each container has its own namespace, providing isolation
Implementing namespace isolation
OS tags objects with the namespace that they belong to (process has both a PID and namespace identifier)
Mappings map container-local ids to a global perspective

- OS tracks processes on a global list
- OS maps from PID within a container to a process on a global list

Filters restrict which objects are visible in a namespace
Resource management
OS manages processes at a process granuarlity by default
Containers manage resources among a set of processes

- Which cores can be utilized, how much of CPU is used, how much memory is used, how much disk and network I/O is used

13 Protection

Mechanisms that prevent accidental or intential misuse of the system

- Authentication: identify a responsible party behind each action
- Authorization: determine which parties are allowed to perform which actions
- Enforcement: control access using authentication and authorization

13.1 Protection Principles

- Permission rather than exclusion (default is no access)
- Check every access to every object (including instructions/memory references)
- Principle of least privilege (only execute with privileges you need)
- User interface to protection must be easy to use (

13.2 Users

- Protection starts with the concept of a user
- Which user you are defines which programs you can run (execute) and which files you can access and how (read,write)
- Cannot do anything on the system until you log in
- Once logged in, everything in the system is performed under your user ID

13.3 Root and Administrator

- Bypasses all protection checks in the kernel
- Running as root can be dangerous (a mistake or exploit can harm the system)
- sudo command runs a process with root privileges (user must be in sudo group)
- su command runs a shell with root priveleges (authenticate using the password for the root user)

13.4 Authorization

- Determines who is allowed to perform which actions
- Subjects: who is performing the action (uers,process)
- Objects: what the action is being performed on (file)
- Actions: what the subject is allowed to do to the object

13.5 Access Control List and Capability List

- Access Control List: (organized by columns) maintains a list of which users are allowed to perform which actions
- Capability List: (organized by rows) for each subject maintains a list of objects and their permitted actions
- Capabilities are faster to check and easier to transfer (like keys, easy to hand off and fast to check)
- ACLs are slower but easier to use (slow to check, object centric and easy to express protection goals)
- OSes use ACLs on objects in the filesystem
- OSes use capabilities when checking access frequently (memory references)

13.6 File System Protection

- For each read/write operation, the OS needs to verify that the process has permission to perform the syscall
- Checking ACL on each read/write is expensive, so Open syscall does path translation and permission check
- Use file descriptors as capabilities (process passes descriptor on each read/write and OS checks descriptor/if action is valid)

13.7 Virtual Memory Protection

- address space defines permissions for a process under execution
- Page table entries are our virtual memory capabilities
- Loading a process and creating the address space
  - Code Pages: set PTE protection bits to read only and execute
  - Data Pages: set PTE protection bits to read/write and not execcute
- As process executes
  - Stack and Heap Pages: set PTE protection bits to read/write and not execute