Graduate Intro. to OS - CS6200

Introduction

This notebook contains my personal notes for CS6200: Graduate Introduction to Operating Systems at the Georgia Institute of Technology. Please find below a description of the course.

Introduction to Operating Systems is a graduate-level introductory course in operating systems. This course teaches basic operating system abstractions, mechanisms, and their implementations. The core of the course focuses on OS support for concurrency (threads) and synchronization, resource management (CPU, memory, I/O), and distributed services. The practical component of the course teaches multithread programming, inter-process communication, and distributed interactions via RPC.

P1: Lesson 1

Theory and Practice

This course covers a multitude of topics to prepare students for Advanced Operating Systems. This course will provide students both the theory of operating system design, as well as an opportunity to put these concepts into practice through programming projects.

This course will cover, at a minimum, these topics:

- Threads, concurrency, and synchronization.
- Single-node OS mechanisms
- Inter-process communication, scheduling
- Multi-node OS mechanisms
- Remote procedure calls (RPC)
- Experimental design and evaluation

Recommended Reading

- Operating System Concepts
- Operating System Concepts: Essentials
- Modern Operating Systems
- Operating Systems: Three Easy Pieces

P1: Lesson 2

Preview

This lesson covers these topics:

- What is an Operating System?
- What are key components of an Operating System?
- Design and Implementation considerations of Operating Systems.

What is an Operating System?

A special piece of software that...

- Abstracts
    - Hides the underlying hardware from the user
- Arbitrates
    - Manages hardware resources for the user

...the use of a computer system.

An Operating System is a layer of software that:

- Has direct, privileged access to the underlying hardware
- Hides the hardware complexity
- Manages hardware on behalf of one or more applications according to some predefined policies
- Ensures that applications are isolated and protected from one another

What do Operating Systems do?

Operating Systems...

- Direct operational resources
    - Control the use of the CPU, memory, peripheral devices...
- Enforce working policies
    - Fair resource access, limiting of resource usage...
- Mitigate difficulty of complex tasks
    - Abstract hardware details (system calls)

The role of the Operating System

- Hide hardware complexity
- Resource management
    - Memory management
    - CPU Scheduling
- Provide process isolation and protection

OS Elements

Abstractions

- processes, threads, files, sockets, memory pages

Mechanisms

- creating, scheduling, opening, writing, allocating

Policies

- least-recently used (LRU), earliest deadline first (EDF)

Design Policies

Separation of mechanism and policy

- Implement flexible mechanisms that support many policies
- e.g, LRU, LFU, random

Optimize for the common case

- Where will the OS be used?
- What will the user want to execute on that machine?
- What are the workload requirements?

OS Organization

- Monolithic OS

    - Historically the original OS.
    - Everything any process could require is already loaded into the OS
    - Very large
    - **Pros**
        - Everything is already included
        - Allows for some compile-time optimizations

    - **Cons**
        - Bad for customization, not very portable, hard to manage the codebase
        - Large memory requirements, impacting performance

- Modular OS

    - Specifies interfaces allowing developers to install modules into the operating system
    - **Pros**
        - Maintainability for upgrades
        - Smaller footprint
        - Less resource needs
    - **Cons**
        - Levels of indirection allowing for modularity can impact performance
        - Maintenance can still be an issue

- Microkernel

    - Provides basic services and mechanisms
    - Everything else, file systems, device drivers, etc., will run in User-Mode - requiring a lot of system-calls and inter-process communication
    - **Pros**
        - Small in size, easy to verify working code
    - **Cons**
        - Not very portable as it's usually written for very specialized hardware
        - Complex software development
        - Commonly conducts user/kernel transitions which can be costly

User/Kernel Protection Boundary

Kernel-Mode

- where the Operating System resides
- privileged, direct access to the hardware

User-Mode

- where applications reside

User-Kernel Switch

- User-Kernel switches are supported by the hardware. When a User-Mode process wants to gain privileged access to the hardware, the User-Mode process will set off a **trap**, stopping the User-Mode process and requiring the Operating System to review what caused the **trap** to occur. The Operating System will either grant the User-Mode process access to the Kernel-Mode resource, or terminate the process if the User-Mode process is attempting to do something illegal.

System-Calls

- Operating Systems provide processes a set of system calls to applications as a means to request the Operating System to execute some sort of privileged operation. Examples include:
    - **open** (file)
    - **send** (socket)
    - **mmap** (memory)

Signals

- A **mechanism** for Operating Systems to pass notifications to applications.

System Call Flowchart

Crossing the User/Kernel Protection Boundary

- The Operating System supports User-Mode applications utilizing system calls, triggering traps, and requiring the kernel to handle instructions with the provided arguments/variables. However, every time the transition from User-Mode to Kernel-Mode occurs, the Operating System must load the required memory to conduct the requested system call into the memory cache. This is a costly operation, and comes with overhead in relation to execution time.

Discussion

Memory pages

- Operating Systems map **memory pages** to parts of physical RAM and allocate memory pages to process for use. This allows processes to avoid having to access physical memory directly, addressing their assigned memory page. This **mechanism** is called **mapping**.

Policies

- In regards to memory mapping, processes often only need to use specific parts of their memory to conduct computations. The **least-recently used (LRU)** policy can be utilized by the Operating System to move under-used memory to a secondary memory device like a hard drive.

Quizzes

Which of the following are likely components of an Operating System?

☐ file editor
- [✓] file system

- [✓] device driver
- [ ] cache memory
- [ ] web browser
- [✓] scheduler

For the following options, indicate if they are examples of abstraction (B) or arbitration (R).

- [**R**] distributing memory between multiple processes
- [**B**] supporting different types of speakers
- [**B**] interchangeable access of hard disk or SSD

On a 64 bit Linux-based OS, which system call is used to...

- [**kill**] send a signal to a process?
- [**setgid**] set the group identity of a process?
- [**mount**] mount a file system?
- [**sysctl**] read/write system parameters?

P2: Lesson 1

Processes and Process Management

This portion of the class will cover processes and process management.

Overview of the questions answered by these lectures:

- **What is a process?**
- **How are processes represented by OS's?**
- **How are multiple concurrent processes managed by OS's?**

What is a Process?

Operating systems manage hardware on behalf of applications.

An application can be a program on the disk, in flash memory, etc. (static entity).

A **process** is the state of an application when executing and loaded into memory (active entity).

Types of state:

- Text and data
    - static state when process first loads
- Heap
    - dynamically created memory space created during execution
- Stack
    - grows and shrinks ~ LIFO queue

Process memory is assigned a virtual address that is mapped to physical memory using a page table.

- Address space
    - 'in memory' representation of a process
- Page table
    - mapping of virtual to phyiscal addresses

Breakdown of Process Components

Processes have these major components:

- State of execution
    - program counter, stack
- Parts & temporary holding areas:

    - data, register state occupies state in memory
- May require special hardware
    - I/O devices, etc.

Process Execution State

How does the OS know what a process is doing?

CPU registers keep track of a process's execution:

- Program counter
- CPU registers
- Stack pointer
- ...

The Operating System maintains a **Process Control Block (PCB)** - a data structure that the Operating System maintains for every process it manages.

- The PCB is created when a process is created.
- Certain fields are updated when a process's state changes.
- Other fields change too frequently, so the CPU saves all of the below registers to the PCB before the process is no longer in an execution state.

Contains:

- Process state
- Process number
- Program counter
- Registers
- Memory limits
- List of Open Files
- Priority
- Signal Mask
- CPU scheduling info

Context Switching

The mechanism used by the operating system to switch the CPU from the context of one process to the context of another.

This operation can be expensive!

- Direct costs:
    - number of cycles for loading & storing instructions
- Indirect costs:

    - Cold cache or cache misses.

Process Life Cycle: States

Processes can be running or idle.

When a process is running, it can be interrupted and context switched.

Schedulers dispatch a ready or idle process to execute again.

Process states:

- **New** - created, admission control completed by operating system, allocated a PCB.
- **Ready** - ready to execute, not actually executing, waiting on the scheduler to move this process to the running state.
- **Running** - currently executed. Can be moved to the states of ready, waiting, or terminated based upon interrupts, I/O or event waiting, scheduler dispatches, or exiting.
- **Waiting** - waiting on some event or I/O input/output
- **Terminated** - exiting either with an error or successfully.

Process Creation

//TODO include (img from P2L1 part 15)

Basic mechanisms for process creation:

- Fork
    - copies the parent PCB into the new child PCB
    - child continues execution at instruction after fork
- Exec
    - replace child image
    - load new program and start from the first instruction

What is the Role of the CPU Scheduler?

A CPU scheduler determines which one of the currently ready processes will be dispatched to the CPU to start running, and how long it should run for.

The Operating System must:

- **preempt** - interrupt and save current context
- **schedule** - run scheduler to choose next process
- **dispatch** - dispatch process and switch into its context

Inter-process Communication

These IPC mechanisms:

- transfer data/info between address spaces
- maintain protection and isolation
- provide flexibility and performance

Message-passing IPC:

- OS provides communication channel, like a shared buffer
  - **Pros:**
    - OS manages all of this communication and provides the API
  - **Cons:**
    - Overhead - requires a copy from User Space into Kernel Memory and then back into User Space
- Processes write(send) / read(recv) messages to/from one another

Shared-memory IPC:

- OS establishes a shared channel and maps it into each process address space. Processes directly read/write from this memory.
  - **Pros:**
    - OS is out of the way!
  - **Cons:**
    - Because the OS is out of the way, developers might have to re-write APIs to use the shared memory region correctly.

Quizzes

For the following sentence, check all options that correctly complete it:

When a cache is hot...

- [ ] It can malfunction so we must context switch to another process
- [✓] Most process data is in the cache so the process performance will be at its best
- [✓] Sometimes we must context switch

The CPU is able to execute a process when the process is in which state(s)?

- [✓] Running

- [✓] Ready

- [ ]

- [ ]

- [`init`] On UNIX-based OSs, which process is often regarded as the "parent of all processes"?

- [`zygote`] On the Android OS, which process is regarded as "the parent of all App processes"?

Which of the following are not a responsibility of the CPU scheduler?

- [✓] maintaining the VO queue
- ☐ maintaining the ready queue
- ☐ decision on when to context switch
- [✓] decision on when to generate an event that a process is waiting on

Shared memory-based communication performs better than message passing communication.

- ☐ True
- ☐ False
- [✓] It depends...

The individual data exchange of shared memory-based communication may be cheap, however, the operation of mapping memory between two processes is expensive. The shared memory-based communication setup is only worth it if the shared memory-based communication setup cost is amortized over the amount of messages sent between the two processes.

P2: Lesson 2

Threads and Concurrency

This portion of the lesson will cover:

- **What are threads?**
- **How are threads different from processes?**
- **What data structures are used to implement and manage threads?**

This lesson will use "*An Introduction to Programming with Threads*" by Birrell to explain:

- Threads and concurrency
- Basic mechanisms for multithreaded systems
- Synchronization

What is a Thread?

A thread is...

- an active entity
    - an executing unit of a process
- works simultaneously with others
    - many threads executing within a process
- requires coordination
    - sharing of I/O devices, CPUs, memory, etc.

Process vs. Thread

Threads represent multiple execution contexts. They share all of the same code, data, and files, however, they require their own stack and set of registers. The PCB for a multithreaded process will contain data shared amongst the threads, as well as all of the execution contexts for each thread.

Why are Threads Useful?

- **Parallelization** - speed up of computation
- **Specialization** - Input processing, display rendering, etc.
    - Hot cache! Because each thread runs on a different processor, it is more likey the thread's state will be already loaded into the cache.
- **Efficiency** - lower memory management required & cheaper inter-process communication

Because threads share the same address space, context switching for threads is less costly than context switch for processes.

Multi-threading is very useful to hide latency associated with I/O operations.

Benefits of Multithreaded OS Kernel

Threads can work on behalf of specific applications. OS-level services like daemons or drivers can be implemented in a multithreaded fashion.

What Mechanisms are Needed to Support Multithreading?

- Thread data structures
    - identify threads, keep track of resource usage...
- Mechanisms to create and manage threads
- Mechanisms to safely coordinate among threads running concurrently in the same address space
- Mutual Exclusion
    - exclusive access to only one thread at a time
    - mutex
- Waiting on other threads
    - specific condition before proceeding
    - condition variables
- Waking up other threads from a wait state

Threads and Thread Creation

- Thread type - thread data structure

    - thread ID
    - Program counter (PC)
    - Stack pointer (SP)
    - Registers
    - stack
    - attributes

- **Fork(proc, args)** - create a thread

    - not the UNIX fork

- **Join(thread)** - terminate a thread

    - the parent thread calls join with the threadID of the child thread, and the child thread will terminate and return its results when computation is complete

Mutual Exclusion

Using mutexes to lock atomic instructions to prevent threads from accessing shared memory at the same time and causing race conditions.

Producer / Consumer Example

What if the processing you wish to perform with mutual exclusion needs to occur only under certain conditions?

The producer / consumer example - multiple producer threads push to a list and the consumer thread retrieves an item, conducts an operation, and clears it from the list. In the example presented, the consumer locks the mutex and checks the list as often as possible. This is wasteful and uses a lot of time.

Condition Variables

Examples shows how to lock a mutex, set a **condition variable** to wait to become true, and then releases the mutex and waits for the condition variable to become true. This thread will then leave the wait state, lock the mutex again, conduct an operation, and then releases the mutex.

At the same time, this example shows how another thread can notify the thread waiting when the condition variable is true using a **signal**.

Condition Variable API

- **Wait(mutex, cond)**
    - mutex is automatically released and re-acquired on wait
- **Signal(cond)**
    - notify only one thread waiting on condition
- **Broadcast(cond)**
    - notify all waiting threads

Quizzes

Do the following statements apply to processes (P), threads (T) or both (B)? Mark you answer in the text boxes.

[**T**] can share a virtual address space [**P**] take longer to context switch [**B**] have an execution context [**T**] usually result in hotter caches when multiple exist [**B**] make use of some communication mechanisms

Midterm Exam Review Q & A

Part 1

1. **What are the key roles of an operating system?**
    1. Hides the underlying hardware from the user. (Hides hardware complexity.)
    2. Manages hardware resources for the user. (Memory management. CPU scheduling.)
    3. Provide process isolation and protection.
2. **What are the distinctions between OS abstractions, mechanisms, and policies?**
    1. Abstractions
        - processes, threads, files, sockets, memory pages
    2. Mechanisms
        - creating, scheduling, opening, writing, allocating
    3. Policies
        - least-recently used (LRU), earliest deadline first (EDF), number of sockets an application can use
3. **What does the principle "separation of mechanism and policy" mean?**
    - Implement flexible mechanisms that support many policies. (e.g. LRU, LFU, random)
        - What this means is that in the operating system we'd like to have some mechanisms to track the frequency or the time when a memory location has been accessed. This will help us keep track of when a page was last used, or when a page was least frequently used, or we can completely ignore that information. The bottom line is, we can implement any one of these policies in terms of how that memory management is going to operate.
4. **What does the principle "optimize for the common case" mean?**
    - Optimize the operating system for the most common use case.
        - We need to understand the common case and then, based on that common case, pick a specific policy that makes sense and that can be supported given the underlying mechanisms and abstractions the operating system supports.
5. **What happens during a user-kernel mode crossing?**

- User-Kernel switches are supported by the hardware. When a User-Mode process wants to gain privileged access to the hardware, the User-Mode process will set off a **trap**, stopping the User-Mode process and requiring the Operating System to review what caused the **trap** to occur. The Operating System will either grant the User-Mode process access to the Kernel-Mode resource, or terminate the process if the User-Mode process is attempting to do something illegal.

6. **What are some of the reasons why user-kernel mode crossing happens?**
   - When a user-mode process attempts to conduct a privileged operation like trying to access a resource: hardware, memory, files, sockets, etc.
7. **What is a kernel trap? Why does it happen? What are the steps that take place during a kernel trap?**
   - When a user-mode process attempts to conduct a privileged operation, a special privileged bit will not be set in the CPU, thus triggering a **trap**. The user-mode process relinquishes control of the CPU to a kernel-mode process, the kernel-mode process reviews policies and determines if it should grant access to the user-mode process or terminate the process for trying to attempt an illegal operation.
8. **What is a system call? How does it happen? What are the steps that take place during a system call?**
   - Operating Systems provide processes a set of system calls to applications as a means to request the Operating System to execute some sort of privileged operation.
     1. The user-mode process uses a system call to request the operating system to perform some privileged operation.
     2. Control is passed to the kernel-mode process, the trap mode bit is set to `0`, and the kernel-mode process reviews policies for the requested privileged operation.
     3. The kernel-mode process executes the system call and then sets the trap mode bit back to `1`.
     4. The kernel-mode process returns the results of the system call back to the user-mode process.
   - To make a system call an application must:
     1. Write arguments
     2. Save relevant data at a well-defined location
     3. Make the system call
9. **Contrast the design decisions and performance tradeoffs among monolithic, modular and microkernel-based OS designs.**

1. Monolithic OS

    - Pros
        - The benefit of this approach is that everything is included in the operating system. The abstractions, all the services, and everything is packaged at the same time. And because of that, there's some possibilities for some compile-time optimizations.
    - Cons
        - The downside is that there is too much state, too much code that's hard to maintain, debug, upgrade. And then its large size also poses large memory requirements, and that can always impact the performance that the applications are able to observe.

2. Modular OS

    - Pros
        - The benefits of this approach is that it's easier to maintain and upgrade. It also has a smaller code base and it's less resource intensive, which means that it will leave more resources and more memory for the applications. This can lead to better performance as well.
    - Cons
        - The downside of this approach is that although modularity may be good for maintainability, the level of interaction that it requires, because we have to go through this interface specification before we actually go into the implementation of a particular service. This can reduce some opportunities for optimizations.
        - Maintenance, however, can still be an issue given that these modules may come from completely disparate code bases and can be a source of bugs.

3. Microkernel OS

    - Pros
        - The benefits of a microkernel is that it's very small. This can not only lead to lower overheads and better performance, but it may be very easy to verify and test that the code behaves as it should.
    - Cons
        - The downsides of the microkernel design are that although it is small, its portability is sort of questionable because it is typically very specialized, very customized to the underlying hardware. The fact that there may be more one-off versions of a microkernel specialized for different platforms makes it harder to find common components of software, and that leads to software complexity as well.
        - And finally, the fact that we have these very frequent interactions

between different processes, these different user-level applications, means that there is a need for frequent user/kernel crossings.

Midterm Exam Review Q & A

Part 2

1. **Process vs. thread, describe the distinctions. What happens on a process vs. thread context switch.**
    - Process
        - A process is an application loaded into memory, and currently executing. It's an active entity.
    - Thread
        - A thread represents an execution context of a process. It is still part of a process and shares virtual address space, code, data, and files, with all of the other threads within the same process.
    - When a process conducts a context switch, it is required to create new virtual to physical address mappings upon being scheduled.
    - When a thread conducts a context switch, because threads share an address space, it is not necessary to create new virtual to physical address mappings upon being scheduled.
    - For these reasons, the time to context switch among threads is less than the time to context switch among processes.
2. **Describe the states in a lifetime of a process?**
    1. New
    2. Ready
    3. Running
    4. Waiting
    5. Terminated
3. **Describe the lifetime of a thread?**
    1. Runnable
    2. Active
    3. Sleeping
    4. Stopped
    5. Zombie
4. **Describe all the steps which take place for a process to transition from a waiting**

(blocked) state to a running (executing on the CPU) state.

   1. A process will enter the waiting (blocked) state if it is waiting on some sort of event to occur like I/O, etc.
   2. After the specific event the process was waiting on occurs, the process will return back to the ready queue for scheduling.
   3. The process will then be scheduled for execution and transition to the running state.

5. **What are the pros-and-cons of message-based vs. shared-memory-based IPC.**

   1. Message-based IPC

      - Pros
        - The operating system manages the communication channel. Both processes will use the exact same system calls and API for sending and receiving data.
      - Cons
        - Overhead as every single message that has to be sent between the two process must be handled by the operating system at some point.

   2. Shared-memory IPC

      - Pros
        - The operating system is not involved in this communication channel. The only overhead required is having the operating system map the shared memory space for both processes.
      - Cons
        - The processes have to be programmed to use a developer-defined API and they must utilize the shared memory in the same way. This is prone to errors if not implemented correctly.`

6. **What are benefits of multithreading? When is it useful to add more threads, when does adding threads lead to pure overhead? What are the possible sources of overhead associated with multithreading?**

   - Multithreading increases the speed at which work can be done. Multithreading allows a process to leverage multiple CPUs. Multithreaded programs are more efficient in their use of resources than multiprocess programs and incur less overhead when conducting inter-process communication (whereas this would be inter-thread communication).

   - It is useful to add more threads when the time for a blocking operations is twice the time of a context switch to a different thread. If you have too many threads being context switched and scheduled, and the time to conduct a blocking operation is less than all the context switches, you have too many threads.

- The main source of overhead for threads is the context switching of active and runnable threads.

7. **Describe the boss-worker multithreading pattern. If you need to improve a performance metric like throughput or response time, what could you do in a boss-worker model? What are the limiting factors in improving performance with this pattern?**

    - The boss-worker model is a multithreading patten in which there is one *boss* thread and an **N** number of *worker* threads. The boss thread receives work and assigns the work to different worker threads and then waits for their results. The primary way to increase performance in the boss-worker multithreading model is to create a work queue, this way the boss assigns work to the workers using the work queue and then the boss doesn't worry about the outcome of the work.

    - The primary limiting factor in improving performance for this pattern is how quickly the boss processes incoming jobs.

8. **Describe the pipelined multithreading pattern. If you need to improve a performance metric like throughput or response time, what could you do in a pipelined model? What are the limiting factors in improving performance with this pattern?**

    - In the pipelined multithreading pattern, multiple threads work together to handle a work item in different stages. In each different stage of the pipeline, the worker thread conducts a specific operation on the work and hands the item to the next thread in line. To increase performance for the pipeline model, one would increase the number of threads available to do work at the stages of the pipeline that take the longest.

    - The limiting factor in improving performance of this pattern is the longest time it takes to conduct an operation.

9. **What are mutexes? What are condition variables? Can you quickly write the steps/code for entering/existing a critical section for problems such as reader/writer, reader/writer with selective priority (e.g., reader priority vs. writer priority)? What are spurious wake-ups, how do you avoid them, and can you always avoid them? Do you understand the need for using a while() look for the predicate check in the critical section entry code examples in the lessons?**

    - Mutexes are short for mutual exclusion. Mutexes are a synchronization mechanism useful for protected shared data between multiple threads or processes.

    - Condition variables are variables used for signalling of an accompanying mutex.

    - Spurious wake-ups are a costly pitfall in which a number of threads are woken up for a condition variable and then attempt to acquire an already locked mutex. To avoid spurious wake-ups, it's best practice to signal a condition variable for waiting threads *after* releasing the mutex. You cannot always avoid them as sometimes the signal will be dependent on a decision statement that the lock might be currently holding. Releasing the lock might mess with the signal's decision statement.

```
#declaration of condition variables and mutex
pthread_mutex = example_mutex;
pthread_cond = read_phase;
pthread_cond = write_phase;
waiting_readers = 0;
num_readers = 0;

#reader with priority

#enter critical section
pthread_mutex_lock(&example_mutex);
  waiting_readers++;
  while(num_readers == -1)
  {
    pthread_cond_wait(read_phase, example_mutex);
  }
  waiting_readers--;
  num_readers++;
pthread_mutex_unlock(&example_mutex);

#read some data

pthread_mutex_lock(&example_mutex);
  num_readers--;
  if (num_readers == 0)
  {
    pthread_cond_signal(write_phase, example_mutex);
  }
pthread_mutex_unlock(&example_mutex);

#writer with priority

#enter critical section
pthread_mutex_lock(&example_mutex);
  while(num_readers != 0)
  {
    pthread_cond_wait(write_phase, example_mutex);
  }
  num_readers--;
pthread_mutex_unlock(&example_mutex);

#write some data
```

```
pthread_mutex_lock(&example_mutex);
  num_readers++;
  if (waiting_readers > 0)
  {
    pthread_cond_signal(read_phase, example_mutex);
  }
  else
  {
    pthread_cond_signal(write_phase, example_mutex);
  }
pthread_mutex_unlock(&example_mutex);
```

- `while()` is used to:
    1. Support multiple consumer threads
    2. Used because we cannot guarantee access to the mutex once the condition is signaled.
    3. The data we're waiting on can change before the consumer gets access again.

10. **What’s a simple way to prevent deadlocks? Why?**

    - If using multiple mutexes, set a defined lock order that all threads must abide by. This way, in order to acquire a lock, the thread must acquire all previous locks as well. This prevents deadlock because each thread will attempt to acquire a lock in the same order, thus preventing two threads from acquiring different resources and waiting for already acquired resources to be freed.

11. **Can you explain the relationship among kernel vs. user-level threads? Think though a general mxn scenario (as described in the Solaris papers), and in the current Linux model. What happens during scheduling, synchronization and signaling in these cases?**

    - In the Solaris many-to-many user-level thread (ULT) vs kernel-level thread (KLT) relationship, each process utilizes lightweight processes as an interface between ULTs and KLTs. ULTs can be scheduled onto LWPs by the thread library's scheduler, and LWPs can be scheduled to KLTs by the kernel thread library's scheduler.

> ## Thread Scheduling
>
> The threads library implements a thread scheduler that multiplexes thread execution across a pool of LWPs. The LWPs are nearly identical, allowing any thread to execute on any of the LWPs in the pool. When a thread executes, it attaches to an LWP and has all the attributes of being a kernel-supported thread.
>
> ## Thread Synchronization

The thread library implements two types of synchronization variables, process-local and process-shared.

Process-local Synchronization Variables

The default blocking behavior is to put the thread to sleep, and place the thread on the sleep queue of the synchronization variable. The blocked thread is then surrendered to the scheduler, and the scheduler dispatches another thread to the LWP. Bound threads will stay permanently to the LWP, and the LWP will become *parked*, not *idle*.

Blocked threads wake up when the synchronization variable they were waiting on becomes available. The synchronization primitives check if threads are waiting on the synchronization variable, then a blocked thread is moved from the synchronization variable's sleep queue and is dispatched by the scheduler to an LWP. For bound threads, the scheduler unparks the thread and the LWP is dispatched by the kernel.

Process-shared Synchronization Variables

Process-shared synchronization objects can also be used to synchronize threads across different processes. These objects have to be initialized when they are created because their blocking behavior is different from the default. The initialization function must be called to mark the object as process-shared - the primitives will then recognize the synchronization variables are shared and provide the correct blocking behavior. The primitives rely on LWP synchronization primitives to put the blocking threads to sleep in the kernel still attached to their LWPs, and to correctly synchronize between processes.

Signals

A challenge is presented for signals because the kernel can send signals, but thread masks are invisible to the kernel, so signal delivery is dependent upon the thread signal mask making it hard to elicit the correct program behavior.

The thread implementation also has to support asynchronous safe synchronization primitives. For example, if a thread calls `mutex_lock()` and then is interrupted, what if the interrupt handler also tried to call `mutex_lock()` and enters a deadlock? One way to make this safe is to signal mask while in a thread's critical section.

Signal Model Implementation

A possibility is to have the LWP replicate the thread's signal mask to make it

> visible to the kernel. The signals that a process can receive changes based upon the threads in the application that cycle through the ACTIVE state.
>
> A problem arises when you have threads that aren't ACTIVE often, but are the only threads in the application that have specific signals enabled. These threads are asleep waiting on signals they'll never receive. In addition, anytime the LWP switches threads with different masks or the thread adjusts its mask, the LWP has to make a system call to notify the kernel.
>
> To solve this problem, we define the set of signals a process can receive equal to the intersection of all the thread signal masks. The LWP signal mask is either less restrictive or equal to the thread's signal mask. Occasionally, signals will be sent by the kernel to ACTIVE threads that have the signal disabled.
>
> When this occurs, the threads library prevents the thread from being interrupted by interposing its own signal handler below the application's signal handler. When a signal is received, the global handler checks the current thread's signal mask to determine if the thread can receive the signal.
>
> If the signal is masked, the global handler sets the LWP's signal mask to the thread's signal mask and resend the signal either to the process (for undirected signals) or to the LWP (for directed signals). If the signal is not masked, the global handler calls the signal's application handler. If the signal is not applicable to any of the ACTIVE threads, the global handler will wakeup one of the inactive threads to run if it has the signal unmasked.
>
> This is all for asynchronous signals. Synchronously generated signals are simply delivered by the kernel to the ACTIVE thread that caused them.

- Linux uses the one-to-one model which provides a lot more simplicity and a lot more transparency. Because a ULT is bound to a KLT in this model, the kernel sees all changes to the ULTs signal mask, and for scheduling and synchronization, if the ULT blocks, the KLT blocks.

12. **Can you explain why some of the mechanisms described in the Solaris papers (for configuring the degree concurrency, for signaling, the use of LWP...) are not used or necessary in the current threads model in Linux?**

    - Linux uses the one-to-one thread model. The optimizations implemented by Solaris aren't necessary because modern computing systems are able to utilize more memory space for the creation of these kernel threads. If systems were still tight on available memory the Solaris implementation would still be relevant, however, because the one-to-one model is easy to implement, the work to achieve the many-to-many thread model's optimizations isn't justified.

13. **What’s an interrupt? What’s a signal? What happens during interrupt or signal handling? How does the OS know what to execute in response to a interrupt or signal? Can each process configure their own signal handler? Can each thread have their own signal handler?**
    - An interrupt is generated by some hardware device and received by the CPU. During signal handling, the CPU has an interrupt table in which it conducts a lookup of what procedure it should execute upon receiving the interrupt. This procedure is known as an interrupt handler. All operations are suspended until the interrupt handler completes all of its required operations.
    - A signal is the same, except it is generated by the OS. A synchronous signal is a signal delivered back to the process that initiated it (SIGSEV, SIGINT, etc.). An asynchronous signal can be from the OS or from a different process. Each process can configure their own signal handlers different from the default. Each thread cannot have their own signal handler.
14. **What’s the potential issue if a interrupt or signal handler needs to lock a mutex? What’s the workaround described in the Solaris papers?**

> ## Signals
>
> A challenge is presented for signals because the kernel can send signals, but thread masks are invisible to the kernel, so signal delivery is dependent upon the thread signal mask making it hard to elicit the correct program behavior.
>
> The thread implementation also has to support asynchronous safe synchronization primitives. For example, if a thread calls `mutex_lock()` and then is interrupted, what if the interrupt handler also tried to call `mutex_lock()` and enters a deadlock? One way to make this safe is to signal mask while in a thread's critical section.
>
> ### Signal Model Implementation
>
> A possibility is to have the LWP replicate the thread's signal mask to make it visible to the kernel. The signals that a process can receive changes based upon the threads in the application that cycle through the ACTIVE state.
>
> A problem arises when you have threads that aren't ACTIVE often, but are the only threads in the application that have specific signals enabled. These threads are asleep waiting on signals they'll never receive. In addition, anytime the LWP switches threads with different masks or the thread adjusts its mask, the LWP has to make a system call to notify the kernel.
>
> To solve this problem, we define the set of signals a process can receive equal to the intersection of all the thread signal masks. The LWP signal mask is either less restrictive or

> equal to the thread's signal mask. Occasionally, signals will be sent by the kernel to ACTIVE threads that have the signal disabled.
>
> When this occurs, the threads library prevents the thread from being interrupted by interposing its own signal handler below the application's signal handler. When a signal is received, the global handler checks the current thread's signal mask to determine if the thread can receive the signal.
>
> If the signal is masked, the global handler sets the LWP's signal mask to the thread's signal mask and resend the signal either to the process (for undirected signals) or to the LWP (for directed signals). If the signal is not masked, the global handler calls the signal's application handler. If the signal is not applicable to any of the ACTIVE threads, the global handler will wakeup one of the inactive threads to run if it has the signal unmasked.
>
> This is all for asynchronous signals. Synchronously generated signals are simply delivered by the kernel to the ACTIVE thread that caused them.
>
> ### Signal Safe Critical Sections
>
> To prevent deadlock in the presence of signals, critical sections that are reentered in a signal handler in both multi-threaded applications and the threads library should be safe with respect to signals. All async signals should be masked during critical sections.
>
> To make critical sections as safe as efficiently possible, the threads library implements `thr_sigsetmask()`. If signals do not occur, `thr_sigsetmask()` makes no system calls making it just as fast as modifying the user-level thread signal mask.
>
> The threads library sets/clears a special flag in the threads structure whenever it enter/exits an internal critical section. Effectively, this flag serves as a signal mask to mask out all signals.

- Another option is to create another thread, entirely, to handle any signals sent to the process by the operating system. The best implementation would be to have this thread created on startup of the process rather than dynamically, as thread creation and tear-down can become costly depending upon how many signals the process is expecting to receive.
- This thread would be able to block on the call for a `mutex_lock()` without completely stopping the execution of the currently running thread and encountering a deadlock. The thread library would schedule this thread when the lock is available to handle the signal/interrupt.

15. **Contrast the pros-and-cons of a multithreaded (MT) and multiprocess (MP) implementation of a webserver, as described in the Flash paper.**

    - Multiprocess (MP) webservers are implemented by running multiple processes all executing the same code, serving clients and conducting IPC to share data and statistics.

    - Pros
        - Each process has its own private address space so no synchronization with other threads of execution is required.
    - Cons
        - Process creation is costly. Because each process has its own private address space, more memory is utilized than in an MT implementation.
        - IPC to share data and statistics is costly and has a lot of overhead.
        - Scheduling and context switches are costly because the virtual address space for each process has to be re-mapped upon execution and suspension of each process.

- Multithread (MT) webservers are implemented in one process with multiple threads all executing the same code, serving clients and conducting communication and synchronization amongst the threads to share data and statistics.

    - Pros
        - Threads share a PCB and private virtual address space, thus, they can collaborate more, share global variables, etc.
        - Scheduling and context switching are less costly because the virtual address space doesn't have to be re-mapped upon the execution and suspension of each thread.
    - Cons
        - Requires more work for the developer to ensure all actions are synchronized correctly. The threads will be sharing global variables within the process.
        - At the time this paper was written, not all operating systems supporting multithreading or thread libraries.

16. **What are the benefits of the event-based model described in the Flash paper over MT and MP? What are the limitations? Would you convert the AMPED model into a AMTED (async multi-threaded event-driven)? How do you think an AMTED version of Flash would compare to the AMPED version of Flash?**

    - The event-based webserver model has smaller memory requirements, no need to context switch, and has no need for synchronization.

    - The event-based webserver model is limited by the operating system's support for asynchronous system calls. If asynchronous system calls are not available, the event-based model must be implemented to utilize helper processes that will conduct blocking I/O for the event dispatcher.

    - I would convert the AMPED model of Flash to the AMTED model due to these reasons:

        - The AMTED model would require less overhead than the AMPED model for communication, as all execution contexts within the AMTED model would share the same address space and state.
        - The AMTED model would have a smaller memory footprint than the AMPED model.
        - The primary issue the AMTED model would have is that it would require more synchronization because the threads would all be accessing the same data.

17. **There are several sets of experimental results from the Flash paper discussed in the lesson. Do you understand the purpose of each set of experiments (what was the question they wanted to answer)? Do you understand why the experiment was structured in a particular why (why they chose the variables to be varied, the workload parameters, the measured metric...).**

    - For the testing of the Flash AMPED webserver implementation, the authors picked a range of different webserver implementations to test against, all on the same hardware using three sets of data:

        - An OWLnet traffic trace from Rice University (small traffic)
        - A CS trace from Rice University (diverse, large traffic)
        - A synthetic workload (a single file) - this works as their base case

    - The authors also used two metrics:

        - bandwidth - total amount of useful bytes transferred
        - connection rate - how they tested concurrent processing by number of clients served

> To summarize, the performance results for Flash show the following. When the data is in cache, the basic SPED model performs much better than the AMPED Flash because it doesn't require the test for memory presence which was necessary in the AMPED Flash. Both SPED and the AMPED Flash are better than the MTed or MP models because they don't incur any of the synchronization or context switching overheads that are necessary with these (MT/MP) models.
>
> When the the workload is disk-bound, however, AMPED performs much better than the single process event-driven (SPED) model because the single process model blocks since there's no support for asynchronous I/O. AMPED Flash performs better than both the MTed and the MP model because it has much more memory efficient implementation and it doesn't require the same level of context switching as in these (MT/MP) models. Again, only the number of concurrent I/O bound requests result in concurrent processes or concurrent threads in this model.

18. **If you ran your server from the class project for two different traces: (i) many requests for a single file, and (ii) many random requests across a very large pool of**

very large files, what do you think would happen as you add more threads to your server? Can you sketch a hypothetical graph?

P3: Lesson 1: Scheduling

This lecture covers scheduling mechanisms, algorithms and data structures. This lecture will cover the aspects of the Linux O(1) and Completely Fair Scheduler (CFS) scheduler implementations. This lecture will also cover scheduling on multi-CPU platforms.

Visual Metaphor

- Dispatching tasks immediately
    - Scheduling is simple (FIFO)
    - First come first served (FCFS)
- Dispatching simple tasks first
    - maximize # of tasks processed over time
    - Shortest job first (SJF)
- Dispatch complex tasks first
    - maximize use of resources

Scheduling Overview

CPU Scheduler

- decides how and when processes (and their threads) access chared CPUs
- schedules tasks running user-level process/threads as well as kernel-level threads
- chooses one of ready tasks to run on a CPU
- runs when:
    - the CPU becomes idle to prevent the CPU from being idle too long
    - a new task becomes ready to execute
    - a task's time-slice expires

Run-to-Completion Scheduling

First-Come First-Serve (FCFS)

- Schedules tasks in order of arrival (FIFO)
- Uses a *runqueue* that is essentially just a regular *queue*.
- Simple, but tasks can have a long wait time.

Shortest Job First (SJF)

- schedules tasks in order of their execution time
- *runqueue* will not be a FIFO, but an ordered queue ordered by execution time.
- *runqueue* can also be ordered like a tree and the scheduler will always select the left-most node of the tree as it will be the shortest execution time

Preemptive Scheduling

Shortest Job First

For SJF, the scheduler needs to know the execution time of the tasks for preemptive scheduling. The scheduler uses heuristics to guestimate the execution time of a task. The scheduler uses history to predict a task's execution time.

- How long did a task run last time?
- How long did a tsk run last *n* times?

This is called a **windowed average**.

Priority Scheduling

- tasks have different priority levels
- the scheduler will run the highest priority task next (preemptive)

The *runqueue* would have to have multiple priority queues or an ordered tree based on priority

Starvation - low priority tasks stuck in runqueue **Priority Aging** - f(actual priority, time spent in runqueue)

Priority Inversion

To avoid priority inversion, temporarily boost the priority of a task that currently holds a common resource needed to execute, like a mutex.

Round Robin Scheduling

- pick up first task from a FIFO queue similar to FCFS
- task may yield due to I/O operation - task will be placed back on runqueue
- next task is executed in a circular queue
- round-robin can included priorities for preemption
- round-robin can also interleave tasks for execution using time-slices

Timesharing and Timeslices

Timeslice - maximum amount of uninterrupted time given to a task (*time quantum*)

- A task can run less than the alloted timeslice time
    - It may have to wait on I/O, synchronization, etc. and will be placed on a queue
    - If priority based scheduling is in play, the task might be preempted and placed on a queue
- Timeslices allow CPU bound tasks to be *interleaved* (timesharing the CPU)
    - CPU bound tasks will be preempted after timeslice expires

Pros

- Short tasks are able to finish sooner
- More responsive schedule for tasks
- Lengthy I/O operations are initiated sooner
- Easier to implement than SJF

Cons

- Pure overhead of interrupting, scheduling, and context switching tasks after each timeslice expires.
    - To minimize overheads, keep timeslices longer than sum of context switch times

How Long Should a Timeslice Be?

Balance the benefits and overheads. Balancing should take CPU-bound tasks vs I/O-bound tasks into consideration when conducting optimization.

Smaller timeslices are better for **I/O-bound** tasks.

- Limits context switching overheads
- Keeps CPU utilization and throughput high

Longer timeslices are better for **CPU-bound** tasks.

- I/O bound tasks can issue I/O operations earlier
- Keeps CPU and device utilization high
- Better user-perceived performance

Runqueue Data Structures

If we want I/O-bound and CPU-bound tasks to have different timeslice values, then we have some options:

- single runqueue data structures, but each task has a type identifier
- two different runqueue data structures, separating I/O-bound and CPU-bound tasks

An option is a multi-queue implementation, separating types of tasks:

- most I/O intensive (highest priority)
- medium I/O intensive (middle priority)
- CPU intensive (lowest priority)(infinite timeslice)

This option provides:

- timeslicing benefits provided for I/O-bound tasks
- timeslicing overheads avoided for CPU-bound tasks

How do we know if a task is CPU-bound or I/O-bound?

This is how we deal with this problem:

1. Tasks enters the topmost of the queue
2. If tasks yield voluntarily, these tasks stay at this level of the queue
3. If tasks use up entire timeslice, these tasks are pushed to a lower level of priority
4. If tasks use up the entire timeslice, these tasks are pushed down to the lowest level
5. If tasks at the lowest level begin voluntarily yielding, their priority will be escalated by the scheduler

This is called the **Multi-level Feedback Queue (MLFQ)** (created by Fernando Corbato *Turing Award Winner*)

MFLQ != Priority Queues

- different treatment of threads at each level
- provides feedback to the scheduler

Linux O(1) Scheduler

The **O(1)** scheduler advertises the ability to conduct its operations in constant time. The O(1) schedule is preemptive and priority-based

- real time priorities (0-99)
- timesharing (100-139)

The default priority for a user process is **120**. These can be adjusted with a *nice value* ranging from (-20 - 19).

The O(1) scheduler is an implementation of the MLFQ scheduler. Timeslice values depend on priority - timeslices are smallest for low priority tasks and highest for high priority tasks.

Feedback the O(1) scheduler utilizes is derived from a process's sleep time (waiting/idling). The longer a process sleeps, its priority is boosted because it is deemed interactive. Smaller sleep time indicates a CPU-bound tasks, its priority is decreased.

The O(1) scheduler's runqueue two (2) arrays of tasks **Active**

- used to pick next task to run
- constant time to add/select tasks
- tasks remain in queue in active queue until timeslice expires

Expired

- inactive list
- when no more tasks are in the active array, the active and expired runqueues are swapped

Problems with the O(1) Scheduler

- suffers with performance of interactive tasks
- not fair to all tasks

Linux Completely Fair Scheduler (CFS)

CFS is the default scheduler since 2.6.23. Its runqueue is implemented in a red-black tree. Tasks are ordered in the tree based upon the time they spend on the CPU (**vruntime**).

CFS scheduling explanation:

- always picks the leftmost (shortest vruntime) of a node
- periodically adjusts vruntime of a currently executing tasks
- compares vruntime of currently running task to leftmost vruntime
- if the currently running task's vruntime is lower than the leftmost node, the currently running task is preempted and the leftmost node's task is scheduled

vruntime progress rate depends on priority and niceness:

- low-priority tasks' vruntime progresses faster
- high-priority tasks' vruntime progresses slower

CFS uses **one** runqueue structure for scheduling.

Performance

Selecting a task => O(1) runtime Adding a task => O(log n) runtime

Scheduling on Multi-CPU Systems

- **LLC** - last level cache
- **SMP** - shared memory multiprocessor

We want to achieve cache-affinity when scheduling on Multi-CPU systems. This means keeping a task running on the same CPU as much as possible. This requires a hierarchical scheduler architecture.

Cache-affinity is achieved by implementing a per-CPU scheduler and per-CPU runqueue to balance the load across CPUs based upon queue length and if a CPU is idle.

We can also implement **memory nodes**, representing a *socket* of multiple processors. Access to a local memory node is faster than accessing a remote memory node. These types of platforms are called **Non-Uniform Memory Access (NUMA) platforms.**

This achieves the goal of keeping tasks on the CPU closer to the memory node where their state resides in the cache. This type of scheduling is called *NUMA-aware scheduling*.

Hyperthreading

- Multiple hardware-supported execution contexts
- Still 1 CPU however...
- Context switches are very fast

Multiple monikers:

- hardware multithreading
- hyperthreading
- chip multithreading (CMT)
- simultaneous multithreading (SMT)

Formulas

Throughput Formula: jobs_completed / time_to_complete_all_job

Avg. Completion Time Formula:

sum_of_times_to_complete_each_job / jobs_completed

Avg. Wait Time Formula:

(t1_wait_time + t2_wait_time + t3_wait_time) / jobs_completed

CPU Utilization Formula

((useful_work_time_per_task) / (useful_work_time_per_task + context_switch_overhead)) * 100

P3: Lesson 2: Memory Management

Summary

All operating systems have memory management systems.

- Uses intelligently sized containers
    - Pages or segments
- Not all memory is needed at once
    - Tasks operate on a subset of memory
- Optimized for performance
    - Reduce time to access state in memory -> improved performance

Memory Management: Goals

Virtual addresses are mapped to physical addresses, as described in P1 of this course. The operating system must be able to allocate and arbitrate memory for processes using this memory.

Allocate

- allocation, replacement, etc.

Arbitrate

- address translation and validation

Virtual addresses are subdivided into fixed-sized segments called pages. Physical memory is divided into page-frames called.

Page-based memory management *Allocate* -> pages mapped to page frames *Arbitrate* -> page tables

Segment-based memory management *Allocate* -> segments *Arbitrate* -> segment registers

Hardware Support for Memory Management

Memory management unit:

- translates virtual to physical addresses
- reports *faults*: illegal access, permissions, page not present

Registers:

- pointers to page table

- base and limit size, number of segments...

Cache - Translation Lookaside Buffer

- valid Virtual Address to Physical Address Translations (**TLB**)

Translation

- actual Physical Address generation done in hardware

Page Tables

Page tables map virtual memory address to physical memory addresses. (Virtual memory pages and physical memory pages are the same size)

Virtual Page Numbers, Physical Frame Numbers, valid bits in the page table to determine if page is actually in memory or not. If not, the operating system will bring the page into memory if the access request is valid.

The operating system contains a page table for every single process that exists. On a context switch, the process will switch to a valid page table. On x86, the pointer to the page table is called **CR3**.

Page Table Entries:

Flags

- Present (valid/invalid)
- Dirty (written to)
- Accessed (for read or write)
- Protection bits => R/W/X

Page Fault

When a page fault occurs, an error code is generated and the processes traps into the kernel. A *page fault handler* determines what action to take based upon the error code:

- bring page from disk into memory?
- protection error (SIGSEV)

Page Table Size

- Process doesn't use entire address space, usually
- Page tables assume an entry per VPN, regardless of whether corresponding virtual memory is needed or not.
- **Hierarchical page tables** can achieve finer referencing of memory to prevent pages from

becoming too large allowing for better use of resources. These level of abstractions will increase the time to translate all the addresses required, however.

P3: Lesson 4: Synchronization Constructs

Visual Metaphor

Synchronization is like waiting for a coworker to finish so you can continue working:

- May repeatedly check to continue
    - Synchronization using spinlocks
- May wait for a signal to continue
    - Synchronization using mutexes and condition variables
- Waiting hurts performance
    - CPUs waste cycles for checking; cache effects

What else is there to learn about synchronization?

Limitations exist for mutexes and condition variables:

- error prone / correctness / ease-of-use
    - unlock wrong mutex, signal wrong condition variable
- lack of expressive power
    - helper variables for access or priority control

The implementation of these synchronization prototypes require low level hardware support for atomic instructions.

Spinlocks

Spinlocks are like mutexes:

- provide mutual exclusion
- lock and unlock (free) call

Spin locks differ from mutexes because the threads that suspend do not sleep or block, but continuously check a lock and burn CPU cycles. Spin locks are a **basic** synchronization construct and can be used to implement more complex synchronization mechanisms.

Semaphores

Semaphores are a common synchronization construct in OS kernels.

- Acts like a traffic light: **STOP** and **GO**
- Similar to a mutex, but used more generally.

Semaphores are represented using an integer value. On initialization, semaphores are assigned some max value (positive integer). On try (wait), if non-zero, the semaphore is decremented and the caller proceeds to wait. If a semaphore is initialized with `1`, it acts as a mutex (binary semaphore). On exit, callers `post` and increment the semaphore allowing future threads to utilize the critical section the semaphore is protecting.

Reader/Writer Locks

Syncing different types of accesses

- Read (never modify) -> shared access
- Write (always modify) -> exclusive access
- R/W Locks
    - Specify the type of access
        - Lock behaves accordingly
- R/W lock support is available in Windows (.NET), Java, POSIX, and more libraries.

Monitors

Monitors specify:

- shared resources
- entry procedures
- possible condition variables

On entry:

- locks and checks are conducted

On exit:

- unlocks, checks, and signals are conducted

A lot of these operations are hidden from the programmer, making **monitors** considered to be a high level synchronization constructs.

- Java natively supports monitors
    - utilizes synchronized methods and generate monitor code
    - notify() explicitly

Final Exam Review Q & A

Part 3

P3L1

1. How does scheduling work? What are the basic steps and data structures involved in scheduling a thread on the CPU?

    - **An operating system scheduler dispatches ready threads/tasks to execute on the CPU. The data structure involved with scheduling is the ready queue. The ready keeps track of all tasks/threads ready to execute their next set of instructions on the CPU. Another data structure involved in dispatching is a task's context.**
    - **The colloquial name for the ready queue is the *runqueue*.**

2. What are the overheads associated with scheduling? Do you understand the tradeoffs associated with the frequency of preemption and scheduling/what types of workloads benefit from frequent vs. infrequent intervention of the scheduler (short vs. long timeslices)?

    - **Overhead associated with scheduling is context switching in order to conduct scheduling itself. Then there's calculations on which tasks should run next, conducting the actual scheduling algorithm. Then when a context switch occurs, the cache might be hot or cold, or the resources the thread requires might not be available.**
    - **Memory-bound tasks, tasks that require some I/O, interrupt, memory access, disk access, etc. benefit from short timeslices as they don't need to hold the processor pipeline for that long.**
    - **CPU-bound tasks, tasks that require a lot of computation and use of the processor pipeline, need longer timeslices so that they aren't interrupted in the middle of their computation. This allows them to complete faster.**

3. Can you work through a scenario describing some workload mix (few threads, their compute and I/O phases) and for a given scheduling discipline compute various metrics like average time to completion, system throughput, wait time of the tasks.

    - **The scheduling algorithms discussed that apply to this particular question are:**
        - First Come - First Served
        - Shortest Job First
        - Longest Job First
        - Priority Scheduling (Preemptive Scheduling)

        - Priority Inversion
        - Round Robin Scheduling

4. Do you understand the motivation behind the multi-level feedback queue, why different queues have different timeslices, how do threads move between these queues... Can you contrast this with the O(1) scheduler? Do you understand what were the problems with the O(1) scheduler which led to the CFS?

    - **The multi-level feedback queue uses feedback from a task's execution behavior to determine what level of the feedback queue it will be prioritized in, next. The highest priority has the shortest timeslice for I/O bound tasks. The medium priority has a medium timeslice. The final priority, the lowest, is for CPU bound tasks that need a long timeslice. A task starts with no history, but based upon its voluntary or involuntary yields or preemption, we can determine it I/O boundness or CPU boundness.**
    - **The difference between the multi-level feedback queue and the O(1) scheduler is that the O(1) scheduler bases feedback off of *sleep* time. Longer sleep means the task has a higher priority because it is interactive. Smaller sleep means it's compute intensive, thus a lower priority. The priority is based upon a numerical value (0 (high) - 140 (lowest)). 0 - 99 are real-time, interactive tasks, 100 - 140 are compute based. Two arrays are maintained: active and expired. Active tasks still have a timeslice that hasn't expired. Expired tasks are tasks whose timeslices are expired. After all active tasks have expired, the arrays swap.**
    - **The O(1) scheduler was replaced by the CFS because the workload changed over time. Typical applications became more time-sensitive and interactive. Jitter was introduced because the tasks were constantly switching between active and expired. O(1) didn't guarantee fairness, either, thus causing starvation.**

5. Thinking about Fedorova's paper on scheduling for chip multi processors, what's the goal of the scheduler she's arguing for? What are some performance counters that can be useful in identifying the workload properties (compute vs. memory bound) and the ability of the scheduler to maximize the system throughput.

    - **She's arguing for a scheduler that can use heuristics of different threads' workloads to effectively load balance tasks across multiple processors in order to achieve high resource utilization.**
    - **Hardware counters useful in determining heuristics include:**
        - Cache misses
        - instructions per cycle (IPC)
        - power and energy data
    - **Hardware counters estimate what kind of resources a thread needs. This enables the scheduler to make an informed decision about which thread needs**

        to be scheduled based upon which resources will be available.
    - **Cycles-per-instruction (CPI) provided, somewhat, of a good metric to identify workload properties.**

P3L1 Formulas

- **Throughput Formula**
    - jobs_completed / time_to_complete_all_jobs
- **Average Completion Time Formula**
    - sum_of_times_to_complete_each_job / jobs_completed
- **Average Wait Time Formula**
    - sum_of_each_job_wait_time / jobs_completed
- **CPU Utilization Formula**
    - ((#_job_type_1 * job_type_1_exec_time) + (#_job_type_n * job_type_n_exec_time)) / ((#_job_type_1 * job_type_1_exec_time) + (#_job_type_1 * context_switch_time) + (#_job_type_n * job_type_n_exec_time) + (#_job_type_n * context_switch_time))

P3L2

1. How does the OS map the memory allocated to a process to the underlying physical memory? What happens when a process tries to access a page not present in physical memory? What happens when a process tries to access a page that hasn't been allocated to it? What happens when a process tries to modify a page that's write protected/how does COW work?

    - **The operating system allocates physical memory to a process and the process will map that physical memory to its virtual memory. The operating system arbitrates the process's access to physical memory by conducting address translation and validation of virtual memory to physical memory mapping.**
    - **The operating system uses a memory management unit (MMU) to translate virtual to physical addresses.**
        - When a process tries to access a page not present in memory, a fault occurs.
        - MMUs also report illegal accesses based upon permissions or access to memory not allocated to a process.
        - Page faults trap into the kernel, and the page fault handler takes control. The page fault handler reads the error code using the page table flags and makes a decision.
        - Copy-on-write (COW) occurs when the operating system identifies that a process wants to modify a piece of memory. The operating system will create a copy that is modifiable. This occurs on the first write, otherwise, the memory is not duplicated to prevent overuse of resources.

2. How do we deal with the fact that processes address more memory than physically available? What's demand paging? How does page replacement work?

  - **Operating systems conduct memory allocation with pages to provide processes memory to work with in main memory. Processes will address more memory than is available, the operating system will conduct swapping between the main memory and secondary memory, or disk memory.**
  - **The process of swapping pages in/out of memory is called demand paging.**
  - **Replacement algorithms, such as least recently used, etc. replace pages currently residing in memory and swap them to secondary memory.**

3. How does address translation work? What's the role of the TLB?

   - **Address translation is conducted by utilizing a page table. The first portion of the virtual address is used to index the page table (virtual page number(VPN)). This produces the physical frame number (PFN). Then we sum the PFN with the offset from the virtual address to create the physical address.**
   - **The translation lookaside buffer speeds up the translation process. It contains a cache of valid translations so that translations don't have to be re-calculated**

4. Do you understand the relationships between the size of an address, the size of the address space, the size of a page, the size of the page table...

   - **Page table size is based upon the number of virtual page numbers, page size, and length of the page table entry.**

5. Do you understand the benefits of hierarchical page tables? For a given address format, can you workout the sizes of the page table structures in different layers?

   - **Hierarchical page tables prevent using too much memory for a page table. Page tables that aren't being utilized or pointed to won't be allocated. Only valid memory regions will have a page table entry.**

P3L2 Formulas

- **Virtual Page Numbers**
  - 2^(architecture size (32 or 64)) / page_size
- **Page Table Size**
  - (virtual_page_numbers / page_size) * page_table_entry
- **Page Size**
  - determined by the offset

P3L3

1. For processes to share memory, what does the OS need to do? Do they use the same virtual addresses to access the same memory?

   - **The operating system creates a piece of memory and the maps the shared memory to both processes. They do not use the same virtual memory**

addresses, however, their virtual memory translations both translate to the same portion of physical memory.

2. For processes to communicate using a shared memory-based communication channel, do they still have to copy data from one location to another? What are the costs associated with copying vs. (re-/m)mapping? What are the tradeoffs between message-based vs. shared-memory-based communication?

    - **For message-based IPC, the operating system manages the format of all the messages, the correct synchronization and fidelity of the data being sent across the communication channel. Unfortunately, the using the communication channel imposes overhead for the operating system. Context switches and copying of data into the process space.**
    - **For shared-memory IPC, there is significantly less overhead imposed by context switching into the kernel. Once the shared memory is mapped and provided, the user-mode processes can do as they please with the memory. Drawbacks include: the developer synchronizing the access to shared memory.**
    - **There are significantly less copies required for shared memory. If a process only needs to read shared memory, there's no reason for it to have to copy it to it's own address space. The same could be said for writing. The process could write directly to the shared memory.**
    - **Costs associated copying (messages) vs mapping (shared memory):**
        - CPU cycles are required for each copy of data to/from port
        - CPU cycles are required to map memory into address space and data into the channel
            - However, mapping is done once
            - Even one map can perform better than continually copying

3. What are different ways you can implement synchronization between different processes (think what kinds of options you had in Project 3).

    - pthreads mutexes and condition variables that are appropriately set
    - OS-supported IPC for sync
    - message queues can be used for command and control / sync mechanism
    - binary semaphores (named and unnamed)

P3L4

1. To implement a synchronization mechanism, at the lowest level you need to rely on a hardware atomic instruction. Why? What are some examples?

    - **Hardware atomic instructions provide guaranteed correctness of the status of a sync mechanism.**

2. Why are spinlocks useful? Would you use a spinlock in every place where you're currently

using a mutex?

  - **Spinlocks allow threads to wait until the lock is free without blocking. The thread will continuously check the lock without giving up the CPU.**
  - **I would not use a spinlock in every location where I'm currently using a mutes. Only on atomic instructions that aren't I/O based or force a thread to block. Else, threads will hold the CPU and continue to spin for way too long.**

3. Do you understand why is it useful to have more powerful synchronization constructs, like reader-writer locks or monitors? What about them makes them more powerful than using spinlocks, or mutexes and condition variables?

  - **Semaphores provide the ability provide more information for a specific synchronization requirement. Imagine if you want more than one reader, a counter semaphore would allow a max number of N readers. Once the semaphore is 0, a writer can read that value and acquire the semaphore for writing.**
  - **Reader-writer locks allow multiple threads to read from a location in memory without having to queue or wait for a mutex/spinlock. Readers can share the memory and, when a write wants exclusive access, a writer lock can provide a writer exclusive access to modify the memory.**
  - **Monitors are powerful because with a monitor you can specify the resource to be protected, entry procedures (permissions, etc.), and condition variables that can be utilized to wake up other threads.**

4. Can you work through the evolution of the spinlock implementations described in the Anderson paper, from basic test-and-set to the queuing lock? Do you understand what issue with an earlier implementation is addressed with a subsequent spinlock implementation?

  - **Sequence of lock prototypes and their downfalls**:
    - Test-and-Set:
      - Low latency and delay, however, with increase in processors, large amount of contention, excessive memory usage to access lock value.
    - Test and Test-and-Set:
      - Similar to Test-and-Set, however, we check the cached version of the lock prior to conducting Test-and-Set. This prevents excessive memory reads, however, if two processes evaluate the lock to "NOT BUSY", contention occurs for a memory read and the process that reads memory first will most likely acquire the lock first. Then the other process also attempts to Test-and-Set, however, the lock will already be acquired by the faster processor.
    - Test and Test-and-Set and Delay (while busy):
      - Processors will delay after test and test-and-set, stopping everyone

        from trying to acquire the lock at the same time. This improves contention, latency doesn't change, but delay is much worse.
    - Test and Test-and-Set and Delay (every read):
        - Processors will delay after every test and test-and-set iteration. This hurts delay the most because we wait even when there might not be any contention.
    - Static Delay
        - Delay based on a fixed value, however, there can be an unnecessary delay during low contention
    - Dynamic Delay
        - Back-off based, similar to CSMA back off over ethernet. Random delay within a range that increases with each failure to acquire the lock. The problem could be that the delay ends up being based upon the length of the critical section.
    - Queuing Lock
        - Upon contention, each processor is assigned a flag in an array of flags. The lock will be passed to next processor within the array, and the pointer to the next processor in the queue will be incremented.

P3L5

1. What are the steps in sending a command to a device (say packet, or file block)? What are the steps in receiving something from a device? What are the basic differences in using programmed I/O vs. DMA support?

    - **Programmed I/O (PIO) steps**:

        1. The operating system polls the device, reading the status register until the register signifies the device is ready for a command.
        2. The operating system then writes some data to the data register.
        3. The operating system writes a command to the command register.
        4. The operating system polls the status register until the device has completed its operation.

    - **Direct Memory Access (DMA) steps:**

        1. The operating system programs the DMA engine by telling it where the data lives in main memory, how much data to copy, and which device to send the data to.
        2. The operating system completes the transfer and continues other work.
        3. The DMA engine completes its operation and raises an interrupt to signal the operating system that the transfer is complete.

    To receive data from a device, first the status is register is checked to see if the operation is complete. Then the operating system reads data from the data register.

The basic differences between programmed I/O and DMA is that DMA frees the CPU from having to copy data to the device interface. Setting up the DMA takes some time because you also have to write instructions to the DMA. PIO should be used for device operations in which you expect the transfer to be very quick. DMA is used for operations in which you expect the transfer to take longer.

2. For block storage devices, do you understand the basic virtual file system stack, the purpose of the different entities? Do you understand the relationship between the various data structures (block sizes, addressing scheme, etc.) and the total size of the files or the file system that can be supported on a system?

    - **Block devices are represented as files in the filesystem for use by processes. The kernel has its own file system it utilizes to determine where and how to find and access a file. The operating system specifies the interface a file is tied to. At the driver level, a device driver provides a generic block layer, allowing the operating system the ability to standardize interactions with the block device.**
        - user process (file)
        - kernel file system (POSIX API)
        - generic block layer (driver)
        - block device (device specific protocols)
    - **virtual file system:**
        - file-system interface
            - VSF interface
                - local file system (type 1)
                - local file system (type 2)
                - remote file system (type 1)
    - **virtual file system abstractions:**
        - file - elements on which the VFS operates
        - file descriptor - operating system representation of a file
            - open, read, write, sendfile, etc.
        - inode - persistent representation of a file "index"
            - list of all data blocks for a file
            - contains permissions, size, device, etc.
        - dentry - directory entry corresponds to each path component
            - /, /users, /users/aheath9
            - dentry cache - most recently visited directory locations
        - superblock - filesystem-specific information regarding the filesystem layout
            - map of all disk blocks
                - inode blocks

        - data blocks
        - free blocks

3. For the virtual file system stack, we mention several optimizations that can reduce the overheads associated with accessing the physical device. Do you understand how each of these optimizations changes how or how much we need to access the device?
    - **caching / buffering** - reduces the # of disk accesses
        - buffer cache in main memory
        - read / write from cache
        - periodically push data to disk (fsync())
    - **I/O scheduling** - reduces random access and disk head movement
        - maximize sequential access
    - **pre-fetching** - increases cache hits
        - leverages locality
    - **journaling / logging** - reduce random access
        - describes a write in a log: block, offset, value
        - periodically applies updates to disk locations that have records of modification

Definitions

- **Peripheral Component Interconnect (PCI)** - connects multiple devices together across a bus
- **device drivers** - responsible for device access, management and control
- **base address register (BAR)** - the register in main memory used to interact with a device
- **I/O port** - instructions that allow writes directly to the I/O port attached to the bus
- **OS Bypass** - operating system configures communication with the device and then gets out of the way. User-level drivers directly access registers and data of a device.

Formulas

- **directly addressed maximum file size** = number_of_addressable_blocks * block_size
- **number_addressable_blocks_indirect** = number_of_direct_blocks + blocks_addressable_by_single_indirect + (blocks_addressable_by_single_indirect ^ 2) + (blocks_addressable_by_single_indirect ^ 3)
- **blocks_addressable_by_single_indirect** = block_size / pointer_length

P3L6

1. What is virtualization? What’s the history behind it? What’s hosted vs. bare-metal virtualization? What’s paravirtualization, why is it useful?
    - **virtualization** allows concurrent execution of multiple operating systems (and their applications) on the same physical machine
        - Consolidation - decrease cost, improve manageability
        - Migration - availability, reliability
        - Security
        - Debugging
        - Support for legacy OS
    - Virtual resources - each OS thinks that it “owns” hardware resources
    - Virtual Machine(VM) - OS + applications + virtual resources(guest domain)
    - Virtual Machine monitor(vmm) - layer that allows VM to exist
        - VMM is in complete control of system resources
    - **history**:
        - Servers were underutilized, data centers were becoming too large, companies had to hire more system admins, companies were paying high utility bills to run and cool servers. All those reasons created a need for virtualization.
    - **bare-metal (hypervisor-based) virtualization**
        - VMM (hypervisor) - manages all hardware resources and supports execution of VMs
        - privileged, service VM - deals with devices and other configuration / management tasks
        - examples: Xen, ESX
    - **hosted virtualization**
        - host operating system owns all hardware
        - special VMM modules provides hardware interfaces for VMs to interact with and deals with VM context switches
        - examples: KVM (kernel-based VM), KVM kernel module + QEMU (hardware virtualization)
    - **paravirtualization** - guest OS is modified so that
        - It know its running virtualized
        - It makes explicit calls to the hypervisor (hypercalls)

        - Hypercall (~system calls)
            - Package context info
            - Specific desired hypercall
            - Trap to VMM
        - Allows gaining performance by giving up on unmodified guests

2. What were the problems with virtualizing x86? How does protection of x86 used to work and how does it work now? How were/are the virtualization problems on x86 fixed?
    - **x86 Pre 2005**
        - 4 rings, no root/non-root modes yet
        - hypervisor in ring 0, guest OS in ring 1
        - 17 privileged instructions did not trap correctly and failed silently
            - e.g. there was an interrupt enable/disable bit in the privileged register; POPF/PUSHF instructions that attempt to access it from ring 1 failed silently
            - hypervisor doesn't know, so it doesn't change settings for the interrupt bit
            - guest operating system never received feedback, so it assumes change was successful
    - **solution** - binary translation
        - goal: full virtualization where guest operating system is not modified
        - approach: dynamic binary translation
            - main idea: rewrite the VM binary to never issue those 17 instructions
        1. inspect code blocks to be executed
        2. if needed, translate to alternate instruction sequence
        - e.g. to emulate desired behavior, possibly even avoiding trap
        3. otherwise, run at hardware speeds
        - cache translated blocks to amortize translation costs

3. How does device virtualization work? What a passthrough vs. a split-device model?
    - **Memory Virtualization**
        - Full Virtualization
            - All guests expect contiguous physical memory, starting at 0
            - Virtual vs physical vs machine addresses and page frame numbers
            - Still leverages hardware MMU, TLB, etc
        - Paravirtualized
            - Guest aware of virtualization
            - No longer strict requirement on contiguous physical memory starting

at 0
        - explicitly registers page tables with hypervisor
        - Can “batch” page table updates to reduce VM exits
        - Overheads eliminated or reduced on newer platforms

- **Device virtualization**
    - High diversity
    - Lack of standard specifications of device interfaces and behavior
    - Adopt 3 key models for device virtualization
        - **Passthrough model** - VMM-level driver configures device access permissions
            - Pro
                - VM provided with exclusive access to the device
                - VM can directly access the device(bypass VMM)
            - Con
                - Device sharing is difficult
                - VMM must have exact type of devices as what VM expects
                - VM migration is tricky
        - **Hypervisor-direct model** - VMM interrupts all device accesses. Emulates device operation: translate to generic I/O op, traverse VMM-resident I/O stack, invoke VMM resident driver.
            - Pro
                - VM decoupled from physical device
                - Sharing and migration and dealing with device specifics
            - Con
                - Adds latency of device operations
                - Device driver ecosystem complexities in hypervisor
        - **Split device driver** - device access control split between frontend driver in guest VM(device API) and backend driver in service VM(or host)
            - Pro
                - Eliminate emulation overhead
                - Allow for better management of shared devices
            - Con
                - Limited to paravirtualized guests

Final Exam Review Q & A

Part 4

P4L1

1. What’s the motivation for RPC? What are the various design points that have to be sorted out in implementing an RPC runtime (e.g., binding process, failure semantics, interface specification…)? What are some of the options and associated tradeoffs?
    - **RPC** - intended to simplify the development of cross-address space and cross-machine interactions
        - High-level interface for data movement and communication
        - Error handling
        - Hides complexities of cross machine interactions
    - **various design points**
        - Binding - how to find the server
        - IDL - how to talk to the server; how to package data
            - IDL(Interface Definition Language) - used to describe the interface the server exports
                - Procedure name, args and result types
                - Version number
        - Pointers as arguments - disallow or serialize pointed data
        - Partial failures - special error notifications
            - Handling partial failure:
                - Special RPC error notification(signal, exception). Catch all possible ways in which the RPC call can partially fail.
2. What’s specifically done in Sun RPC for these design points – you should easily understand this from your project?
    - Binding - per-machine registry daemon(server registers with local registry daemon)
    - IDL - XDR (for interface specification and for encoding). In SunRPC they use language agnostic RPC called XDR (.x file)
    - Pointers - allowed and serialized
    - Failures - retries; return as much information as possible
3. What’s marshalling/unmarshalling? How does an RPC runtime serialize and deserialize complex variable size data structures? What’s specifically done in Sun RPC/XDR?
    - **marshalling**

        - Converts operation and arguments used for that operation into contiguous buffer.
    - **unmarshalling**
        - Decodes contiguous memory buffer into operation and arguments to perform operation on.
    - This encoding process is typically automatically generated for a particular structure by the RPC toolchain from an IDL specification.

P4L2

1. What are some of the design options in implementing a distributed service? What are the tradeoffs associated with a stateless vs. stateful design? What are the tradeoffs associated with optimization techniques – such as caching, replication, and partitioning – in the implementation of a distributed service?

    - One of the most important design decisions in a distributed service is regarding redundancy: to what extent is state duplicated across machines?

    - Though most designs incorporate both, there are two primary approaches: a replicated system, in which every server stores all data, and a partitioned system, in which each server stores part of the data.

    - **Tracking State**

        - stateless - stateless model requires requests to be self-contained
            - Pros
                - prevents supporting caching and consistency management
                - results in larger requests
            - Cons
                - makes server-side processing simpler and cheaper
        - stateful -
            - offer better performance at the cost of complexity in check-pointing and recovery

    - **Caching**

        - An important optimization to minimize network latency. Clients maintain some file state and operate on it locally. This introduces the problem of consistency and coherence across clients, which is much like the consistency problem between caches in multi-processors

2. The Sprite caching paper motivates its design based on empirical data about how users access and share files. Do you understand how the empirical data translated in specific design decisions? Do you understand what type of data structures were needed at the server- and client-side to support the operation of the Sprite system (i.e. what kind of

information did they need to keep track of, what kinds of fields did they need to include for their per-file/per-client/per-server data structures)?

- **Write-Back Delays**
    - Sprite adopted an approach in which every 30 seconds, dirty blocks that had been since unmodified during that period (i.e. in the last 30 seconds) are written out of a client's cache to the server's cache. After another 30 seconds under the same conditions, it would be written to the server's disk. This design choice was made under the observation that "20-30% of new data was deleted within 30 seconds, and 50% was deleted within 5 minutes;" as such, it made sense to delay disk writes in hopes that they wouldn't be necessary in the first place, since the original write would be deleted.
- **Rejecting Write-on-Close**
    - Part of the rationale to implementing delayed write-back was rejecting the write-on-close alternative. The BSD study indicated to the authors that "75 percent of files are open less than 0.5 seconds and 90 percent are open less than 10 seconds," which indicated that on-close writes would occur very frequently, and thus still incur a high cost overall.
- **Benchmark Estimates**
    - The authors used the BSD study to approximate how many simultaneous clients a machine running Sprite could maintain. They used the "average file I/O rates per active user" to approximate load, and extrapolated this to the maximum total of concurrent users under the various networked filesystem implementations.
- In the Sprite NFS, various parts of the system track different metadata points. For example, to prevent the use of out-dated caches, both the clients and the servers store version information about their files; the clients compare their local versions to that of the server when they open them, in order to see whether or not their version is out-dated. Both endpoints track per-file timers for the delayed write-back, as well as "cacheability" flags that kick in when there's a writer to a file. The server also tracks both the current and the last writer of a file; the last writer is referenced when the server needs its dirty blocks (since these are sometimes fetched on-demand instead of pushed).
- On a per file basis, the client keeps track of:
    - cache (overall yes/no)
    - cached blocks
    - timer for each dirty block
    - version number

    - On a per file basis, the server keeps track of:
        - readers
        - writer
        - version

P4L3

1. When sharing state, what are the tradeoffs associated with the sharing granularity?

    - The choice of granularity is correlated to the layer in the software stack in which the shared memory implementation resides. For example, sharing by the page is typically tightly-involved with the operating system, since the operating system also manages memory by the page. Other approaches, like per-object sharing, are application- or runtime-specific. One thing to be careful of as granularity increases is false sharing, which is the phenomenon that occurs when two applications operate on different portions of the same shared memory region. For example, given some 1KiB page, application instance A might need the first 100B and instance B might need the last 100B. Neither of these technically care that the other is operating in the same page, but since granularity is page-based, there will be a lot of (pointless) synchronization overhead between them.

2. For distributed state management systems (think distributed shared memory) what are the basic mechanisms needed to maintain consistency – e.g. do you understand why it is useful to use "home nodes," and why we differentiate between a global index structure to find the home nodes and a local index structures used by the home nodes to track information about the portion of the state they are responsible for?

    - When distributing state it helps to permanently associate a particular piece of state with a "home." In the context of distributed shared memory, this is called a home node, and it's responsible for maintaining metadata about the state. This includes tracking access and modifications to its memory segments, cache permissions (i.e. whether or not a particular memory location can / should be cached), locking for exclusive access, etc. The home node is different than the state owner, which is the (temporary) owner and operator on a particular memory segment. Of course, the home node does track the current owner; all of this metadata, indexed by the object identifier itself, is maintained locally in a local metadata map.
    - Since the home node is a static property of a memory segment, the mapping between memory and its respective home (called the global map) is distributed and replicated across every node in the system. The prefix of a shared memory address identifies its home node, whereas the suffix identities the resulting frame number. In a way, the address behaves akin to how a virtual address behaves under virtual address translation: the first part is an index into a global mapping to nodes (like the index into the page table), and the second part is a frame number (like the

offset into physical memory).

- If we introduce explicit replicas to the system – existing alongside the on-demand replicas that are already temporarily present in local caches – we can implement features like load balancing, hotspot avoidance, and redundancy. Management of such replicas can either be done by the home node or a higher-level management node, which effectively acts as a home node for one or more backing nodes.

3. What's a consistency model? What are the different guarantees that change in the different models we mentioned – strict, sequential, causal, weak... Can you work through a hypothetical execution example and determine whether the behavior is consistent with respect to a particular consistency model?

   - A consistency model is a convention defined by the DSM implementation; it makes certain guarantees about the availability, validity, and "current-ness" of data to higher-level applications as long as the applications follow the rules. There are different consistency models that guarantee different things with varying levels of confidence, but the most important part is that the applications need to follow the model for those guarantees to hold.

   - The two main operations for managing consistency are push invalidations and pull modifications. The former is a pessimistic approach that assumes any changes will need to be visible elsewhere immediately; it proactively pushes invalidations to other systems as soon as they occur. The latter is a lazier approach with a more optimistic outlook; as memory is needed, modification info is reactively pulled from other nodes on demand.

   - **Strict Consistency**

     - This is the simplest model: every update is seen everywhere, immediately, and in the exact order in which it occurred.
     - This model is purely theoretical: in practice, even with complex locking and synchronization, it is still impossible to guarantee (or sustain) this level of consistency because of physical complexities introduced by packet loss and network latency.

   - **Sequential Consistency**

     - Sequential Consistency This alternative model is the "next best thing" regarding an actually-achievable consistency. Here, we only guarantee that any possible ordering of the operations – in other words, any ordering that would be possible in execution on a regular, local shared memory system (think data-races between threads).
     - In summary, memory updates from different processes may be arbitrarily interleaved. There is a catch, though: this model must guarantee that the interleaving is the same across all processes. In other words, another process

P4 must see the same order as P3.

- **Casual Consistency**
  - The causal consistency model enforces consistency across relationships between memory locations.
  - As in sequential consistency, writes occurring on the same processor must be seen in the same order that they occur. The causal consistency models make no guarantees about concurrent writes (i.e. writes that are not causally related), though. Like in the sequential consistency model, reads occur in any possible ordering of the writes, but unlike in the sequential consistency model, these interleaves do not need to be consistent across processes.
- **Weak Consistency**
  - This model enables the programmer to define explicit synchronization points that will create consistency with the system.
  - A synchronization point does the following:
    - It enforces that all internal updates can now be seen by other processors in the system. • It enforces that all external updates prior to a synchronization point will eventually be visible to the system performing the synchronization.
  - A weak consistency model allows the programmer to define explicit synchronization points that are important to their application, essentially managing causal relationships themselves. The model makes no guarantees about what happens between sync operations. Updates can (or can not) arrive in any order for any variable. Variations on the model include a global sync operation that synchronizes the entirety of the shared memory, or more granular synchronization that operates on a per-object or per-page basis.

P4L4

1. When managing large-scale distributed systems and services, what are the pros and cons with adopting a homogeneous vs. a heterogeneous design?
   - A functionally homogeneous approach to distributed systems means every system is equal: it executes every possible step necessary for the processing of a request. This is much like the standard "boss-worker" model, in which workers are generic threads that process all of the steps in a certain task. Alternatively, a functionally heterogeneous approach isolates certain steps to certain machines, or for certain request types. This is somewhat akin to the "pipeline" model, in which individual steps of a task are isolated and can be scaled according to the time it takes to complete them.
   - The tradeoffs between each approach are similar to the tradeoffs we discussed in

Part I of this course regarding boss-worker and pipeline models, with some additional tradeoffs that are a part of the "distributed" aspect of things. A homogeneous model is simpler; the front-end is more straightforward and just behaves as a simple dispatcher. Scaling is simpler, as it just requires adding more nodes to the system. Of course, the simplicity and generality comes at a performance cost, since specific things can't be optimized. Caching is difficult, since a simple front-end isn't aware of which node processed which task, and thus can't make any inferences about existing workload state. With a heterogeneous design, both the front-end and the overall management is more complex, but now the dispatcher (and workers themselves) can make meaningful inferences about data locality and caching in its workers.

2. Do you understand the history and motivation behind cloud computing, and basic models of cloud offerings? Do you understand some of the enabling technologies that make cloud offerings broadly useful?

    - Cloud computing is beneficial due to its elasticity and abstraction. There's no need to manage infrastructure; the scale of your application is directly correlated with its demand, saving money and resources. Management of your application is standardized and easily-accessible via a web API provided by the cloud computing service. Billing is based on usage, typically by discretized "tiers" that correspond to the power of a rented system.

    - The cloud operates on a massive scale and requires technologies that support such a scale. Aside from the obvious need for virtualization to isolate tenants, clouds also need resource provisioning technologies to fairly allocate, track, and manage resources for their tenants. Furthermore, big data technologies are essential in storing and processing the insane amounts of data that their customers (and the providers themselves) produce or consume. The necessity of flexible "virtual infrastructure" by customers also means software-defined solutions for networking, storage, etc. are essential in providing these definitions. Finally, monitoring at scale requires complex real-time log processing solutions to catch and analyze failures.

3. Do you understand what about the cloud scales make it practical? Do you understand what about the cloud scales make failures unavoidable?

    - The Law of Large Numbers makes cloud services viable. Different clients, of which there are many, have different usage patterns and different peak times. Averaging these factors leads to a relatively steady pattern of resource usage for the provider.
    - Unfortunately, this same law leads to unavoidable failures, too. The scale at which cloud services operate means massive amounts of hardware. Even if the failure rate of a particular component – say, a hard drive – is 0.01% a month (that's 1 in 10,000), a sufficiently-large data center might have 100,000 hard drives. That means about 10 hard drives will fail a month, every month! Of course, that's just hard drives; there

are hundreds, if not thousands, of individual components in a particular system. Thus, individual failures happen with a fairly high frequency.