CS 6200 Introduction to Operating Systems Lecture Notes

Jie Wu, Jack

Spring 2022

1 MODULE 1

1.1 Lecture 1 Course Overview

1.1.1 Course Overview

- processes and process management
- threads and concurrency
- resource management (scheduling, memory management)
- OS services for communication and I/O
- OS support for distributed services (RPC, distributed files and memory)
- systems software for data center and cloud environment

1.1.2 Theory and Practice

project 1: threads, concurrency, and synchronization

project 2: single-node OS mechanisms (inter-process communication, scheduling etc.)

project 3: multi-node OS mechanisms (remote procedure calls)

project 4: experimental design and evaluation

1.1.3 Recommended Textbooks

- Abraham Silberschatz, Peter B. Galvin, Greg Gagne, *Operating System Concepts*, Wiley, 9th edition (2012)
- Abraham Silberschatz, Peter B. Galvin, Greg Gagne, *Operating System Concepts Essentials*, Wiley, 2nd edition (2013)
- Andrew Tanenbaum, Herbert Bos, *Modern Operating Systems*, Pearson, 4th edition (2014)
- Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, CreateSpace Independent Publishing, 1st edition (2018)

1.2 Lecture 2 Introduction to Operating Systems

1.2.1 Lesson Preview

abstract: to simplify what the hardware actually looks like

arbitrate: to manage, oversee, and control hardware use

1.2.2 Visual Metaphor

An operating system

directs operational resources: control use of CPU, memory, peripheral devices etc.

enforces working policies: fair resource access, limits to resource usage

mitigates difficulty of complex tasks: abstract hardware details (system calls)

1.2.3 What is An Operating System?

hide hardware complexity: storage: read/write files, network: send/receive sockets

resource management: storage management, CPU scheduling

provide isolation and protection:

1.2.4 Operating System Definition

An operating system is a layer of systems software that

- directly has privileged access to the underlying hardware (vs. application software don't have)
- 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

1.2.5 Operating System Components Quiz

The OS does NOT directly manage the cache. It is the hardware that manages itself. Thus cache memory is unlikely a component of an OS.

1.2.6 Abstraction or Arbitration Quiz

abstraction: look for keywords such as "support", "access", "use"

arbitration: look for keywords such as "distribute", "allocate"

1.2.7 Operating Systems Examples

desktop: Windows-based vs. Unix-based (including Mac OS and Linux)

embedded: Android (embedded form of Linux), iOS, Symbian

This course will focus on Linux.

1.2.8 OS Elements

abstraction: process, thread which correspond to the applications that the OS executes; file, socket, and memory page which correspond to the hardware resource that the OS needs to manage

mechanism: create, schedule (for process, thread); open, write, allocate (for file, socket, and memory page)

policy: e.g. least-recently used (LRU), earliest deadline first (EDF)

1.2.9 OS Elements Example

least recently used (LRU): the pages that have been least recently used over a period of time are the ones that will no longer be in physical memory and instead will be copied on disk (termed "swapping"), because the pages that have been least recently used are likely not to be as important, or likely will not be used in the near future

1.2.10 Design Principles

separation of mechanism and policy: implement flexible mechanisms to support many policies (e.g. LRU, LFU, or random for memory management)

optimize for common case: where will the OS be used? what will be executed? what are the workload requirements? need to understand the common case and then based on that common case pick a specific policy that can be supported given the underlying mechanism and abstraction

1.2.11 OS Protection Boundary

Computer platforms distinguish between at least two modes

user mode: unprivileged, the applications operate in unprivileged user mode

kernel mode: privileged, the OS must operate in privileged kernel mode in order to have direct access to the hardware

Switching between user mode and kernel mode is supported by the hardware on most modern platforms, which can be caused by

trap instruction: an instruction in user mode that attempts to perform privileged operation which will be forbidden ("trapped" in CPU), when such instruction arrives the operation will be interrupted, and the hardware will give the control to the OS to check what caused the trap and verify if it should be granted the access or terminated

system call: to perform certain operations the application may need to invoke the OS to export a system call interface which provides required privileged access (e.g. open a file, send a socket, allocate memory), an OS also supports signals which is a mechanism for to pass notifications to the applications

1.2.12 System Call Flowchart

Executing a system call involves

1. changing the execution context from the user mode to the kernel mode, which passes the control to the OS

2. passing arguments necessary for the OS to cooperate
3. jumping into the memory of the kernel so that the instruction sequence that corresponds to the system call can be implemented
4. changing the execution context from the kernel mode to the user mode, which returns the result and the control to the calling process once the system call completes

To make a system call an application must

- write arguments, the arguments can either be passed directly to the OS or indirectly by specifying their address in the register
- save relevant data at a well-defined location, so that the kernel can determine which arguments and where it should retrieve based on the system call number
- make system call using specific system call number

There are two types of system calls:

synchronous system call: the calling process will wait until the system call completes

asynchronous system call: discussed later

1.2.13 Crossing the OS Boundary

User-kernel transition is a necessary step during application execution. The hardware provides support for performing user-kernel transition (e.g. traps on illegal instructions, memory access requiring special privilege because the application cannot change the contents of certain registers). The drawback is

- it involves a number of instructions thus takes some time (50–100ns on a 2GHz Linux machine)
- it switches locality thus consumes hardware cache (much faster than memory: ~ 1 cycle vs. ~ 100 cycles) which may otherwise be used to store application content (a cache is hot/cold if it contains/does not contain the data or address the application needs, in the later case the application is forced to retrieve the required data or address from the much slower main memory)

Therefore user-kernel transition is not cheap (Errata: the performance of the application can benefit or suffer based on how a context switch changes what is in cache at the time it is accessing).

1.2.14 System Calls Quiz

On a 64-bit Linux-based OS the system calls 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

1.2.15 Monolithic OS

Historically the operating system had a monolithic design—including every possible service that any application can require or any hardware can demand. The pro and con are

pro: everything is included and packaged at the same time, which allows compile time optimization

con: too many states, too much code to maintain, debug and upgrade, large size poses large memory requirement which impacts the performance

1.2.16 Modular OS

A more common approach today is the modular approach as with the Linux OS, in which the OS specifies the interfaces that any module must implement in order to be part of the OS, and a module can be dynamically installed depending on the workload. The pro and con are

pro: easier to upgrade, less code to maintain, less resource required thus leaving more memory for the applications

con: indirect interaction through module interface impacts the performance (typically not very significant though), maintenance is still an issue since modules may come from completely disparate code bases

1.2.17 Microkernel

Commonly used in embedded devices and control systems, microkernel only requires the most basic primitives at the OS level. All other applications as well as software that we typically think of as an OS component like file system and device driver will run outside the OS kernel at the unprivileged user mode. For this reason microkernel requires lots of user-kernel transitions and inter-process interactions (so typically the microkernel itself will support inter-process communications as one of its core abstractions and mechanisms along with address spaces and threads). The pro and con are

pro: small thus lower overhead, less code thus easy to verify

con: very specialized thus affects portability resulting in software complexity, frequent inter-process communications and user-kernel transitions are costly

1.2.18 Linux and Mac OS Architecture

The Linux architecture consists of

user mode: including

- **user:** users
- **standard utility programs:** shell, editor, compiler
- **standard library:** open, close, read, write

kernel mode: including

- **Linux OS:** process management, memory management, file system, I/O
- **hardware:** CPU, memory etc.

The system call interface connects the standard library to the modular OS.
The Mac OS architecture consists of

graphical user interface: Aqua

application environments and services: Cocoa, Quicktime

kernel environment: Mac microkernel and BSD (Berkeley software distribution)

I/O kit, kernel extensions:

2 MODULE 2

2.1 Lecture 1 Processes and Process Management

2.1.1 Process and Process Management Preview

Synonymous with task or job, a process is an instance of an executing program.

2.1.2 What is a Process

application: program on disk, flash memory etc. (static entity)

process: state of a program when executing loaded in memory (active entity)

Once started executing, an application becomes a process. A process represents the execution state of an active application.

2.1.3 What Does a Process Look Like

Every single element of the process state has to be uniquely identified by its address. An OS abstraction used to encapsulate all of the process states is **address space**, which is defined by a range of addresses from V_0 to $V_{\max}$. The types of state include

text and data: static state when process first loads

heap: dynamically created during execution to allocate memory, store temporary result, and read data from files, there may be portions of it that is not used or even inaccessible by the process

stack: grows and shrinks like a LIFO queue

2.1.4 Process Address Space

The addresses in the address space are called **virtual addresses**, since they are used by the process to reference its states and don't have to correspond to actual locations in the physical memory. The memory management hardware and the OS components responsible for memory management like page tables maintain a mapping between the virtual addresses and the physical addresses. This mapping decouples the address space from the physical memory to allow simple physical memory management.

2.1.5 Address Space and Memory Management

Not all processes require the entire address space from V_0 to $V_{\max}$. Also there may not be enough physical memory to store all the process states, in which case the OS dynamically decides which portion of the address space of which process will be present in the physical memory.

2.1.6 Virtual Addresses Quiz

The fact that we have decoupled the virtual addresses that are used by the processes from the physical addresses makes it possible for different processes to have exactly the same address space range.

2.1.7 Process Execution State

program counter: at any given point of time the CPU needs to know where in the instruction sequence the program currently is, this is realized by program counter, which is maintained on the CPU while the process is executing in a register, since it changes frequently the CPU has a dedicated register to track the current program counter for the currently executing process, however it is the OS's job to collect, and save all the information that the CPU maintains for a process and store it in the PCB whenever that particular process is no longer running on the CPU (ref. the section on PCB)

stack pointer: the top of the process stack is defined by the stack pointer

To maintain all of this information for every single process, the OS maintains what is called a process control block or PCB.

2.1.8 Process Control Block

A **process control block** (PCB) is a data structure that the OS maintains for every process it manages. Besides memory mapping, program counter, and stack pointer, it include other useful information such as registers, memory limits, a list of open files, priority, signal mask, and CPU scheduling information. A PCB is created and initialized when a process is created.

2.1.9 How is a PCB Used

Each time a swapping between processes is performed, the OS performs what is called context switch, which involves saving the PCB of the process that is interrupted and restoring the PCB of the process that is to be executed.

2.1.10 Context Switch

Context switch is the mechanism used by the OS to switch execution from the context of one process to that of another process. This operation can be expensive because

direct cost: number of cycles for loading and storing the values of the PCBs to and from the memory

indirect cost: *cache miss*—some or all the data of process #1 stored in the processor cache will be replaced to make room for the data needed by process #2, thus next time process #1 is scheduled to execute its data will not be in the cache ("cold cache") but have to be read from the memory

As a result we need to limit the frequency of context switching.

2.1.11 Process Life Cycle: States

The states that a process is going through throughout its life cycle is depicted in Figure 1 below.

admitted: the OS performs admission control, if passed the OS will create and initiate a PCB for this process

interrupt: the OS performs context switching

2.1.12 Process State Quiz

The CPU is able to execute a process when the process is in (1) running or (2) ready state. A process enters the ready queue from the running state due to either scheduler interrupt or I/O and other event wait.

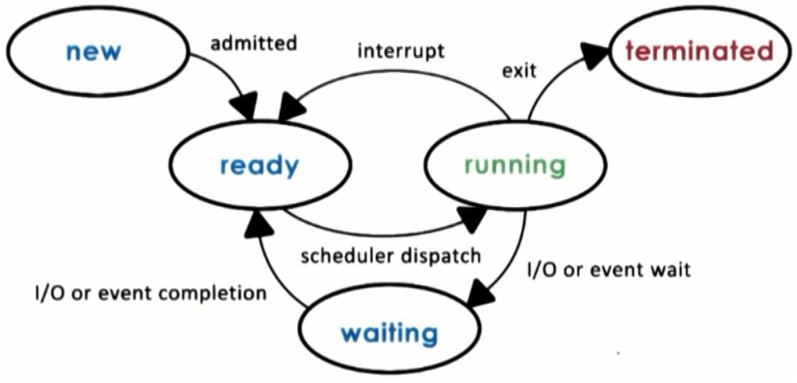

Figure 1: various states in a process life cycle

2.1.13 Process Life Cycle: Creation

In OS a process can create child processes. Once the initial boot process is done and the OS is loaded on the machine, it will create some number of initial processes (root processes). When a user logs into a system, a user shell process is created. When the user types in commands, new processes get spawned from that shell parent process.
Most OS support two basic mechanisms for process creation

fork: copy the parent PCB into the PCB of the new child, and because the PCB includes the program counter, both the parent and the child will start their execution at exactly the <u>same</u> point

exec: still take the PCB created by fork, but the OS will replace the child's image and load a new program, and the program counter of the child's PCB will now point to the <u>first</u> instruction of this new program

2.1.14 Parent Process Quiz

On Unix-based OS the parent of all processes is init. On Android-based OS the parent of all processes is Zygote—the OS forks the Zygote process every time a new app needs to be created.

2.1.15 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. This means that in order to manage the CPU the OS must be able to

preempt: interrupt the executing process and save its current context

schedule: run the scheduler to choose next process

dispatch: dispatch next process to the CPU and switch to its current context

Given that the CPU resources are precious, the OS needs to ensure that the CPU time is spent running processes i.e. minimizing the CPU time spent on the above three tasks.

2.1.16 Length of Process

Let T_p denote the time allocated to a process on the CPU which is usually referred to as "time slice", and T_{sche} the time spent scheduling next process to dispatch, the useful CPU work in percentage is

$$\text{useful CPU work} = \frac{2 \cdot T_p}{2 \cdot T_p + 2 \cdot T_{\text{sche}}}$$

Scheduling design decision involves

- what are the appropriate time slice values
- metrics to choose next process to run

2.1.17 What about IO

A process can make its way to the ready queue through the following ways

- waiting for an interrupt which ultimately occurs
- waiting for an I/O event which ultimately occurs
- its time slice expired
- created via the fork call

2.1.18 Scheduler Responsibility Quiz

The scheduler has no control over when I/O operations occur, one exception being the timer interrupt. The scheduler has no control over when to generate the external event that a process is waiting for either, but again the timer interrupt is one exception.

2.1.19 Inter Process Communication

Inter process communication (IPC) mechanisms

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

There are two mechanisms of IPC that the OS supports:

message passing based: the OS establishes a communication channel e.g. a shared buffer, and the processes interact with it by writing/sending a message into or reading/receiving a message from that buffer. The pro and con are

> **pro:** exactly the same APIs and the same system calls for all processes
>
> **con:** overhead—every message has to be copied to the buffer first

shared memory based: the OS establishes a shared memory and maps it to the address spaces of both processes, which are then allowed to directly read into and write from this memory as any memory location that is part of their virtual address space. The pro and con are

> **pro:** the OS is not in the path of communication thus no overhead
>
> **con:** the OS no longer supports fixed and well-defined APIs how the shared memory is used, as a result it becomes more error prone, or the developer has to re-implement the code to use the shared memory

2.1.20 Shared Memory Quiz

Since the operation of mapping the address spaces of both processes into the shared memory is expensive, shared memory based communication is better than message passing based communication only if the mapping cost can be amortized across a sufficiently large number of messages.

2.2 Lecture 2 Threads and Concurrency

2.2.1 Threads and Concurrency Preview

To take advantage of multi-CPU systems a process needs to have multiple execution contexts. We call such execution context within a single process "thread".

2.2.2 Process vs. Thread

Threads share all of the virtual to physical address mappings and all the data, code, and files. However they would potentially execute different instructions and access different portions of the address space. This means that they need to have different program counters, stack pointers, and registers.

2.2.3 Benefits of Multithreading

parallelization: each thread executes the same code but for a different subset of the input thus speed up input processing

specialization: each thread executes different portions of the code, specializing different threads to run different tasks permits execution with hotter cache (store the more repetitive portion in the cache)

address space and execution context sharing: lower memory requirement, the process is more likely to fit in memory and not to require as many swaps from disk compared to a multi-process alternative, and inter process communication can be avoided

2.2.4 Benefits of Multithreading: Single CPU

The question can be phrased more generally as: are threads useful when number of threads > number of CPUs. As long as the idle time is greater than the time of context switching to and back from another thread, it is worth context switching to another thread to hide the idle time. Recall that one of the most costly steps in context switch is the creation of new virtual to physical address mapping, which can be avoided by the threads sharing the address space. Hence the time of context switch among threads is shorter than the time of context switch among processes, and therefore more idle time can be hidden by context switch. In summary multithreading allows us to hide more latency that is associated with IO operations, and this is useful even on a single CPU.

2.2.5 Benefits of Multithreading: Apps and OS

Multithreading the OS's kernel allows the OS to support multiple execution contexts (on behalf of applications or for OS-level services such as daemons and device drivers).

2.2.6 Process vs. Threads Quiz

thread: can share a virtual address space, and usually result in hotter caches

process: takes longer to context switch

both: have an execution context, make use of some communication mechanisms

2.2.7 Basic Thread Mechanisms

mutual exclusion: a mechanism where only one thread at a time is allowed to perform an operation, which is necessary to avoid data race that might occur, since multiple threads share the same address space and thus the same virtual physical address mapping

condition variable: waiting for other threads based on the condition specified by a condition variable (ref. Birrell's paper)

These two mechanisms are referred to as synchronization mechanisms.

2.2.8 Thread Creation

The thread type proposed by Birrell is the data structure that contains all the information that can describe a thread including thread ID, program counter, stack pointer, register etc. For thread creation Birrell proposes a fork call with two parameters *Fork(proc, args)* where

proc: the procedure that the created thread will start executing

args: arguments for the procedure

When a thread $T0$ calls a fork, a new thread data structure of $T0$ type is created and its fields are initialized such that its program counter will point to the first instruction of the procedure, and the arguments will be available on the stack of this new thread $T1$. The parent thread $T0$ will have to wait until the child thread $T1$ finishes the processing before it can exit so as not to force early termination. To deal with this issue Birrell proposes a join call *Join(child thread ID)*, which will return to the parent thread the result of the child thread's computation. At that point any data structure state and resource allocated to the child thread will be freed.

```
T1 = fork(safe_insert, 4);
safe_insert(6);
join(T1);
```

2.2.9 Mutual Exclusion

When a thread locks a mutex, it has exclusive access to the shared resource. The portion of the code protected by the mutex is called **critical section**.

```
lock(m){
    // critical section
}//unlock
```

We will use Birrell's lock construct throughout this lecture, although common thread API uses lock call and unlock call explicitly before and after critical section.

2.2.10 Mutex Quiz

Since both $T2$ and $T4$ have attempted to get the lock before it was released, their requests will be in the queue of pending requests, although the specification of the mutex does not make any guarantee regarding which one will get the lock first. However $T5$ requests the lock at the same time $T1$ releases the lock, $T5$ is the one that actually gets the lock.

2.2.11 Condition Variable

Instead of locking to check the condition frequently, Birrell argues that a condition variable should be used in conjunction with mutex, which invokes suspension and the release of lock if the condition is not fulfilled.

```
lock($m$){
    while (my_list.not_full())
        wait($m$, list_full);
    my_list.print_and_remove_all();
}//unlock
```

Note that the waiting operation is inside the critical section, which means that when the thread exits the wait state upon receipt of notification, it has to reacquire the mutex.
The producer on the other hand checks condition variable after each operation and sends out the signal to the waiting thread when the condition is fulfilled.

```
lock($m$){
    my_list.insert(my_thread_id);
    if my_list.full()
        signal(list_full);
}//unlock
```

2.2.12 Condition Variable API

A condition variable API should have a data structure that contains the following components:

- condition type
- wait(mutex, condition), where mutex is automatically released and reacquired on wait
- signal(condition), which notifies only one waiting thread, or broadcast(condition), which notifies all waiting threads

Note that since the waiting threads have to reacquire the mutex when they exit the wait state, it means that although broadcast is able to wake up all the threads on wait queue, these threads can reacquire the mutex only one at a time.

2.2.13 Condition Variable Quiz

Only while loop can support multiple waiting threads, because only while loop can ensure that when the thread wakes up upon receipt of the signal, the condition is still fulfilled, as it is possible that other waiting threads may wake up and reacquire the mutex first and change the condition.

2.2.14 Readers and Writer Example

The fact that there can be many readers but only one writer in operation results in the following

enter critical section: the predicate while loop checks is

$$\text{[reader] resource\_counter} == -1 \text{ vs. resource\_counter != } 0 \text{ [writer]}$$

s update predicate

$$\text{[reader] resource\_counter++ vs. resource\_counter} \leftarrow -1 \text{ [writer]}$$

exit critical section: update predicate

$$\text{[reader] resource\_counter}-- \text{ vs. resource\_counter} \leftarrow 0 \text{ [writer]}$$

signal/broadcast

$$\text{signal(writer\_phase) if resource\_counter} == 0 \text{ [reader]}$$

$$\text{vs. signal(writer\_phase) and broadcast(read\_phase) [writer]}$$

2.2.15 Common Pitfalls

Safety tips are

- use a single mutex to access a single resource
- the order of signal or broadcast doesn't make any priority guarantee regarding which thread will execute next

2.2.16 Spurious Wake Ups

Spurious wake ups occurs when we wake threads up by sending out signal or broadcast before releasing the mutex, so that these awaken threads cannot get the lock and proceed with their operations. This can be avoided by unlocking the mutex before signal or broadcast (e.g. for readers). This solution is viable only if the signal or broadcast does not depend on the condition variable which requires mutex protection because it is shared (e.g. for writers).

2.2.17 Deadlocks Introduction

Deadlock occurs when two or more competing threads are waiting on each other to complete, yet none of them ever do because each waits on the other one.

2.2.18 Deadlocks

Deadlock occurs when two threads are waiting for the resources each other has. Maintaining a lock order will prevent it from occurring. But it takes effort to ensure the order if there are many mutexes.

2.2.19 Deadlock Summary

A cycle in the wait graph (pointing from the thread waiting on a resource to the thread owning that resource) is necessary and sufficient for a deadlock to occur. There are three strategies to cope with it:

prevention: every time when a thread is about to issue a request for a lock, check if it will cause a cycle in the wait graph, and change the code to make this thread release some resource first if it does, which is expensive

detection and recovery: it requires the ability to roll back the execution once deadlock occurs and is detected, which requires maintaining enough execution states and is impossible if external I/Os are involved

Ostrich algorithm: do nothing hoping for no deadlock, reboot if deadlock does occur

The first two are expensive and thus are applied only to performance critical systems.

2.2.20 Kernel vs. User-Level Threads

For a user-level thread to actually execute, first it must be associated with a kernel-level thread, and then the OS-level scheduler must schedule that kernel-level thread onto a CPU. There are three possible relationships/models between the user-level threads and the kernel-level threads.

2.2.21 Multithreading Models

one-to-one model: the pro and con are

- **pro:** the OS can see all of the user-level threads and understands thread needs such as synchronization, blocking etc.
- **con:** every operation has to go to the OS via system call which can be expensive, and the OS may have limitation on the policy it can support (which affects portability since only the kernel that supports the policy can be used) and on the number of threads available

many-to-one model: all the user-level threads are mapped onto a single kernel-level thread, which requires a thread management library at the user level, the pro and con are

- **pro:** portable since it does not depend on the OS and policy, cheap since user-kernel transition via system call is avoided
- **con:** the OS has no insight into application needs, and the OS may block the entire process if one user-level thread blocks on I/O

many-to-many model: can have the best of both one-to-one and many-to-one, and the user-level threads can either be scheduled onto any of the kernel-level threads (unbound mapping) or mapped one-to-one permanently onto a kernel-level thread for higher priority or better responsiveness etc. (bound mapping)
the downside is that it requires some coordination between user-level thread management (many-to-one) and kernel-level thread management (one-to-one)

2.2.22 Scope of Multithreading

system scope: at kernel level system-wide thread management by OS-level thread managers e.g. CPU scheduler

process scope: at user level different processes will be managed by different instances of the same or entirely different user-level management libraries

If the user-level threads have a process scope and the OS does not see all of them, then the available physical resource may be allocated equally to different processes; if the user-level threads have a system scope thus they will be visible at the kernel level, then the available physical resource may be allocated in proportion to the number of user-level threads the processes have.

2.2.23 Multithreading Patterns

We will look at (1) boss-workers pattern (2) pipeline pattern (3) layered pattern.

2.2.24 Boss/Workers Pattern

boss: one thread, assigns work to workers

worker: multiple threads, perform the task

The pro and con are

pro: workers don't need to synchronize

con: the throughput of the system is limited by the boss' performance, and since the boss must track what each worker is doing, the throughput is low

Worker tracking can be avoided by assigning work through placing work in producer/consumer queue, which however would require the synchronization of the boss (check whether the queue is full) and the workers' access to the shared queue. Despite this need of synchronization the queue mechanism still improves throughput thus will be adopted here.

2.2.25 How Many Workers

Boss-worker communication via producer/consumer queue and a pre-assigned dynamically adjusted worker pool is the most effective boss-worker pattern. The advantage is its simplicity, but the shortcoming is the overhead needed for worker pool management as well as the shared queue synchronization. Another disadvantage is that since the boss doesn't track the work, it ignores locality—a worker will be more efficient at performing exactly the same task in the future or a worker already has some of the tools required for a particular task.

2.2.26 Boss/Workers Variant

We can have different workers specialize in different tasks for better locality. However this brings in the challenge of load balancing—how many workers should be assigned to different tasks?

2.2.27 Pipeline Pattern

The overall task is divided into subtasks, and each of the subtasks is performed by a separate thread. Multiple tasks can still be run concurrently in the system. It is just that different tasks will be in different pipeline stages. The throughput of this model depends on the weakest link—the longest stage in the pipeline. To cope with it we can assign a larger thread pool to a longer stage, so the pipeline overall will still be balanced. The best way to pass work among these different pipeline stages is using a shared-buffer based mechanism similar to the producer/consumer queue—the worker in the earlier stage places the output in the queue, which will be picked up by the later stage worker when it is free.

pro: higher specialization and better locality (more likely that the execution states are in cache)

con: complicated to maintain balanced pipeline and synchronized queue

2.2.28 Layered Pattern

A layered model is one in which each layer is assigned a group of related tasks, and the threads that are assigned to a layer can perform any one of the subtasks that correspond to it. Different from the pipeline model a task must pass up and down through all the layers i.e. go in both directions.

pro: high specialization and good locality, less fine-grained than the pipeline model thus easier to decide how many threads should be allocated to a layer

con: may not be suitable for all applications (when it does not make sense to group subtasks), more complex synchronization (now bidirectional)

2.2.29 Multithreading Patterns Quiz

For the boss-worker model the time formula is

$$\text{time to finish } n \text{ orders} = \text{time to finish one order} \times \left\lceil \frac{n}{\text{number of concurrent workers}} \right\rceil$$

For the pipeline model the time formula is

$$\text{time to finish } n \text{ orders} = \text{time to finish first order} + (n-1) \times \text{time to finish last stage}$$

2.3 Lecture 3 Threads Case Study: PThreads

2.3.1 PThreads Preview

PThreads stands for POSIX threads where POSIX is the acronym of "Portable Operating System Interface". Within POSIX PThreads describes the threading related API that operating systems need to support. It is the de facto standard of Unix system.

2.3.2 PThread Creation

PThread implementation of Birrell's mechanisms is

thread: pthread_t t_id

fork(proc, args): int pthread_create(pthread_t *t_id, const pthread_attr_t *attr,
void *(*start_routine)(void *), void *arg)

join(thread): int pthread_join(pthread_t t_id, void **status)

PThread attributes can be

initialize: int pthread_attr_init(pthread_attr_t *attr)

destroy: int pthread_attr_destroy(pthread_attr_t *attr)

set/get: pthread_attr_{set/get}{attribute}

In PThreads the default behavior of thread creation is *joinable*—parent thread creates children threads which can join parent thread at a later time. The mechanism that is implemented in PThread but not considered by Birrell is *detachable* threads. In PThreads children threads can be detached from parent thread. Once detached they cannot join parent thread later. Detachable threads can be created by int pthread_detach(t_id) or phthread_attr_setdetachstate(attr, PTHREAD_CREATE_DETACHED)

2.3.3 PThread Creation Quiz 3 Question

Data race occurs when one thread tries to read a variable that another thread is modifying.

2.3.4 PThread Mutexes

Birrell's mutex mechanism is implemented in PThreads as follows

pthread_mutex_t m;
int pthread_mutex_lock(pthread_mutex_t *m);
int pthread_mutex_unlock(pthread_mutex_t *m);

pthread_mutex_lock can be replaced by pthread_mutex_trylock, which checks whether the mutex is available before attempting to lock it.
You need to initialize the mutex before using it, and make sure that you free it up at the end

```
int pthread_mutex_init(pthread_mutex_t *m, const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *m);
```

Passing NULL to pthread_mutexattr_t specifies the default behavior which makes the mutex private to a process thus only visible to the threads within a process.
Mutex safety tips are as follows

- shared data should always be accessed through a single mutex
- mutex scope must be visible to all threads of interest i.e. must declare mutexes as global variables
- globally order locks i.e. for all threads lock mutexes in order to prevent deadlock
- always unlock a mutex

2.3.5 PThread Condition Variables

Birrell's condition variable mechanism is implemented in PThreads as follows

```
pthread_cond_t c;
int pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
int pthread_cond_signal(pthread_cond_t *c);
int pthread_cond_broadcast(pthread_cond_t *c);
```

Identical to Birrell's condition variable mechanism

- a thread entering the wait operation will automatically release the mutex and place itself on the wait queue that is associated with the condition variable
- when the thread is woke up it will automatically reacquire the mutex before exiting the wait operation

Similar to mutex you need to initialize the condition variable before using it, and make sure you free it up at the end

```
int pthread_cond_init(pthread_cond_t *c, const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *c);
```

Again passing NULL to pthread_condattr_t specifies the default behavior, which makes the condition variable private to a process thus only visible to the threads within a process.
Condition variable safety tips are as follows

- do not forget to notify waiting threads, signal/broadcast the correct condition variable when the predicate changes
- when in doubt use broadcast instead of signal to figure out the what the desired behavior is, note that if signal is desired but broadcast is used instead, you will lose performance, thus the choice between signal and broadcast does matter
- since you don't need a mutex to signal/broadcast, it may be appropriate to signal/broadcast after unlocking the mutex

2.4 Lecture 4 Threads Design Considerations

2.4.1 Thread Design Considerations Preview

We will be looking at

- two types of threads: kernel-level vs. user-level
- two notification mechanisms: interrupt vs. signal

2.4.2 Kernel vs. User Level Threads

Thread abstraction, scheduling, synchronization etc. are provided by

user level: user-level library

kernel level: OS kernel

2.4.3 Thread Data Structures: Single CPU

From the perspective of the user-level threading library, the underlying kernel-level threads look like virtual CPUs. The user-level library decides which user-level threads will be scheduled onto the underlying kernel-level threads.

2.4.4 Thread Data Structures: At Scale

Kernel-level threads that belong to different processes point to different process control blocks (PCBs) and have different virtual address mappings. For multi-threaded processes the PCB of the first kernel-level thread is saved. If context switching to a second kernel-level thread occurs, the PCB of the second kernel-level thread will be restored.

2.4.5 Hard and Light Process State

The information contained in the PCB is separated into

hard process state: relevant for all the user-level threads within the process

light weight process state: only relevant for a subset of the user-level threads currently associated with one particular kernel-level thread

2.4.6 Rationale for Multiple Data Structures

Single PCB → multiple data structures brings in the following benefits

scalability: large contiguous data structure → pointers pointing to much smaller data elements

overhead: private, one copy for each entity → easy to share those components that are identical across threads or processes, create new elements only when different information is needed

performance: save and restore on each context switch → only save and restore what needs to change, modifications impact only subset of data elements

flexibility: update for any change → user-level library needs only update a portion of the state

2.4.7 Thread Structures Quiz

The name of the kernel thread structure is kthread_worker. The name of the data structure contained in the above structure that describes the process the kernel thread is running is task_struct.

2.4.8 User Level Structures in Solaris 2.0

- The thread is not POSIX thread but similar. Its ID is not a pointer to the actual data structure, instead it is an index in a table of pointers. It is the table pointer that points to the actual thread data structure. By indexing to encode information into the table entry this design provides meaningful feedback for debugging. In addition data structure layered in a contiguous way help us achieve locality
- The problem of this threading library is that the private stack of the threads can grow without check, and the error of one thread may be due to another unknown thread writing into its stack making debugging difficult. The solution is to separate the information belonging to different threads with a "red zone". This helps identify which thread causes the error (the one that steps into the red zone of others).

2.4.9 Kernel Level Structures in Solaris 2.0

It consists of four data structures

process: include a list of kernel-level threads, virtual address space mapping, user credentials, signal handlers

light-weight process (LWP): include user-level registers, system call arguments, resource usage information (at the OS level the kernel tracks resource use on per kernel thread basis), signal mask, it is similar to user-level library but visible to kernel, it is not needed when process is not running thus swappable

kernel-level thread: include kernel-level registers, stack pointer, scheduling information, and pointers to the associated LWP and CPU structures, it is always needed even when the thread is not live because there are OS services that need to access some of the information thus not swappable

CPU: include current thread scheduled, list of other kernel-level threads, dispatching & interrupt handling information on peripheral devices, with the knowledge of current thread we can find the information about all the other data structures that is needed to rebuild the entire process state, thus on the SPARC architecture used in the Solaris paper there is dedicated register devoted to current thread at any given point of time

2.4.10 Basic Thread Management Interactions

When the process starts, the kernel will first give a default number of kernel-level threads and the accompanying lightweight threads. If the process requests additional kernel-level threads, the kernel supports a system call named "set_concurrency". In response to this system call the kernel will create additional threads and allocate them to the process. This is useful because the user-level library does not know what is happening in the kernel, and the kernel does not know what is happening in the user-level library either, and it often occurs that the pre-assigned kernel-level threads and the associated user-level threads are blocked by queuing in certain I/O events. Allocating additional user-level threads can make progress in the process while these threads are blocked. In the same vein the performance can improve if the kernel can also notify the user-level library the availability of additional kernel-level threads in addition to those blocked. This is the reason why in the Solaris system there are many system calls and special signals that allow the kernel and the user-level library to interact and coordinate.

2.4.11 Pthread Concurrency Quiz

In the pthread library the function that sets the concurrency level is pthread_setconcurrency(). Passing integer 0 to this function instructs the implementation to manage the concurrency level as it deems appropriate. Passing an integer $n \neq 0$ sets the number of concurrent threads $= n$.

2.4.12 Thread Management Visibility and Design

kernel: sees kernel-level threads, CPUs, kernel-level scheduler

user-level library: sees user-level threads, available kernel-level threads to the process

Even it is not a one-to-one model, the user-level library can request that one of its user-level threads be bound to a kernel-level thread, similar to a multi-CPU system in which a kernel-level thread is permanently associated with a CPU.
Thread spinning occurs when the user-level library makes the user-level scheduling decision to change the ULT-KLT mapping (ULT = user-level thread, KLT = kernel-level thread) and also data structures like mutexes and wait queues, which are invisible to kernel. This problem of invisibility of state and decision between kernel and user-level library suggests that we look at one-to-one models.
Process jumps to user-level library scheduler when

- user-level threads explicitly yield
- timer set by user-level library expires
- user-level threads call library functions like lock/unlock for synchronization
- blocked threads become runnable

2.4.13 Issues on Multiple CPUs

The user-level library that is operating in the context of one thread on one CPU needs to impact what is running on another thread on another CPU. For example when the user-level thread with the highest priority (say T3) is waiting for mutex, the thread with the second highest priority (say T2) needs to preempt the thread with the lowest priority (say T1) currently running on another CPU. T2 cannot directly modify the registers of the CPU that is running T1. Instead T2 needs to send an interrupt from its context to that CPU to tell that CPU to execute the library code, because the library needs to make scheduling decision and change which thread to execute (schedule for the thread with the highest priority).

2.4.14 Synchronization-Related Issues

adaptive mutex: if critical section is short then don't block the thread requesting the mutex, just let it spin on the CPU it is currently running
if critical section is long then use the default blocking behavior—context switching to it, blocking it, queueing it up on a mutex queue
Adaptive mutex is one example of the usefulness of maintaining mutex ownership information.

thread reuse: when a thread exits, don't destroy it immediately, instead put it on a "death row", periodically a special reaper thread will perform garbage collection; if a request for a thread comes in before it is destroyed by the reaper thread, then its data structure and stack can be reused

2.4.15 Number of Threads Quiz

In the Linux kernel's code base, the minimum number of threads needed to allow a system to boot is 20. The name of the variable used to set this limit is max_threads.

2.4.16 Interrupts and Signals Intro

interrupt: events generated externally to a CPU by components other than the CPU (e.g. I/O devices, timers, or other CPUs), which interrupts can occur is determined based on the physical platform, appear asynchronously (i.e. not direct response to specific action taking place in the CPU)

signal: events triggered by the CPU and software running on it, which signals can occur is determined based on the OS, can appear asynchronously or synchronously (i.e. as a direct response to specific action taking place in the CPU)

The similarities and differences between interrupts and signals are

- both have a unique ID, depending on the hardware (interrupt) or OS (signal)
- both can be masked and disabled/suspended via corresponding mask, per CPU for interrupt mask vs. per process for signal mask
- both trigger corresponding handler if enabled, handler set for the entire system by the OS (interrupt) vs. handler set on per process basis by the process (signal)

2.4.17 Interrupt Handling

Based on the pins where the interrupt occurs or the message signal interrupter (MSI) we know which device generated the interrupt. Based on the interrupt number an interrupt handler table is referenced, which specifies the starting address of the interrupt handling routines. All of this happens in the context of the thread that was interrupted. What interrupts can occur depends on the physical platform, but how they are handled is determined by the OS.

2.4.18 Signal Handling

Once the OS generates the signal, the rest of the processing is similar to interrupts—the OS maintains a signal handler table which specifies the starting address of a handling routine. The difference is that, the OS specifies some default actions for handling signals such as terminate, ignore, terminate and core dump, stop or continue. However the process is also allowed to install its own custom handling routine, although there are certain signals which are exception to this and are referred to as signals that cannot be caught.
Examples of synchronous and asynchronous signals include

synchronous: SIGSEGV (access to protected memory, table #11 in Linux), SIGFPE (divide by zero), SIGKILL (kill, id)

asynchronous: SIGKILL (kill, self), SIGALARM (time out)

2.4.19 Why Disable Interrupts or Signals

If the handling routine needs to use mutex and the thread to be interrupted has exactly the same mutex, we then have a deadlock situation, because the interrupted thread will not release the mutex until the handling routine completes the execution on the thread's stack and returns. To prevent this there are two solutions

- keep the handler code simple i.e. prohibit the handling code to use mutexes, but this limits what handler can do thus is very restrictive
- use interrupt/signal masks, which is a sequence of bits where each bit corresponds to a specific interrupt/signal with 0 = disabled vs. 1 = enabled, if the interrupt/signal is disabled it remains standing and will be handled later when the thread resets the mask value after unlocking the mutex

Note that handling routine will typically be executed only once. So if we want to ensure that a signal handling routine is executed more than once, it is not sufficient to generate the signal more than one.

2.4.20 More on Signal Masks

interrupt mask: per CPU, if mask disables interrupt, then hardware interrupt routing mechanism will not deliver interrupt to CPU

signal mask: per execution context, if mask disables signal, then kernel will not interrupt the corresponding thread

2.4.21 Interrupts on Multicore Systems

In a multi-CPU system interrupts can be directed to any CPU that has them enabled at the delivery time. Thus in a multi-CPU system we can designated one of the cores for handling the interrupts, the only CPU that has the interrupts enabled, so as to avoid any overheads and perturbations on all the other cores.

2.4.22 Types of Signals

There are two

one-shot: n signals pending is equivalent to 1 signal pending (overriding), handler is executed only once and the handling routine must be re-enabled every single time, and any further signals will be handled by the OS default action

real-time: if n signals raised then handler is executed n times (queuing, supported by Linux)

2.4.23 Signals Quiz

SIGINT: terminal interrupt signal

SIGURG: high bandwidth data is available on a socket

SIGTTOU: background process attempting to write

SIGXFSZ: file size limit exceeded

2.4.24 Interrupts as Threads

One solution in the SunOS paper to the deadlock problem is to promote an interrupt to a full-fledged thread so that it has its own context, its own stack, and therefore can remain blocked (vs. by default the handling routine should start executing in the context of the interrupted thread). Nevertheless if the handling routine is going to perform synchronization operations, the routine will execute in the context of a separate thread.
The rule to the dynamic decision of whether executing the handling routine in the stack of the interrupted thread or turn it into a real thread is that

- if the handler doesn't include locks, then execute it in the interrupted thread
- if the handler can block, then turn it into a real thread

To eliminate the need to dynamically create threads, the kernel pre-creates and pre-initializes a number of threads for the interrupt routines it supports.

2.4.25 Interrupts: Top vs. Bottom Half

The division into top and bottom half is a common technique to allow interrupt-handling routines to have arbitrary complexity without worrying about deadlocks.

top half: perform minimum amount of processing, required to be non-blocking, fast, executes immediately when an interrupt occurs

bottom half: arbitrary complexity, can block, can be scheduled at a later time

2.4.26 Performance of Threads as Interrupts

Overall overhead of 40 SPARC instructions per interrupt, saving of 12 instructions per mutex because no need to lock the mutex and change the interrupt mask and switch it back and unlock the mutex. This observation is also one of the most important lessons in system design—optimize for the common case (the mutex lock/unlock operation in this case).

2.4.27 Threads and Signal Handling

In the Solaris implementation there is a signal mask associated with each user-level thread that is visible to the user-level library, there is also a signal mask associated with the light-weight process visible at the kernel level.

2.4.28 Threads and Signal Handling: Case 2

When the kernel-level thread has the signal enabled whereas the user-level thread currently active has the signal disabled and yet there is another user-level thread that has the signal enabled, we can have a special library routine that wraps the signal handling routine. When the signal occurs we execute the library provided handler, which can see the masks of all the user-level threads and switch to the user-level thread that has the signal enabled so that the signal can be handled.

2.4.29 Threads and Signal Handling: Case 3

If the user-level thread with the signal enabled is currently associated with another kernel-level thread with the signal enabled, the user-level library will generate a directed signal to that kernel-level thread to execute the signal.

2.4.30 Threads and Signal Handling: Case 4

If the user-level library cannot find any user-level thread that has the signal enabled, it would execute system call to request the signal mask update for the kernel-level threads one by one. When one user-level thread becomes enabled for the signal again, the user-level library will perform a system call to update the signal mask of one of the kernel-level threads to reflect the fact that the process can now handle the signal.
This interaction between the user-level library and the kernel is in the same spirit of optimizing the common case—signals occur much less frequent than signal mask updates, to make the more common signal mask updates cheap we only update the user-level threads to avoid system calls

2.4.31 Tasks in Linux

- the main abstraction that Linux uses to represent an execution context is called a **task**, a task is essentially the execution context of a kernel-level thread, a single-threaded process has one task whereas a multi-threaded process has many tasks, one per thread
- some of the key elements in a task data structure (approx. 1.7KB!) are

  pid_t pid: task id, not process id, although for single-threaded processes they are identical, for multi-threaded processes process id is the task id of the first task, this information is also stored in the task group id (tgid)

  struct list_head tasks: list of tasks

- to create a task Linux uses an operation called *clone*

  clone(function, stack_ptr, sharing_flags, args)

  where sharing_flags is a bit map that specifies which portion of the state of a task will be shared between the parent task and the child task, when all cleared it is equivalent to fork (albeit fork in Linux is internally implemented via clone)

- the current thread implementation in Linux is called native POSIX threads library (NPTL), it is one-to-one model thus the kernel sees each user-level thread, which replaces the older many-to-many model and is made possible by
  1. cheaper kernel traps (user-kernel crossings)
  2. more resources (memory, range of IDs etc.) to create as many kernels as needed

2.5 Lecture 5 Threads Performance Considerations

2.5.1 Performance Metrics

- platform efficiency is a combination of throughput and how well resources are utilized to deliver the throughput
- SLA violations stands for service level agreement violations

2.5.2 Performance Metrics Summary

Measurable quantity can be obtained from

- experiments with real software deployment, real machines, real workloads
- toy experiments representative of realistic settings
- simulation

We refer to these experimental settings as testbed, which specifies where the experiments were carried out and what were the metrics measured.

2.5.3 Multi Process Web Server

Multi process = multiple instances of the same process, each of which is single-threaded. The pro and con are

pro: simple, once the sequence of steps for one process has been correctly developed, we just spawn multiple processes

con: have to allocate memory for every process thus high memory usage, hard and costly to maintain shared state across processes (tricky port setup, system call needed)

2.5.4 Multi Threaded Web Server

Multi threaded = every single thread executes all the steps of the process, or alternatively a single boss thread performs the accepted connection operation and the worker threads perform the remaining operations. The pro and con are

pro: shared address space and shared state thus low memory requirement, context switching is cheap as system call is not needed

con: not simple implementation as synchronization is required, and OS support for threads is required

2.5.5 Event-Driven Model

It consists of a single address space, a single process, and a single thread of control, and an event dispatcher that is continuously looping to look for incoming events and invoke registered handlers upon the occurrence of the events. In the context of web server events include receipt of request, completion of send, and completion of disk read. The event dispatcher operates like a state machine, and invoking the handler is jumping to the implementation code in the process address space. The key feature of the handlers is that they run to completion—if they need to block, they will initiate blocking operation and pass control to the event dispatcher.

2.5.6 Concurrency in the Event-Driven Model

multi-process & multi-threaded model: one request per execution context (process/thread)

event-driven model: many requests interleaved in an execution context

2.5.7 Event-Driven Model: Why

On one CPU, an event-driven model processes a request until wait is necessary and then switches to another request, this realizes the benefit of threads hiding latency via context switching (ref. Lecture 2). This works for multiple CPUs as well—each CPU hosts a single event-driven process. This involves less overhead compared to multi-process and multi-threaded model, in which each CPU has to context switch among multiple processes and multiple CPUs where each of them is handling a separate request.

2.5.8 Event-Driven Model: How

An OS typically uses socket and file as the abstractions of network and disk. But fortunately they are both represented by a file descriptor. An event is an input to a file descriptor. To determine which file descriptor has input, either select() or poll() are called to scan through a range of file descriptors to return the first one that has an input regardless of whether it is a socket or a file. An alternative is to use an API called epoll() supported by Linux to alleviate the search time of select() and poll().
The benefits of event-driven model are

- single address space, single flow of control
- smaller memory requirement, no context switching thus low overhead
- no synchronization is needed thus simpler programming

Recall that calling different handlers involves jumping in the code base of the process which leads to the loss of localities. But that will still be significantly lower than a full blown context switching.

2.5.9 Helper Threads and Processes

A problem with event-driven model is that a blocking request will block the entire event-driven process. One way to circumvent this problem is to use asynchronous I/O operations. Asynchronous call have the property that

1. when an asynchronous system call is made, the kernel captures all relevant information about the caller, and where and how the data should be returned once it becomes available
2. it allows the caller to proceed executing something else, and then come back later to check if the results of the asynchronous operation are available

Asynchronous call is supported if the OS kernel is multi-threaded or the device can perform the blocking I/O operation via e.g. direct memory access (DMA). However

- asynchronous call may not be available for all types of devices. To deal with this limitation Pai proposed helpers which are designated for blocking I/O operations only, so that only helpers will block but the main event loop will not. The communication with the helper can be via socket based interface or a type of messaging interface that is available in OS called *pipes*, both of which present a file descriptor-like interface thus select(), poll(), or epoll() can be used.
- not all kernels are multi-threaded. To deal with this limitation Pai proposed making helpers processes—asymmetric multi-process event-driven model (AMPED), asymmetric in the sense that the helper process only deals with blocking I/O operations and the main process is responsible for everything else. In the same vein we can promote helpers to threads instead of processes which becomes asymmetric multi-threaded event-driven model (AMTED)

The key benefits of the asymmetric model are

1. resolves the portability limitations of basic event-driven model
2. allows the achievement of concurrency with smaller memory than multi-threaded or multi-process model, in which a worker has to perform everything and there needs to be the same number of threads or processes as the number of concurrent requests regardless of blocking or not

Nonetheless the asynchronous model has the drawback that

1. it is not generally applicable to any application
2. the routing of events in multi-CPU systems can be complex

2.5.10 Models and Memory Quiz

Of the three models event-driven model is likely to require the least amount of memory, because

event-driven model: extra memory is required only for helper threads associated with concurrent blocking I/O, not for all concurrent requests

boss-worker model: extra memory is required for all concurrent requests

pipeline model: concurrent requests will demand multiple threads to be available in a stage of the pipeline if the level of concurrency is beyond the number of pipeline stages

2.5.11 Flash Web Server

Flash is an event-driven webserver that follows the AMPED model with asymmetric helper processes to deal with blocking I/O operations. The communication from the helpers to the event dispatcher is performed via pipes. The helper reads the file in memory via the *mmap* call, and then the dispatcher checks if the pages of the file are already in memory via *mincore* to decide whether to use the handler or helper to perform I/O, in the former case reading would not result in a blocking I/O operation. This extra check results in big savings because it prevents blocking the entire process if a blocking I/O operation is necessary otherwise.
Additional optimizations Flash applies are

- application-level caching (data and computation) at multiple levels (e.g. caching file path, response header, mapped file)
- align data structures so that it is easy to perform DMA operations without copying data, use DMA operations that have scatter-gather support so that the header and file data don't have to be contiguous in memory

2.5.12 Apache Web Server

The flow of control is similar to the event-driven model in the sense that each request passes through all the modules which play the role of handlers. However Apache is a combination of a multi-process and multi-threaded model, in which a single process is internally a multi-threaded boss-worker process with a dynamic thread pool, and the number of processes can also be dynamically adjusted.

2.5.13 Experiment Methodology

what systems to compare: the following systems are compared with Flash (AMPED model)

- multi-process single-threaded configuration of Flash
- multi-threaded boss-worker model
- single-process event-driven model (SPED)
- Zeus (SPED with 2 processes)
- Apache (v1.3.1, multi-process)

All integrate some optimizations that Flash introduced except Apache.

what workloads to use: realistic (distribution of web page access over time) + controlled and reproducible (trace-based, from real web servers)

what metrics to measure: bandwidth = bytes/time, connection rate = requests/time, both were evaluated as a function of file size because with a large file size

- the connection cost can be amortized and more bytes can be pushed out leading to higher bandwidth
- more work per connection leading to lower connection rate

2.5.14 Experimental Results

bandwidth vs. file size: bandwidth increases with file size for all, SPED > Flash AMPED due to extra check for memory presence > anomaly for Zeus arisen from misalignment of some DMA operations > multi-threaded (MT)/multi-process (MP) slower because of extra synchronization and context switching > Apache due to lack of optimizations

Owlnet trace vs. CS trace: Owlnet trace is similar to the synthetic workload above since it is the smaller trace and most of it will fit in the cache, but sometimes blocking I/O is still required thus Flash > SPED
CS trace is the larger trace thus mostly requires I/O, which makes SPED second worst as it lacks asynchronous I/O, MT > MP since (a) MT has smaller memory footprint thus more memory available to cache files which in turn leads to less I/O (b) context switching between threads is cheaper than between processes, Flash is the best because of (1) smaller memory footprint (2) fewer blocking I/O (3) no synchronization needed

connection rate vs. file size: the curve for Flash shifts upward when various optimizations are added (directory lookup caching, file caching, header caching)

2.5.15 Performance Observation Quiz

At about 100MB file size Flash starts to outperform SPED because (1) the workload becomes I/O bound and (2) Flash can handle I/O operations without blocking. Note that Flash and SPED have comparable memory footprints thus the file size that can be cache is comparable. In fact helpers once created will occupy memory, thus Flash will have less memory available for file caching than SPED.

2.5.16 Advice on Designing Experiments

- rule of thumb of picking metrics: (1) choose standard metrics for broader audience (2) choose metrics that answer the “why, what, who” question
- after choosing metrics, think about the system factors that affect those metrics: (a) system resources—hardware + software (b) workload, then (1) choose a subset of configuration parameters that are the most impactful to the metrics chosen (2) pick realistic ranges for each of these variables (3) include the best/worst scenario to demonstrate certain limitation or opportunity—the only times when picking unrealistic workload makes sense (4) pick useful combinations of factors
- compare your system to the baseline model to show the proposal improves the state of the art or common practice, or compare with extreme conditions in terms of the workload or resource assignment to reflect scaling property etc.

3 MODULE 3

3.1 Lecture 1 Scheduling

3.1.1 Visual Metaphor

An OS scheduler has the following choices:

- assign tasks immediately → simple scheduling to avoid overhead
- assign simple tasks first → maximize throughput
- assign complex tasks first → maximize utilization of CPU, devices, memory etc.

3.1.2 Scheduling Overview

The CPU scheduler decides how and when processes (and their threads, processes and threads are the same in this contexts so we will use "tasks" to describe both of them) access shared CPUs. The scheduler concerns the scheduling of both user-level threads as well as kernel-level threads. The scheduler will have to look at all the tasks in the ready queue and decide which to dispatch to run on the CPU. We have to run the scheduler when

- the CPU becomes idle because the thread that is running on the CPU enters the wait state
- the time slice of the task running on the CPU expired
- a new task becomes ready because a thread exits wait state or a new task was created

The details of the scheduler implementation depends very much on the runqueue data structure that we use to implement the ready queue.

3.1.3 Run to Completion Scheduling

This type of scheduling assumes that as soon as a task is assigned to a CPU it will run until it finishes i.e. there will be no preemption. The metrics we will use to compare the various scheduling algorithms are ($\Rightarrow$ note that time to complete all tasks $\neq \sum_i$ (time to complete task i) thus throughput and average completion time are not reciprocal to each other)

$$\text{throughput} = \frac{\text{number of tasks completed}}{\text{time to complete all tasks}}$$

$$\text{average completion time} = \frac{\sum_i (\text{time to complete task } i)}{\text{number of tasks completed}}$$

$$\text{average waiting time} = \frac{\sum_i (\text{waiting time for task } i \text{ to be dispatched})}{\text{number of tasks completed}}$$

first-come-first-serve (FCFS) scheduling: we can use queue structure (FIFO)

shortest-job-first (SJF) scheduling: we can use ordered queue or tree structure instead

3.1.4 Preemptive Scheduling: SJF + Preempt

The scheduler needs to inspect the execution times of the tasks in the ready queue, and decide whether to preempt the currently running task. However in reality it is hard to know exactly the execution time of a task, thus we need to use some heuristics to estimate the execution time of a task e.g. past execution time.

3.1.5 Preemptive Scheduling: Priority

Besides execution time tasks have different priority levels. Thus the scheduler should be able to preempt tasks with lower priorities to run tasks with the highest priority. To achieve this we can have multiple runqueue structures, one for each priority level. Another option is tree structure ordered based on priority.

One danger with priority-based scheduling is **starvation**, a situation in which low priority tasks stuck in a runqueue indefinitely because there is always some higher priority task that show up in some other runqueues. One mechanism to protect it is called "priority aging"—the scheduling priority is a function of not only the actual priority but also the time the task spent in the runqueue, the longer a task spent in a runqueue the higher its priority would become.

3.1.6 Priority Inversion

Priority inversion occurs when lower priority tasks own the mutex that higher priority tasks request. A solution to this problem would be to temporarily boost the priority of the mutex owner to that of the task requesting the mutex, and then lower its priority after the release of the mutex.

3.1.7 Round Robin Scheduling

Round robin scheduling is similar to FCFS in that it also picks up the first task from the queue. The difference is that when a task needs to yield to wait on an I/O event, it will be placed at the end of the queue. Round robin scheduling can be generalized to include priorities as well. The scheduler will just go round robin between the tasks with the same priority until they complete. A further modification to round robin is interleaving/time slicing—giving each task a time slice of some time units.

3.1.8 Timesharing and Timeslices

Timeslice, also called time quantum, is the maximum amount of uninterrupted time given to a task. Task may run less than timeslice due to e.g. waiting on an I/O, or synchronization, or higher priority tasks become runnable. The use of timeslice allows task interleaving. Note that FCFS is equivalent to round robin without timeslices. The benefits of this timeslice-based method are that

- short tasks are able to finish sooner
- it becomes more responsive and allows lengthy I/O operations to be initiated sooner

The downside is that there is some overhead as we have to (1) interrupt a running task (2) run the scheduler to pick again which task to run next (3) perform context switching, the latter two being the dominant part of the overhead. But as long as the timeslice value is significantly larger than the context switching time, we should be able to minimize these overheads.

3.1.9 How Long Should a Timeslice Be

The balance between the benefits and overheads of using timeslices differs if we consider I/O-bound tasks and CPU-bound tasks.

3.1.10 CPU Bound Timeslice Length

CPU bound tasks are tasks that are mostly just running on the CPU and don't perform any I/O. For CPU bound tasks we care more about average completion time than average waiting time, and we will be better off with choosing a larger timeslice.

3.1.11 I/O Bound Timeslice Length

For I/O bound tasks the value of the time slice is not really relevant because the I/O operations would interrupt anyway. For I/O bound tasks using a smaller timeslice is better, because it allows the task to issue an I/O request sooner keeping both the CPU and the I/O device busy.

3.1.12 Summarizing Timeslice Length

CPU bound tasks prefer longer timeslices to

- limit context switching overheads
- keep CPU utilization and throughput high

I/O bound tasks prefer shorter timeslices to

- issue I/O operations earlier
- keep CPU and device utilization high
- achieve better user-perceived performance

3.1.13 Timeslice Quiz

CPU utilization is defined as

$$\text{CPU utilization} = \frac{\text{CPU running time}}{\text{CPU running time} + \text{context switching overhead}}$$

Both CPU running time and context switching overhead should be calculated over a consistent recurring interval.
For 10 I/O bound tasks each issuing an I/O operation every 1ms and 1 CPU bound task, the CPU running time for a time interval of 20ms is $10 \times 1 + 1 \times 10 = 20$ms, while the total context switching overhead is $11 \times 0.1 = 1.1$ms if the overhead is 0.1ms per switch.

3.1.14 Runqueue Data Structure

If we want I/O bound and CPU bound tasks to have different timeslice values, we can

- maintain a single runqueue but make it easy for the scheduler to check out the task type
- use two different runqueues, one for I/O bound and one for CPU bound

More generally we can have multi-queue structure, assigning a timeslice of say 8ms for the most I/O intensive tasks (given highest priority), 16ms for tasks with mixed I/O and CPU processing, and ∞ms (i.e. FCFS) for CPU intensive tasks (given lowest priority). When a new task arrives, it enters the I/O intensive queue first. If the task yields voluntarily before 8ms, then keep it in this queue after the I/O operation completes; if the task uses up the entire timeslice, then push it down to the next queue; if the task ends up getting preempted in the next queue, then push it down further. If a task in the bottom queue repeatedly releases the CPU earlier due to I/O waits, then push it up to the queue one level above. This data structure is called the **multilevel feedback queue** (MLFQ). Note that MLFQ is different from priority queue because of

- the different policies at different levels of the queue
- the feedback mechanism

The Linux O(1) scheduler uses some of the mechanisms of MLFQ and the Solaris scheduler is essentially a 60-level MLFQ.

3.1.15 Linux O(1) Scheduler

The O(1) scheduler receives its name because it is able to add/select task in constant time regardless of the number of active tasks in the system. It is a preemptive and priority-based scheduler with a total of 140 priority levels (0 being the highest). These priority levels are organized into two different classes

real-time tasks: level 0 to 99

time-sharing tasks: level 100 to 139

All user processes have one of the time-sharing priority levels with a default level of 120 and can be niced by -20 to 19.
The O(1) scheduler borrows from MLFQ in that it

- associates different timeslice values with different priority levels
- uses feedback from how the tasks behaved in the past to determine how to adjust their priority levels in the future

However it differs in how to assign the timeslice values to priorities and how it uses the feedback

timeslice: assigns the smallest timeslice value to the lowest priority i.e. CPU intensive, opposite for more I/O intensive or more interactive tasks

feedback: the feedback it uses for time-sharing tasks is based on sleep time—longer sleep time implies more interactive operation thus warrants higher priority (niced by -5), opposite for shorter sleep time

The runqueue of the O(1) scheduler is organized as two arrays of task queues, each array element pointing to the first runnable task at the corresponding priority level

active array: featuring

- used to pick next task to run
- takes constant time to add/select, because it relies on certain instructions that return the position of the first set bit in a sequence of bits, if the sequence of bits corresponds to the priority level and a bit value of one indicates that there are tasks at that priority level, then it will take a constant amount of time to detect what is the first priority level that has tasks on it, and it also takes constant time to index into this array and select the first task from the runqueue associated with that level
- even if yielding the CPU to wait for an event or preempted, the task remains in active array until the timeslice expires, and once expired it will be placed on the appropriate queue in the expired array

expired array: contains inactive tasks, inactive in the sense that the scheduler will not select as long as there are still tasks on any queue in the active array, when there are no more tasks in the active array, the pointers of these two arrays will swap so that active becomes expired and expired becomes active, this serves as an aging mechanism to allow lower priority tasks the chance to run

This design justifies why the O(1) scheduler gives longer timeslices to higher priority tasks—as long as the high priority tasks have any time left in their timeslices, they will remain in the active array and keep getting scheduled. With shorter time slices lower priority tasks will get a chance to run. However they won't disrupt the higher priority tasks and they won't delay them either.

As the workloads become more time sensitive in the Linux environment, the jitter that was introduced by the O(1) scheduler becomes unacceptable. It was thus replaced with the completely fair scheduler (CFS). Nonetheless both the O(1) and the CFS scheduler are part of the standard Linux distribution, with the CFS being the default scheduler.

3.1.16 Linux CFS Scheduler

The problem with the O(1) scheduler is that

- once tasks are placed on the expired list, they wouldn't be scheduled until all remaining tasks from the active list have a chance to execute, as a result the performance of interactive tasks is affected (jitter)
- the scheduler in general doesn't make any fairness guarantees, where *fairness* can be intuitively defined as that in a given time interval all tasks should be able to run for an amount of time that is proportional to their priority

The completely fair scheduler (CFS) is the default scheduler for all non-real time tasks. The real time tasks are scheduled by a real time scheduler. CFS uses a single red-black tree as a runqueue structure which will self-balance itself as nodes are added or removed, so that all the paths from the root to the leaves of the tree are approximately of the same size. Tasks are ordered in the tree based on the amount of time that they spent running on the CPU called *virtual runtime*. CFS tracks this virtual runtime in a nanosecond granularity. Each of the internal nodes in the tree corresponds to a task and the nodes to the left/right of the task correspond to those tasks which had less/more virtual runtime on the CPU. The leaves in the tree don't play any role in the scheduler.
To summarize CFS

- always picks leftmost node i.e. schedules the tasks with the least virtual runtime
- periodically increments the virtual runtime of the task that's currently executing on the CPU, and compares the virtual runtime with that of the leftmost task in the tree, if smaller the task will continue executing; if larger the task will be preempted and placed in the appropriate location in the tree, and the task corresponding to the leftmost node will be the one to run next
- the virtual runtime rate of increment depends on priority, with slower rate for higher priority tasks which as a result will be executed on the CPU longer

With the red-black tree structure, selecting a task takes O(1) time as it typically is just a matter of selecting the leftmost node, whereas adding a task takes $O(\log N)$ time

3.1.17 Linux Scheduler Quiz

The main reason why the Linux O(1) scheduler was replaced by the CFS is that, the O(1) scheduler interactive tasks could wait unpredictable amount of time to be scheduled, since once the task was moved to the expired list, it has to wait there until all the low priority tasks consumed their entire time quantum.

3.1.18 Scheduling on Multiprocessors

In a shared memory multiprocessors (SMPs) there are multiple CPUs each of which have their own private caches like L1 and L2, but their last level cache (LLC) may or may not be shared among the CPUs, and there is a system memory DRAM that is shared across all the CPUs. The OS sees all these CPUs as well as the cores in these CPUs as entities onto which it can schedule all execution context i.e. the scheduler sees the cores as CPUs. What we want to achieve with a scheduling on multi-CPU systems is to try to schedule the thread back on the same CPU where it executed before, because it is more likely that its cache will be hot. We call this **cache affinity**. To achieve this we want the scheduler to keep a task on the same CPU as much as possible. To achieve this we can maintain a hierarchical scheduler architecture, where

- a load balancing component at the top level divides the tasks among CPUs, to achieve load balancing it can look at information such as the length of each of the runqueues, or potentially when a CPU is idle it can start looking at the other CPUs and try to get some work from them

- a per-CPU scheduler with a per CPU runqueue repeatedly schedules those tasks on a given CPU as much as possible

In addition to having multiple processors it is possible to also have multiple memory nodes. A memory node can be connected to some subset of the CPUs via e.g. a socket that has multiple processors. In this case the access from that subset of the CPUs to the memory node will be faster than from the other CPUs (both types of accesses are possible because of the interconnections among all these components e.g. Intel's QuickPath Interconnect or QPI). We call this type of platforms non-uniform memory access (NUMA) platform. Hence it makes sense for the scheduler to divide tasks in such a way that tasks are bound to those CPUs that are closer to the memory node where the state of those tasks is. We call this type of scheduling **NUMA scheduling**.

3.1.19 Hyperthreading

The reason why we have to context switch among threads is because the CPU has one set of registers to describe the active execution context, which include stack pointer and program counter in particular. One way to reduce this overhead is to use CPUs with multiple registers each of which describe the context of a separate thread. This multiple hardware-supported execution contexts are referred to as **hyperthreads**. However note that there is only one CPU thus only one of these hyperthreads can execute at a particular time. Nonetheless the context switch between these hyperthreads is very fast. In addition to hyperthreading this mechanism also has other names such as hardware multithreading, chip multithreading, or simultaneous multithreading (SMT). One of the features of today's hardware is that you can enable or disable this hardware multithreading at boot time. If it is enabled each of these contexts appears to the OS as a virtual CPU onto which a thread can be scheduled.

Recall that it is worth context switching to hide latency if the idle time is more than twice of the context switching time. Because in SMT system the context switching time is in the order of several cycles whereas the time to perform a memory access remains in the order of 100 cycles, hyperthreading can hide memory access latency.

Hyperthreading does raise the issue of what threads should be co-scheduled on the hardware threads in the CPU, we will discuss this question in the context of the paper *Chip Multithreaded Processors Need a New OS Scheduler* by Fedorova et. al.

3.1.20 Scheduling for Hyperthreading Platforms

We will make the same assumptions as in Fedorova's paper

- a thread can issue an instruction on every single cycle, thus a CPU bound thread will be able to achieve a maximum metric in terms of instructions per cycle (IPC) = 1 (given that we have only one CPU we cannot have an IPC > 1)

- a memory access takes 4 cycles, thus a memory bound thread will experience idle cycles while it is waiting for the memory access to return

- SMT with two hardware threads

- context switching among different hardware threads is instantaneous i.e. no overhead

Given only one CPU, we have the following options

1. Two CPU bound threads if co-scheduled will interfere i.e. contend for the CPU pipeline resource. The best case is one remains idle while the other issues instructions. As a result for each of the threads the performance degrades by a factor of 2. Furthermore the memory component is idle as there is nothing scheduled to perform any memory access.

2. Alternatively if two memory bound threads are co-scheduled, we end up again with some idle cycles. As we need to wait for 4 cycles until the memory access returns 2 of the 4 cycles are unused. So the strategy to co-schedule memory bound threads leads to waste of CPU cycles.

3. With a mix of one CPU bound thread and one memory bound thread, we can fully utilize each processor cycle—whenever the memory bound thread needs to perform a memory reference we context switch to the CPU bound thread until the memory reference completes.

Therefore a mix of CPU and memory intensive threads allows us to avoid or at least limit the contention on processor pipeline. In addition both the CPU and memory component will be well utilized. Nevertheless there is still some degradation due to the interference between the two threads (the CPU bound thread can issue instruction in only 3 out of the 4 cycles).

3.1.21 CPU Bound or Memory Bound

We will use historic information to determine whether a thread is CPU bound or memory bound. However the sleep time used in the O(1) scheduler won't work here, because

- the thread is not sleeping when waiting on a memory reference, it is waiting in some stage of the processor pipeline rather than some type of software queue

- to keep track of sleep time we need to use some software method which however takes too much time to compute, in general we cannot execute in software the computation to decide whether a thread is CPU bound or memory bound because the context switch takes order of cycles

Therefore we need information from the hardware to determine whether a thread is CPU bound or memory bound. Fortunately modern hardware has lots of hardware counters that get updated as the processor is executing and keeps information about various aspects of the execution such as (1) L1, L2, or LLC misses (2) IPC (3) power and energy usage. There are also many software tools that can be used to access these hardware counters e.g. oprofile or the Linux perf tool in Linux (note that hardware counters are not uniform on every single platform). From hardware counters the scheduler can estimate the kind of resources a thread needs and pick a good mix of the threads, so that all the components of the system are well utilized and the threads don't interfere with one another etc. However there is not a unique way to interpret the information provided by hardware counters (cache miss may be due to large memory footprint or entering a new stage of execution) so we really need to guess what hardware counters are telling about the thread's resource use. Nevertheless the scheduler can still make informed decisions based on the information from hardware counters. Typically it

- uses multiple hardware counters to build a more accurate picture of the threads' resource use

- relies on some models that have been built for a specific hardware platform and have been trained using some well understood workloads

3.1.22 Scheduling with Hardware Counters

Memory bound threads have high cycles per instruction (CPI) whereas CPU bound threads have 1 or low CPI. Given that there is not a CPI counter on the processors in Fedorova's work, and computing 1/IPC would require software computations thus is unacceptable, Fedorova uses simulation based evaluation to determine whether CPI is a good metric.

3.1.23 CPI Experiment Quiz

- with mixed CPIs the processor pipeline is well utilized resulting in high IPC
- with the same CPI there is contention on some cores (low CPI) and wasted cycles on other cores (high CPI) resulting in much lower IPC

Based on this comparison Fedorova concludes that CPI is a good metric. However realistic workloads don't have such distinct and widely spread CPIs as in the synthetic workloads used for the simulation. Instead the CPIs often clutter around some values. Therefore CPI won't be a useful metric. Nonetheless there are still some useful takeaways from the paper

- there is resource contention in SMTs for processor pipeline
- hardware counters can be used to characterize workloads
- schedulers should be aware of resource contention not just load balancing, this principle generalizes to other types of resources not just the processor pipeline in SMTs

In Fedorova and others' follow-up work it is contended that LLC usage would be a better choice.

3.2 Lecture 2 Memory Management

3.2.1 Visual Metaphor

In an OS

- memories typically manage the granularity of pages or segments
- tasks operate on only a subset of memory
- how the state required for the tasks is brought in and out of memory and into memory pages or segments is optimized so as to reduce the time that is required to access that state

3.2.2 Memory Management: Goals

allocation: The amount of virtual memory can be much larger than the amount of physical memory. Thus the OS must be able to allocate physical memory and arbitrate how it is being accessed. In addition because physical memory is smaller than virtual memory, it is likely that some of the contents needed in virtual address space are not present in physical memory, which may be stored on secondary storage like disk. Thus the OS must have mechanisms to decide how to replace the contents that are currently in physical memory with needed contents that are on temporary storage.

arbitration: the OS should be able to quickly translate virtual addresses into physical addresses and verify that it is a legal access

page-based memory management: virtual address space is subdivided into fixed sized segments called **pages**, physical memory is divided into page frames of the same size

allocation: the role of the OS is to map pages from virtual memory into page frames of physical memory

arbitration: the arbitration of the access is done via page tables

segment-based memory management: paging is not the only way to decouple virtual memory and physical memory

allocation: virtual address space is subdivided into flexibly sized segments that can be mapped to physical memory as well as swapped in and out of physical memory

arbitration: segment registers are used to either translate or validate access

Paging is the dominant method used in current OS. We will focus our discussion on page-based memory management.

3.2.3 Memory Management: Hardware Support

Memory management is not purely done by the OS alone. Hardware also support a number of mechanisms to make it easier, faster, or more reliable to perform allocation and arbitration, which include

- every CPU package is equipped with a memory management unit (MMU), the CPU issues virtual address to the MMU and is responsible for translating them into the appropriate physical address, or the MMU can generate a fault which indicates illegal access, permission denial, not present in memory
- use designated registers during address translation, e.g. in a page-based system there are registers pointing to the currently active page table and in a segment-based system there are registers that indicate the number of segments and the base address and size limit of the segments
- a small cache of valid virtual to physical address translation called the translation lookaside buffer (TLB) to make the translation process much faster
- the actual physical address generation is done in hardware, note that although the OS maintains certain information necessary for the translation, it is the hardware that performs the actual translation, this implies that the hardware will dictate what type of memory management modes are supported

We will focus on the software aspects of memory management which are more flexible in terms of their design.

3.2.4 Page Tables

Page tables are used to translate virtual memory addresses into physical memory addresses. By keeping the size of the pages and page frames the same, we don't have to keep track of the translation of every single individual virtual address. Instead we only need to translate the first virtual address in a page to the first address of the corresponding page frame in physical memory. It means that only the first portion of the virtual address is used to index into the page table. We call this part of the virtual address the *virtual page number*, and the rest of the virtual address the actual *offset*. The virtual page number is used as an offset into the page table which would produce the *physical frame number* (PFN)—the physical address of the physical frame in DRAM. To complete the full translation the PFN

needs to be sent with the offset that is specified in the latter part of the virtual address to produce the actual physical address.
The first time the OS accesses an array initialized in a virtual address, there is no physical address that corresponds to this range of virtual addresses. It will then take a page of physical memory and establishes a mapping between this virtual address and the physical address of the allocated page frame in physical memory. We refer to this as allocation on first touch. This is to make sure that physical memory is allocated only when it is needed because sometimes programmers may create data structures that they don't really use. If a process hasn't used some of its memory pages for a long time, it is likely that those pages will be reclaimed and the content will no longer be in physical memory. In order to detect this, page table entries do not just consist of the physical frame number, instead they also have a number of bits that tell the validity of the access e.g. 1 representing the presence in physical memory and 0 otherwise. If the MMU sees 0 in this bit, it will raise a fault and will trap the OS. If the hardware determines that the mapping is invalid and false, then control is passed to the OS to decide should the access be permitted, where is the page located, where should it be brought to DRAM etc. As long as a valid address is being accessed, ultimately there will be a mapping re-established between a valid virtual address and the valid location in physical memory, albeit the location in physical memory may be completely different from previous access. In summary the OS will maintain a page table for every process that exists, and upon context switch the OS needs to make sure that it switches to the page table of the new process (the supporting hardware register will point to the new table as well).

3.2.5 Page Table Entry

Every page table entry will have the physical page frame number (PFN) at the beginning, and at least a valid bit called present bit since it indicates whether the contents of the virtual memory are present in physical memory or not. There are some other flags used by the OS and supported by the hardware such as

- a dirty bit which is set whenever a page is written to, used to indicate whether the file on disk needs to be updated or not
- a accessed bit, used to indicate whether the page has been accessed
- protection bits, used to indicate whether a page is read-only or write/execution permissible

The MMU relies on these bits to establish the validity of the access. If the hardware determines that a physical memory access cannot be performed, it causes a page fault, and then the CPU will place an error code on the stack of the kernel and generate a trap into the kernel, which in turn will generate a page fault handler. Based on the error code and the faulting address the handler determines an action.

3.2.6 Page Table Size

A page table has the number of entries that is equal to the number of virtual page numbers (VPN) that exist in a virtual address space. In a 32-bit architecture each of the page table entries is 4 bytes including PFN and flags. Thus the number of VPNs is $2^{32} \div$ page size. Different hardware platforms support different page sizes. If we pick a page size of 4KB, then the page table size for every single process is ($\Rightarrow$ page size $\neq$ page table size)

$$\text{page table size} = \frac{2^{32}}{2^{12}} \times 4\text{B} = 4\text{MB}$$

With many active processes in an OS today this can get quite large. The problem is that page table assumes that there is an entry for every single VPN regardless of whether the corresponding virtual memory region is needed by the process or not.

3.2.7 Multi Level Page Tables

Page tables have evolved from a flat page map to a more hierarchical multilevel structure. For example a two-level page table has

- the outer level referred to as page table directory, the elements of which are pointers not to actual pages but rather to page tables
- the inner level has proper page tables as its components that point to actual pages, these inner page tables exist only for those virtual memory regions that are valid

If necessary a new inner page table may be allocated via malloc, and the appropriate page table directory will be set to correspond to that entry.
To find the right element in this page table structure, the virtual address is split into yet another component—$p_1 + p_2 + d$, where p_1 and p_2 are the indices of the outer page table directory and the inner page table respectively and d is still the offset needed to compute the offset within the actual page. If 10 bits are used for p_2 then 2^{10} pages can be addressed in the inner page table. If further 10 bits are used for d then the page size is 2^{10}. Combined the inner page table can address 2^{20} or 1MB memory. It means that whenever there is a gap in the virtual memory that is 1MB in size, we don't need to allocate that inner page table. As such the overall size of the page table required for a process is reduced. This is in contrast with the single level page table design, where the page table has to be able to translate every single virtual address and have entries for every single virtual page number. This reduction scheme can be extended to have additional layers, and is particularly important for 64-bit architectures, where not only the required page table size is larger but also the virtual address spaces of processes on these 64-bit architectures tend to be more sparse.
There is a trade-off in supporting multiple levels in the page table hierarchy. As we add multiple levels

pro: the page table directories and inner page tables end up covering smaller regions of the virtual address space, as a result it is more likely that the virtual address space will have gaps that match the granularity thus more room to reduce page table size

con: there will be more memory accesses required for translation leading to an increase in translation latency

3.2.8 Memory Management: Multi Level Page Table Quiz

The number of virtual pages addressable is determined by the number of bits allocated to the various levels of the page table e.g. $2^6 = 64$ pages for a 2-bit outer page directory p_1 + 4-bit inner page table p_2, which is the same as a 6-bit single-level page table.
Given 4-bit p_2 and 6-bit d each entry in the outer page directory corresponds to 2^{10} = 1KB virtual addresses. Thus if there is a gap of 1KB or more, not all entries in the outer page directory needs to be populated. For the quiz only the first two (the first 2KB virtual addresses) and the last one (the last 1KB virtual addresses) are populated with inner page tables, resulting in $3 \times 2^4 = 48$ pages instead of $4 \times 2^4 = 64$ pages.

3.2.9 Speeding Up Translation TLB

For each memory reference, a single-level page table requires two accesses, one to page table entry and the other to physical memory, whereas a four-level page table requires five accesses, four to each hierarchical level of the page table entries and one to physical memory.
The standard technique to avoid repeated accesses to memory is to use a page table cache. On most architectures the MMU integrates a hardware cache that is dedicated for caching address translations,

which is called the translation lookaside buffer (TLB). In addition to address translation the TLB contains all the necessary protection and validity bits to verify the access. It turns out that even a small number of entries in the TLB can result in high TLB hit rate, because we typically have a high temporal and spatial locality in memory references.

3.2.10 Inverted Page Tables

Today on the most high-end platforms physical memory is on the order of 10TB whereas virtual memory can reach PB and beyond. Clearly it would be much more efficient to have a page table structure that is on the order of physical memory instead of virtual memory. To achieve this the page table is searched based on the process ID and the first part of the virtual address i.e. pid + p + d. The problem with this inverted page table design is that we have to perform a linear search for the matched pid and pids are not ordered. To address this issue inverted page tables are supplemented with hashing page tables. A hash is computed on a part of the address, which is an entry into the hash table that points to a linked list of possible matches for this part of the address. This allows us to speed up the linear search by narrowing down the search to only a few possible entries.

3.2.11 Segmentation

In addition to paging virtual to physical memory mapping can be performed using segments. With segments the address space is divided into components of arbitrary size, and typically different segments will correspond to some logically meaningful components of the address space like the code, the heap data etc. A virtual address in the segmented memory mode includes a segment descriptor and an actual offset. The segment descriptor is used in combination with a descriptor table to produce information regarding the physical address of the segment. This segment selector, when combined with the offset, produces the actual address of the memory reference (called linear address). In its pure form a segment could be represented with a contiguous portion of physical memory. In that case the segment would be defined by its base address and its limit registers which together determine the segment size.
In practice segmentation and paging are used together through the following flowchart

$$\text{CPU} \xrightarrow{\text{logical address}} \text{segmentation unit} \xrightarrow{\text{linear address}} \text{paging unit} \xrightarrow{\text{physical address}} \text{physical memory}$$

e.g. the Intel x86_32 platform supports both segmentation and paging, while Linux allows up to 8K segments per process and another 8K global segments. However the default mode of the Intel x86_64 platform is paging, with segmentation supported only for backward compatibility.

3.2.12 Page Size

The number of bits allocated to the offset determines the page size e.g. a 10-bit offset corresponds to a 1KB page size. In practice system supports different page sizes. For Linux x86 platforms the common page sizes are 4KB (default), 2MB, and 1GB. For the large 2MB and huge 1GB pages as more bits are used for offset, fewer bits are used to represent the virtual page number and thus fewer entries in the page table. As a result there is a factor of 512 and 1024 reduction in the page table size (fewer number more than offsets larger size). Hence the benefit of using larger page sizes is that they require smaller page tables. Another benefit is that fewer tables implies more TLB hits. The downside of larger pages is that if this large virtual memory page is not densely populated, there will be a larger unused gap within the page itself leading to internal fragmentation and waste of memory. Because of this issue smaller pages of 4KB are more commonly used. But there are some settings such as database where these large and huge page sizes are necessary.

3.2.13 Memory Allocation

Memory allocator decides the physical pages that will be allocated to particular virtual memory regions. Memory allocators can exist at the kernel level as well as the user level

kernel-level allocator: responsible for allocating memory regions for kernel states and static process states (code, stack, initialized data), and keeping track of the free memory available in the system

user-level allocator: used for dynamic process states such as heap data, once the kernel allocates some memory to a malloc call, the kernel is no longer involved in the management of that space, which will then be managed by a user-level allocator

We will focus on kernel-level allocators in this course, but the same design principle applies to user-level allocators.

3.2.14 Memory Allocation Challenges

External fragmentation occurs where we have multiple interleaved allocate and free operations which results in free memory that is not contiguous. To avoid or limit the extent of external fragmentation the allocator needs to know the coming memory requests to permit memory coalescing.

3.2.15 Linux Kernel Allocators

To address free space fragmentation the Linux kernel relies on two basic allocation mechanisms

buddy allocator: starts with an area of size equal to a power of 2, whenever a request comes in the allocator will subdivide this large area into smaller chunks ("buddies") such that every one of them is also a power of 2, it will continue subdividing until it finds the smallest chunk that can satisfy the request, fragmentation is still there, but the allocator has a fast way to check out whether free space aggregation is possible because

- it is very easy to figure out the start of the adjacent allocation, as the addresses of adjacent buddies differ only by 1 bit
- checking what are free areas can be propagated up the tree of 2^x buddies

slab allocator: the power of 2 granularity requirement of the buddy allocator will result in internal fragmentation e.g. task struct has a size of 1.7KB, to fix this issue the slab allocator builds custom object caches on top of slabs which represent contiguous physical memory, when the kernel starts it will pre-create caches for the different object types e.g. task struct etc. when an allocation comes from a particular object type it will go straight to the cache, if none is available in the cache then the kernel will create another slab and allocate a portion of contiguous physical memory to it, by doing so

- internal fragmentation can be avoided because the slabs are of exactly the same size as the common kernel objects
- external fragmentation can be avoided because future requests will match the size of the freed slabs

3.2.16 Demand Paging

Because virtual memory is much larger than physical memory, virtual memory pages are not always present in physical memory. Instead the backing physical page frame can be repeatedly saved and restored to and from some secondary storage like disks. This process is referred to as **demand paging**. With demand paging pages are moved between main memory and a swap partition in a storage device. The steps of demand paging is depicted in Figure 2 below.

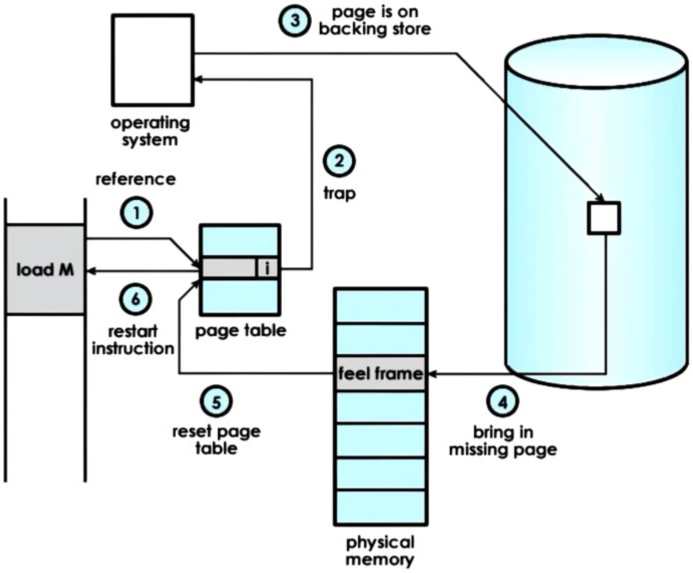

Figure 2: steps of demand paging

If a page is required to be constantly present in memory or maintain the same physical address during its lifetime, then we will have to “pin” the page and disable the swapping, because the physical address after swap is probably different from the original physical address. This is useful when the CPU is interacting with devices that support direct memory access (DMA).

3.2.17 Page Replacement

when to swapped out: when the amount of occupied memory raises above a threshold or when the CPU usage drops below a threshold, the OS will run some page out daemon that will look for pages that can be freed

which to be swapped out: pages that probably won't be used in the future predicted based on history e.g. least recently used or LRU (tracked by access bit), or that don't need to be written out to secondary storage (tracked by dirty bit), but avoid non-swappable pages such as those containing important kernel states or used for I/O operations

In Linux and most OS a number of parameters are available to allow the configuration of the swapping nature of the system, which include (1) parameters such as the thresholds mentioned above (2) page types which help narrow down the decision process such as non-swappable. Note that the default replacement algorithm in Linux is a variation of LRU, but it gives a second chance—it performs two scans before determining which page to swap out.

3.2.18 Least Recently Used (LRU) Quiz

The first page is the least recently used one.

3.2.19 Copy on Write

In addition to performing translation, tracking access, and enforcing protection, MMU can be used to build other services such as copy-on-write (COW), which is motivated by the following observation. On process creation we usually need to copy the entire parent address space for the new process. However many parent pages are static. To avoid unnecessary copying a portion of the virtual address space of the new process will be mapped to the original content of the parent process. Thus the same physical address may be referred to by two completely different virtual addresses, and we also have to make sure to write protect that physical address so as to track concurrent access to it. However if a write request is issued for this physical address via either one of the two virtual addresses, then the MMU will generate a page fault and the OS will create an actual copy. Hence pages will only be copied when they need to be updated—copy on write. Whether the write protection will be removed or not once this copy is performed will depend on who else this page is shared with.

3.2.20 Failure Management Checkpointing

Another OS service that can benefit from MMU is checkpointing, which is part of the failure and recovery management. The idea is to periodically save process states so that upon failure the process can restart from the nearest checkpoint rather than the beginning so the recovery will be much faster. A simple approach to checkpointing would be to pause the execution and copy its entire state. A better approach will be to use MMU to optimize the disruption the checkpointing will cause on the execution through

- write protect the entire address space and copy everything at once
- track the dirty pages and copy only the differences for incremental checkpoints, which however makes the recovery process more complex since we will have to rebuild the process from multiple differences

The basic mechanism used in checkpointing can also be used in other services, e.g.

debugging: rewind-replay which gradually goes back to older checkpoints until error is found

migration: checkpoints the process to another machine (repeated in a fast loop until pause-and-copy becomes acceptable or enough dirty pages have accumulated) and then restarts it on that machine, used for disaster recovery and consolidation (load as few machines as possible)

3.2.21 Checkpointing Quiz

The more frequently you checkpoint

- the more states you will checkpoint
- the higher the overhead of the checkpointing process
- the faster you will be able to recover from a fault

3.3 Lecture 3 Inter-Process Communication

3.3.1 Inter -Process Communication

Inter-process communication are OS-supported mechanisms for interaction among processes. IPC mechanisms are broadly categorized as

message based: sockets, pipes, message queues

memory based: shared memory (pages of physical memory or memory mapped files)

higher-level semantics: files (multiple processes read and write), remote procedure call (RPC), higher-level because it goes beyond supporting communication channel to prescribe communication protocol, data format etc.

synchronization primitives: since process synchronization involves communicating something about the point in their executions

This lecture will focus on the first two.

3.3.2 Message Based IPC

The OS creates and maintains a channel (such as buffer, FIFO queue etc.) for message passing and an interface (such as a port). The OS kernel is required to both establish the channel and perform every single IPC operation. It means that both the send and receive operation require a system call and a data copy. As a result a simple request-response interaction among two processes will require four user-kernel crossings and four data copies. The associated overheads are the drawback of this approach. The advantage of this approach is its simplicity—the OS kernel takes care of all the operations.

3.3.3 Forms of Message Passing

pipe: the simplest form of message passing IPC, part of the POSIX standard, only two processes can communicate, one popular use of pipes is to connect the output from one process to the input of another

message queue: a sending process must submit a properly formatted message, the OS management includes understanding priorities and scheduling message delivery, the use of message queues is supported through different APIs, in Unix-based systems these include the POSIX API and the SysV API

The most familiar message passing API is the socket API. The notion of ports for message passing IPC is the socket abstraction supported by the OS. The socket call itself creates a kernel-level socket buffer. It also associates necessary kernel-level processing needed along with the message movement e.g. TCP/IP protocol stack. If the communication is between different machines, then the channel is essentially between a process and a network device; if the communication takes place in the same machine, then a lot of the full protocol stack is bypassed.

3.3.4 Shared Memory IPC

In shared memory IPC processes read and write into a shared memory region. The OS is involved in mapping certain physical pages of memory into the virtual address spaces of both processes, note that

- although mapped to the same physical addresses, the virtual addresses of the two process don't need to be the same
- the physical addresses backing the shared memory buffer do not have to be contiguous

The pro and con of this approach are

pro: once the physical memory is mapped, the OS is out of the way, i.e. system calls are only used for the setup, data copies are potentially reduced but not eliminated, because for the data to be visible to both processes, they must be explicitly allocated to the virtual addresses corresponding to the shared memory region

con: since the shared memory can be concurrently accessed by both processes, the processes must explicitly synchronize their shared memory operation, in addition it is the developer's responsibility to determine any communication protocol related issues such as format and delimiter

Unix-based systems including Linux support two popular shared memory APIs—SysV API and POSIX API. In addition shared memory communication can be established using a file based interface i.e. the memory mapped files in both address spaces, which is analogous to the POSIX shared memory API. By the way the Android system uses a form of shared memory IPC called Ashmem.

3.3.5 IPC Comparison Quiz

message passing: must perform multiple copies

shared memory: must establish mappings between processes' virtual address spaces and shared memory page

So there are drawbacks on both sides, and it depends on the specific task which one is better.

3.3.6 Copy vs. Map

copy/message passing: require CPU cycles to copy data to and from port

map/shared memory: require CPU cycles to both map memory into address spaces and copy data to channel, but there are no user-kernel transitions required, the setup of mapping is expensive but can be amortized, thus for large data the time to copy is much greater than the time to map

To leverage this difference windows system exercises an internal tradeoff—if the data that needs to transfer is smaller than a certain threshold then the data is copied in and out of port, otherwise the data is copied once to the shared memory which is then mapped into the address spaces of the processes. This tradeoff mechanism that the windows kernel supports is called local procedure calls (LPC).

3.3.7 SysV Shared Memory

The OS supports segments of shared memory that don't have to correspond to contiguous physical pages. In addition the OS treats shared memory as a system-wide resource using system-wide policies. This means that there is a limit on the total number and total size of segments of shared memory.

create: when a process requests that a shared memory segment is created, the OS assigns to it a unique key, any other process can refer to this shared memory segment using this key, if the creating process wants to communicate with other processes, it needs to pass this key through some other forms of IPC or as a file or a command line argument

attach: using this key the shared memory segment can be attached by a process by the OS establishing a valid mapping between the virtual addresses of the process and the physical addresses that backs this segment, multiple processes can be attached to the same shared memory segment

detach: detaching a shared memory segment means invalidating the address mappings for the virtual address region within the process that corresponds to the segment

destroy: a shared memory segment isn't destroyed once it is detached, once a shared memory segment is created it is persistent until there is an explicit request to destroy it or until the system reboots, this persistent property makes it very different from regular non-shared memory that is malloc and destroyed as soon as the process exits

3.3.8 SysV Shared Memory API

create: shmget(shmid, size, flag) is used to create or open a shared memory segment, the unique key schmid is explicitly passed to the OS by the application, this unique identifier is generated by ftok(pathname, project_id), the same argument yields the same key

attach: shmat(shmid, addr, flag), addr = NULL allows the OS to choose and return arbitrary virtual addresses available in the process address space, it is the programmer's responsibility to cast the returned virtual address to the appropriate type

detach: shmdt(schmid) invalidates the virtual to physical memory mapping

destroy: shmctl(shmid, cmd, buffer) passes the control and command to the OS, which can destroy the segment with the IPC_RMID command

3.3.9 POSIX Shared Memory API

Although it is supposed to be the standard, the POSIX API is not as widely supported as say the SysV API.

create: shm_open() returns a file descriptor in "tmpfs", the most notable difference is that the POSIX shared memory standard doesn't use segments, instead it uses files, which are not the real files that exist in some file system, but only exist in the so called tmpfs file system and are in essence just a bunch of states present in physical memory

attach & detach: mmap() and unmmap(), the POSIX uses the same representation and data structure as those used for representing a file to represent shared memory pages, for this reason there is no need to generate a unique key, instead shared memory segments can be referenced by the file descriptor

destroy: shm_close(), analogous to fclose() for files but only removes the file descriptor from the address space of the process
shm_unlink(), analogous to unlink() for files and necessary to delete all the shared memory related data structure and free up the shared memory segment

3.3.10 Shared Memory and Sync

Just like threads accessing shared states in a single address space, inter-process synchronization is required to avoid data race condition for concurrent access to shared memory by multiple processes. Inter-process synchronization can be handled by

- the same mechanisms supported by threading library
- OS-supported IPC for synchronization

Either method must coordinate

- concurrent accesses to shared segment (mutex)
- when data is available in shared segment (condition variable)

3.3.11 PThreads Sync for IPC

Recall that passing NULL to pthread_mutexattr_t and pthread_condattr_t makes the mutex and condition variable private to/only visible within the process (the default). To share them among processes we need to pass PTHREAD_PROCESS_SHARED.

```
pthread_mutexattr_t(&m_attr);
pthread_mutexattr_set_pshared(&m_attr, PTHREAD_PROCESS_SHARED);
pthread_mutex_init(&shm_ptr.mutex, &m_attr);
```

In addition the synchronization data structure must be shared among processes, analogous to the need to declare mutex and condition variable as global variable so as to make them visible to all threads in a multithreaded process. To achieve this the synchronization construct should be allocated in the shared memory region that is visible to both processes.

```
typedef struct{
    pthread_mutex_t mutex;
    char *data;
} shm_data_struct, *shm_data_struct_t;
```

To use it don't forget to cast the virtual address returned by shmat to this shared data structure type

```
segid = shmget(ftok(arg[0],120), 1024, IPC_CREATE|IPC_EXCL);{
shm_address = shmat(segid, (void*)0, 0);
shm_ptr = (shm_data_struct_t) shm_address;
```

Once initialized there is no difference in their actual usage, i.e. they can be used just as regular mutexes and condition variables in a multithreaded PThread process.

3.3.12 Sync for Other IPC

The PThreads synchronization mechanism is not supported on every single platform. Instead we rely on other forms of IPC for synchronization such as

message queue: implement mutual exclusion via send/recv, e.g. the sending process writes data to shmem and sends READY to queue, the receiving process reads data from shmem and sends OK to queue

semaphore: for a binary semaphore, if its value is 0 then the process will be blocked, if its value is 1 then a process will automatically decrement that value (equivalent to lock) and proceed execution

3.3.13 Message Queue Quiz

For message queue the Linux system calls used for

msgsnd: send message to a message queue

msgrcv: receive message from a message queue

msgctl: perform a message control operation

msgget: get a message identifier

3.3.14 IPC Command Line Tools

ipcs: list all IPC facilities
-m displays information on shared memory IPC only

ipcrm: delete IPC facility
-m[shmid] deletes shm segment with given id

3.3.15 Shared Memory Design Considerations

There are different APIs/mechanisms for synchronization. Moreover once the OS provided the shared memory it is out of the way. All data passing and synchronization protocols are up to the programmer.

3.3.16 How Many Segments?

one large segment: need some memory manager for allocating to/freeing memory from shared segment

many small segments: use a pool of segments, one for each pairwise communication, and a queue of segment ids (nonetheless may need to communicate segment ids among processes via other IPCs)

3.3.17 Design Considerations

- segment size = message size works well for well-known static sizes, or to set max data size limit
- segment size < message size requires transferring data in rounds, however for this the programmer needs to include protocol to track the transfer progress (likely need to cast the shared memory area as some data structure consisting of synchronization construct + data buffer + progress flag)

3.4 Lecture 4 Synchronization Constructs

3.4.1 Visual Metaphor

Processes may

- repeatedly check whether it is OK to continue using a construct called spinlocks
- wait for a signal to continue using mutexes and condition variables

Regardless of which synchronization mechanism we use, wait hurts performance (wasted CPU cycles, cache misses).

3.4.2 More about Synchronization

Limitation of mutexes and condition variables include

- error prone (unlock wrong mutex, signal wrong condition variable) thus affects ease of use
- lack of expressive power (need helper variables for access or priority control)
- require lower level support from the hardware via atomic instructions

3.4.3 Spinlocks

Spinlock has constructs that are equivalent to mutex

```
spinlock_lock(s);
    // critical section
spinlock_unlock(s);
```

The way spinlocks differ from mutexes is that, when the lock is busy the thread that is suspended in its execution isn't blocked like in the case of mutexes but instead is spinning. It is running on the CPU and repeatedly checking whether the lock becomes free. With mutexes the thread would have to relinquish the CPU and allow another thread to run on it.
Because of their simplicities spinlocks are a basic synchronization primitive, and can be used to implement more sophisticated synchronization constructs.

3.4.4 Semaphores

A semaphore is like a traffic light. It either allows threads to go or stop them. On initialization a semaphore is assigned a positive integer. Threads arriving at the semaphore will try it out (operation P by Dijkstra)—if the semaphore's value is nonzero then they will decrement it and proceed, if the semaphore's value is zero then they will have to wait. This means that the number of threads allowed to proceed is equal to the positive integer initially assigned to the semaphore. Hence one of the benefits of semaphore is that it allows us to express count-related synchronization requirements. If a semaphore is initialized to 1 (binary semaphore), then it is equivalent to a mutex. All threads that are leaving the critical section will increment the semaphore's counter (operation V by Dijkstra). For a binary semaphore this is equivalent to unlocking a mutex.

3.4.5 POSIX Semaphores

POSIX semaphore API is illustrated as follows

```
#include ⟨semaphore.h⟩
sem_t sem;
sem_init(sem_t *sem, int pshared, int count);
sem_wait(sem_t *sem);
sem_post(sem_t *sem);
```

where the pshared flag will indicate whether the semaphore is shared by threads within a single process or across processes.

3.4.6 Mutex via Semaphore Quiz

With pshare = 0 and count = 1 a semaphore will behave identically to a mutex used by threads within a process.

3.4.7 Reader/Writer Locks

When specifying synchronization requirements it is useful to distinguish among different types of accesses e.g. accesses that never modify the shared resource (e.g. read) vs. accesses that always modify the shared resource (e.g. write). For the former shared access is permitted whereas for the latter exclusive access is necessary. For this reason OS and language runtime support so called reader/writer locks.

3.4.8 Using Reader/Writer Locks

In Linux a reader/writer lock can be defined as follows

```
#include ⟨linux/spinlock.h⟩
rwlock_t m;
read_lock(m);
    // critical section
read_unlock(m);
write_lock(m);
    // critical section
write_unlock(m);
```

Sometimes referred to as shared/exclusive locks, reader/writer locks are supported in many OS and language runtimes. However there are possible semantic differences including

- the implementation of read_unlock for recursive read_lock (for one or for all)
- option to upgrade to a writer lock/downgrade to a reader lock or needs to release and reacquire
- interaction with scheduling policy (e.g. block readers if a higher priority writer is waiting)

3.4.9 Monitors

Monitors explicitly specify (1) the shared resource being protected (2) all possible entry procedures to the resource (3) possible condition variables. On thread entry all the necessary locking and checking will take place automatically. Similarly on thread exit all the necessary unlocking and checking and signaling for condition variables will occur automatically and be hidden from the programmer. Because monitors automatically take care of the pairwise lock/unlock operations and signals, they are referred to as a high-level synchronization construct. Monitors are supported by Java runtime.

3.4.10 More Synchronization Constructs

serializer: make it easier to define priorities and hide the need for explicit use of condition variables and signaling

path expression: need to specify the expression that captures the correct synchronization behavior e.g. many reads and single write

barrier: opposite to semaphore in the sense that it will block all threads until n threads arrive

rendezvous point: similar to barrier

wait-free sync: effort to achieve concurrency without explicitly locking and waiting (e.g. read-copy update log or RCU in Linux kernel)

All the above constructs require support from the underlying hardware to atomically make updates to a memory location so as to avoid data race.

3.4.11 Sync Building Block Spinlock

Anderson's paper *The Performance of Spin Lock Alternatives for Shared Memory Multiprocessors* discusses alternative implementations of spinlocks and generalize the atomic techniques to other constructs.

3.4.12 Spinlock Quiz 2

```
spinlock_init(lock):
    lock = free;
spinlock_lock(lock):
    while (lock == busy);
    lock = busy;
spinlock_unlock(lock):
    lock = free;
```

Even with while loop check the implementation is still incorrect, because multiple threads will exit this while loop and proceed to set the lock to be busy. In fact there is no way purely in software that can guarantee that race condition will not occur, and some hardware support is necessary.

3.4.13 Need for Hardware Support

We need checking the lock value and setting the lock value to happen indivisibly and atomically.

3.4.14 Atomic Instructions

Each hardware will support a number of atomic instructions such as test_and_set, read_and_increment, and compare_and_swap. Different instructions may be supported on different hardware platforms. The hardware guarantees that the set of operations included in the instruction will happen atomically (all or none), in mutual exclusion, and queue all other concurrent instructions. For example test_and_set(lock) == busy tests whether the original lock value is busy and sets the new lock value to be busy. Note that the same as the first thread that acquires the lock the other threads spinning in the while check loop also set the lock value to busy repeatedly. This is OK because they are not attempting to change the lock value set by the first thread.

3.4.15 Shared Memory Multiprocessors

A shared memory multiprocessor system (also called symmetric multiprocessors or SMPs) consists of more than one CPU and some memory that is accessible to all these CPUs. The shared memory may be a single memory component that is equidistant from all the CPUs or there will be multiple memory modules. There are two types of connections between CPUs and memory

bus based: only one memory reference can be in flight, more common in the past

interconnect based: can have multiple memory references in flight, more common in current systems

Each of these CPUs can have caches which are useful to hide memory latency. Memory latency is more of an issue in shared memory systems because there is contention on memory modules and thus certain memory references have to be delayed. When CPUs perform a write several things can happen

no-write: no write is allowed to cache, a write will directly go to memory and any cached copy of that will be invalidated

write-through: write to both cache and memory

write-back: write to cache immediately but to memory later (e.g. after the cache is evicted)

3.4.16 Cache Coherence

non-cache-coherent (NCC) architecture: the hardware is not responsible for ensuring that the data appearing in multiple caches are consistent, it has to be dealt with purely in software

cache-coherent (CC) architecture: the hardware will ensure that the data appearing in multiple caches are consistent

There are two basic methods to manage cache coherence

write-invalidate (WI): after one CPU changes the value of a variable, the hardware will invalidate that variable in any other cache if exists, thus any future reference will result in a cache miss and will be pushed to memory
the advantage of this method is lower bandwidth requirement, because only the address is needed for invalidation, moreover invalidation only needs to be done once thus the cost of coherence traffic can be amortized

write-update (WU): after one CPU changes the value of a variable, the hardware will update the value of that variable accordingly in any other cache if exists, thus any future reference will result in a cache hit
the advantage of this method is the immediate availability of the new value

As a programmer you don't have a choice between these two methods, because this is a property of the hardware architecture.

3.4.17 Cache Coherence and Atomics

The purpose of atomic instructions is to deal with issues that are related to arbitrary interleaving of threads. To avoid the hassle of applying atomic instructions to all caches, atomic operations bypass caches and always directly access memory. The pro and con of forcing all atomics to go directly to the memory controller are

pro: all memory references can be ordered and synchronized thus race condition would not occur

con: it takes much longer to apply and is more expensive due to (1) the bus or interconnect contention (2) the cache bypass and the cache coherence traffic generation

3.4.18 Spinlock Performance Metrics

The three objectives that we want to achieve in a good spinlock design are

1. reduce latency—the time to acquire a free lock

2. reduce delay—the time to stop spinning and acquire a lock that has been freed

3. reduce contention—over both bus/interconnect and coherence traffic, contention is bad because it will delay not only any other CPU that is trying to access the memory but also the owner of the spinlock

3.4.19 Conflicting Metrics Quiz

Out of the three objectives above

- 1 conflicts with 3 because immediate execution of atomic instructions will potentially create additional contention on the network

- 2 conflicts with 3 because continuous spinning will create contention

3.4.20 Test and Set Spinlock

test_and_set is a very common atomic construction that most hardware platforms support it, so as code it will be very portable.

latency: minimal, since only atomic execution

delay: potentially minimal, since spinning continuously on the atomic instruction

contention: not well, as spinning means all threads will repeatedly go on the shared bus/interconnect to the shared memory location where the lock is stored creating contention, note that with this implementation even if we do have coherent caches, they will be bypassed because we are using atomic instruction, thus in every single spin we will go to memory regardless of cache coherence

3.4.21 Test and Test and Set Spinlock

For test_and_set the while loop check is

```
spinlock_lock(lock):
    while(test_and_set(lock) == busy);
```

For test and test_and_set an extra test is added to the atomic check

```
spinlock_lock(lock):
    while((lock == busy) OR (test_and_set(lock) == busy));
```

The intuition is that for the extra test we can potentially test the cached copy of the lock value involving no atomic operation, and only when the cached copy of the lock is freed do we apply the atomic test and set which involves memory access. This test and test_and_set spinlock is referred to as spin on read in Anderson's paper (also spin on cached value).
From the latency and delay perspective this implementation is slightly worse than the test_and_set spinlock because of the extra cached value check but is still OK. The real problem is associated with contention.

NCC architecture: this extra test makes no difference since every single reference will go to memory just like test_and_set

CC-WU architecture: all the CPUs with the right cached value will see the lock is freed, thus all of them will try to execute test_and_set at the same time

CC-WI architecture: the worst case because every single attempt to acquire the lock will not only generate contention for the memory module but also generate invalidation traffic

3.4.22 Test and Test and Set Spinlock Quiz

In an SMP system with n processors, what is the complexity of the memory contention/accesses that will result from test and test_and_set spinlock when the lock is freed?

CC-WU architecture: all the CPUs will be able to see the lock is freed immediately and thus will issue a test_and_set operation, hence there will be as many memory references as there will be test_and_set operations, therefore the complexity is $O(n)$

CC-WI architecture: all the CPU caches will be invalidated after the lock is freed, for some of the CPUs when they re-read the lock value from memory the lock would have been set to busy again, so they will spin on the newly read cached copy of the lock value, while for others they will see the lock is freed and so will execute test_and_set, only one of these test_and_set operations will succeed, which will then invalidate every CPU's cached copy including those re-read and found and cached the busy lock value, therefore the complexity is $O(n^2)$

3.4.23 Spinlock Delay Alternatives

A simple way to alleviate contention of the test and test_and_set spinlock is to introduce delay after lock release so that most of the threads will not execute test_and_set because their double check (the outer while loop) will find the lock is busy again.

```
spinlock_lock(lock):
    while((lock == busy) OR (test_and_set(lock) == busy)){
        while(lock == busy);
        delay();
    }
```

As a result the contention is improved. The latency is similar to the case without delay. However from the delay perspective this lock is clearly worse, because the delay would be a waste of time if there is no contention for the lock.
A variant of this delay-based test and test_and_set spinlock is to introduce delay after every single lock reference without inner while loop check. The main benefit is that it works in NCC architecture and the delay helps reduce the contention of memory modules (in a NCC architecture we always have to go to memory).

```
spinlock_lock(lock):
    while((lock == busy) OR (test_and_set(lock) == busy)){
        delay();
    }
```

The downside of this implementation is that it hurts the delay much more, because we are building up delay even when there is no contention on memory.

3.4.24 Picking a Delay

There are two basic strategies

static: based on some fixed value e.g. CPU ID and the length of critical section

- **pro:** simple implementation, and under high loads it is likely that this static delay will spread out over the atomic references so that there is no contention
- **con:** unnecessary delay under low contention (for high CPU ID)

dynamic: random delay in a range that increases with perceived contention level, more popular, tend to be equivalent to the static delay strategy under high loads

- **pro:** dynamically adjusted to suit contention level
- **con:** perceived contention level may not be the actual contention level, a good metric is the number of failed test_and_set operations, however it is complicated by the length of critical section (high contention or long critical section?)

3.4.25 Queuing Lock

Queuing lock uses an array of flags with n elements where n is the number of CPUs. Each flag has one of the two values—has-lock or must-wait. In addition there are two pointers indicating current lock owner (with has-lock flag) and the last element in the queue. When a new thread arrives at the lock it will be added after the existing last element in the queue. Since multiple threads may arrive at the same time, this last element pointer needs to be incremented atomically thus this queuing spinlock relies on hardware support for atomic read_and_increment, which however is not as common as for test_and_set. This is another downside in addition to the much larger $O(n)$-size memory location. When a thread completes the critical section and releases the lock, it needs to signal the next flag in this queuing array that it currently has the lock i.e. queue[ticket+1] = has-lock.

3.4.26 Queuing Lock Implementation

Anderson proposed the following implementation of queuing lock

```
init:
    flags[0] = has-lock;
    flags[1...$p-1$] = must-wait;
    queuelast = 0; //global variable
lock:
    myplace = read_and_increment(queuelast); //get ticket
    while(flags[myplace mod $p$] == must-wait);
    //critical section
    flags[myplace mod $p$] = must-wait;
unlock:
    flags[myplace+1 mod $p$] = has-lock;
```

latency: not efficient due to the extra and more complex read_and_increment operation which actually takes more cycles than test_and_set, in addition modular shift is required before lock value check

delay: negligible because when the lock is freed the next-to-run thread is directly signaled by changing the value of its flag

contention: much better than any of the other alternatives because

- there is only one thread at a time that sees the lock is freed and tries to acquire the lock
- the atomic operation is executed only once and up front and is not part of the spinning code
- the variable the threads spin on (the elements of the flag array) is not the variable the atomic read_and_increment operates on, hence the invalidation triggered by the atomic instruction will not affect the threads' ability to spin on local caches

however this implementation requires (1) a CC architecture (2) cache line aligned elements (otherwise changing the value of one element of the flag array will invalidate the caches of the other elements)

3.4.27 Queueing Lock Array Quiz

If a system has 32 CPUs, how large is the array data structure required to implement Anderson's queuing spinlock? The answer depends on the cache line size. Because for the queuing implementation to work correctly, each of the flag array elements has to be in a different cache line so as not to be invalidated upon the update of some other element. If the cache line is 64 bytes, then the size of the data structure required is 32 × 64 bytes.

3.4.28 Spinlock Performance Comparisons

The metric is the overhead compared to ideal performance—the theoretical time limit of executing the critical section for a fixed number of times.

high loads: queuing lock is the best because it is the most scalable, test and test_and_set lock is the worst because we have a CC-WI architecture thus test and test_and_set requires $O(n^2)$ memory references
static delay is slightly better than dynamic delay since it spreads out the atomic operations and random thread collision is avoided
delay after memory reference is slightly better than delay after lock release because additional invalidation is avoided

light loads: test and test_and_set lock is the best because its spinning has low latency, queuing lock is the worst due to the extra complex read_and_increment operation and modular shift
dynamic delay is better than static delay because it doesn't have extreme delay as in static case

3.5 Lecture 5 I/O Management

3.5.1 Visual Metaphor

protocol: the OS incorporates different interfaces for different types of I/O devices, which determines the protocols used for accessing these devices

handler: device drivers, interrupt handlers

decouple: abstract the details of the I/O devices and hide them from applications or upper level of system software

3.5.2 I/O Device Quiz

input device: keyboard, microphone

output device: display, speaker

both: hard disk drive, network interface card (NIC), flash card

3.5.3 I/O Device Features

Any device can be abstracted to have the following set of features

control registers: a set of control registers accessible by the CPU to realize CPU-device interaction, including

- **command register:** used by the CPU to control what the device will be doing
- **data register:** used by the CPU to control the data transfer in and out of the device
- **status register:** used by the CPU to check the status of the device

micro-controller: the device's CPU, controlling the operation of the device

on-device memory: DRAM or SRAM or both

other hardware-specific logic device: e.g. analog-to-digital converter

3.5.4 CPU-Device Interconnect

Devices interface with the rest of the system via controller that is typically integrated as part of the device packaging. It determines what type of interconnects the device can directly attach to Peripheral component interconnect (PCI) is one of the standard methods for connecting devices to the CPU. Today's platforms typically support PCI express interconnect (PCIe) which are more advanced than the original PCI and PCI extended (PCI-x). Other types of interconnects include SCSI bus connecting to SCSI disks, peripheral bus connecting mice and keyboard. Bridging controller handles differences between different types of interconnects.

3.5.5 Device Drivers

OS supports devices via device drivers, which are device-specific software components and are responsible for all aspects of device access, management, and control. Thus an OS has to include a device driver for every type of different hardware devices in the system. OS standardizes their interfaces to device drivers. Typically this is achieved by providing some driver framework so that device manufacturers can develop the specific device driver within that framework. Thus there is standardization in terms of both the interaction with devices and the development of devices. In this way we achieve both device independence (an OS does not have to be specialized or integrate a particular functionality for a specific device) and device diversity (an OS can support different types of devices).

3.5.6 Types of Devices

block devices: devices that operate at a granularity of blocks of data e.g. disks, a key property is that individual blocks can be directly accessed

character devices: devices that work with a serial sequence of characters e.g. keyboard

network devices: somewhat in-between in the sense that they deliver characters in blocks but the granularity is not necessarily a fixed block size, more like a data stream

In this manner the interfaces from the OS to the devices are standardized based on the types of the devices e.g.

block devices: support read/write in blocks

character devices: support put/get a character

Internally an OS typically uses file abstraction to represent different devices. As such the OS can use mechanisms that are used to manipulate files such as read and write to access devices albeit handled in some device-specific manner. On Unix-like systems all devices appear as files underneath the /dev directory. But as special files they are treated by special file systems such as tmpfs and devfs.

3.5.7 Pseudo Devices Quiz

Linux supports pseudo or virtual devices which don't represent actual hardware e.g.

- /dev/null accepts and discards all output (i.e. produces no output)
- /dev/random produces a variable-length string of pseudo-random numbers
- /dev/lp0 represents first line printer

3.5.8 CPU-Device Interactions

There are two CPU-device interaction models

memory-mapped I/O: The main way in which PCI interconnects devices to the CPU is by making devices accessible in a manner that is similar to how CPUs access memory. The device registers appear to the CPU as memory locations. When the CPU writes to these locations the integrated memory PCI controller realizes that this access should be routed to the appropriate device. This requires part of the physical memory of the host to be dedicated to device interactions, and this reserved portion of physical memory is controlled by the base address registers (BAR) and configured during the boot.

I/O port: The CPU can also access devices via special instructions, each of them has to specify the target device (the I/O port) and some value that is to be stored in registers.

The path from the device to the CPU can take two routes

interrupt: devices can generate interrupts to the CPU

> **pro:** can be generated as soon as needed
>
> **con:** interrupt handling overheads

polling: the CPU can poll the device by reading its status register

> **pro:** do when it is convenient for the OS which reduces overhead
>
> **con:** occasional polling introduces delay whereas continuous polling introduces CPU overhead

3.5.9 Device Access PIO

With basic PCI interconnect the CPU can program a device by writing instructions into the command register of the device and controlling data movement via access to the data register of the device. This is called **programmed I/O**.

3.5.10 Device Access DMA

DMA stands for direct memory access and relies on special hardware support in the form of DMA controller. The way CPU interacts with a device via DMA consists of writing instructions into the command register of the device and configuring the DMA controller. The latter involves specifying the physical address of the data and the amount of the data. This requires that the data buffer must be present in physical memory until data transfer is complete, hence the memory region involved must be pinned and non-swappable. Although from the CPU perspective the number of steps is much fewer than in programmed I/O, DMA controller configuration is much more complex and takes many more cycles than controlling data movement via access to the data register of the device. Therefore for small data transfer programmed I/O is still preferred.

3.5.11 DMA vs. PIO Quiz

Which device access method is better for the following devices

keyboard: PIO, because it is unlikely to transfer much data for each keystroke

NIC: depends, PIO for small packets while DMA for large packets (cutoff: number of cycles to configure the DMA controller)

3.5.12 Typical Device Access

1. user process issues system call to kernel to specify the operation (send data, read file)
2. kernel runs in-kernel stack (form packet, determine disk block)
3. device driver configures device request and sends it to device via PIO/DMA (transmission record, disk head move)
4. device performs request (transmission, read blocks)

$$\text{user process} \xrightarrow{\text{system call}} \text{kernel} \xrightarrow{\text{form}} \text{driver} \xrightarrow{\text{request}} \text{device}$$

3.5.13 OS Bypass

It is possible for a user process to directly access a device bypassing the kernel. The OS is involved in configuring the access but once done it is out of the way. To support this operation the device manufacturer must provide user-level driver/library that the user process has to be linked with.
Although bypassed, the OS still needs to exercise some coarse-grain control such as enabling the device and adding permission to add more user processes to access the device etc. To do this the OS relies on some device features such as

- sufficient registers so that the OS can map some to potentially multiple user processes for performing default device functionality and retain others necessary for coarse-grain control i.e. can't reuse
- the device should be able to demultiplex so as to figure out which data to send to which user process

3.5.14 Sync vs. Async Access

synchronous I/O operations: the calling process will be blocked and placed on the wait queue associated with the corresponding device, and become runnable when the response to the request becomes available

asynchronous I/O operations: the calling process is allowed to continue after issuing the I/O call, the process will either come back later to check whether the result is ready or be notified by the device that the I/O operation has completed

3.5.15 Block Device Stack

Block devices like disks are typically used for storage, and the typical storage-related abstraction is a file. Below this file-based interface used by user process is the kernel file system. The OS typically provides some flexibility in the actual details of the file system, e.g. the OS allows the file system to be modified or replaced. To make this easy the OS specifies something about the file system interface, which includes both the interface used by the application to interact with the file system where the norm is POSIX API, and the interface used by the file system to interact with the underlying block device and other OS components.
At the lowest level the file system needs to interact with block devices via their device drivers, which requires protocol specific API. To mask all these differences the block device stack introduces a generic block layer. The purpose is to provide a standard to all types of block devices. The full device details are still accessible through device drivers, but user processes can invoke generic read/write operation through POSIX API that will be interpreted by the kernel file system and the generic block layer.

3.5.16 Block Device Quiz

In Linux the command *ioctl* is used to directly access and manipulate devices.

```
int fd;
unsigned long numblocks = 0;
fd = open(argv[1], O_RDONLY);
ioctl(fd, BLKGETSIZE, &numblocks);
close(df);
```

3.5.17 Virtual File System

The virtual file system layer hides from applications all details regarding the underlying file systems (how many devices, local or remote, which file system), and specifies a more detailed set of file system related abstractions that every underlying file system must implement.

3.5.18 Virtual File System Abstractions

The virtual file system (VFS) supports the following key abstractions

file: elements on which VFS operates

file descriptor: the OS representation of file

inode: persistent representation of file “index”, index in the sense that it includes a list of all data blocks of the file along with other information such as size and permission, it is a standard data structure in Unix-based systems, it is necessary because files do not need to be stored contiguously on disk

dentry: directory entry, each dentry object corresponds to a single path component to be traversed to reach the file (e.g. both / and /users are stored as dentry object), the file system will maintain a cache of all the directory entries that have been visited and we call it the dentry cache, note that contrary to inode for files there is no persistent on-disk representation of dentry objects, it is only in the memory maintained by the OS

superblock: a map the file system maintains so that it can figure out how has it organized on disk the various persistent data elements along with some additional metadata, exactly what and how information is stored is file system specific

3.5.19 VFS on Disk

The VFS data structures are software entities. Other than dentries the remaining components actually correspond to blocks that are present on disk.

file: data blocks on disk

inode: track file’s blocks, also resides on disk in some block

superblock: overall map of disk blocks, specifying which blocks are inode blocks, data blocks, or free blocks (used for file creation, file write, and file lookup)

3.5.20 ext2: Second Extended Filesystem

ext2, which stands for extended file system version 2, was the default file system in several versions of Linux until it was replaced by ext3 and then ext4 more recently. It is also available for other operating systems and not just Linux specific. A disk partition that is used as an ext2 Linux file system consists of boot, the first block containing the code to boot the computer, followed by block groups. Each block group is organized as follows

superblock: information about the overall group such as number of inodes, number of disk blocks, start of free blocks

group descriptor: location of bitmaps, number of free nodes, number of directories

bitmap: used to quickly find a free block or a free inode

inode: numbered from 1 to some max number, 1 per file, every inode is 128 bytes in ext2

data block: file data

3.5.21 Inodes

A file is uniquely identified by its inode. In VFS inodes are uniquely numbered so to identify a file we use the number identifier of the corresponding inode. The inode contains a list of the indices of all the blocks that correspond to the file, along with some other metadata information that is useful to keep track of whether certain file access is legal and whether the file is locked etc. Free blocks are identified by -1 and when allocated will be updated by the block index. The pro and con of this approach are

pro: easy to perform both sequential and random access (compute block index based on block size)

con: the file size that can be indexed is very limited (given 4 bytes per block pointer and 2KB per data block, a 128-byte inode can address only $128/4 \times 2 = 64$KB file size)

3.5.22 Inodes with Indirect Pointers

Indirect pointers are used to extend the number of data blocks that can be addressed by a single inode while keeping the size of the inode small. They are pointers that point to blocks of block pointers instead of data blocks. Given 4 bytes per block pointer and 2KB per data block

direct pointer: points to 2KB data per entry

indirect pointer: points to $2048/4 \times 2$KB $= 1024$KB data per entry

double indirect pointer: points to $2048/4 \times 1024$KB $= 512$MB data per entry

The pro and con of using indirect pointer are

pro: small inode is able to address large file size

con: slow file access (direct pointer: 2 disk accesses to reach data block vs. double indirect pointer: 4 disk accesses to reach data block)

3.5.23 Inode Quiz

Maximum file size is calculated by

$$\text{max file size} = \text{max number of addressable blocks} \times \text{block size}$$

where maximum number of addressable blocks is given by

$$\begin{aligned} \text{max number of addressable blocks} &= \text{number of direct data block pointers} \\ &+ \text{number of data blocks addressable by single indirect pointers} \\ &+ \text{number of data blocks addressable by double indirect pointers} \\ &+ \text{number of data blocks addressable by triple indirect pointers} \end{aligned}$$

Given a block size of 1KB, 12 direct data block pointers, and 4 bytes per block pointer, the maximum number of data blocks addressable is $12 + 1024/4 + (1024/4)^2 + (1024/4)^3$; if the block size is increased to 8KB the maximum number of data blocks addressable is $12 + 8096/4 + (8096/4)^2 + (8096/4)^3$.

3.5.24 Disk Access Optimizations

The following methods are used to reduce file access overheads

caching/buffering: to reduce number of disk access by

- buffer cache in main memory to enable read/write from cache
- periodically flush any changes to the file that have not been backed up on permanent storage from memory to disk via fsync() system call, periodic flushing enables amortizing disk access over multiple intermittent cache hits

I/O scheduling: to reduce disk head movement by maximizing sequential accesses against random accesses (e.g. reorder write block 25 and write block 17 into write block 17 and write block 25)

prefetching: to increase cache hits by also fetching subsequent blocks when a single block is accessed (e.g. read also block 18 and 19 when reading block 17), this does use up more disk bandwidth to move larger blocks, but leveraging locality reduces access latency

journaling/logging: to reduce random access to disk by writing updates in log (describing the write that is supposed to take place e.g. block index, offset, value) instead of directly in disk, note that a journal eventually has to be updated periodically into a proper disk location

3.6 Lecture 6 Virtualization

3.6.1 What is Virtualization

virtualization: the technique to allow concurrent execution of multiple OSs (and their applications) on the same physical machine to deal with diverse workloads

virtual resources: each OS deployed on the same physical platform has an illusion that it owns the underlying hardware resources or some small portion of them

virtual machine: each OS together with its applications as well as the virtual resources that it thinks it owns is called a virtual machine or VM, VM are often referred to as guest VM or guest domain

virtualization layer: also referred to as virtual machine monitor or hypervisor, responsible for allocating and managing physical resources and providing isolation guarantees across VMs

3.6.2 Defining Virtualization

In order for a virtual machine to become an efficient isolated duplicate of the real machine, the virtual machine monitor (VMM) that enables virtualization must have three essential characteristics

fidelity: provide an environment that is essentially identical to the real machine i.e. the same components and devices albeit the capacity might differ, thus the representation of the hardware that is visible to the VM should match the hardware that is available on the physical platform

performance: deliver the VM performance as close to that of the real machine as possible if given the exact amount of resources as the physical machine

safe isolation: give the VM complete control of system resources which is isolated from other VMs

3.6.3 Virtualization Tech Quiz

JVM is a language runtime which provides system services and portability to Java applications that is very different from the underlying physical machine, thus it is not a duplicate of the real machine and hence not a VM.
Virtual Gameboy emulates some hardware platform that is very different from the hardware the emulator is running on, thus it is not a duplicate of the real machine and hence not a VM.

3.6.4 Benefits of Virtualization

consolidation: by running multiple OSs and their applications on a single physical platform, virtualization reduces cost and improves manageability

migration: by encapsulating an OS and its applications in a VM, virtualization makes it easier to migrate the OS and the applications from one physical machine to another or clone them onto multiple physical machines, therefore provides greater availability of the applications and greater reliability of the services (migrate from overheated physical nodes)

security: virtualization makes it easier to contain any bug or any malicious behavior to those resources that are available to the VM only

debugging: virtualization has become a very important platform for OS research as it allows researchers to quickly test new OS features

legacy support: virtualization provides affordable support to legacy OS without need to designate hardware resources for the old system

3.6.5 Benefits of Virtualization Quiz 1

The reason why virtualization hasn't been widely used since it was first proposed in 60s is two-fold (1) mainframe machines were not ubiquitous (2) other hardware was cheap. This trend of simply buying more machines if a different OS is needed to support different applications continued for decades.

3.6.6 Benefits of Virtualization Quiz 2

Recently people started caring about virtualization because

- servers were underutilized (at most 20%)
- data centers were becoming too large

- companies had to wire more system admins to manage large data centers
- companies were paying high utility bills to run and cool data centers (70% of IT budget)

3.6.7 Virtualization Models Bare Metal

The two main virtualization models are (type 1) bare-metal or hypervisor-based (type 2) hosted. The hypervisor-based model uses a virtual machine monitor or hypervisor to manage all hardware resources and support execution of VMs. One issue with this model are devices because device manufacturers have to provide device driver for different hypervisors. To cope with it the hypervisor model integrates a privileged service VM that runs a standard OS to run all the device drivers and control the devices. It also runs the management tasks that specify how the hypervisor would share resources across VMs. The two prevailing products based on hypervisor are

Xen: opensource or Citrix XenServer, the privileged domain is called dom0 and the guest VMs are referred to as domUs, all device drivers are running in dom0

ESX: VMware, owning the largest virtual server core market share it mandates device manufacturers to provide drivers for the hypervisor, nonetheless it provides many open APIs to support developers, it used to have Linux control core which are now replaced by remote APIs

3.6.8 Virtualization Models Hosted

At the lowest level there is a full-fledged host OS that manages all the hardware. The host OS integrates a virtual machine monitor module responsible for providing VMs with virtual platform interface and managing all context switch scheduling. One benefit of this model is that it can leverage all the services of the host OS thus much less functionality development is needed.
One example of the hosted model is kernel-based VM (KVM) which is hosted by Linux. The support for running guest VM is through a combination of a kernel module and a hardware emulator called QEMU for hardware virtualization. A benefit of KVM is that it is able to leverage all the advances made by the Linux open-source community.

3.6.9 Bare Metal or Hosted Quiz

Hypervisor-based virtualization products include VMware ESX, Citrix XenServer, Microsoft Hyper-V. Host OS based virtualization products include KVM, Fusion, VirtualBox, VMware Player.

3.6.10 Virtualization Requirements Quiz

Virtualization requirements include

- VMM must provide VMs with a virtual platform interface to all the hardware resources
- VMM must isolate VMs from one another (using mechanisms similar to those used by OS)
- virtualization must protect guest OS from faulty or malicious applications, thus cannot run guest OS and applications at the same protection level
- virtualization must protect VMM from guest OS bringing down the hypervisor of the entire machine, thus cannot run guest OS and VMM at the same protection level

3.6.11 Hardware Protection Levels

To meet the requirement of offering different protection levels (see the quiz above), commodity hardware usually has more than 2 protection levels. For example the x86 architecture has 4 protection levels called rings. Ring 0 has the highest privilege and can access all resources and execute all hardware-supported instructions which is where OS resides in a native model, whereas ring 3 has the lowest which is where applications reside. When a trap occurs in ring 3, control would be switched to ring 0. These protection levels can be used for virtualization by putting the hypervisor in ring 0, guest OS in ring 1, and applications in ring 3.
More recent x86 architectures also introduce two different protection modes: root and non-root, each of which has 4 protection levels. When running in root mode all operations are permitted. In contrast in non-root mode certain types of operations are not permitted even for ring 0. Thus the hypervisor should reside in ring 0 of root mode whereas guest OS should reside in ring 0 of non-root mode. Attempts by guest OS to perform privileged operations cause traps that are called VMexits, which trigger a switch to root mode and pass control to the hypervisor. When the hypervisor completes checking it passes control back to guest OS in non-root mode by performing a VMentry.

3.6.12 Processor Virtualization

To achieve efficiency at near native speeds, guest OS instructions are executed directly by hardware. The VMM does not interfere non-privileged operations which will then operate at hardware speeds for efficiency. When a privileged operation is initiated, guest OS causes a trap and control is switched to the hypervisor, which then determines whether it is a

illegal operation: terminates VM

legal operation: emulates (slightly different operation) the behavior guest OS expects from the hardware

This trap-and-emulate mechanism is a key method for virtualization to achieve near native efficiency.

3.6.13 x86 Virtualization in the Past

Trap-and-emulate doesn't solve all problems of pre-2005 x86 architecture which doesn't have root/non-root modes yet. Because there are 17 privileged instructions that if issued from ring 1 do not trap and fail silently (e.g. interrupt enable/disable bit in privileged register).

3.6.14 Problematic Instructions Quiz

As mentioned above, in earlier x86 platforms the flags privileged register was accessed via the instructions POPF and PUSHF that failed silently if not called from ring 0. As a result guest VM could not request interrupt enabled/disabled or even figure out the status of the interrupt enabled/disabled bit.

3.6.15 Binary Translation

One approach to solve the problem with the 17 instructions is to rewrite the binary of guest VM so that it never issues any of these 17 instructions. This method is called binary translation.
The goal pursued by VMware is to run unmodified guest OS (without any special drivers or policies etc.). This type of virtualization is called *full virtualization.* The basic approach consists of

1. dynamically capture the instruction sequences to be executed (in granularity of loop or function—likely to be input dependent thus cannot be static)

2. inspect the code block to see whether any of the 17 instructions is to be issued
    - if not, then allow it to execute natively at hardware speed
    - if yes, then translate it to an alternative instruction sequences that avoids issuing the instruction and emulates the desired behavior (possibly avoid trap)

To reduce the overhead added by binary translation, translated blocks are cached and kernel code is distinguished from application code.

3.6.16 Paravirtualization

A different approach is to give up the goal of unmodified guest OS. Instead the goal is to offer a virtualization solution that offers efficient performance. This type of virtualization is called *paravirtualization*. With paravirtualization guest OS is modified so that

- it knows it is running in a virtualized environment on top of a hypervisor
- it makes explicit calls to the hypervisor called hypercalls, which behave similar to system calls

Paravirtualization was originated and popularized by the Xen hypervisor, which was commercialized as XenSource and acquired by Citrix.

3.6.17 BT and PV Quiz

Accessing a page that is swapped out will cause a trap and exit to the hypervisor for both binary translation and paravirtualization VMs. Updating to page table entry will not if the guest OS has write permission for the page table.

3.6.18 Memory Virtualization: Full

For full virtualization a key requirement is that guest OS observes contiguous physical memory starting at address 0. To achieve this we distinguish among three types of addresses—virtual vs. physical vs. machine, and similarly for page numbers.

option 1: guest OS maps virtual addresses → physical addresses followed by hypervisor maps physical addresses → machine addresses, hardware support such as MMU and TLB can be leveraged to expedite the second translation, however it is still expensive because every single address goes through two translations

option 2: guest OS maps virtual addresses → physical addresses while hypervisor maintains a shadow page table mapping virtual addresses → machine addresses directly, which can be used by MMU to achieve execution at native speed, but then the hypervisor has to be responsible for maintaining consistency with guest page tables, invalidating shadow page tables upon context switch, and write-protecting guest page tables so that installing new guest page tables will cause a trap to the hypervisor

3.6.19 Memory Virtualization: Paravirtualized

Since guest OS knows it is running on a virtualized environment, there is no longer the strict requirement on contiguous physical memory and starting at address 0. Besides guest OS can explicitly register its page tables with hypervisor so there is no need for hypervisor to maintain dual page tables. Although guest OS still doesn't have write permissions to its page tables, but now we can modify guest OS to achieve batch processing guest page table update for a single hypercall.

3.6.20 Device Virtualization

Virtualization is relatively simple for CPUs and memory, because there is a significant level of standardization at the instruction set architecture (ISA) across different platforms. In contrast for devices virtualization is more complicated, because there is a much greater diversity and there is lack of standardization in terms of the specifics and semantics of device interface. To deal with this diversity virtualization adopts three key models (see below). Note that these models were developed before any virtualization friendly hardware extensions were made, which make some aspects of device virtualization much simpler.

3.6.21 Passthrough Model

Passthrough is achieved by the VMM-level driver configuring device access permissions thus allowing guest VM to have access to the memory where the control registers of the device are mapped. The pro and con of this model are

pro: there are two

- guest VM has exclusive access to a device
- guest VM can directly access the device bypassing the hypervisor (thus also called VMM-bypass model)

con: there are three

- device sharing is difficult due to the exclusive access
- the device has to be of exactly the same type as expected by guest OS, because the hypervisor is out of the way thus the device driver in guest VM directly operates on the device
- VM migration is tricky, because the passthrough directly binds a device to guest VM

3.6.22 Hypervisor Direct Model

The second model of device virtualization, originally adopted by VMware ESX hypervisor, is to allow hypervisor intercept all device accesses by guest VM, which then emulates the expected device operation by

1. translating guest VM's request to a generic I/O operation for that particular family of devices
2. traversing the VMM-resident I/O stack to the bottom, which is the real device driver
3. invoking the VMM-resident device driver to perform the I/O operation on behalf of guest VM

The pro and con of this model are

pro: guest VM is decoupled from physical devices, which facilitates device sharing and VM migration

con: increased latency of device operations due to the emulation steps, and the hypervisor needs to support all the device drivers in the device family

3.6.23 Split Device Driver Model

The model is called split because device access control is split between

front-end device driver: device API, resides in guest VM

back-end device driver: real device driver, resides in service VM (type 1 virtualization) or host OS (type 2 virtualization)

The pro and con of this model are

pro: it eliminates emulation overheads and allows for better management of device sharing, because device access management is centralized by the back-end device driver

con: since the front-end device driver has to be modified because it needs to explicitly wrap up applications' device requests and pass them to the back-end device driver, it applies to paravirtualization only

3.6.24 Hardware Virtualization

Key virtualization-related hardware features for x86 architectures introduced by AMD Pacifica and Intel Vanderpool Technology (VT) starting from 2005 are

- close the holes of the 17 non-virtualizable instructions
- introduce non-root/guest mode
- VM control structure to track the state of virtual CPUs, and allow hypervisor to decide whether an operation should not cause a trap in root mode and should instead be handled by ring 0 of non-root mode
- extend page tables by tagging memory with VM IDs, also tag TLB with VM IDs for more efficient switch among VMs
- multiqueue devices so that different logical interfaces can be assigned to different VMs, interrupt routing to the CPU the VM is running on
- stronger security guarantees to protect VMs from one another and from hypervisor, more virtualization friendly management interface

4 MODULE 4

4.1 Lecture 1 Remote Procedure Calls

4.1.1 Why RPC

RPC was born to capture all the common steps that are related to remote IPCs to simplify them.

4.1.2 Benefits of RPC

RPC offers

- higher-level interface that captures all aspects of data movement and communication
- error handling
- hiding complexities of cross-machine interactions (machines may be of different types, the network between them or themselves may fail etc.)

4.1.3 RPC Requirements

client-server interaction: RPC needs to manage the interaction

procedure call interface: RPC needs to simplify the development of distributed applications underneath a procedure call interface (where the name RPC comes from) through synchronous call semantics—when a process makes RPC, the calling thread will block and wait till the RPC is completed

type checking: error handling similar to regular procedure call, packet interpretation

cross-machine conversion: RPC should hide the difference in the representation of integers, floating points etc. one way to deal with it is to have both endpoints agree upon a single data representation

higher-level protocol: RPC should support different kinds of protocols (e.g. regardless of UDP or TCP), access control, fault tolerance, authentication etc.

4.1.4 Structure of RPC

Suppose the client wants to add two integers. The client calls RPC just like a regular procedure call and the following takes place in order

1. the client process calls RPC which directs it to the client stub implementation
2. the client stub creates a buffer and builds a message in it with the description of the operation and the required arguments
3. the RPC runtime sends the message to the server
4. the server OS hands the message to the server stub
5. the server stub extracts from the message the operation requested and the associated arguments
6. the server stub makes a regular procedure call to perform the operation
7. the server stub creates a buffer and builds a message in it containing the operation result

8. the server stub sends the message to the client
9. the client OS hands the message to the client stub
10. the client stub extracts from the message the operation result and the handle is returned to the calling client process

4.1.5 Steps in PRC

0. register: the server announces the available procedures and connections

1. bind: the client finds and binds to the desired server

2. call: the client makes RPC, passes the control to the client stub, the rest of the code blocks

3. marshal: the client stub packs the arguments (serialize arguments into a buffer—the RPC runtime will need to send a contiguous buffer)

4. send: the RPC runtime sends the message to the server

5. receive: the server receives the message, passes it to the server stub, access control check if necessary

6. unmarshal: the server stub unpacks the arguments and creates data structures to hold them

7. actual call: the server stub calls the local procedure implementation

8. result: the server performs the operation and computes the result

9. return: the result is passed to the server stub which packs it into a message and sends the message to the client

4.1.6 Interface Definition Language

The client and the server can be developed independently using different languages. But the agreement should be in place specifying the operations that can be performed and the associated arguments, so that the client can decide which server to bind and the RPC runtime can automate stub generation. This can be achieved by using PRC interface definition language (IDL).

4.1.7 Specifying an IDL

An IDL is used to describe the interface the server exports, which at the minimum should include (1) the procedure name, the arguments, and the result type (2) version number. The RPC system can use an IDL that is

language-agnostic: e.g. XDR in SunRPC, which is different from and simpler than any programming language

language-specific: e.g. Java in Java RMI

Note that IDL is used only for specifying the RPC interface, not the actual implementation of RPC.

4.1.8 Marshalling

The marshalling process needs to encode the data into some agreed upon format, which needs to be serialized into some contiguous memory location.

4.1.9 Unmarshalling

Developers do not need to write the marshalling and unmarshalling routines. Instead the RPC system typically includes a special compiler that takes an IDL specification that describes (1) the procedure prototype and (2) the argument data types, from which the marshalling and unmarshalling routines are generated.

4.1.10 Binding and Registry

binding: used by the client to determine which server to connect and how to connect it

registry: to support binding the system software needs to publish a database of available services, it can be

> **distributed:** any RPC server can register with
>
> **machine specific:** only for services that are running on the same machine, thus the client must know the machine address
>
> the procedure name protocol can be exact match or depend on some fuzzy logic

4.1.11 Visual Metaphor

The registry provides details about the available services through IDL.

4.1.12 Pointers in RPCs

In RPC passing a pointer to the remote server makes no sense, since it points to some location in the caller's address space. To solve this problem the marshalling routine needs to serialize the pointer by copying the data it refers to into the send buffer. On the server side after unpacking the message the RPC runtime needs to record the address to the decoded data and passes it to the procedure call as the pointer argument.

4.1.13 Handling Partial Failures

Being either unable or too costly to figure out the exact failure, RPC systems provide a special error notification that tries to capture what went wrong with an RPC request (timeout, exception etc.) without providing the exact detail.

4.1.14 RPC Design Choice Summary

The choice encompasses four aspects: (1) binding (2) IDL (3) pointer argument (4) partial failures.

4.1.15 What is SunRPC

The SunRPC made the following design choices

binding: per-machine registry daemon, as it assumes that the server machine is known up front

IDL: language-agnostic XDR for both interface specification and message encoding

pointer argument: allowed and serialized

failure: internal retry mechanism, return as much information as possible

4.1.16 SunRPC Overview

In SunRPC the client and the server interact via a procedure call interface. The server specifies the interface that it supports in an .x file written in XDR. A compiler called *rpcgen* will compile the .x file to language specific stub. It will generate separate stubs for the client and the server. The server process when launched will register itself with their registry daemon that is available on the local machine i.e. per-machine registry. When the binding happens the client creates an RPC handle, which is used whenever the client makes any RPC, and the RPC runtime uses the handle to track per-client RPC state.

4.1.17 SunRPC XDR Example

The .x file describes data types

```
struct square_in{          struct square_out{
    int arg;                   int res;
};                         };
```

and procedure (name, version etc.)

```
program SQUARE_PROG{                          // service name (client use)
    version SQUARE_VERS{
        square_out SQUARE_PROC(square_in) = 1;  // procedure id (server use)
    } = 1;                                    // version number (client & server use)
} = 0x31230000;                               // service id (server use)
```

Note that it is possible for a single server to support multiple versions of the same procedure.

4.1.18 Compiling XDR

rpcgen -C square.x produces

- square.h contains the language specific data types and function definitions
- square_svc.c contains the server stub and the skeleton of the server i.e. the main() function
- square_clnt.c contains the client stub
- square_xdr.c contains the common code of marshalling and unmarshalling routines

In particular

square_svc.c contains

- the main() function of the server, for registration and housekeeping cooperations
- square_prog_1, internal code, request parsing, argument marshalling
- square_proc_1_svc, actual procedure to be implemented by the developer

square_clnt.c contains

- squareproc_1, wrapper for the RPC call to square_proc_1_svc
- y = squareproc_1(&x), invoked by the developer to execute the RPC call

4.1.19 Summarizing XDR Compilation

The developer is responsible for

- implementing square_proc_1_svc on the server side
- calling squareproc_1 on the client side
- including the .h header files to link with stub objects

The RPC runtime is responsible for

- interacting with OS
- managing communication (socket, protocol etc.)

Note that the following is not thread safe

```
rpcgen -C square.x
y = squareproc_1(&x, client_handle)
```

The thread safe version is

```
rpcgen -C -M square.x
status = squareproc_1(&x, &y, client_handle)
```

However it does not automatically create a multithreaded server. On Solaris platform provides an -a option to generate multithreaded server code, but on Linux this has to be implemented manually.

4.1.20 SunRPC Registry

In SunRPC the registry daemon is a process that runs on every single machine, and is called *portmapper*. To start this process in Linux we need to *sudo ./sbin/portmap*. This process has to be used by both the server (to register a service) and the client (to find the server). Once the daemon is running we can check what are the services that are registered with it using *./usr/sbin/rpcinfo -p*. Note that the port mapper service is registered with both TCP and UDP on the same port number 111.

4.1.21 SunRPC Binding

The binding process is initiated by the client using the following commands

```
CLIENT *clnt_handle;
clnt_handle = clnt_create(rpc_host_name, SQUARE_PROG, SQUARE_VERS, “tcp”);
```

where SQUARE_PROG and SQUARE_VERS are auto-generated in the compilation of the .x file, and will be included in the .h file as hash defined values. It means that if the client wants to use a different version of the procedure, then re-compilation is required.

4.1.22 XDR Data Types

Besides the default types that are common to other programming languages such as char, byte, int, float, XDR supports additional data types

const: equivalent to hash defined value in C

hyper: 64-bit integer

quadruple: 128-bit float

opaque: uninterpretable binary data such as those of image, similar to byte in C

XDR allows two types of array specifications

fixed-length array: e.g. int data[90]

variable-length array: e.g. int data⟨90⟩ where 90 is the maximum, when compiled it translates into a data structure that contains a length integer field and a value pointer field

Note that variable-length specification is used for all strings, although they are stored as normal null-terminated strings in memory.

4.1.23 XDR Data Types Quiz

The number of bytes needed to represent a fully occupied int data ⟨5⟩ on a 32-bit machine is 4 (for length = 5) + 4 (for data pointer) + 4 × 5 (for the 5 integers) = 28 bytes.

4.1.24 XDR Routines

- marshalling and unmarshalling routines, found in square_xdr.c
- xdr_free, to free up the memory used by the RPC, called by square_prog_1_freeresult after results are returned

4.1.25 Encoding

The buffer/message contains

transport header: protocol, destination address

RPC header: procedure id, version number, request id (for retry)

data: arguments or results, which need to be serialized into a byte stream, sometimes there may be a 1-to-1 correspondence between the in-memory representation and the encoded representation but it is not necessary

4.1.26 XDR Encoding

Roughly speaking XDR = IDL + message encoding. XDR encoding rules are

- all data types are encoded in multiples of 4 bytes (padding if not)
- big endian is the transmission standard
- 2's complement is used for integers
- IEEE format is used for floating point

For example for string data ⟨10⟩ = "Hello", 4 (for length = 5) + 5 (1 byte for each of the 5 characters, note that the null terminator is not to be transmitted) + 3 (for padding to a multiple of 4) = 12 bytes are needed to encode the data.

4.1.27 XDR Encoding Quiz

For a fully occupied int data $\langle 5 \rangle$, 4 (for length $= 5$) $+ 4 \times 5$ (for the 5 integers) $= 24$ bytes are needed to encode the data.

4.1.28 Java RMI

RMI stands for remote method invocations. It is among address spaces in JVM and matches Java object-oriented semantics. Java is an OOP language in which objects interact via method invocations rather than procedure calls, and RMI is especially designed for such remote interaction. Thus it is natural that RMI is language specific. It also has client stub and server stub, the latter referred to as a skeleton.
RMI runtime is separated into two components

remote reference layer: support different reference semantics (e.g. unicast, broadcast + return-first, broadcast + return-if-all-match)

transport layer: implement transport protocol (TCP, UDP, shared memory)

4.2 Lecture 2 Distributed File Systems

4.2.1 Visual Metaphor

access via well-defined interface: access via virtual file system, which allows the OS to take advantage of multiple storage types

maintain consistent state of the shared files: tracking state, file updates, cache coherence etc.

mixed distribution model: replicated vs. partitioned, peer-like systems etc.

4.2.2 Distributed File Systems

Modern OS supports different file systems by exporting a virtual file system interface. Underneath this interface it can also hide the fact that the files are maintained on a remote machine that is being accessed over the network.

4.2.3 DFS Models

There are

- the client and the server are on different machines, the focus of this lesson
- the file server is distributed on multiple machines, which may be

  replicated: each server maintains all files, it helps in the event of server failure

  partitioned: each server maintains part of the files, this makes the system more scalable than the replicated model

  both: files partitioned + each partition replicated, big data companies like Google and Facebook use this
- files stored on and served from all machines, which blurs the distinction between client and server because all the nodes are responsible for both maintaining the shared files and providing the file system services

Note that there are a lot of similarities between distributed file management and distributed memory management (next lecture).

4.2.4 Remote File Service: Extremes

upload/download model: when the client wants to access a file, it downloads the entire file, performs the operations locally, and uploads the file back to the server, this is more similar to the FTP server or SVN server

pro: local reads/writes are fast

con: the client needs to download and upload the entire file even for small modifications + server gives up access control

true remote file access model: all reads/writes are done by the server

pro: the server retains full control of file access, and has full knowledge of client modifications which make it easy to maintain file system consistency

con: every file operation incurs the cost of remote network latency even when clients are only reading read-only files + since every single file operation goes to the server the server is easily overloaded which limits server scalability

4.2.5 Remote File Service: A Compromise

A more practical remote file access (with caching) will

- allow clients to store parts of the files locally (download and/or prefetch)

  pro: low latency on file operations + server load is reduced making it more scalable

- force clients to interact with server more frequently, because clients need to notify server any modifications they made and be informed any modifications other clients made

  pro: server has insights into what clients are doing and control over which access is permitted, making it easier to maintain consistency

  con: server becomes more complex as it has to perform additional tasks and maintain additional states for maintaining consistency + clients have to understand different file sharing semantics

4.2.6 Stateless vs. Stateful File Server

stateless server: server keeps no information regarding which client accesses which file, every client request has to be self-contained, suitable for upload/download model or true remote file access model

pro: no resource is needed to maintain states + just need restart upon server failure

con: cannot support caching because we cannot achieve consistency management without state information + more data need to be transferred because every request must be self-contained

stateful server: server keeps client state information, suitable for the more practical model that tracks what is cached/accessed

pro: can support locking, caching, and incremental operations

con: all states need to be recovered upon server failure which requires checkpoints and recovery mechanisms + overhead for maintaining state information and state consistency determined by caching mechanism and consistency protocol

4.2.7 Caching State in a DFS

Keeping the cached portions of the file consistent with the on-server representation of that file requires some coherence mechanism. The problem is similar to maintaining cache coherence in shared memory multiprocessors, where write-update or write-invalidate is used upon file update. In DFS coherence mechanism may be triggered on demand, periodically (server-driven), or whenever the client opens the file (client-driven).

4.2.8 File Sharing Quiz

In a DFS files can be cached in (1) client memory (2) client storage device (3) buffer cache in memory on server—may not be very useful if there is high request interleaving from many clients, because there may not be much locality among the accesses.

4.2.9 File Sharing Semantics on a DFS

Given the fact that message latency may vary, there is no way of determining how much delay for read operation is sufficient to ensure reading the most recent file version. Thus a DFS typically sacrifices some consistency and accepts some relaxed file sharing semantics. In contrast to Unix semantics where every write is immediately visible, a DFS employs

session semantics: whenever a file is closed, client writes back to server all the changes it has applied to the file in its cache, whenever client needs to open a file, it checks with server whether there is a more recent version, the period between file open and file close is referred to as one session, this semantics is insufficient for situations in which clients concurrently write to a file, and can result in long period of inconsistency

periodic update: clients write back periodically as if they have a lease on cached data albeit exclusive writing is not necessary + server invalidates periodically to provide time bounds on inconsistency, this is augmented with flush and sync API as clients do not know about the start and end of synchronization periods

immutable files: files are deleted and created with a new name instead of being modified (e.g. photos)

transactional guarantees: export API for clients to specify the collection of files or the collection of operations that need to be treated like a single transaction and be atomically committed

4.2.10 DFS Data Structure Quiz

For a DFS where file sharing is implemented via a server-driven mechanism (periodic invalidation) and with session semantics (change visible upon file close), server per file data structure should include (1) current readers (2) current writers (3) version number—because it is possible for overlapping write sessions to see different versions of the file.

4.2.11 File vs. Directory Service

File systems have two different types of files which differ significantly in terms of access pattern, locality, and lifetime, and hence merits different file semantics

regular files: session semantics, or periodic update with less frequent write-back

directories: Unix semantics, or periodic update with more frequent write-back

4.2.12 Replication and Partitioning

replication: each machine holds all the files

> **pro:** easy load balancing across replicas, better server availability, more fault tolerant
>
> **con:** more complex writes (either synchronously write to all replicas or write to one and propagate to others, and reconciliation such as voting is necessary for any discrepancy among replicas)

partitioning: each machine holds a subset of all the files

> **pro:** better server availability, greater scalability, simple writes
>
> **con:** data loss upon server failure, load balancing is more difficult and if not balanced hot spot is possible

These two can be combined i.e. replicating partitions, and various optimization is possible such as replicating read-only files to a greater extent and smaller partitions and more replicas for files accessed more often.

4.2.13 Networking File System (NFS) Design

In NFS clients access remote server over a network hence the name NFS, which is one motivation for developing RPC. It consists of

client: requests a file using the same type of file descriptor as that of local files and virtual file system interface, which determines whether the file is local or needs to be pushed to the NFS client, the NFS client interacts with the NFS server via RPC

server: the NFS server accepts the request, forms it into a local file system operation, and then issues it to local virtual file system interface, which is serviced as local file requests, upon receiving a file open request the NFS server creates a file handle and returns it to the NFS client, which will be passed with every subsequent request

4.2.14 NFS File Handle Quiz

File handle becoming stale means the file on the remote server has been removed. For the other cases

- the file is outdated only implies consistency issue
- the remote server is not responding only implies an RPC layer error
- that the file has been open for too long will not result in a stale file handle, as NFS allows clients to keep file open for an indefinite period of time

4.2.15 NFS Versions

Current versions are stateless NFS3 and stateful NFS4. Note that although stateless NFS3 includes extra module that supports file caching and logging.
The caching semantics in NFS are as follows

session-based: for files that are not accessed concurrently, i.e. on close changes are flushed to remote server and on open a check is performed and the cached portion is updated if needed

periodic update: break the session semantics when there are multiple clients that are concurrently updating, default period = 3 seconds for regular files vs. 30 seconds for directories

NFS4 further incorporates a delegation to clients all the rights to manage a file for a period of time to avoid update checks
With server side state NFS can support locking using a lease-based mechanism—server assigns a requesting client a particular time period during which the lock is valid. This helps to deal with client failure via lease expiration. NFS4 also supports a reader/writer lock called share reservation.

4.2.16 NFS Cache Consistency Quiz

Immutable is not supported because NFS allows files to be modified. Unix is not supported because NFS as a distributed file system cannot guarantee that file update is immediately visible. Given the option for configuring and disabling periodic updates, NFS is neither purely session based nor purely periodic update based.

4.2.17 Sprite Distributed File System

Although it is a research DFS rather than a production DFS, it is valuable because it used trace data on file access patterns to analyze DFS design requirements and justify DFS design decisions.

4.2.18 Sprite DFS Access Pattern Analysis

The study on how are files accessed in the file system used by the author's department shows that

- 33% of all file accesses are writes ⇒ caching can be important to improving performance, however write-through is not sufficient
- 75% of files are open less than 0.5 second and 90% of files are open less than 10 seconds ⇒ with session semantics periodic updates within 0.5 second is necessary for many files and almost all within 10 seconds, but then the overhead of session semantics would be too high
- 20–30% of new data are deleted within 30 seconds and 50% of new data are deleted within 5 minutes ⇒ write-back on close is unnecessary
- file sharing is rare ⇒ no need to optimize for concurrent access

4.2.19 Sprite DFS from Analysis to Design

Based on the above workload analysis Sprite DFS made the following design decisions

- supports caching with write-back every 30 seconds for files that have not been modified for the last 30 seconds
- when a client opens a file that is currently being written by another client, the server will contact this writer client and collect all the outstanding dirty blocks, with this policy Sprite allows a file to be open, modified, and closed multiple times before any data is written back to the server
- every open operation goes to the server, directories are not cached

In summary Sprite distinguishes between two situations

sequential write sharing: caching and sequential semantics

concurrent write sharing: no caching

Thus one unique feature of Sprite is that, it dynamically enables and disables caching depending on whether it is sequential write or concurrent write.

4.2.20 File Access Operations in Sprite

Assume R_1, R_2, …, R_n readers and W_1 writer, W_1 keeps time stamps for each modified blocks for the 30-second periodic update semantics, and checks with server whether its cached value is the same thus version number is needed. To support these operations

client: needs to track (1) cache presence (2) cached blocks (3) time for each dirty block (4) file version

server: needs to track (1) current readers (2) current writer (3) file version (4) file cachable

sequential sharing: assume a sequential writer W_2 shows up after W_1 has closed the file, the server will contact W_1 to gather all dirty blocks and update the new file version, then W_2 can proceed to cache the file

concurrent sharing: assume a concurrent writer W_3 shows up while W_2 is still writing to the file, the server will again contact the last writer W_2 to gather all dirty blocks, but since W_2 hasn't closed the file, it will write back all the dirty blocks and the server will disable caching the file for all clients, both W_2 and W_3 will continue to have access to the file, except that they will not be able to use their caches and all file accesses must go to the server, when either W_2 or W_3 closed the file, the server will change the file to become cachable

4.3 Lecture 3 Distributed Shared Memory

4.3.1 Visual Metaphor

placement: place memory (pages) close to relevant processes

migration: when to copy memory (pages) from remote to local

sharing rule: ensure memory operations are properly ordered

4.3.2 Reviewing DFSs

A distributed file system is an example of a distributed service, in which

client: send request of file access

caching: improve performance (client) and scalability (server)

server: own and manage state (files), provide service (file access)

In this lesson we will discuss the scenarios in which (1) there are multiple file servers thus shared file states (2) all nodes provide both file service and send file access request thus there isn't clear distinction between client and server.

4.3.3 Peer Distributed Applications

In a "peer" distributed application, the state of the application is distributed and stored across all nodes, thus each node owns some portion of the state, provides access to this locally stored state, and at the same time requests access to state that is stored in other nodes. The quotation mark for peer is due to the fact that some designated nodes will perform overall management and configuration for the entire system, thus control is centralized rather than evenly distributed among all nodes. In a truly peer-to-peer system even system control and management are done by all nodes.

4.3.4 Distributed Shared Memory (DSM)

DSM is a service that manages the memory across multiple nodes. Each node owns some portion of the physical memory and provides memory reads from/writes to other nodes together with consistency protocols (e.g. reads and writes must be ordered in a meaningful way so that they appear as reading/writing on a shared memory).
DSM permits scaling beyond single machine memory limit at low cost. Although this leads to slower memory access, there are ways to hide this latency, some of which will be discussed later. DSM is becoming more prevalent today also because commodity interconnect technologies now offer low-latency remote direct memory access (RDMA).

4.3.5 Hardware vs. Software DSM

DSM can be supported in either hardware or software

hardware: relies on high-end interconnect, the OS is allowed to establish virtual-to-physical memory mappings that correspond to memory locations in other nodes, the memory access that references memory locations in other nodes is passed to a network interconnect card (NIC) which knows how to translate the memory operation, these are high-end NICs which are involved in all aspects of memory management and support atomic operations
very expensive and used only for supercomputing platforms

software: distinguishes remote from local access and creates and sends messages to appropriate nodes, can be done at the level of OS or language runtime

4.3.6 Implementing DSM Quiz

A common task that is implemented in both software and hybrid (hardware + software) DSM implementations is prefetching pages. Address translation and triggering invalidation are more concretely defined and easier to implement in hardware.

4.3.7 DSM Design: Sharing Granularity

In SMP (shared memory multiprocessor) systems the granularity of sharing is a cache line, which is too granular and thus too costly for DSM to write a message for. Instead DSM looks at larger granularity such as

variable granularity: still too fine grained

page granularity: the only choice for OS-level DSM because OS doesn't understand objects, since page size is large it is possible to amortize the cost of remote access

object granularity: with the help of compiler application-level objects can be shared, the benefit is that the OS does not need to be modified

A problem with increased granularity is false sharing, e.g. two variables are stored in the same page or object, which will trigger unnecessary coherence operation.

4.3.8 DSM Design: Access Algorithm

single reader/single writer (SRSW): the main role of DSM is to provide additional memory

multiple readers/multiple writers (MRMW): consistency is a challenge, the focus of this lesson

4.3.9 DSM Design: Migration vs. Replication

Obviously the performance metric for DSM is its memory access latency. Since accessing local memory is faster than accessing remote memory, we should maximize the number of local memory access. There are two ways to achieve it

migration: copy the state to the requesting node, which makes sense for SRSW but the cost of data movement is not amortizable

replication: more general, similar to caching (close to vs. at the requesting node), but requires consistency management because the state will be accessed concurrently on multiple nodes, the overhead of which is proportional with the number of replicas

4.3.10 DSM Performance Quiz

For MRMW migration can increase rather than reduce access latency

4.3.11 DSM Design: Consistency Management

In SMP systems consistency is managed using two mechanisms—write-invalidate and write-update. The coherence operations are triggered by the shared memory support in the hardware on every single write, which will lead to too high overhead for DSM.
There are two options of coherence operation for DSM

proactive/eager/pessimistic: push invalidation messages when data is written to, similar to the server-based approach for DFS

reactive/lazy/optimistic: pull modifications periodically or on-demand

When these methods get triggered depends on the consistency model for the shared state.

4.3.12 DSM Architecture

A page-based OS-supported architecture consists of nodes each contributing all or a portion of its memory. All the pages contributed by the nodes form a memory pool, in which each memory address is uniquely identified with the node ID it belongs to and the page frame number it resides. The node where a page is located is referred to as the home node of that page.
For MRMW local caches are needed to achieve low-latency performance. Home node or manager node is responsible for driving coherence operations. In this way all nodes are responsible for part of the distributed memory management. The information maintained by home node is similar to that of DFS, which includes the page accessed, modifications, caching enabled/disabled, locked etc. Note that to optimize frequent access of a page, home node may not be the current owner node of the page.

home node: track which is the current owner node, only one

owner node: control state updates, drive consistency operations, multiple (whichever that is accessing the page)

Explicit page replication may be used for load balancing, hot spot avoidance, or reliability. In this case the replica consistency is controlled by either home node or designated manager node.

4.3.13 Indexing Distributed State

Each page (or object) has an address consisting of node ID (which identifies its home node) and page frame number. A global manager map that allows the determination of manager node for each page is replicated on each node, whereas per page metadata is partitioned and distributed across all manager nodes. For every page there is a fixed manager node. For additional flexibility we can take object identifier as an index into a mapping table, the position of which in the table identifies the associated manager node, so that only relocation in the table is required for any change of manager nodes leaving object identifiers intact.

4.3.14 Implementing DSM

The overhead resulted from DSM having to intercept all accesses to DSM states so as to determine whether to send request for remote access or whether to trigger coherence operation should be avoided for local non-shared state access. This requires dynamically engaging or disengaging DSM depending on the nature of access, which can be achieved by leveraging the hardware support from memory management unit (MMU). In particular

remote address mapping: trap in OS since mapping is invalid and pass page information to DSM to send remote request

cached content: trap in OS since access is not permitted (write protected) and pass modification to DSM to trigger coherence operation

Other information maintained by MMU such as dirty pages may also be useful to DSM.
For object-based DSM implemented at language runtime, similar mechanism is available with the aid of compiler.

4.3.15 What is a Consistency Model

How a DSM should be designed and how the coherence operation should be triggered depends on the consistency model. A consistency model is a guarantee that memory (state) behaves correctly as long as upper software layer follows specific rules. Correct behavior encompasses (1) accesses are ordered (2) updates will be visible and propagated to relevant parties. Rule following means either using specific API to request access or write updates, or setting certain expectations based on the guarantee (e.g. no lock no non-race guarantee).

4.3.16 Strict Consistency

Strict consistency requires (1) immediate visibility of updates to everybody (2) all writes observed in the same order as they occur by everybody. Strict consistency cannot be achieved in SMP systems without use of synchronization constructs. In DSM latency and message reorder/loss make strict consistency impossible. For this reason strict consistency remains a nice theoretical model, but in practice other consistency models are used instead.

4.3.17 Sequential Consistency

In sequential consistency model

- memory updates from different processors may be arbitrarily interleaved as long as it is equivalent to some possible outcome of sequential execution of SMP systems without lock, however all processors should observe the same interleaving
- updates from the same processors always appear in the order they were issued

4.3.18 Causal Consistency

In causal consistency model

- causally related writes should be observed in the order they were issued, for all other concurrent writes the requirement that all processors should observe the same interleaving is relaxed
- same as sequential consistency, updates from the same processors always appear in the order they were issued

4.3.19 Weak Consistency

In addition to read/write, in weak consistency model synchronization points are inserted to distinguish causally related writes from other writes. All updates prior to a sync point will become visible to other processors. But other processors are not guaranteed to observe the updates, because the sync point has to be called by both the processor that performed the update and the processor that wants to observe the update. Variants of single synchronization operation are

- separate sync points are associated with different subsets of states
- separate sync points into entry/acquire (want to observe) vs. exit/release operations (make visible)

The motivation of the variants is to limit data movement and coherence operation. But extra states are required to track the additional operations.

4.4 Lecture 4 Datacenter Technologies

4.4.1 Internet Services

An internet service is any service that is accessible via web interface. Most commonly internet services are composed of

presentation: static content that interfaces with end users

business logic: dynamic content that is user specific

database tier: data storage and management

which are not necessarily separate processes running on separate machines. There are various integrated software or middleware that provide commonly used functionality. For services that are organized as multiple processes, the inter-process communication between those processes is carried out via some form of IPC that were covered in previous lessons.

4.4.2 Internet Service Architectures

For services that need to deal with highly variable request rates, choosing the multi-process multi-node configuration becomes necessary, because it is the easiest architecture to deal with scaling. In fact it is referred to as scale out architecture. A boss-worker pattern can be used for this architecture, in which a front-end dispatches requests to available nodes. The workers have the following two variants

functionally homogeneous/general-purpose: all nodes execute all steps in request processing for any request

functionally heterogeneous/specialized: some nodes execute some specific steps in request processing for some request types

4.4.3 Homogeneous Architectures

The pro and con are

pro: keeps front-end simple, as it doesn't need to track which node is for which step

con: little opportunity to benefit from caching

Note that this architecture doesn't require each node has all data, just each node has access to all data.

4.4.4 Heterogeneous Architectures

For this architecture it makes sense that data is not uniformly accessible. The pro and con are

pro: benefit of specialization, locality, and caching

con: more complex front-end, more complex management for scaling and avoiding hot spot

4.4.5 Cloud Computing Requirements

Traditional approach is to determine the capacity based on expected peak demand, and then buy and configure resources to provide that capacity. Inaccurate demand forecast will lead to lost opportunity and dropped requests. Ideally

- capacity should scale elastically with demand
- scaling should be instantaneous, both up and down
- cost should be proportional to demand and revenue

All of the above should happen automatically without hacking wizardry and should be doable anytime anywhere. The goal of cloud computing is to provide

- on-demand elastic resources and services
- fine-grained pricing based on usage
- professionally managed and hosted
- API-based access

4.4.6 Cloud Computing Overview

Cloud computing provides

shared resources: infrastructure (virtual machine rather than physical) and software/services (e.g. database and email), it may be easier to rent the infrastructure along with the properly configured software

API: for access and configuration, including web-based, libraries, command line interface etc.

billing services: due to the overhead billing is typically not done on actual usage, but done in some discrete step function

All these are managed by cloud provider via some software stack such as VMware vSphere.

4.4.7 Why Does Cloud Computing Work

Two basic principles backing cloud computing are

law of large number: demand averaged across many customers is roughly constant

economies of scale: unit cost of resources drops at large scale

4.4.8 Cloud Computing Vision

The oldest description of cloud computing was proposed by John McCarthy in 1961, who advocated that computing should become a fungible public utility. Nonetheless limitation exists to prevent this from becoming true such as hardware dependence, API lock-in, privacy, security etc.

4.4.9 Cloud Deployment Models

public: provided to third party customers/tenants for a fee

private: leverage technology internally

hybrid: private cloud interfaced with public cloud for fail over, redundancy, dealing with spikes, workload simulation etc.

community: public but used by certain type of users

4.4.10 Cloud Service Models

The second way clouds are commonly differentiated is based on the service model they provide

on-premises: any standalone software

- **you manage:** applications, data, runtime, middleware, OS, virtualization, servers, storage, networking
- **other manage:** nil

infrastructure as a service (IaaS): e.g. Amazon EC2

- **you manage:** applications, data, runtime, middleware, OS
- **other manage:** virtualization, servers, storage, networking

platform as a service (PaaS): e.g. Google app engine (Android dev platform)

- **you manage:** applications, data
- **other manage:** runtime, middleware, OS, virtualization, servers, storage, networking

software as a service (SaaS): e.g. Google gmail

- **you manage:** nil
- **other manage:** applications, data, runtime, middleware, OS, virtualization, servers, storage, networking

4.4.11 Requirements for the Cloud

- fungible resources that can easily be repurposed to support different customers and hide resource heterogeneity
- elastic dynamic resource allocation methods
- management at scale, scalable resource allocations for customers
- deal with failures
- guarantee performance and isolation among multiple tenants
- security and tenant privacy

4.4.12 Cloud Failure Probability Quiz

Given a p failure probability of each component, for an n-component cloud the probability of a failure somewhere in the cloud is

$$\text{failure probability} = 1 - (1 - p)^n$$

4.4.13 Cloud Enabling Technologies

virtualization: to provide fungible resources that can be dynamically repurposed

resource scheduling & provisioning: e.g. Mesos, Yarn

big data storage & processing: processing (Map Reduce, Spark), storage (DFS, NoSQL, distributed in-memory caches)

software-defined resource: customers are provided with their own software-defined data centers, storage, network etc.

monitoring: real-time log processing (Flume, CloudWatch, LogInsight)

4.4.14 The Cloud as A Big Data Engine

The layers that is needed to provide such a big data infrastructure are

- data storage layer
- data processing layer
- caching layer
- language-specific front-end (for queries)
- analytic libraries (e.g. machine learning)
- continuous streaming data

4.4.15 Example Big Data Stacks

The two most popular big data stacks are

Hadoop: Hadoop distributed file system or HDFS (storage), Yarn (schedule), Map Reduce (processing), Hbase (data table), Hive (query), R connectors (statistics), Mahout (machine learning), Pig (scripting), Oozie (workflow), Flume (log collector), Zookeeper (coordination)

Berkeley Data Analytics Stack (BDAS): HDFS (storage), Tachyon (memory), Mesos (schedule), Map Reduce/Spark (processing), Shark (SQL API), Hive (query), MLBase (machine learning), Storm (log collector), Spark Streaming (streaming)