CS 111: Midterm Review

Winter 2017, Prof. Reiher

Introduction

Lecture 1: Introduction

What is an Operating System (OS)

- System software that supports higher level application
- Sits between hardware and everything else
  - **RULE: we do not interact directly with hardware — instead, ask OS for help**
- Hides complexity
- Virtualization of hardware and software

How to Work with OSes

- Configure them
- Use the features
- Rely on *services*
  - Memory management
  - Persistent storage
  - Scheduling and synchronization
  - Interprocess communication
  - Security

OS Wisdom

- Services viewed as objects and operations
- Interface vs. Implementation
  - Implementation: actual implementation of interface not a specification
    - Many implementations possible for an interface
    - Do not rely on this as a specification - may cause dependencies that will be problematic when implementations change
  - Interface: specification describing how something works
    - Contract between producers and consumers
    - Ensure that interface is accurate regardless of implementation
- Modularity and Functional Encapsulation
  - Hide complexity and abstract appropriately
  - Module + interface → easy to build large complex pieces of software
  - Separate pieces of functionality build as independent modules instead
- Separate policy and mechanism
  - Policy: what can/should be done
    - Must be changeable
  - Mechanism: implements basic operations (should implement multiple policies)
    - Should not dictate policies
- Parallelism and asynchrony and powerful and vital — but also dangerous

- Performance and correctness are at odds
    - Sometimes performance is favored over correctness

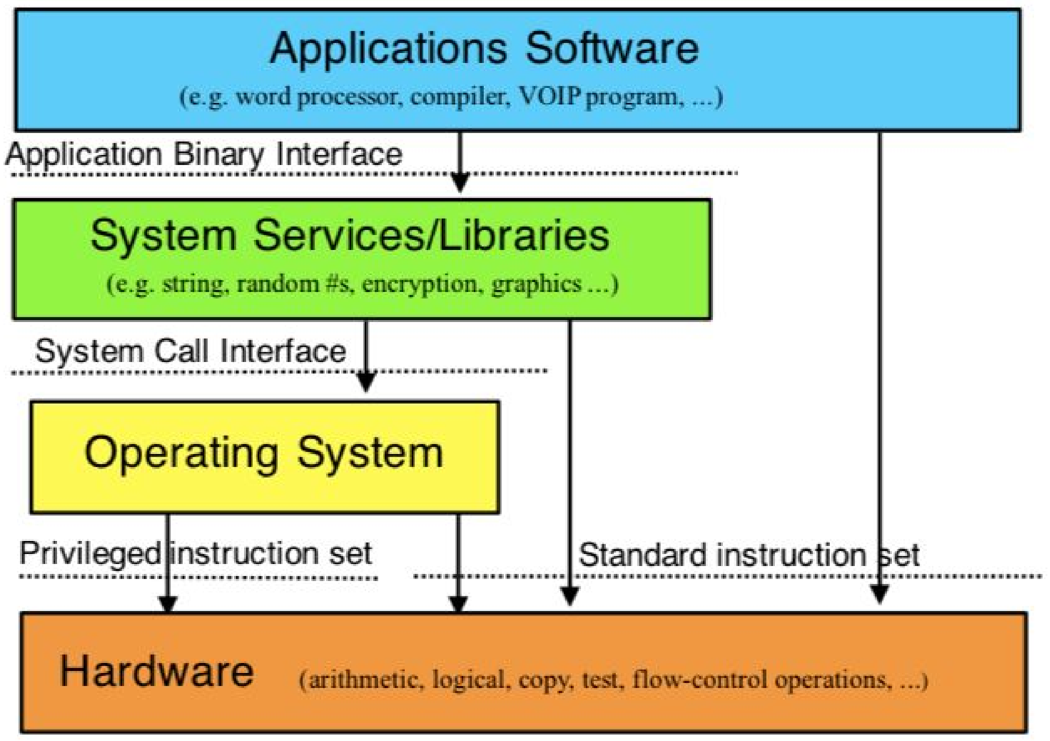
What an OS Does

- Manages hardware for programs
- Abstracts hardware
    - Hardware errors → OS is the first line of defense
- Provides abstractions for applications

What does an OS Look Like?

- Set of management and abstraction services
- Applications see objects and their services
- OS extends a computer
    - Creates rich virtual computing platform
    - Gives you more power than you appear to have

Where Does the OS Fit In?


Applications Software
(e.g. word processor, compiler, VOIP program, ...)
Application Binary Interface
System Services/Libraries
(e.g. string, random #s, encryption, graphics ...)
System Call Interface
Operating System
Privileged instruction set
Standard instruction set
Hardware
(arithmetic, logical, copy, test, flow-control operations, ...)

- **Application - (what you want to run in)**
    - Accesses hardware in standard instruction set
    - More efficient, accesses certain set of instructions without OS help
- ***Application Binary Interface (some performance hit, ex: subroutine calls)***
    - Binary makes calls to system calls
    - Links up to other interfaces/computers
    - Allows you to have portability
- **System Services/Libraries Provides some libraries, not necessarily privileged**

        - Unprivileged instructions accessing the hardware in standard instruction set
    - *System Call Interface (expensive)*
        - Switches to **kernel mode**, invoking the OS because it needs special commands/instructions
    - Operating System
        - Invoked by system call
        - Needs to use privileged instructions
    - *Privileged Instruction Set*
        - Cannot be used by system service library or application software
        - Only used by OS

What's Special About the OS?

- Always controlling the hardware
    - Automatically loaded by **boot loader** when machine boots
    - First software to have access to hardware
    - Continues running constantly
- Has **complete access** to hardware alone
- Mediates applications' access to OS
    - Bypassing the OS means that the OS cannot manage resources for yo
- Trusted
- If it crashes → everything else will crash along with it

Instruction Set Architectures (ISAs)

- Set of instructions supported by computer
- Incompatible among different machines
- Usually upwards-compatible

Privileged vs. General Instructions

- ISAs divide instruction set between privileged and general instructions
- Any code can run general instructions
- Processor must be put in **special mode** to execute privileged instructions
    - Typically "dangerous" things require privilege

Platforms

- ISA doesn't define computer
    - Functionality beyond user mode instructions
    - I/O devices
- Variations - **platforms** on which the OS must run
    - ISA + functionality + I/O = platform

Portability to Multiple ISAs

- Successful OS must run on many ISAs → need to abstract the ISA
- Minimal assumptions about hardware
- How can we do this?

Binary Distribution Model

- Binary - derivative of source code
    - OS written in binary

    - Source distribution must be compiled — binary distribution ready to run
- OS - distributed in binary
- One binary/ISA

Binary Configuration Model

1. Binary figures out which platform you have
2. Automatically re-configures itself

- Eliminate manual/static configuration
- Automatic hardware discover
- Automatic resource allocation
- **Key to software portability and software industry today**

Interface Stability

- OS sometimes breaks old applications - how do we prevent this?
- Well specified application interfaces
    1. Application Programing Interfaces (APIs)
    2. Application Binary Interfaces (ABIs)

Functionality in the OS

- As much as necessary yet as little as possible
- Functionality required if it:
    - Requires privileged instructions
    - Manipulates OS data structures
    - Must maintain security, trust, and resource integrity
- Functions in libraries are for:
    - Uncommonly used services
    - Do not actually need to be in OS
- BUT some things are faster in the OS

The OS and Abstraction

- OS - abstracts resources instead of using physical resources
- Implements abstract resources using physical resources

Why Abstract Resources?

- Simpler and better suited for programmers and users
    - Primarily helping programmer
    - User rarely interacts with OS

1. **Easier to use than original**
2. **Compartmentalize/encapsulate the complexity**
3. **Eliminate irrelevant behavior**
4. **Create more convenient behavior**

Generalizing Abstractions

- Allows us to use a high level abstraction to work with any hardware/software
- Common unifying model (ex: PDF)
- Federation Framework
    - Translate between top & low-level functionality

Types of OS Resources

1. Serially Reusable Resources

2. Partitionable Resources
3. Sharable Resources

1. Serially Reusable Resources

- Multiple clients, but one at a time
- Processes use one at a time but can switch between resources
- Require **access control** (other process should not be able to use it even if they had previously or will in the future)
    - Ensure exclusive use
- Require **graceful transitions** from one user to the next
    - Switching resources from one to another means that the resource is **reset**
    - We want a **CLEAN TRANSITION**

Graceful Transition?

- Switch that hides the fact that the resource was previously used
- Incurs a performance cost

2. Partitionable Resources

- Divide into pieces for multiple clients
    - Spatial multiplexing
- Need access control to ensure:
    - Containment: you cannot access resources outside of your share
    - Privacy: nobody else can access your resources

Graceful Transitions?

- Yes
- Partitionable resources are not permanently allocated → when they are taken away, need a graceful transition

Shareable Resources

- Usable by multiple concurrent clients
    - No dividing
    - No transitions
- Limitless resources perhaps
- Under-the-covers multiplexing
- Typically **read-only** resources
    - Writable resources incur synchronization problems

Graceful Transitions?

- Typically no
- Isn't changing its state or being reused

Reading

Arpaci-Dusseau Ch. 2 - Introduction to Operating Systems

- What happens when a program runs? → it executes instructions

1. Processor fetches instruction
2. Decodes it
3. Executes it

- OS (Virtual Machine) responsible for
    - Making it easy to run programs
    - Allowing programs to share memory
    - Enabling programs to interact with devices
- Achieves this using **virtualization** - taking physical resource and transforming it to a virtual form
- Allows users to use OS to:
    - System calls
    - Standard library of applications
- OS: resource manager

→ 2.1 Virtualizing the CPU

- CPU: system with a single processor
- Virtualizing the CPU means: turning a single CPU into a **seemingly infinite number of CPUs**
- **Policy of the OS:** determines which program should run at a particular time

→ 2.2 Virtualizing the Memory

- Model of physical memory:
    - Memory as an **array of bytes**
    - Read memory/write memory by **specifying the address and/or the data**
- Virtualizing memory: each running program shares the same physical memory as other programs yet, each has its **own virtual address space**

→ 2.3 Concurrency

- Appear with OS and multi-threaded programs

→ 2.4 Persistence

- Data can easily be lost → DRAM stores in a volatile manner
- Need to store data persistently
    - Hardware: I/O devices, hard drives
    - Software: file systems
- Share files → used by system calls linked to file systems
- Large groups of data written at once
    - Intricate write protocol used, created by OS

→ 2.5 Design Goals

- OS’s functions:
    - Takes physical resources and virutalizes them
    - Handles concurrency
    - Stores Files Persistently
- Goals:
    - Make system easy to use with abstractions
    - High performance with minimal overhead
    - Provide protection between applications (**isolation**)

    - High degree of reliability

Kampe: Operating System Principles

1. Introduction

- Issues with size and complexity increase
    - Sophisticated software → more code
    - Increased complexity → more work & more difficult to understand
    - Systems assembled from independently developed and delivered pieces → emergent behavior
    - Testing is difficult
    - Severe failure
- OSes - large and complex
    - Interact between subsystems
    - Asynchronous interactions and externally originated events
    - Resource sharing
    - Coordinated actions among heterogeneous computer
    - Changing requirements
    - Portable and Reliable

2. Complexity Management Principles

→ 2.1 Layered Structure & Hierarchical Decomposition

- Break system down into **components that are easy to understand**
- Hierarchical decomposition - decomposing system into top-down fashion

→ 2.2 Modularity & Functional Encapsulation

- Requirements for each group (module)
    - Coherent purpose
    - Functions entirely within the group
    - Union of groups achieve the purpose
- Look at groups to:
    - Examine responsibilities and role
    - Examine internal structure and operating rules
- Module: possible to understand and use the functions of a component
- Implementations are **encapsulated** within the module
- Characteristics of good modularity (cohesion):
    - Small/simple components
    - Combine closely related functionality
    - Compartmentalize operations

→ 2.3 Appropriately Abstracted Interfaces & Information Hiding

- **Appropriate abstraction** interface that enables a client to specify parameters that are most meaningful
    - Poorly abstracted interface does not do this
    - Does not rely on the underlying implementation (**opaque**) and exhibits good **information hiding**

→ 2.4 Powerful Abstractions
- Profitably applied in many situations
    - Common paradigms: understand wide range of phenomena
    - Common architectures: fundamental models for new solutions
    - Common mechanisms: visualie and construct solutions

→ 2.5 Interface Contracts
- Specification - promise to deliver specific results

3. Architectural Paradigms
- Few broad concepts

→ 3.1 Mechanism/Policy Separation
- Mechanisms to manage resources - shouldn't dictate or constrain policies for those mechanisms
- Resources are divided into two parts:
    - Mechanisms that keep track of resources and give/revoke client access
    - Configurable/Plug-in policy engine that controls to whom and when each resources is given

→ 3.2 Indirection, Federation, & Deferred Binding
- Accommodate multiple implementations with similar functionality
- Use plug-in modules that share features:
    - Common abstraction/class interface
    - Implementation accessed indirectly
    - Indirection achieved using **federation framework**
        - Registers available implementation
        - Enables client to select desired implementation
        - Routes future request to that specific implementation
    - **Binding** of a client → implementation is deferred when necessary
    - **Preferred binding** may be beyond selection @ runtime
        - Implementation module dynamically loaded/discovered

→ 3.3 Dynamic Equilibrium
- Tunable parameters - optimize behavior for specific situation
    - Need **dynamic equilibrium**

→ 3.4 The Criticality of Data Structures
- Data structures determine:
    - Speed of operations
    - Complexity
    - Locking requirements
    - Speed of error recovery

OS Services & Interfaces

Lecture 2: Services, Resources, and Interfaces

Outline
- OS services
- System service layers and mechanisms
- Service interfaces and standards
- Service and interface abstractions

OS Services
- Offers services to other programs as abstractions
- Basic categories:
    - CPU/Memory
    - Persistent Storage
    - Other I/O

Services: Higher Level Abstractions
- Higher level than basic categories
- Cooperating parallel processes
- Security
- User interface

Services: Under the Covers
- Not directly visible to users
- Enclosure management
- Software updates and configuration registry
- Dynamic resource allocation and scheduling
- Networks, protocols and domain services

How Can the OS Deliver These Services?
- Several possible ways
1. Applications can call subroutines
2. Applications can make system calls (asking OS to do something)
3. Applications can send messages to software that performs the services
- Each option - different *layer* of software stack

OS Layering
- Software built on top of other software
- More powerful at the top
- Services - layer of hardware and software
- Everything ultimately mapped down to simple hardware

1. Service Delivery via Subroutines
- Access services with direct subroutine calls
    - Push parameters, jump to subroutine, return values in registers on the stack

- High level
- Advantages:
    - Fast, optimized in hardware
    - Implementation binding possible
- Disadvantages
    - Implemented in same address space → in order to jump to subroutine → if you provide each process an address space, this is a problem
    - Limited ability to combine languages
    - Can't use privileged instructions
        - If process running subroutine has privileged access → everyone has to run in privileged mode
        - OR switch to a system call, no longer just a subroutine

Service Delivery via Libraries

- Programmers don't write code for all programs, can make a subroutine call
- Library: collection of object modules (not standalone code)
    - Always compiled, can be used directly
- Systems come with standard libraries
- Programmers can build libraries

Library Layer

- In the **general libraries** layer
- Usually already compiled, therefore, underneath the ABI layer
- Not yet privileged mode

Characteristics of Libraries

- Advantages:
    - Programming easier
    - Single well written/maintained version
    - Encapsulates complexity — better building blocks
- Multiple bind-time options
    - Static: include in load modules at link time
        - Takes up space in the program
        - Used often
        - Each lod module has its own copy
        - Program needs to be relinked for new libraries
    - Shared: map into address space at execute time
        - Not included in library, instead, always at a certain place in memory (**shared resource**)
            - 1st program that needs it must load it → library is in system as long as somebody is using and is shared among processes
        - EX: libc library
        - Many processes use
        - One in-memory copy, shared by all processes
        - Library - separate from load modules

        - OS loads library along with program
    - Dynamic: choose and load at runtime
        - Add notation in program to these libraries
        - Code only pulled when necessary
        - **Results in a delay**
- No special privileges

<u>Advantages of Shared Libraries</u>

- Reduce memory consumption
- Faster startup
- Simple updates
    - Update copy, delete old version, and reload new version

<u>Limitations of Shared Libraries</u>

- Not all modules will work
    - Modules cannot change state
    - Cannot define/include global data storage
- Added into program memory
    - Regardless of whether or not they are necessary
    - Take room in memory
- Called routines must be known at compile-time
- Dynamically Loadable Libraries - more general
    - Eliminate limitations at a performance cost

<u>2. Service Delivery via System Calls</u>

- Change from user process ➔ more privileged OS process (big switch)
- Code for these services - in OS
- Return values and parameters (similar to subroutine calls)
- Forces an entry into the OS
- Advantages:
    - Able to allocate/use privileged resources
        - Can work on OS data structures themselves
    - Able to share/communicate with other processes
        - Only capable with OS mode because OS knows about all
        - Each process has their own memory space that usually is separate
- Disadvantages
    - All implemented in local mode - only local machine
    - 100x - 1000x slower than subroutine calls **– huge disadvantage**

<u>3. Providing Services via the Kernel</u>

- Functions that require privilege (and need system calls)
    - Privileged instructions
    - Allocations of physical resources
    - Ensuring process privacy and containment
    - Ensuring the integrity of resources
- Some may be out-sourced
    - System call gets to service itself

        - OS does some work but outsources this to another process
        - System daemons (provide services), server processes (created as necessary)
    - Some plugins less trusted
    - Exist in the OS kernel layer

System Services Outside the Kernel

- Not all trusted code in kernel
- Only some part may require permission
    - May not need data structure or privileged instructions
- Some are only **somewhat privileged processes**
    - **Superuser mode**
    - Privilege given by OS
    - May not run privileged instructions but **can do more than regular processes**
- Some are merely trusted
- **OS > SUPERUSER > REGULAR**
- **SUPERUSER =/= PRIVILEGED MODE**
    - Still at application level
    - Cannot run privileged instructions

System Service Layer

- System services (above ABI) and are not in privileged mode

3. Service Delivery via Messages

- Exchange messages with a server (via system calls)
- Somewhere, there is a piece of code capable of performing function
    - Not part of the OS
    - Located somewhere else
- Advantages:
    - Server can be anywhere on earth
    - Highly scalable and available
    - Can be implemented in user-mode code
- Disadvantages
    - 1000-100000x slower than subroutine
        - Requires a system call at the minimum
        - Includes networking cost to deliver and receive messages, causing delays
    - Limited ability to operate on process resources
        - Service provider can only see the message, nothing else

System Services via Middleware

- Extremely powerful
- Software that is part of the application and service platform but **not part of the OS**
- Kernel code - very expensive
    - Usermode code is:

        - Easier to build, test, and debug
        - More portable
        - Can crash and be restarted
- Lies below user and system applications, above ABI

OS Interfaces
- OS meant to support programs
- Usually very general
- Interface required between OS and other programs

Interfaces: APIs (for programmers)
- Application Program Interface
    - Source level interface specifying certain things
- Basis for software portability
    - Recompile for desired architecture
    - Linkage edit
    - Resulting binary runs on architecture and OS
- API compliant program will compile and run on any compliant system

Interfaces: ABIs (for users)
- Add to a compliant system and run — it **should work**
    - Basis of the software industry
- Application Binary Interface
    - Binary interface specifying
        - DLLs
        - Data formats, calling sequences, and linkage conventions
    - Binding of an API to a hardware architecture
- Basis for binary compatibility
- ABI compliant program runs on any compliant system

Libraries and Interfaces
- Normal libraries - accessed through an API
- Dynamically loadable libraries also called through an API
    - Loading mechanism is ABI-specific

Interfaces and Interoperability
- Strong, stable interfaces - key to allowing programs to operate together
- Can’t change interface
- Need to maintain interface so OS upgrades do not break programs

Interoperability Requires Stability
- No program is an island
- If interfaces change, programs fail
- API requirement, **frozen at compile time**
    - We have to recompile if an APi changes
    - All partners and services need to support protocols
    - Future upgrades must support older interface
- Maintain interface → ensure programs don’t crash

Side Effects

- Occur when an action on one object has non-obvious consequences
    - Effects not specified by interfaces or other objects
- Often due to a shared state
    - Optimization this way causes issues
- Unexpected behaviors

<u>Standards</u>

- Different from interfaces
- Interfaces: specific to OS
- Standards: global, for everybody

<u>The Role of Standards</u>

- Key standards - widely required
    - Non-compliance:
        - Reduces application capture
        - Raises prices for consumers

<u>Abstractions</u>

- Creates, manages, and exports abstractions
- Makes life easier for application programmers if there is a simple abstraction

<u>Simplifying Abstractions</u>

- Hardware - fast but complex and limited
- Abstractions
    - Hide the complexity and combine it with simplicity
    - Encapsulate implementation details
        - Eliminate irrelevant behaviors
    - Provide more convenient or powerful behavior
    - Combine operations of hardware → more power for user and better result

<u>Critical OS Abstractions</u>

- Core abstractions that computation model relies on
    1. **Memory Abstractions**
    2. **Processor Abstractions**
    3. **Communications Abstractions**

<u>1. Abstractions of Memory</u>

- Necessary for computing
- Many resources used by programs and people require data storage
- All programs require memory
- Persistent data storage wanted
- All memory have similar properties
    - read/write

<u>Some Complicating Factors</u>

- Persistent vs. Transient Memory
    - Persistent: stays, doesn't go away
    - Transient: may disappear
- Size of operations
    - Size the user/application wants to work with

    - Size the physical device actually works with → specific unit
    - **Size user wants =/= size hardware works with**
- Coherence and atomicity
    - Are you guaranteed that any number of operations occur?
    - How do we guarantee this?
    - **Coherence:** write then read immediately, we expect to see what we just wrote
    - **Atomicity:** operations either all happen or don't happen at all
- Latency: how long it takes for operations to happen
- Same abstractions - implemented with different devices
    - These differences are hidden

Where do Complications Come From?

- We do not have abstract hardware
- Particular physical devices with **unchangeable** and **inconvenient** properties
- Core OS abstraction problem:
    - Creating abstract device with the properties we want with the physical device that lacks them

Example: a file

- read/write a file
- read/write arbitrary amounts of data
- We expect/observe coherence
- Several reads/writes → expect an order
    - **We want to hide the fact that it may not be in order**

What is Implementing the File

- Hard disk drive
    - Only works when the magnetic moving head is at a certain location at a given moment
- Peculiar characteristics
- OS needs to smooth out the oddities
- **Physical movements → vastly slower than pure electronics**

What does that Lead To?

- **File system component** of the OS that puts things at the right place
- Reorder disk operations to optimize
    - Atomicity complications
- Optimize based on caching and read-ahead
    - Consistency maintenance complications
- Sophisticated organizations to handle failure

2. Abstractions of Interpreters/Processors

- Interpreter - something that performs commands
- Physical level: CPU processor
- OS provides us with higher level abstraction

Basic Interpreter Components

- An instruction reference - tells interpreter where to go next

- A repertoire - the set of things the interpreter can do
- An environment reference - current state on which next instruction should be performed
- Interrupts - situations in which instruction reference is overridden
  - Usually due to external stimuli

<u>An Example</u>

- A process: higher level abstraction of an interpreter
  - Runs on hardware
  - Multiple interpreters using the same core
- Program counter maintained by OS
  - Instruction reference
- Repertoire: source code
- Environment reference: stack, heap, register components
- Self-contained, no interference from other processes

<u>Implementing the Process Abstraction in the OS</u>

- Easy for a single process
- Difficult for multiple processes
- Limited memory
  - Need to SHARE the RAM
- Share a CPU/Core

<u>What Does That Lead To?</u>

- **Scheduler**: share the CPU
  - Built into OS
- **Memory Management**: hardware and software
  - Multiplex memory to use among processes
- **Access Control Mechanisms**: for memory abstractions
  - Protect one process from the others

<u>3. Abstractions of Communications</u>

- Communication link allows one process/interpreter to talk to another
- Communication between and within different computers/one computers and/or devices
- At physical level, memory and cables
- At abstract levels, networks and interprocess communication mechanisms
- Similarities to memory but also differences

<u>Why Are Communication Links Distinct from Memory?</u>

- Does not depend on whether messages are on the same machine or not
- Highly variable performance
  - Cannot depend on resource sharing
- Asynchronous
  - Usually involves issues synchronizing the parties
- Receiver sometimes may be blocked until it receives something from a sender
- Complications in remote machines

<u>Implementing Communication Link Abstraction in the OS</u>

- Easy if both ends are on the same machine
    - Usually copying and reading memory
- Same machine: use memory for transfer
- Complications in remote

What Does That Lead To?

- Optimize copying costs
- Memory management
- Complex network protocols need to be in the OS itself
- Security concerns

Generalizing Abstractions

- How do applications deal with many abstractions
- Make different things appear the same
    - Applications deal with a single class
    - Often Lowest Common Denominator + sub-classes
- Common/Unifying Model
    - What/How much do you expose
- Usually involves **federation framework**

Federation Frameworks

- Allows many similar, but somewhat different; things are treated uniformly
- One interface that all must meet
- Plug in implementations for particular things
- Common software layer you can plug into
    - Provides a **high level view** of what software does
    - **Translates** what comes in and what you can manage

Too Limiting?

- “Lowest Common Denominator” model required?
    - No
    - Can include “optional features”
- Special applications may be needed for special devices

Abstractions and Layering

- Complex services by layering abstractions
- Layering - allows modularity

Downside of Layering

- Performance penalties
- Going from one layer to another - interface changes may limit actions
- Expensive going from one layer to the next
    - May have to change data structures or representations
    - Always involves extra instructions
- Lower layers may limit upper layers

Reading

Kampe: Software Interface Standards

<u>Introduction</u>

- Independent software vendors (ISVs) & killer applications created situation where:
    - Cost of building version of application of different hardware platforms is high → reduce ports
    - Sales driven by applications computers can support
    - Software portability is crucial
- We need: detailed specifications, comprehensive compliance testing
- Changes in customer base (more people use computers)
    - New applications need downward compatibility
    - Upgrades cannot break existing applications

<u>Challenges of Interface Standardization</u>

- Why are standards good?
    - Extensive review, broad, well-considered
    - Platform-neutral
    - Clear and concise specifications, well-developed testing
    - Give suppliers freedom to explore implementations
- Standards - oppose diversity and evolution
    - Constrain possible implementations
    - Impose constraints on consumers
    - Difficult to evolve
        - What new requirements?
        - Changes affect stakeholders

→ Confusing Interface with Implementation

- Interface standards - specify **behavior**, not design
- Implementation-neutral
- Interface specifications written after design are BAD
    - May not comply
    - Determining fundamental behavior difficult

→ The Rate of Evolution for Technology & its Applications

- Technology evolving quickly - both computers and applications
- Forces us to choose between
    - Maintaining compatibility with old interfaces & not supporting new applications → leads to obsolescence
    - Developing new interfaces - incompatible with old interfaces/applications → sacrifice old customers

    - Compromise partially supports new & mostly compatible with old → uncompetitive
- Conflict: stable interfaces and embracing changes

→ Proprietary vs. Open Standards

- **Proprietary interface:** developed & controlled by a single organization
- **Open interface:** developed and controlled by a consortium of providers and/or consumers
- Tech provider must decide whether to:
    1. Open interface defers to competitors
       PROS: - better standard
       - higher chance of wide adoption
       CONS: - reduce freedom
       - give up competitive edge
       - force to re-engineer existing implementation
    2. Keep interface proprietary
       PROS: - patent protection
       - competitive advantage
       CONS: - competing open standard appears that is better - lose
       - fragment market, reduce adoption
       - no partners, shoulder all costs

Application Programming Interfaces (APIs)

- Typically APIs include:
    - List of writing/methods and signatures
    - Macros, data types, data structures
    - Discussions of semantics of operations
    - Discussion of options
    - Return values and error discussion
- Written at the source programming level → describes how source code should be written
- Basis of portability
- PROS:
    - App development - API is easily recompilable and executable
    - Platform suppliers - any application should port
- PROS only happen if API is platform-independent
- Open Standard finds and corrects this problem

Application Binary Interface (ABIs)

- API - good but not enough
- Problem with finding source for many applications
- Application Binary Interface solves this problem
    - **Binding of an API to an ISA**
- ABI: describes the machine language instructions and conventions to call routine on a particular platform, contains:
    - Binary representations of key data types

        - Instructions to call and return from subroutines
        - Stack frame structure and responsibilities from caller/callee
        - How parameters are passed and return values exchanged
        - Register conventions
        - System call and signal delivery conventions
        - Formats of load modules, shared objects, dynamically loaded libraries
    - Higher portability, **especially compared to an API**
    - Application → written to supported API → compiled and linkage edited by ABI → binary load module should run correctly

→ Who uses the ABI?

- The compiler - generate code for entry, exit, calls
- The linkage editor - create load modules
- Program loader - read load modules into memory
- OS - process system calls

Kampe: Interface Stability

Interface Specifications

- Parts manufactured and measured within specifications → reduce cost and difficulty of building working system

The Criticality of Interfaces in Architecture

- System architecture designs:
    - Components
    - Functionality
    - Interfaces (critical)

The Importance of Interface Stability

- Software interface specification is a contract:
    - Describes interface and functionality
- Implementers agree that their systems conform
- Developers agree to limit use to what is described in the specification

Interfaces vs. Implementations

- Interfaces should exist independently from any implementations

Altering Interfaces

- Adding upwards-compatible extensions
- Adding incompatible changes to old interfaces without breaking backwards compatibility
    - Interface polymorphism - different method signature with different parameters and return types
        - Different versions are very distinguishable and can provide new interfaces
    - Versioned interfaces - backwards compatibility requires us to continue providing old interfaces
        - Application may specify which version of an interface it requires

Interface Stability and Design

- When designing a system - need to consider the change that would happen
- External component interfaces need to accomodate all evolution
- We may:
    - Rearrange functionality distribution → simpler interface
    - Design features we never implement → ensure that implementations will support
    - Introduce abstraction → make room for future changes

Processes

Lecture 3: Processes, Execution, and State

Outline
- What are processes?
- How does OS handle processes?
- How do we manage processes' state?

Process
- Type of interpreter
- Executing instance (not the same as the file itself)
- Virtual private computer
    - Has its own **memory**, **CPU**, and **a copy of all its own resources**
        - This is not true, it is sharing the CPU with other processes
        - This is HIDDEN by the OS
- An *object*
    - Characterized by its properties (*state*)
    - Characterized by its *operations*
    - Not all OS objects are processes BUT processes are a central and vital OS object type

"State"
- data/information bits → we can **save the state**
- Important in OS, which stores the state of processes and later **restore the state**

Examples of OS Object State
- Scheduling priority
- Pointer into a file
- Completion status of an I/O operation
- List of memory pages allocated
- OS state - managed by OS itself
    - For a process to **change state**, we must **ask the OS**
    - Must **ask the OS** to **access or alter state** of OS objects

What are Operations?
- Activities performed by object → resulting in changes to its state
    - Usually results in changes to hardware devices
    - Sometimes results in changes to other objects' states

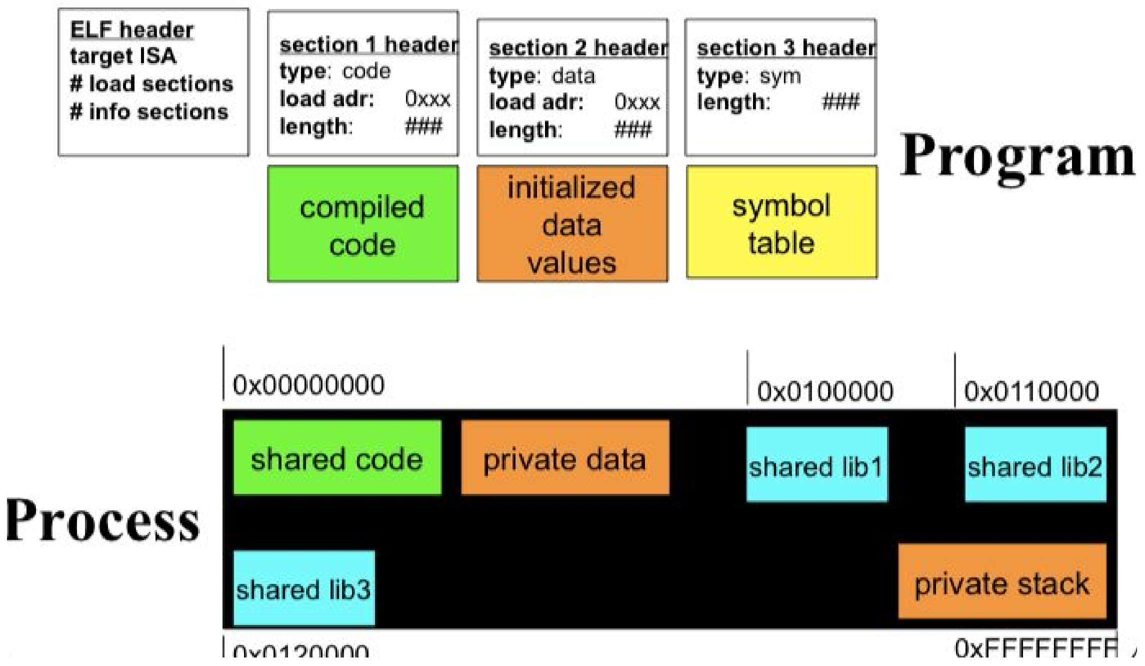
Examples of OS Object Operations
- Create a process
- Deallocate memory
- Open a file
- Write to display

- Cannot be performed by user process → almost always results in OS object state changing so need the help of the OS

Process Address Spaces

- Addresses that you can **issue legally** → in the address space
- Addresses you **cannot access** → NOT in the address space → cause a *segmentation fault*
- Process has some memory addressed reserved for its private use
    - Pieces of memory available to your process
- Set of addresses given to a process is an **address space**
    - All memory locations the process **can access**
- Modern OSes - pretend that every process' address space includes all of memory
    - Impossible, processes must share
    - This is an illusion, created so we do not have to worry about this when writing programs

Program vs. Process Address Space

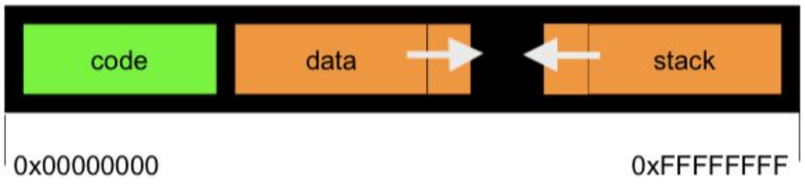


- Process thinks it is in RAM
- Process thinks it has entire set of addressees (0x0120000 - 0xFFFFFFFF) – illusion
- Black space - unallocated space in process

Process Address Space Layout

- All required memory elements are put somewhere in the process' address space
- Different memory elements have different requirements
    - Different segments of memory have different characteristics
        - Some parts are writable, but not executable, readable but not writable, etc.

Layout of Unix Processes in Memory



- Code segments statically sized
    - Not modified at compile time
- Data segment and Stack segment grow towards each other
    - We don't know how much of each we will need
    - This strategy allows them to have more address space
    - Need a bit to indicate which direction each segment grows
- Data and Stack segments are **not allowed to meet**
    - If they start to intersect, PROBLEMS
    - If you ask for more data, you will be denied
        - DATA: malloc() will fail
        - STACK: will kill a process if this happens

Address Space: Code Segments

- Load module: file that is **ready to be turned into a process**
    - Output of a **linkage editor**
    - External references have been resolved
    - Modules are combined into multiple segments (text, data, BSS)
        - Different information provided by the module builds the segments
- Code is loaded into memory → must be put in RAM, copied from the load module to RAM
    - Where do we put the address?
    - Instantiate it in RAM so we can run it
    - Code segment is created by reading in from the loadmodule
    - Map the segment into the address space
- Code segments are read/execute only and shareable
    - Processes can share the same code segments for efficiency

Address Space: Data Segments

- Data is also initialized in the address space
    - Process data segment is created and mapped into the address space
    - Initial contents are copied from the load module
    - BSS segments are initialized to all 0s
- Data segments
    - read/write
    - Process private
        - Process cannot read/write another process's data → even child processes have their own address space
    - Program can **grow** or **shrink** the data segment using system calls

        - sbrk → gives you more address space, not memory

Processes and Stack Frames

- Stack-based programs
- Procedure calls allocate a **new stack frame**
    - New memory is needed
    - This is set up in the address space for a stack (usually there is enough)
    - Storage for the procedure local variables
    - Storage for invocation parameters
    - Save and restore registers (popped off when the stack returns)

Address Space: Stack Segment

- Size of stack **depends on program activities**
    - Depends on variables, dynamic allocation, subroutine calls
    - Grows larger as calls nest more deeply
    - After calls return, stack frames are recycled
- Each recursive call → adds a stack frame
- Each subroutine call → adds a stack frame
- OS manages the process' stack segment - we don't have to worry about stack sizing
    - Stack segment - created at the same time as the data segment
    - Some OSes allocate a fixed size stack at **program load time**
        - OS guesses
        - If it guesses incorrectly, there is an exception that is caught by the OS who then increases the size of the stack
    - Some dynamically extend stack if the program needs it
- Stack segments are read/write and **process private**
    - Usually not executable

Address Space: Libraries

- Anything static is built into the load module
- Static libraries are **added to the load module**
    - Machine language instructions in loadable form are built into the load module
    - Each load module has a copy of each library
    - Program must be relinked to get the new version
- Shared libraries - less space
    - Not contained in load module, **already loaded elsewhere**
    - **If not loaded, we will load** since another process eventually will use it
    - Library separate from load modules
    - An in-memory copy is shared by all processes
    - OS loads library along with other program
- Reduced memory use and faster load times

Other Process State

- Not in the address space

- Registers (keep track of register data)
    - General registers
    - Program counters, processor status, stack pointer, frame pointer
- Process has its own OS resources
    - Open files, directories, locks, etc.
- Also has OS-related state information
    - Priority, cycles, etc.
- OS has a data structure to keep track of all of this information
    - Is **stored by the OS**
    - **Data structure associated with the process by OWNED by the OS**
        - Cannot be accessed by process

Process Descriptors

- **OS data structure** for dealing with processes
- Stores all info relevant to the process that **is not part of the address space**
    - State to restore when process is dispatched
    - References to allocated resources
    - Information to support process operations
- Managed by the OS
- Used for **scheduling, security decisions, allocation issues**

Linux Process Control Block (PCB)

- Data structure Linux (and other Unix systems) use to handle processes
- Type of process descriptor

Other Process State

- Process state not entirely stored in process descriptor
- Other process state information is in multiple other places
    - Application **execution stack** is on the stack and in registers
    - **Supervisor-mode stack** in Linux processes
        - Retain state of in-progress system calls
        - Save state of an interrupt
        - 2 stacks: user mode + supervisor mode
        - System calls done on this stack

Handling Processes

- Creating processes
- Destroying processes
- Running processes

Where do Processes Come From?

- Created by the OS
    - We must ask the OS to create a new process/run a particular program
    - Used some method to initialize their state
    - Set up a particular program to run
- At the request of other processes
    - Specify which program to run & other details
    - Other aspects of initial state

- Parent process - process **creating**
    - Process that created your process
- Child process - **created** process
    - Processes that your process created
- Parent → Child relationship

Creating a Process Descriptor

- Process descriptor - OS' basic per-process data structure
- To create a new process descriptor for a new process - OS gets memory to hold the process descriptor
- OS puts the descriptor in a **process table**
    - Data structure that keeps track of **all process descriptors for all processes**
    - Organizes all **currently active processes**
    - One entry/process

What does a New Process Need?

- **An address space**: issued by OS
    - Holds all segments
- OS creates a **new process space**
    - Allocates memory for code, data, and stack
- **OS loads program code and data into new segments**
- **Initializes stack segment and sets up initial registers**

Choices for Process Creation

1. "Blank" process
    - No initial state or resources
    - Need to have a way of filling in the vital stuff
2. Use the calling process (parent) as a template - make a copy of the parent
    - Give new process same stuff as the old one
    - Starts at the same place the parent process just finished

1. Starting with a Blank Process

- Create a brand new process
- System call creating it needs to provide parameters
- New process is created by command of an existing process

2. Process Forking

- Cloning the existing parent process so the child is almost exactly the same
- Assumption that the new child process is a lot like the old one (almost never true)
    - True for parallel programing
    - Not for typical user computer
    - Advantages

What Happens After a Fork?

- Two different process (parent and child)
    - Different IDs (parent and child know who's who)
    - Mostly exactly the same

- We need to write code to figure out which is which → **use difference in ID to make two different paths to follow (one for parent, one for child)**
  - Usually parent goes “one way” and child goes “the other”
  - Can be distinguished with a system call

Forking and Data Segments

- Forked child - shares parent’s code
- **Has its own stack**
  - **Initialized to match the parents**
- **Child has its own data segment**
  - Do not share data segments

Forking and Copy on Write

- Parent with a big data area means that setting up a copy for the child is expensive
- BUT if neither parent nor child write into parent’s data area - no copy is necessary
- Set up as a **copy-on-write**
  - If one process writes, then you make a copy and let the process write the copy
    - Other process keeps original
- Start off with something identical, if they don’t write **it is shared data**
  - Otherwise, if they write, **trap, and make a copy of what to write**
- Cheaper, if you don’t write or even need the data segment

But Fork Isn’t What I Want!

- Want the process to do something else entirely
- Use the exec() system call
  - “Remakes” a process
  - Switch from the current program to another program
  - Changes the code associated with a process
  - Resets the state as well
  - When exec() returns → you are in the main routine of the new program
- Can be called anytime on any process
- Usually:
  - Figure out parent and child
  - Have the child run exec()

The exec Call

- Handles the common case
- **Replaces process’ existing program with a different one**
  - **We don’t care about the parent’s data → the data area is reset**
  - **New code**
  - **New address space**
  - **Different set of resources**
  - **Different PC (change the program to the first line of the new program) and different stack (empty stack for a new program)**

        - **The file descriptor table is copied**
- Calling exec() → goes into other program, any other code after the exec() is not executed by the process calling exec()

How does the OS Handle Exec?

- OS needs to get rid of child's old code, stack, and data areas
    - Stack: set pointer to bottom
    - Data: get rid of all data pointers for the child
        - Easy if you use copy-on-write
- Load new set of code for that process
    - Looks as if you created a process from scratch
    - Can share some files from the parent and child (useful for piping)
- Initialize child's stack, PC, and other relevant control structures
    - Start a fresh program for the child process

Destroying Processes

- Processes terminate
    - All terminate when machine is turned off
    - Some do work and then exit
    - Others are killed by OS or another process
- OS needs to **clean it up** after it terminates
    - Looks for all resources associated to the resources (which are tracked)
    - Gets rid of these resources so they can be reused later
    - Allows simple reclamation

What Must the OS Do to Terminate a Process?

- Reclaim resources the process may be holding
- Inform other processes that need to know
    - Interprocess communications
    - Parent and maybe child process
        - Parent may be waiting for child to do something
        - Some parents may not be able to exit until the children are gone; those parents are ZOMBIE processes
            - Process sits in the table but the resources are deallocated
- Remove process descriptor from the process table → **this means the process is now gone**

Running Processes

- Each process runs on a single core (unless you have kernel level threads)
- Processes execute code - OS needs to give them access to a processor core
- BUT, usually, # processors > # cores
    - They need to share
- Subset of current processes is running the code
    - Switch between which process is actually running
- Process not running on a core will eventually need to be put on one
- Need to switch between processes on the core

    - Some may or may not be able to or be ready to run (blocked) → sits on the process table and waits

<u>Loading a Process</u>

- To run a process on a core, the core's hardware needs to be initialized
    - To initial state or previous state of the process
- State info: kept in the PCB and in the process' stack
- Must **load core's registers**
- Must **initialize the stack** and **set the stack pointer**
- Must **set up memory control structures**
- Must **set the program counter**
- Better to keep process running on a single core rather than switching it around
- Set it up in a way so that the **process works as soon as it starts** and is able to run the next instruction
    - To do so, we need a lot of state info/reference info about the process
- Enter "**limited direct execution"** model

<u>How a Process Runs on an OS</u>

- Execution model called **limited direct execution**
- Most instructions are **executed directly** ("Direct Execution" by the process on the core
    - Without asking the OS for permission
    - Helps run as fast as possible, speed depends on number of instructions we have to run
    - Includes **standard set of instructions**
- Some instructions **cause a trap** ("Limited") to the OS
    - Privileged instructions that need to be executed in supervisor mode
    - OS works from here

<u>Limited Direct Execution</u>

- CPU directly executes most application code
    - Time interrupts
    - Traps
- Maximize this is the goal
- Go to OS for the fewest possible things we can
    - If we go to the OS, we want to get back to the application as quickly as possible
- Want to spent most cycles running user level instructions

<u>Exceptions</u>

- Affects execution of a program
- Not usual
- Indicate that we need to do something else, instead of going to the next instruction
- What happens when the process can't (or shouldn't) run an instruction
- Some are routine

        - EOF, Overflow, Conversion
        - We need to check for these
    - Some are unpredictable
        - Segmentation fault
        - ^C abort, power failure
        - **Asynchronous exceptions**

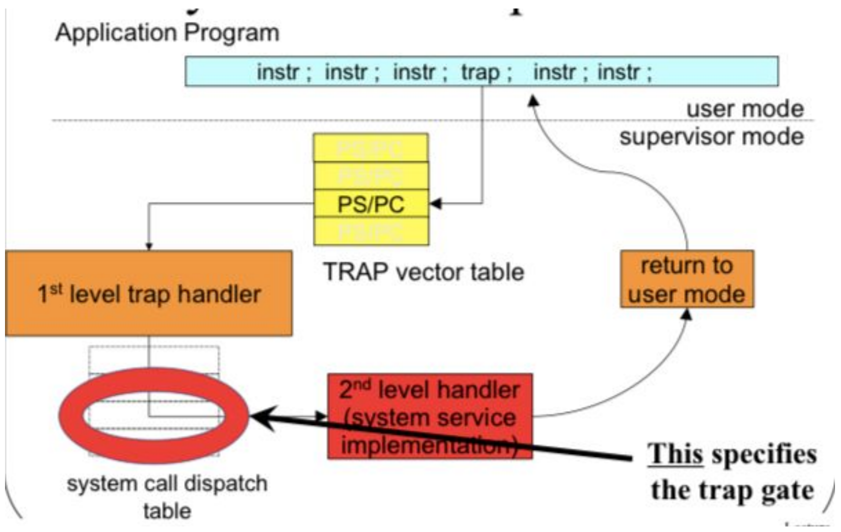
Asynchronous Exceptions

- Unpredictable
- Can't check for them - programs have no knowledge of when they will happen
- Doesn't have anything to do with the program, usually because another thing is happening
- Hardware (signals on lines of bits) and OS (interrupts program and does OS work instead) support **traps**
    - Catch exceptions and **transfer the control to the OS**
- OS also uses this for **system calls**
    - Requests from programs for OS services
- Transition from user code → OS/privileged code
    - We need to ask the OS by causing an **exception**/interrupt
    - Causes the OS to take over
    - One way: try to run a privileged instruction → gets the OS's attention
        - Have a user level program run an instruction they are not allowed to run
        - Particular instruction exists in hardware that does this
            - Switch from user → OS mode
            - Each associated in the ABI with a number
            - Placed as a parameter for a special instruction ("trap")
                - Used in the trap table

Traps for System Calls

- Traps: type of exception
- Made possible in the hardware
- We can **reserve one privileged instruction for system calls**
- Define a **system call linkage editor**
- **Prepare arguments** for the desired system call
- Execute the **designated system call** instruction (which is not allowed)
- Causes an exception that **traps to the OS** - tells which service you want to use
- OS recognizes this and performs the **requested operation**
    - Enter the OS through a point called a **gate**: leads to OS, **one gate/system call**
- Return to the instruction after the system call

System Call Trap Gates


Application Program
instr ; instr ; instr ; trap ; instr ; instr ;
user mode
supervisor mode
PS/PC
TRAP vector table
1st level trap handler
return to user mode
2nd level handler (system service implementation)
system call dispatch table
This specifies the trap gate

- 1st level trap handler: general, done for all traps
    - **Saves the state of the program**
- **TRAP vector table:** set up during boot time by OS
    - For this trap → go to this location
- **2nd level handler (system service implementation)**: specific to the system call that was requested

Trap Handling

- Hardware + software working together
- Hardware portion:
    - Trap indexes into trap vector table for PC/PS
    - Load processor status word, switch to supervisor mode
    - Puch PC/PS of program that caused trap onto stack (for when we return)
    - Load the PC with the address of the 1st level handler
- Software portion:
    - 1st level handler pushes all other registers, gathers info, and selects 2nd level handler
    - 2nd level handler actually deals with the problem

Traps and the STack

- Code to handle a trap is just code
    - Run in privileged mode
- Requires a stack - where should this stack be?

Stacking and Unstacking a System Call

- Two stacks: user-mode and supervisor-mode stacks
- User-mode stack contains:
    - Stack frame from application computation
    - Resumed computation
- Supervisor-mode stack contains:
    - User mode PC & PS
    - Saved user mode registers

    - Parameters to system call handler
    - Return PC
    - System call handler stack frame
- When we done, we load back everything
- Choice:
    - We can have one supervisor mode stack to share
    - BETTER: each process **has its own supervisor mode stack**

Returning to User-Mode

- Return - opposite of interrupt/trap entry
    - 2nd level handler returns to 1st level handler
    - 1st level handler restores registers from the stack
    - Use privileged return instructions to restore PC/PS
    - Resume **user-mode execution** at the next instruction
        - Change the processor status mode
    - Saved registers can be changed before returning

Asynchronous Events

- Some things cannot be done quickly
    - Ex: read()
- Other times, waiting does not make sense
    - Want to do something else while waiting
    - Some events require prompt attention
- Need **event completion call-backs**: tell me when this event occurs
    - Programming paradigm
    - Computers support **interrupts** (like traps)
    - Associated with **I/O devices and timers**
    - Support multiple ongoing activities simultaneously
    - Let you know when something is complete
    - **Interrupt-based programs**
    - Event recognized by the OS → checks if this is something we are waiting for → makes an **interrupt** if this is the case

User-Mode Signal Handling

- Different types of **signals**
- Process can **control their handling**
    - Ignore
    - Designate a handler
    - Default action
- Analogous to hardware traps/interrupts
    - Implemented by the OS
    - Used by user-mode processes

Managing Process State

- Shared responsibility
- Process takes care **of its own stack and memory**
- OS keeps track of **resources allocated to the stack**

Blocked Processes

- Process state element: whether a process is **ready to run**
    - Process that aren't ready to run are **blocked**
        - Cant run any useful instructions until an event has unblocked the process
- OS keeps track of whether a process is blocked
    - In order to eventually schedule it — **scheduling**

Blocking and Unblocking Processes

- Why block?
    - Make sure OS schedules them only when they are ready
    - Notes to the scheduler so the scheduler knows not to choose them
    - Other part os OS know if they later need to be unblocked
- Any part of OS can set blocks and remove them
    - And a process can ask to be unblocked

Who Handles Blocking?

- **Resource manager**
    - Process needs an **unavailable resource**
        - Change the state to "blocked"
        - Call the scheduler and yield the CPU so another process can run
    - When process is available → become unblocked
        - Change process' scheduling state to "ready"
        - Notify scheduler
        - Processor can choose to unblock itself

Swapping Processes

- Processes only run while in **main memory**, RAM holds all info for that process
- Sometimes we **move processes** to **secondary storage**
    - Use previously occupied RAM for another process - **swapping**
- Usually due to resource shortages (particularly, memory)

Why We Swap

- Make good use of limited memory
    - Not ready, blocked → doesn't need to be in memory
- Don't swap out all blocked processes
    - Expensive to swap back and bring back in
    - Done when resources are tight (memory management)

Basic Mechanics of Swapping

- Process' state is stored as data in main memory
- Copy it to secondary storage
- Alter the process descriptor to note that you did
    - Indicate this process is currently not in main memory and has been swapped out
- Give the freed resources to another process
    - **Entire reason we swap**

Swapping Back

- Whenever what blocked the process is cleared, you can swap back if there is space
- Reallocate memory and copy state back
- Unblock the descriptor to make it eligible for scheduling
- Ready swapped process do not need to be brought back immediately

Reading

Arpaci-Dusseau Ch. 4 - The Abstraction: The Process

- Process: a running program
- Our problem: the OS needs to provide the illusion of an endless supply of CPUs
    - Virtualize the CPU
    - “Time sharing”: sharing for a bit of time
    - “Space sharing”: dividing up the resource
- To implement, we need:
    - Low level machinery - mechanisms: methods or protocols of functionality
        - Context switch
    - Policies: algorithms that make decisions in the OS
        - Scheduling policy
    - Separation of HOW and WHICH → modularity

4.1 The Abstraction: A Process

- Process: abstraction of a running program
- Machine state: what a program can read/update while running
    - Memory - address space: memory the program can address
    - Registers - read/update registers
        - Special registers
            - Program counter/instruction pointer
            - Stack pointer
            - Frame pointer
    - Persistent storage devices = I/O information

4.2 Process API

- General look @ the API:
    - Create: create a new process
    - Destroy: destroy processes forcefully
    - Wait: wait for a process to stop running
    - Misc: other controls, ex: suspend and resume
    - Status: get status information about a process

4.3 Process Creation: A Little More Detail

- Q: How does process creation work?

1. **OS loads code and static data into memory** → address space of the process
    - Programs reside on disk in executable

    - Eagerly done in earlier OS → all at once before running
    - TODAY: done lazily → load code only when need
        - Paging, swapping
    - Loading: takes on-disk program and reads it into address space of the process
2. **OS must allocate memory to the run-time stack** and give it to the process, initializing the **stack with arguments**
    - Also allocate in heap dynamically within the data case
3. Initialization tasks related to I/O
- Set the stage for execution
- Final task: jump to the main() → **transferring control from the OS to the process**

4.4 Process States

- Processes can be in one of three states
    - **Running:** process running/executing instructions
    - **Ready:** process ready to run but OS is not running it
    - **Blocked:** process not ready until another event takes place

        Running → Ready (descheduled)
        Ready → Running (scheduled)
        Running → Blocked (I/O initiate)
        Blocked → Ready (I/O done)
- Scheduled: moved from ready to running
- Descheduled: moved from running to ready
- Deciding between scheduled/descheduled are decisions made by an OS scheduler

4.5 Data Structures

- **Process list**: tracks state of each process and other information
- Track blocked processes
- Important information OS tracks about a process:
    - **Register context**: holds contents of registers when a process is stopped → register saved and restored when resumed (**CONTEXT SWITCH**)
    - Other states:
        - Initial: when process is created
        - Final: exited but not cleared yet - "zombie"
            - Allows other processes ("parent" usually) to inspect return result
- Process control block (PCB): structure that stores information about a process

Arpaci-Dusseau Ch. 5 - Interlude: Process API

- UNIX process creation using fork(), exec(), and wait()
- fork() creates a new process ("child") identical to the parent
    - In child, returns 0
    - In parent, returns PID of child

- exec() runs a new program
    - Loads code from the executable
    - **Overwrites** current code segment
    - **Heap and stack are reinitialized**
- wait() waits for child to die before parent dies

Arpaci-Dusseau Ch. 6 - Mechanism: Limited Direct Execution

- Virtualize the CPU → time sharing: run one process for a little and then another
- Challenges: performance (avoid overhead) & control (retain control of CPU)

6.1 Basic Technique: Limited Direct Execution (LDE)

- **Limited direct execution:**
    - "Direct execution": run program directly on CPU
        - OS starts program by creating process
        - Problems:
            - OS needs to ensure program doesn't misbehave
            - How to implement time sharing?
                - Stop a process while its running and switch to another process

6.2 Problem #1: Restricted Operations

- Direct execution is fast, program can run natively in the hardware
- Problem: process may perform restricted operations
    - Introduce a new processor mode, **user mode**, restricting what it can do
- **Kernel mode**: the OS runs in this mode
    - Challenge: what if process wants to perform privileged operations → **use system calls** that allow kernel to expose some functionality to user programs

  → Executing a System Call
    1. Program executes a **trap** instruction
        - Jumps into kernel and raises privilege level to kernel mode
    2. PRogram does required work
    3. OS calls a **return-from-trap** instruction once complete
        - Returns caller to user mode
- Hardware is careful when trapping
    - Caller's registers are to return onto a per-process **kernel stack**
        - Return from trap instruction pops values off of the stack and **resumes execution as normal**
- Kernel must control what code is executed upon a trap
    - The code can't **specify an address to jump to** as this would be dangerous
    - Creates a **trap table @ boot time in kernel mode**
        - OS tells hardware **what code** to run when certain events occur

        - This code is called a **trap handler**, the locations of which are given to the hardware
        - Hardware remembers what to do when a system call happens
- **System-Call-Number**: assigned to each system call
    - User code places these numbers into registers or in the stack
    - OS looks at this number and executes in a system call case
- This **indirection**: provides protection as user cannot specify an address, instead, requests a service number
- Instructions tell hardware where trap tables are - privileged
- Two phases in LDE: boot time and runtime
    - Boot time: kernel initialized the **trap table** and CPU remembers its locations
    - Runtime: kernel sets things up and then uses the **return-from-trap** instruction to start the execution of the process

6.3 Problem #2: Switching Between Processes

- Switching is difficult
- While processes are running, the OS is not
    - How does the OS regain control?

→ Cooperative Approach: Wait for System Calls

- Cooperative approach: **OS trusts processes to behave reasonably**
- Process: will give up control of the CPU by **frequently asking for system calls**
- Include explicit **yield** system call that transfers control to the OS
- Transfer control by **doing something illegal**
    - The OS will regain control by waiting for a system call or an illegal operation
    - **Will generate a TRAP**
- OS regains control by **waiting for a system call** or an **illegal operation**
- Issue: when a process gets stuck, to regain control, you have to reboot the machine

→ A Non-Cooperative Approach: The Os Takes Control

- **Timer interrupt**: used to gain control of the CPU when processes are not being cooperative
    - Timer: can raise an interrupt → process currently running is halted and an **interrupt handler** in OS runs
- OS specifies a **pre-configured interrupt handler** at boot sequence
    - Hardware needs to save enough of the state of the program such that later it can be continued correctly
    - **Similar to a system call**
- OS starts the timer (privileged operation) during the boot sequence
    - It can also be turned off (privileged operation)

→ Saving and Restoring Context

- **Scheduler**: decides to continue current process or to switch to a different one
- If switching → the OS then executes a **context switch**

    - OS saves the register values for the current process onto its **kernel stack** & restores them for a soon-to-be executing process
- Saving context of currently running process means the OS executes low-level assembly code to save the different registers and restore all of this for the next process
- Switch to the kernel stack for soon-to-be executing process
- **Switching stacks - kernel enters the call to the switch code in the context of one process and returns from another**
- Return from trap → soon-to-be-executing process becomes the current process
- Two types of register saves:
    1. When a timer interrupt occurs
        - User registers saved by hardware using the kernel stack
    2. OS switches from A → B
        - Kernel registers are saved by the software into memory in process structure
            - Moves the system from running as if it were trapped in A to as if it had been trapped in B

<u>6.4 Concurrency</u>

- What if another interrupt occurs while in the system call/handling an interrupt?
    - **Concurrency issue**
- How does the OS handle this?
    - May **disable interrupts** during interrupt processing
    - **Locking** schemes protect concurrent access to internal data structures

Kampe: Object Modules, Linkage Editing, Libraries

<u>Introduction</u>

- Process: executing instance of a program

<u>The Software Generation Tool Chain</u>

- Classes of files that represent programs:
    - **Source modules**: editable text in a language that is translated to machine language by a compiler/assembler
    - **Relocatable object modules**: sets of compiled/assembled instructions created from source modules, not yet complete
    - **Library**: collections of object modules, we can fetch functions from them
    - **Load Modules**: complete programs that are ready to be loaded and executed

[.c & .h files] → (1. compiler) → [.s files] → (2. assembler) → [.o files] → (3. Linkage editor) → [executable] → (4. Program loader) → PROCESS

- Components of this Software Tool Chain

1. Compiler: reads the source modules and header files
    - Parses the language
    - Infers intended computation

        - Generates the lower level code
        - Usually generates assembly language to allow more flexibility
2. Assembler: translates each line to machine language instructions/data items
    - User mode assembly includes:
        - Performance critical string and data manipulation
        - Routines to implement calls in the OS
    - Kernel mode/OS assembly includes:
        - CPU initialization
        - First level trap/interrupt handlers
        - Synchronization
    - Output: object module containing the machine language code
        - Corresponds to a single input module for the linkage editor, so:
            - Some functionality is not present, addresses not yet filled in
            - Locally defined symbols are expressed as offsets
3. Linkage Editor: reads object modules and places them in the virtual address space
    - Notes where objects were placed
    - Notes unresolved external references
    - Searches libraries (the specified set) to find object modules that satisfy these references
    - Places the libraries in address space
    - Finds references to relocatable/external symbols and updates the module to reflect the new addresses
    - **Result: program ready to be loaded and executed**
        - The new file is called an "executable load module"
4. Program Loader: part os the OS
    - Examines information in load module
    - Creates the virtual address space
    - Reads the instructions/initialized data values in the virtual address space
    - adds/maps additional shared libraries if needed

- The virtual address space is created, code is loaded in there → then the program can be executed

<u>Object Modules</u>

- Programs: combination of multiple modules
    - Fragments: relocatable object modules
    - Differences from executable (load) modules:
        - Incomplete - reference code/data items from other modules
        - Addresses not yet determined
- Code in object modules is ISA specific
- Object module formats: common across different ISAs
    - One popular format is **ELF (Executable & Linkable Format)**

→ ELF Format: divided into different multiple consecutive sections
- Header: describes size/type/locations of other sections
- Code & Data: bytes to be loaded contiguously into memory
- Symbol Table: lists external symbols needed/defined
- Relocation entries collection, each describes:
    - Location of a field that requires relocation
    - width/type of field
    - Symbol table entry

<u>Libraries</u>
- Library: collection of object modules
- Not always independent
    - Implementing higher level libraries using lower level libraries is common
    - Using alternative implementations for some library functions
    - Intercepting calls to standard functions
- Which library we use is important

<u>Linkage Editing</u>
- Three things are done in order to turn collection of relocatable object modules into runnable programs:
    1. **Resolution:** search specified libraries to find object modules that can satisfy unresolved references
    2. **Loading:** lay text and data from object modules into single virtual address space
    3. **Relocation:** go through all relocatable entries and relocate each reference to reflect the chosen addresses
- “Linkage editing”: fill in all linkages in loaded modules
    - Done by a linkage editor

<u>Load Modules</u>
- Contains multiple sections
- Doesn’t need relocation - it is complete
    - Unlike an object module
- When OS is instructed to load a new program:
    1. Consults the load module to check size and locations of text/data
    2. Allocates segments within virtual address space
    3. Reads contents of text and data segments from modules into memory
    4. Creates a stack segment and initializes the stack pointer
- Why a symbol table?
    - Exceptions → symbol table allow us to detect where the error occured

<u>Static vs. Shared Libraries</u>
- Static linking: library modules directly (and permanently) incorporated into load modules
- 2 significant disadvantages

    - Libraries are reused often - many copies of the same code will decrease performance
    - Popular library can change - programs will have been built frozen with the version available at the time it was last linkage edited
- Issues addressed by runtime loadable, shared libraries
- Possible implementation
    1. Reserve an address for each shared library
    2. Linkage edit each shared library into a **read-only code segment**
    3. Assign a number to each routine, place a **redirection table** at the beginning of the segment with an address to each routine
    4. Create a stub library that defines symbols for every entry point in the shared library
        a. Implement each as a branch through entry in the redirection table
        b. Include symbol table information → tells OS what segment this program requires
    5. Linkage edit the client program with the stub library
    6. When the OS loads the program into memory, notice that the program requires a shared library and open the code segment, mapping it onto the address space
- PROS:
    - Single copy shared among all programs
    - Version chosen at **program load time**, can be controlled
    - Client programs - not affected by size changes or additions
        - Calls between client and library are vectored through a table, the **redirection table**
    - Possible for one shared library to make calls to another
- Limitations:
    - Read-only code, no static data
    - Shared segment - not linkage edited, no static calls/global variables to/from the client program
        - Knows nothing about client program
    - May be large and expensive and seldom used
        - Slows down the start-up because of the huge memory
    - The name of the library to be loaded must be known at linkage time

<u>Dynamically Loaded Libraries</u>

- Dynamically loadable libraries (DLLs): libraries that are not loaded (or even chosen)
- General model:
    1. Application choses a library to be loaded
    2. Application asks OS to load library into the address space
    3. OS returns the address of standard address pointers
    4. Application calls supplied initialization entry point - application and DLL bind to each other

5. Application requests services from DLL through vector
6. Application calls shutdown method when done and asks OS to unload

- In Linux: dlopen(3)
- Calls from the client application or the DLL are handled **through a table or vector of entry points**
- Calls from the library to the application
    - Application can regulate a call-back routine by passing its address to an **appropriate library registration method**
        - This can be generalized
        - Application can call the library to register a vector of entry points for standard functions
    - DLL: allowed to access function and data structures in the kernel, enabled by the run-time loader
        - Linkage edits the loaded module against the OS

<u>Implicitly Loaded Dynamically Loadable Libraries</u>

- DLL Implementations exist that are more similar to shared libraries
    1. Application linkage edited against set of stubs to create a program linkage table (PLT) entry in the load module
    2. PLT entries are initialized to point to calls to a run-time loader
    3. When entry point is called for the first time, the run-time loader is called by the stub to open and load the DLL
    4. After loading, the PLT entry is changed to point to routine — now, all calls from PLT entry go directly to this routine
- Implicitly loaded DLLs - indistinguishable from statically loaded libraries or shared libraries
- DLL - only available if client applications know about them
- Shared libraries:
    - Allow delayed binding
    - Consume memory entire time
    - Constraints on code
    - Indistinguishable, simple
- DLLs:
    - Library chosen at runtime
    - Unloadable
    - Complexity is OK
    - Require more work to load
- **Summary: shared libraries are more efficient but DLLs can extend the functionality of a client application**

Kampe: Stack Frames & Linkage Conventions

<u>Introduction</u>

- Fundamental questions

1. What is a state, how can it be stored?
2. How to request services & receive results?

The Stack Model of Programming Languages

- Modern programming languages - support procedure-local variables
    - Allocated automatically
    - Visible only to code within procedure
    - Each distinct in location
    - Automatically deallocated
- Local variables, parameters, intermediate results - stored in a LIFO stack
    - New call frames pushed onto stack when procedure called
    - Old frames are popped when a procedure returns
- Universal - processor architecture provides hardware instructions for stack management
- If stack doesn't work, use a heap instead
    - May not work because some data items need to last longer than the procedure that created them

Subroutine linkage conventions

- For Intel x86 architecture
- Basic elements of subroutine linkage:
    - Parameter passing: send information that will be passed as parameters to the routing
    - Subroutine calls: save return address (in calling routine) on the stack and transfer control to the entry point
    - Register saving: save contents of the register that the linkage conventions declare to be **non-volatile** so they can be resotred later
    - Allocating space for local variables
- When a subroutine completes, there is a return process
    - Return value: place return value where the calling routine expects it
    - Popping local storage off of stack
    - Register saving: restore non-volatile registers to the values they had
    - Subroutine return: transfer control to the return address that the calling routine had saved before beginning the call
- General Observations:
    - Register saving is the responsibility of the **called routine**
        - Only the **called routine knows which registers it will use**
    - Cleaning parameters of the stack is the responsibility of the **calling routine**
        - Only the **calling routing knows how many parameters were passed**

Traps and Interrupts

- **Interrupts**: inform software an external event has occurred
- **Traps**: inform software of an execution fault
- Both supported by ISAs
- Similar to procedure calls in that we must:

    - **Transfer control** to an interrupt/trap handler
    - **Save the running state**
    - **Restore the saved state** and resume after
  - Differences:
    - Procedure cals are **requested by the software**
      - Calling software expects function to run and return
      - Linkage conventions are under software control
    - Interrupts/traps are **initiated by the hardware**
      - Unexpected, state should be restored once completed
      - Linkage conventions are defined by the hardware
  - Typical trap/interrupt mechanism
    - # associated with every possible interrupt or exception
    - Table initialized by OS associates a **program counter and processor status word (PC/PS pair)** with each possible interrupt or exception
    - When trap/interrupt is triggered
      1. CPU uses **number to look up in the vector table**
      2. CPU loads a new PC/PS pair **from the interrupt/trap vector**
      3. CPU pushes PC/PS pair (associated with interrupted computation) into CPU stack (associated with the new processor status word)
      4. Execution continues from new PC
      5. Selected code **(first level handler)**
         a. **Saves register to stack**
            i. **Specifically to the kernel stack**
         b. Gathers info from hardware on cause of trap/interrupt
         c. **Choses second level handler**
         d. Makes procedure call to second level handler
    - After **second level handler** finishes, we return to the first level handler
      1. First level handler restores saved registers
      2. First level handler executes “return from interrupt” or “return from trap” instruction (privileged)
      3. CPU **reloads the PC/PS pair that was saved at the time of interrupt/trap**
      4. Execution resumes

→ Stacking and Unstacking of a (System Call) Trap
  - interrupt/trap mechanism
    - More complete job
    - Translates hardware driven call to the first level handler to a higher level language to the second level handler (chosen by the first level handler)
  - Much more expensive than a procedure call
    - Because a new processor status word is loaded → moves us to a more privileged processor mode

Summary

- Procedure and trap/interrupt linkage provide:
    - Low-level definition of the **state of a computation and what it means to save and restore that state**
    - Intro to CPU-supported mechanisms for synchronous and asynchronous transfer of control
    - How to interrupt an on-going computation

Scheduling

Lecture 4: Scheduling

<u>Outline</u>

- What is scheduling
    - Scheduling goals
- What resources should we schedule
- Example scheduling algorithms and their implications

<u>What is Scheduling?</u>

- OS has choices on what to do next
- In particular, for resources that can only serve one client at a time
- Making decisions on who gets the resource and for how long is scheduling

<u>How do we Decide How to Schedule?</u>

- Choose a goal we wish to achieve
- Design a **scheduling algorithm** that in order to achieve our goal
- Different scheduling algorithms optimize different things, try to achieve different goals
- Changing scheduling algorithms can dramatically change system behavior

<u>The Process Queue</u>

- OS typically keeps a queue of processes ready to run
    - Ordered by the scheduling algorithm
- Processes that aren't ready to run (blocked)
    - Aren't in the queue
    - Are at the end
    - Or are ignored by the scheduler

<u>Potential Scheduling Goals</u>

- Maximize **throughput**: get as much work done as possible
- Minimize **average waiting time**: avoid delaying too many for too long
- Ensure some **fairness**: ensure processes get relatively equal time on the CPU
- Meet explicit **priority goals**: prioritize certain processes, schedule based on a priority
- **Real time** scheduling: scheduled items with a real life deadline
- No scheduler that meets all of these goals

<u>Different Kinds of Systems, Different Scheduling Goals</u>

- Time Sharing
    - Fast response time to interactive programs
    - Each use should get an equal share of CPU (fairness)
- Batch
    - Maximize throughput
    - Delays of processes aren't important

- Real-time
    - Critical operations must happen on time
    - Non-critical operations may not happen at all

Preemptive VS. Non-Preemptive Scheduling

- Non-preemptive: let a piece of work use the resource until it finishes
    - Process decides when to stop
    - **Scheduled work always runs to completion**
- Preemptive: virtualization techniques to interrupt process part of the way through
    - OS decides when to stop
    - Allow other pieces to run instead
    - **Scheduler temporarily halts running jobs to run something else**

Pros and Cons of Non-Preemptive Scheduling

+ Low scheduling overhead (not many decisions)
+ High throughput (jobs are always completed)
+ Conceptually simple

- Poor response time (large processes will take up entire CPU)
- Bugs can cause machine to freeze up (process that is buggy can take control of the entire CPU)
- Not good fairness
- Make real time and priority scheduling difficult

Pros and Cons of Preemptive Scheduling

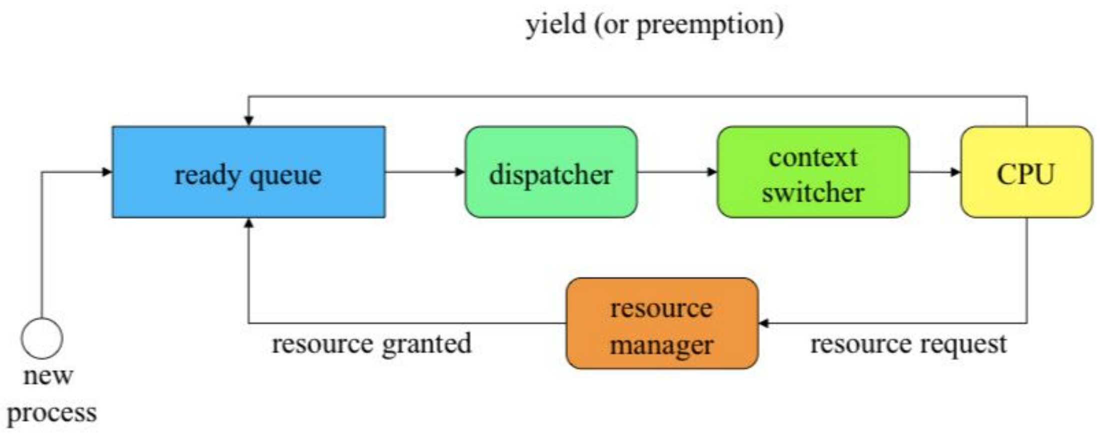
+ Good response time (when you need something, you can get it)
+ Fair usage
+ Works well with real-time and priority scheduling

- More complex
- Requires clean halting of processes and the saving of process states
    - To help ensure sharing and fairness
    - Need to continue as if it isn't preemptive
    - Done by the OS
- May not get good throughput
    - Lots of scheduling and jobs to be done

Scheduling: Policy and Mechanism

- Policy: who runs next
- Mechanism: how do we decide and swap
- **Dispatcher:** scheduler moves job in and out of a processor
    - Part of scheduling mechanism
- How dispatching is done does not depend on policy
    - Able to change policy with the same functionality
- Separate **choice of who runs** and the dispatching mechanism
- Able to plug in different policies and keep the same mechanism

Scheduling the CPU



- Yield: stop process, let another process run
- Preemption: Os decides to stop the process
- **Context switch**:
    - Context: info that describes the state of the process
    - Keeps track of page tables, registers, priority, etc.
    - Mechanism: **context switcher**
- Resource request: blocking operation
    - Resource manager: does the request

Scheduling and Performance

- Important system activities scheduling has a major effect of performance
- May not be able to optimize for all aspects of performance
- Different characteristics under heavy and light load
- Increasing one aspect → may damage another
- Need to understand performance basics

General Comments on Performance

- Goals should be quantitative and measurable
    - Need to quantify “goodness”
- Metrics: the way and units in which measure
    - Choose characteristic to be measured
    - Unit to quantify that characteristic

How Should We Quantify Scheduler Performance?

- Throughput (processes/second)
    - Different processes need different runtimes
    - Not controlled by scheduler
- Delay (milliseconds)
    - What delays should we measure?
    - Some delays are not the scheduler's fault

Other Scheduling Metrics

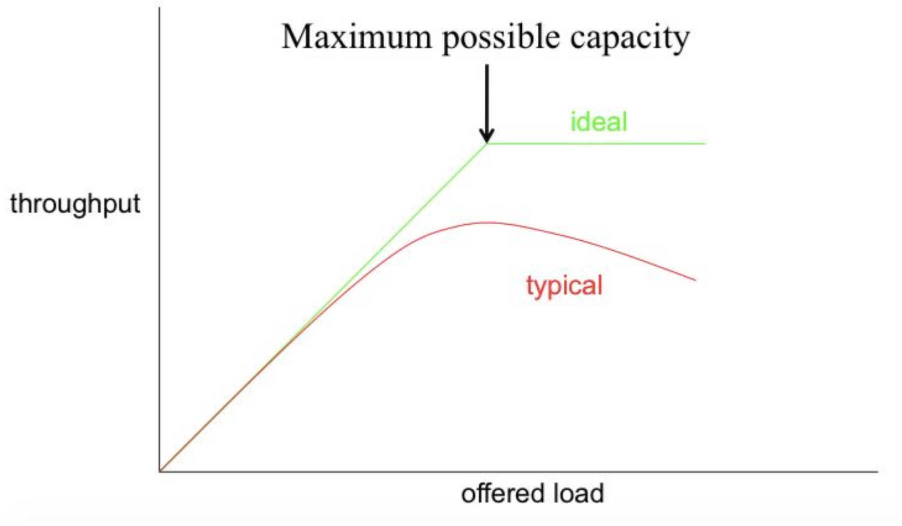
- Fairness
- Mean time to completion (seconds)
    - For a particular job mix

- Throughput (operations/second)
    - For a particular activity or job mix
- Mean response time (milliseconds)
    - Time spent on ready queue
- Over "goodness"
    - Based on customers

Example: Measuring CPU Scheduling

- Process execution phases
    - Time spent running
        - Process controls this
    - Time spent waiting for resources or completion
        - Resource manager controls this
    - Time spent waiting to be run
        - Scheduler controls this
- Metric: time spent on the "ready" queue

Typical Throughput vs. Load Curve



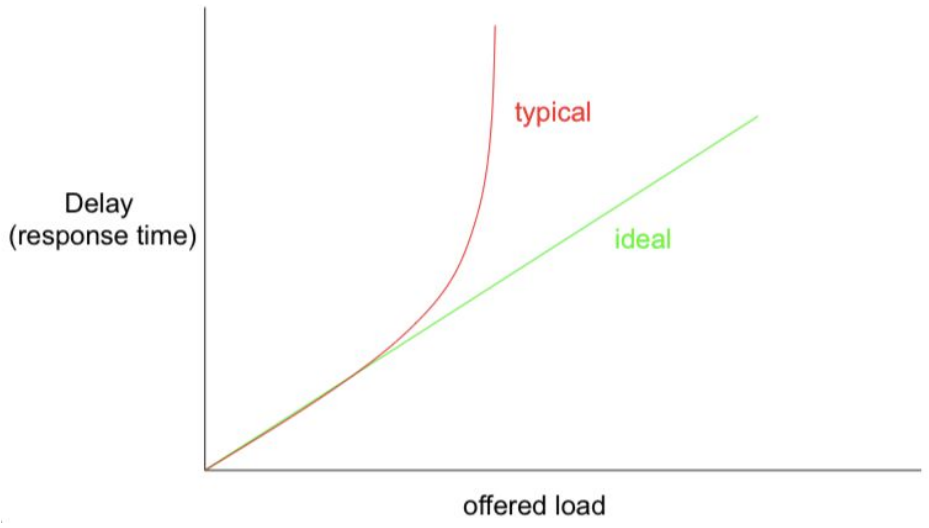
- Ideal: max work system is capable of doing
    - More load, still constant
    - Plateau
- Typical: less and less work done
    - Real systems

Why do we not Achieve Ideal Throughput?

- Scheduling is not free
    - We have to **spend the resource**
    - **Overhead** to dispatch a process
    - More dispatches → more overhead
    - Less time is available per process
- Minimize the performance gap by:
    - Reducing **overhead per dispatch**

- Minimize number of dispatch
  - Don’t dispatch when unnecessary
  - Don’t want to switch jobs often

Typical Response Time vs. Load Curve



- Typical: essentially approaching infinite response time
- Ideal: delay is infinite still (possibility)

Why Does Response Time Explode?

- Real systems have **finite limits**
  - Impact what you can do with what you have
- When limits are exceeded, requests are **dropped**
  - Not enough memory
  - Requests have infinite response time
  - May be automatic retires but those can be dropped
- Overheads during **heavy load explode**
  - Explosion of time doing **non-useful work → overhead**

Graceful Degradation

- What we want
- Overloaded system: no longer able to meet service goals
- **What do we do when we are overloaded?**
  - Continue service with degraded performance
  - Maintain performance by rejecting work
  - Resume normal service when the load returns to normal
- **What should we NOT do when overloaded?**
  - Allow the throughput to drop to 0
    - Try not to spend all time on overhead
  - Allow response time to grow without limit
    - Want work accepted to have a reasonable response time

Non-Preemptive Scheduling

- Scheduled process runs until it yields CPU

- OK for simple systems
    - Small number of processes
    - Natural producer, consumer relationships
        - Runs, produces work, waits for process to be consumed
        - Do one, switch, do other, switch
- Maximizes throughput
- Depends on process to voluntarily yield
    - Complete or yield the processor is built into the program
    - Piggy process: starve others
    - Buggy process: can lock up entire system

Non-Preemptive Scheduling Algorithms

- First come, first served
- Shortest Job Next
- Real Time Schedulers

First Come, First Served

- Simplest
- Run first process on ready queue
    - Until it completes or yields
- Run next process
    - Processes not yet ready are placed at the end
- Highly variable delays
    - Depends on implementations → which processes and how they are implemented on the system
- All processes eventually served

When Would First Come First Served Work Well?

- Very simply
- Poor response time
- Makes sense in:
    1. Batch systems: response time not important
    2. Embedded systems: computations are brief or natural producer/consumer relationships exist

Real Time Schedulers

- Things **must** happen
- Synchronization with the real world
- Need things to happen in real time
- Schedule on the basis of **real-time deadlines**
    - Moments in real world where something must happen
    - Code needs to be run at a particular time
- Either *hard* or *soft* deadlines
    - Hard: strict, vital
    - Soft: less strict, less vital

Hard Real Time Schedulers

- Absolutely must meet deadline

- System fails if deadline is not met
- Require very, very careful analysis
    - Complete understanding of code is necessary

Ensuring Hard Deadlines

- Deep understanding of code
- Avoid non-deterministic behavior
    - Even if they may speed things up
- Turn off interrupts
- Scheduler is non-preemptive
- Pre-defined schedule
    - No runtime decisions
- Success depends on your scheduler

Soft Real Time Deadlines

- Highly desirable to meet deadlines but deadlines can be missed
- Goal: avoid missing deadlines
- May have different classes of deadlines
    - Some are “harder” than others
- May require less analysis

What Happens When You Don’t Meet a Deadline?

- Depends on system
- May drop the job
- May allow system to fall behind
- May drop jobs in the future

What Algorithms Do You Use For Soft Real Time?

- Earliest Deadline First algorithm
- Each job has a deadline
    - Based on a common cock
- Queue sorted by these deadlines
- Prune the queue/remove jobs when deadlines are missed
- Minimizes *total lateness*

Preemptive Scheduling

- Thread or process is chosen to run
- Run until it yields or until the OS decides to interrupt it
    - **Should not affect the process instructions or results**
    - Forces us to be able to have **context switchessh**
- Another process is run in its place
- Later, the interrupted process is restarted

Implications of Forcing Preemption

- Processes **can be forced to yield at any time** (can be interrupted at any time)
    - If a more important process is ready
    - If a running process’ important is lowered
- Interrupted process may not be in a **“clean” state**
    - Timing may be inconvenient BUT, you still have to save the state

  - Complicates **saving and restoring the state**
- Enables **"fair share"** scheduling
- Introduces gratuitous **context switches** (based on dynamics of process, will be extraneous)
- Creates **potential resource sharing problems**
  - Switching problems back in first but they may want to share a resource

<u>Implementing Preemption</u>

- Need to **get control away from the process** without cooperation from the process
  - We can guarantee interrupts **using a clock**
- Need to consult scheduler before returning to a process
- Scheduler finds highest priority ready process

<u>Clock Interrupts</u>

- Clock
- Generates an interrupt at a fixed time interval
- Halts running processes
- Ensures that runaway process doesn't keep control forever

<u>Round Robin Scheduling Algorithm</u>

- GOAL: **fair share scheduling**
  - All processes are offered an **equal share** of the CPU with an **equal amount of time**
  - Experience **similar queue delays**
- Processes assigned a **time slice**
  - Usually same sized slice for all processes
- Each process is **scheduled**
  - They each run until they block or until they used up the entire time slice
  - Then they are placed at the end of the queue
- Eventually all processes get to run

<u>Properties of Round Robin Scheduling</u>

- Processes have a **relatively quick change** to do some kind of computation
  - Longer time slice → more things done
  - Cost: processes not finished as quickly
  - Win for interactive processes
- **More context switches** that can be expensive
- Runaway processes do little harm

<u>Comparing Round Robin to FIFO</u>

- More expensive (more context switches)
- Longer to complete some processes
- More responsive

<u>Choosing a Time Slice</u>

- Performance depends on time slice
- **Long time slices**: avoid context switches
  - Context switches waste cycles

    - Better throughput and utilization
- **Short time slices**: better response time to processes
- We need to balance this, but how? → Based on the context switch which is defined by the OS

Costs of a Context Switch

- Entering the OS: taking the interrupt, saving registers, calling the scheduler
- Cycles to **choose who to run**: scheduler/dispatcher does work
- **Changing the OS context**: switching the stack, non-resident process description information
- **Switching process address space**
- **Losing instruction** and **data caches**
    - Slow down the next hundred or so instructions

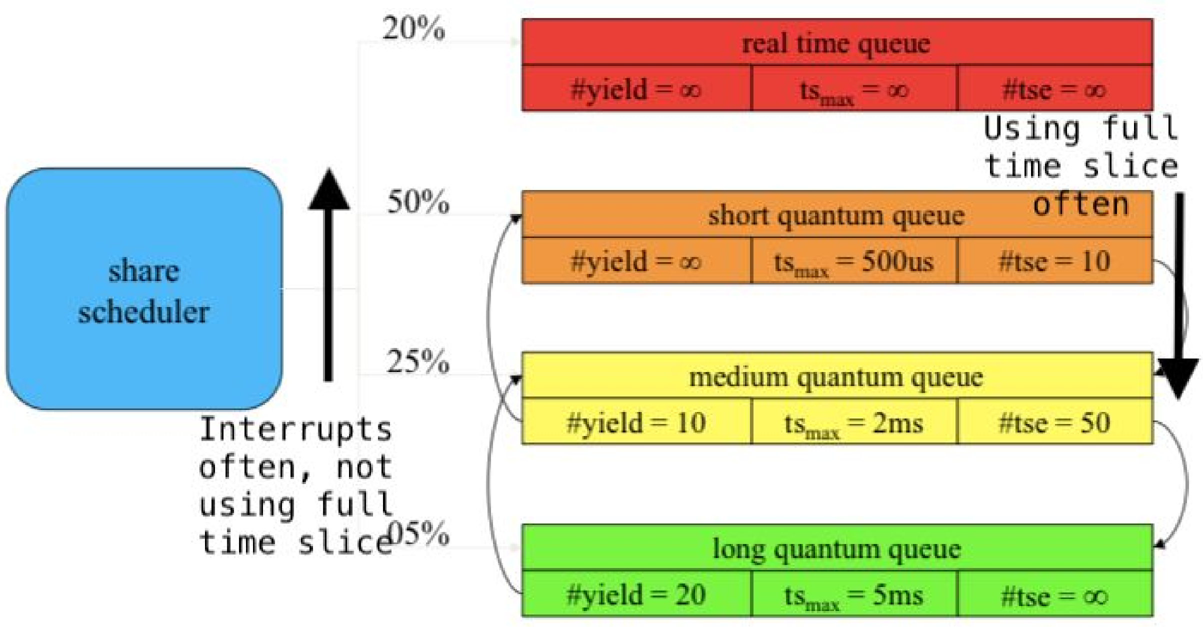
Multi-Level Feedback Queue (MLFQ) Scheduling

- A single time slice may not suit all running processes
- Instead, **choose time slices dynamically** among **dynamically preloaded time slices**
- Create multiple ready queues (with different time slices for each queue)
    - **Short quantum** (foreground) tasks that finish quickly
        - Short but frequent time slices to optimize response time
    - **Long quantum** (background) tasks that run longer
        - Longer, but infrequent time slices, minimize overhead
    - Different queues may get different shares of the CPU
- **Finds a balance** between good response time and good turnaround time
- GOAL: move processes into correct queue automatically based on needs

How Do I Know Which Queue To Put a New Process Into?

- Start all processes in short quantum queue
    - Move **downwards** if too many time-slices end
        - If it uses all of its time-slice consistently
    - Move **upwards** if too few time-slices end
        - If it doesn't use up time-slice consistently
    - Process will **dynamically find the right queue**
    - If **most of the time** it doesn't use the full time slice, it is in a good position
- With real-time tasks, you already know where it belongs (soft real time)
    - Start them in the real-time queue and don't move them

Multiple Queue Scheduling



What Benefits Do We Expect From MLFQ?

- **Acceptable response time** for interactive jobs
    - Jobs with regular external inputs
    - Don't want to wait too long before they are scheduled
    - Also won't waste too much time when they are scheduled
    - Scheduled **fairly frequently** but with smaller time slices
- **Efficient but fair use of CPU** for non-interactive jobs
    - Run for a long time slice without interruption
    - Typically computation heavy jobs
    - Scheduled in larger quanta queue
- Predictable real-time response
- **Dynamic and automatic adjustment** based on actual behavior of jobs

Priority Scheduling Algorithm

- Some processes have **a higher priority** in the list of tasks
- How do we schedule these processes?
    - Assign priority numbers and run according to these numbers

Priority and Preemption

- In **non-preemptive** scheduling, priority scheduling means ordering based on priority
- Preemptive scheduling is more complicated
    - Might preempt a running process if the priority is higher

Problems with Priority Scheduling

- Possible starvation
    - Stream of high priority requests can cause starvation
- May adjust priorities

    - Processes that have run for a long time may have their priority **temporarily lowered**
    - Processes that have not been able to run may have their priority **temporarily raised**
- Have a temporary priority to ensure that starvation doesn't happen

Hard Priorities vs. Soft Priorities

- Higher priority has absolute precedence - hard priority
- Higher priority has larger share of resources - soft priority

Priority Scheduling in Linux

- Each process has a priority called a **nice value**
    - Soft priority value describing share of CPU a process gets
- Commands can be run to change priority
- Anybody can lower priority
- Only **privileged user** can request higher (superuser)
- **Super user =/= supervisor**
- Super user: running in a mode with a little more privilege
- Supervisor: OS code can be run

Reading

Arpaci-Dusseau Ch. 7 - Scheduling: Introduction

- **Scheduling policies** (disciplines) - how do we develop this?

7.1 Workload Assumptions

- Workload: process running in the system
- Assumption → help develop a full-operational scheduling discipline
- Assumptions about processes: (AKA jobs)
    1. Each job runs for the same amount of time
    2. All jobs arrive at the same time
    3. Once started, all jobs run to completion
    4. All jobs only use the CPU (no I/O)
    5. Runtime of each job is known

7.2 Scheduling Metrics

- Enable us to compare different scheduling policies
- Single metric: turnaround time - time at which the job completes - time it entered the system
    - Assume all jobs arrive at the same time so turnaround time = completion time
- Turnaround time - performance metric
- Jain's Fairness Index - fairness metric
    - Both at odds with each other

7.3 First In, First Out (FIFO)

- Most basic algorithm - first come, first served
- Positives:
    - Simple, easy to implement
    - Works well without assumptions
- BUT, if we relax assumption (1), each job has a variable amount of time
    - PROBLEM: **the convoy effect**: relatively short consumers get queued behind heavyweight consumers

7.4 Shortest Job First (SJF)

- **Shortest Job First (SJF): runs the shortest job first, then the next shortest, etc.**
    - Optimal scheduling algorithm
- If we relax assumption (2) → jobs can arrive at any time
- If a long job arrives before short jobs arrive, the convoy effect problem appears again

7.5 Shortest Time-To-Completion First (STCF)

- Relax assumption (3) - all jobs run to completion
- Scheduler can **preempt** a longer job and decide to run another job and continue A later
    - If B and C arrive and they require less time than A, STCF can **preempt** job A and decide to run another job
- SJF: non-preemptive scheduler
- If we add preemption to SJF → **shortest time-to-completion first (STCF)**
- **Determines which remaining job has the least amount of time left and schedules that one whenever a new job enters the system**

7.6 A New Metric: Response Time

- STCF works if we know job lengths, if jobs only used the CPU, and the only metric is turnaround time
- If we add a new metric, **response time**: the time the job first arrives to the first time it is scheduled
- STCF - good for turnaround, bad for response time
    - Another job may have to wait for the first 2 to finish completely
- Build a scheduler sensitive to response time

7.7 Round Robin

- **Round robin scheduling**: instead of running to completion, run a job for a **time slice/scheduling quantum** and then switch
    - Repeats until all jobs are completed
- AKA time slicing
- Shorter time slices - better response time
    - Problematic: **cost of context switching** hurts performance
    - HIGH OVERHEAD
    - We need to reduce the initial cost of switching
- Bad metric for turnaround time - stretches out jobs for as long as possible
- **Policies that are fair → usually perform poorly on turnaround time**

7.8 Incorporating I/O

- Relax assumption (4)
- Scheduler has decision when job requests I/O operation
    - “Blocked” while waiting for I/O completion
- Scheduler decides after I/O is complete whether to run that job again
- Common approach: treat each **I/O request** and **computing request** separately
    - Can schedule the I/O and computing of another job at the same time - **overlap**

7.9 No More Oracle

- OS doesn’t actually know how long each job will take, we relax assumption (5)
- We can build a scheduler
- **Multi-Level feedback queue**

Arpaci-Dusseau Ch.8 - Scheduling: The Multi-Level Feedback Queue

- **Multi-Level Feedback Queue (MLFQ)**: tries to optimize turnaround time and minimize response time

8.1 MLFQ: Basic Rules

- MLFQ has different queues, each with a different priority level
    - Priorities used to decide which job to run at a given time
- Two basic rules:
    - **RULE 1: If Priority(A) > Priority(B), A runs (B doesn’t).**
    - **RULE 2: If Priority (A) = Priority (B), A & B run in Round Robin**
- MLFQ sets priority by varying it based on **observed behavior**
    - It tries to learn about processes and use past behavior to predict future behavior

8.2 Attempt #1: How to Change Priority

- Decide how MLFQ changes priority level of a job
    - **RULE 3: When a job enters the system, it is placed at the highest priority**
    - **RULE 4a: If a job uses up an entire time slice while running, its priority is reduced.**
    - **RULE 4b: If a job gives up the CPU before the time slice is up, it stays at the same priority level.**
- Goal of algorithm: it doesn’t know whether a job will be long or short
    - First **assumes that it will be short** and gives it a high priority
    - If it isn’t a short job, it **moves slowly down the queue**
- But what about I/O?
    - RULE 4b - if a process gives up the processor before the end of the time slice, we keep it at the same level
    - Intent of this rule
        - If a job is doing I/O, we don’t want to punish it
- Problems with our current MLFQ:

1. **Starvation**: if there are "too many" interactive jobs, they will consume all of the CPU
    - **Prevent long-running jobs from receiving CPU time,** starving them
2. **Gaming the Scheduling**: can trick the scheduler by having I/O operations occur before the time slice is over
    - Allows you to remain in the same queue → get more CPU time
3. Changing Behavior: program may transition

<u>8.3 Attempt #2: The Priority Boost</u>

- Avoid issue of starvation
- Periodically **boost** the priority of all jobs in the system
    - **RULE 5: After some time period S, move all jobs in the system to the topmost queue**
- Solves 2 problems
    - Guarantees that processes will not starve because in the top queue, there is RR among all processes in that queue
    - If a CPU-bound job is inactive, it will get a priority boost
- What should S be set to?
    - High - long running programs could starve
    - Low - interactive jobs may not get CPU

<u>8.4 Attempt #3: Better Accounting</u>

- Prevent gaming of the system with better accounting of the CPU time at each level of the MLFQ
    - Scheduler keeps track of how much of a time slice a process as used
    - **RULE 4: Once a job uses up its allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced**

<u>8.5 Tuning MLFQ and Other Issues</u>

- Issues with MLFQ scheduling
    - How do we parametrize a scheduler?
        - How many queues, how big are the time slices, etc.
- MLFQ: allows for different time slices at each level
- Some reserve high level for OS work

Kampe: Real-Time Scheduling

<u>Introduction</u>

- Priority scheduling: sometimes the best effort approach is not enough

<u>What are Real-Time Systems?</u>

- **Real time systems**: one whose correctness depends on timing as well as functionality
- Characterized by different metrics:
    - **Timeliness**: how closely does it need to meet its timing requirements
    - **Predictability**: deviation in delivered timeliness
- New concepts related:

    - Feasibility: whether it is possible to meet these requirements
    - Head real-time: stron requirements that specified tasks be run at specified time intervals
        - Failure may result in system failure
    - Soft real-time: good response time
        - Consequences are degraded performance or recoverable failure
- May be easier to perform scheduling
    - We know the length/duration of each task
    - Starvation is acceptable
    - Workload is fixed

<u>Real-Time Scheduling Algorithms</u>

- Simple real-time systems - no scheduler needed
- Some require **dynamic** scheduling algorithms → two key questions
    1. How they choose the next ready task
        - Shortest first
        - Static priority
        - ASAP - soonest start time deadline
        - Soonest completion time (slack time)
    2. How they handle overload (infeasible requests)
        - Best effort
        - Periodically adjust - lower priority less often
        - Work shedding - stop lower priority tasks
- Preemptive scheduling: beneficial in ordinary systems
    - May not be in real-time scheduling
    - Preempting a task may cause it to miss its completion deadline
    - No need
    - Infinite loop bugs are rare

<u>Real-Time and Linux</u>

- Linux: not designed to be embedded or real-time

Basic Memory Management

Lecture 5: Memory Management

Outline

- What is memory management?
- Strategies:
    - Fixed partition strategies
    - Dynamic partitions
    - Buffer pools
    - Garbage collection
    - Memory compaction

Memory Management

- Key assets in computing
- **Memory abstractions** that are used by running programs (RAM)
    - RAM: attached to BUS
    - You can only **execute instructions that are in RAM**
    - Stack must be in ram
    - Low level instructions (%mov) involve RAM
- Limited amount of memory
    - What the hardware provides
- But all processes need to use it
    - Greedy, not enough RAM to satisfy all of our processes
- Question: how do we manage it?
    - Limited resource

Memory Management Goals

1. Transparency
    - We want **privacy among processes**
    - Hide the face that RAM is being shared – processes should think they each have their own RAM
    - A process can only see **its own address space** and is **unaware that memory is shared**
2. Efficiency
    - High effective memory utilization
    - RAM is a critical resource, make sure it is being used effectively and not being wasted
    - We want a **low run-time cost** for allocation/reallocation
        - Cheap allocation means **low overhead**
3. Protection and Isolation
    - Private data **is not corrupted** and **cannot be seen by other processes**
    - Ensure that other processes cannot see your data or modify it

Physical Memory Allocation

- Divided between: OS Kernel, Process private data, shared code segments

Physical and Virtual Address

- Important distinction
- Cells of RAM have **particular physical addresses**
  - Previously used to name RAM location
  - Processes had been using the **physical address**
- Instead, we have processes use **virtual addresses**
  - Process are issued an address that **isn't the real physical address** → a **translation** is required
  - May not be the same as physical addresses
- Provide **more flexibility** in memory management but require a **virtual to physical translation**
  - No longer tied to a physical address
  - We can put things in memory where the process doesn't expect
  - Translation - requires hardware help

A Linux Process' Virtual Address Space

- Process thinks it has the full address space
- Illusion that all segments are present in memory when the process runs
- Black space is unused and still belongs to the process
- Virtual address space does not contain OS or other process segments

Memory Management Strategies

- Fixed partition allocations
- Dynamic partitions
- Relocation

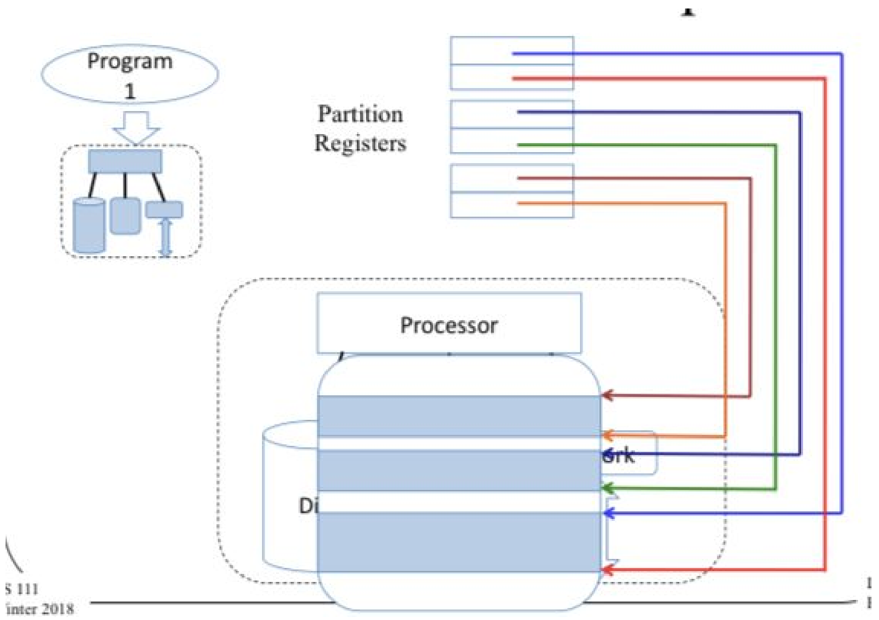
Fixed Partition Allocation

- **Pre-allocate partitions** for *n* different processes
  - One or more are given to each process when they ask for it
  - They reserve space for the largest possible process
- Partitions may come in **one or a few set sizes**
- This is easy to implement
  - Can be managed with a bit map
  - allocation/deallocation is cheap and easy

Memory Protection and Fixed Partition

- We need to **enforce partition boundaries**
  - Use hardware typically
  - Ensure that process A & B don't access each other's memory
- Use the hardware
  - **Partition registers:** Special registers that contain the partition boundaries
  - We **only accept addresses within the register values**
- Basic scheme: doesn't use virtual addresses

The Partition Concept



- The **partition registers** are a part of the CPU
    - Check here to see if the memory address is within the boundaries of the given memory
    - Point to **the beginning and end of each partition**
- Program shares everything but it **looks like it has its own**

Problems with Fixed Partition Allocation

- Presumes you **know how much memory will be used** ahead of time
- Limits of processes supported
    - Cannot support more processes than we have memory
- Not good for sharing memory
- **Fragmentation** will cause inefficient memory use
- We cannot run a process that **uses more memory/RAM** than we actually have

Fragmentation

- Ineffective use of the memory have
- Some memory is being wasted
- Problem for all memory management systems
    - **Fixed partitions suffer especially badly**
- Based on process **NOT using all the memory they requested**
    - The process requests some memory in a fixed size but doesn't use all of it → the unused memory is wasted
- Result: you can't provide memory for other processes/as many processes as you theoretically could
- **Memory wasted** because it cannot be used by another process unless it is released
- Say process A requests 6MB but your chunk size is 8MB
    - You must give 8MB and the 2MB given to process A is wasted because **only A can use that memory**

Internal Fragmentation

- Two kinds of fragmentation
  - Internal and external
- **Internal fragmentation**: given a fixed size allocation, the difference between what is needed and what is given is wasted
- **Wasted space in fixed size blocks**
  - Requester given more than needed
  - Unused part is wasted
- **Internal fragmentation** occurs whenever you have **fixed-size partition** allocation

More on Internal Fragmentation

- Caused by a **mismatch** between
  - Chosen size of fixed-size blocks
  - Actual sizes that programs use
- Average waste: 50% of each block
- Waste can be reduced by having more sizes → more overhead

Summary of Fixed Partition Allocation

- Very simple - low overhead
- Inflexible - not many choices
- Internal fragmentation
- Not used in modern systems but sometimes used in special purpose systems

Dynamic Partition Allocation

- Like fixed partitions except:
  - Variable sized, **usually any size requested**
  - Each partition is **contiguous** in memory (restriction)
  - Potentially can be **shared** between processes
  - Partitions have **access permissions**
- Each process can have multiple partitions
  - With different sizes and characteristics
- Give processes **exactly what they ask for** → helps to deal with internal fragmentation

Problems with Dynamic Partitions

- **Not relocatable**
  - Partitions cannot be moved
- They cannot be expanded
- Impossible still to support applications with larger address spaces than physical memory
  - Cannot support applications whose total needs are greater than physical memory
- Still subject to fragmentation

Relocation and Expansion

- Partitions are **tied to particular physical address ranges**
- We cannot just move the contents of a partition to another set of addresses

    - (ex: when a stack needs more space)
    - If we copy all pointers, it is WRONG, **they still point to the od partition in memory**
    - Pointers in the contents will be wrong
    - Generally, we can't tell which memory locations have pointers
        - No memory "tags"
- We cannot expand because **there may not be space nearby**

The Expansion Problem

- Partitions allocated on request
- Processes can ask for new ones later
- BUT, partitions that have been given **cannot be moved**
- Memory management system may have also allocated all space before or after a partition
    - Cannot expand
- Problematic in fixed-size allocation as well, **we cannot move chunks of data around**
- Say process A and B lay next to each other
    - If A wants to expand its size, it will step on process B's memory and start trying to access B's memory
    - Implications on correctness, BAD

How To Keep Track of Variable Sized Partitions?

- Begin with a large "heap" of memory
- Maintain a **free list**
    - Lists the free memory to be allocated
    - Keeps track of unallocated memory
- Go through the list when a process requests more memory
    - We find a large chunk of memory
    - Carve of the requested memory
    - Put the remainder back
- When a process frees memory, we place it back on the list

Managing the Free List

- Fixed sized blocks - easy to track using a bitmap
- Variable chunks require more information
    - Linked list of **descriptors**, one/chunk
    - Descriptors: list size of the chunk and whether it is free
    - Each has a **pointer to the next chunk** on the list
    - Descriptors are often kept at the front of the chunk
- Allocated memory may also have descriptors

Free Chunk Carving

1. Find a large enough free chunk
2. Reduce its "len" to the requested size
3. Create a new header for the residual chunk
4. Insert the new chunk into the list

5. Mark the carved piece as in use

Variable Partitions and Fragmentation

- Not as subject to internal fragmentation
    - Unless a requester asks for more than needed (common)
- Subject instead to **external fragmentation**

External Fragmentation

- Gradually build up **small, unusable** memory chunks scattered through memory
- A **gradual accumulation** of small, unusable chunks of memory in the address space

External Fragmentation: Causes and Effects

- Allocation creates **leftover chunks** that become smaller and smaller
    - Eventually so small they cannot be used for any allocation requests
- Small leftover fragments are useful
- Solutions:
    - Try not to create tiny fragments
    - Try to recombine fragments

How to Avoid Creating Small Fragments?

- Be smart about which free chunk of memory you use
- Smartness costs time
- Algorithms to try and avoid creating small fragments:
    - Best fit
    - Worst fit
    - First fit
    - Next fit

Best Fit

- Search for the “best” fit chunk
    - Smallest size **greater than or equal to** requested size
- Advantages:
    - Finding a perfect fit is possible
- Disadvantages/Problems:
    - Have to **search entire list**
    - Creates **small fragments**
    - How do you know which fit is best? → expensive overhead

Worst Fit

- Search for the “worst fit” chunk
    - **Largest size greater than or equal to requested size**
- Advantages:
    - Creates **very large fragments** for a while (still end up with small segments)
- Disadvantages
    - Still have to **search the entire list every time**

First Fit

- Find the **first chunk** that is big enough

- Advantages:
    - Short search, low overhead (for a while at least)
    - Creates random sized fragments
- Disadvantages:
    - First chunks **fragment quickly** (any time you carve from there)
    - Searches **eventually become longer**
    - Ultimately, **as bad as best fit**

Next Fit

- After searching, we set the **guess pointer** to the chunk after the one that was chosen
- **Next fit** starts the search at a different place every time, **where the special "guess pointer" points to next**

Next Fit

- Tries to get **advantages of first and worst fit**
    - Short search
    - Spread out fragmentation
- **Guess pointers**: general technique
    - Lazy (non-coherent) caches
    - Save a lot of time if right
    - If wrong, algorithm still works
    - Applicable in many problems
    - Spreads out searches

Coalescing Partitions

- All variable sized partition allocation have external fragmentation
    - Spreading out fragments means that small fragments are not in one place which would cause an expensive search
- We need to **reassemble fragments**
    - We can **check neighbors when chunks are free**
    - **Recombine the free neighbors** if possible
    - The free list can help with this
        - Can use the free list and keep memory in order of addresses
        - We can then combine the neighbors into a larger, more useful segment
- **Counters** external fragmentation

Fragmentation and Coalescing

- Opposing processes that operate in parallel
- Fragmentation gets worse with time
- Chunks held for a long time - cannot be coalesced
- Coalesced works better with more free space
- Highly variable chunk size requests increase the fragmentation rate

Variable Sized Partition Summary

- Eliminates **internal fragmentation**
- Implementation is **more expensive**

    - Long searches
    - Carving and coalescing
- External fragmentation **is inevitable**

Another Option

- Strike a balance between fixed partition allocation and dynamic partition allocation
- Fixed partition allocation: internal fragmentation
- Dynamic partition allocation: external fragmentation

Special Case for Fixed Allocations

- Internal fragmentation occurs due to a mismatch in chunk size and request size
- Allocation requests **aren't random at all**
- There exist special cases where certain sized chunks are requested more often than others (spikes)
    - **Set aside a chunk of memory and do FIXED PARTITION ALLOCATION for these specific sizes**

Memory Request Sizes Aren't Random

- Some sizes **request more often than others**
- Use a **fixed-size buffer** often
    - Handle these cases specifically
- No need to do variable sized partitions for these fixed sizes
    - Simpler, no fragmentation
    - Low overhead
    - Doesn't solve full problem but helps
- All modern OS's do this → have a **buffer pool with buffers of fixed sizes**

Buffer Pools

- For popular sizes:
    - Reserve special pools of **fixed size buffers**
    - And satisfy **matching requests** from these pools
- Only **allocated when the EXACT size** is requested
- Benefit: improved efficiency
    - Simpler than variable partition allocation
    - Reduces external fragmentation
- We need to know how much to reserve for the pool
    - Too little: bottleneck
    - Too much: unused buffer space
- We **only** satisfy perfectly matching requests

How Are Buffer Pools Used?

- Process requests a piece of memory for a special purpose (may implicitly request one of the special sizes)
- System supplies an element from the buffer pool
- Process uses it, completes, and frees the memory
    - Reclamation is simple

Dynamically Sizing Buffer Pools

- Benefits: things are not constant!
- Dynamically change the buffer pool size
- If we **run low** on fixed size buffers
    - Get more memory **from the free list**
    - We can **carve it up into more fixed size buffers**
    - **There is no need for continuity**
- If our free buffer list **is too large**
    - Return some buffer to the free list
- If the free list gets **dangerously low**
    - Ask each major service with a buffer pool **to return space**
- Tuned by parameters:
    - Low space threshold (need more)
    - High space threshold (too much)
    - Normal allocation

Lost Memory

- Occurs with **buffer pools** and **self** allocation
- Problem: **memory leaks**
    - Process doesn't free memory when done with it
- Solution: **garbage collection**

Garbage Collection

- Solution to memory leaks
- We don't rely on processes to release memory
- Monitor our free memory
- When we run low - start to collect garbage
    - Search data for object pointers
    - Not the address and size
    - Compute the compliment
    - Add inaccessible memory to the free list

How Do We Find Accessible Memory?

- Object oriented languages enable this
- System resources

General Garbage Collection

- What do you do?
- **Find all pointers (difficult)**
- Determine "how much" and what is and is not being pointed to
- Free what isn't pointed to
- This is difficult, why?

Problems with General Garbage Collection

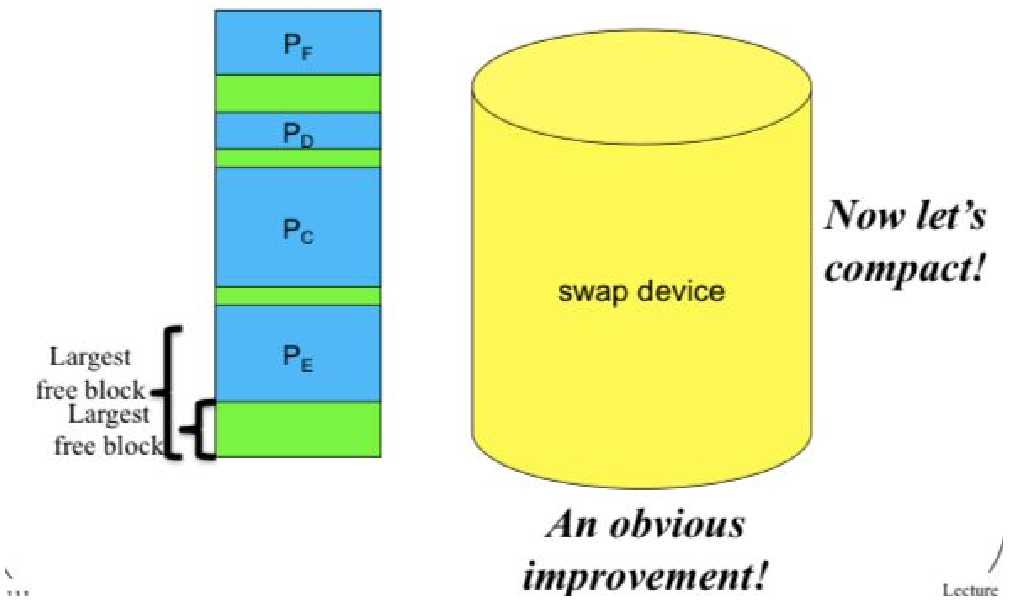
- Location **may seem** to contain addresses but:
    - Are those really pointers?
    - Are they still accessible if they are pointers?
    - We can infer this in dynamically allocated structures but what about statically allocated areas?

- How much is “pointed” to?

Compaction and Relocation

- Garbage collection just one way
- Ongoing activity starves coalescing
- Halting memory allocations hurts throughput
- Need to **rearrange active memory**
    - Repack all process memory into one part of memory
    - Create a large free chunk at the other end

Memory Compaction



- Compact Processes F, D, C, and E into one large chunk, leaving a large free block at the end
- **Swap device**: storage that has a LOT of memory (not RAM), slower than RAm
    - We copy the partitions into the **swap space** and then we can **copy them back** into the memory, putting them all together

All This Requires is **Relocation**

- Ability to move a process
    - From where it was initially loaded
    - To a new and different region of memory
- Problem: moving memory without changing the pointers means that:
- **All addresses in the program will be WRONG**
    - Reference in the code segment
    - New pointers created during execution

The Relocation Problem

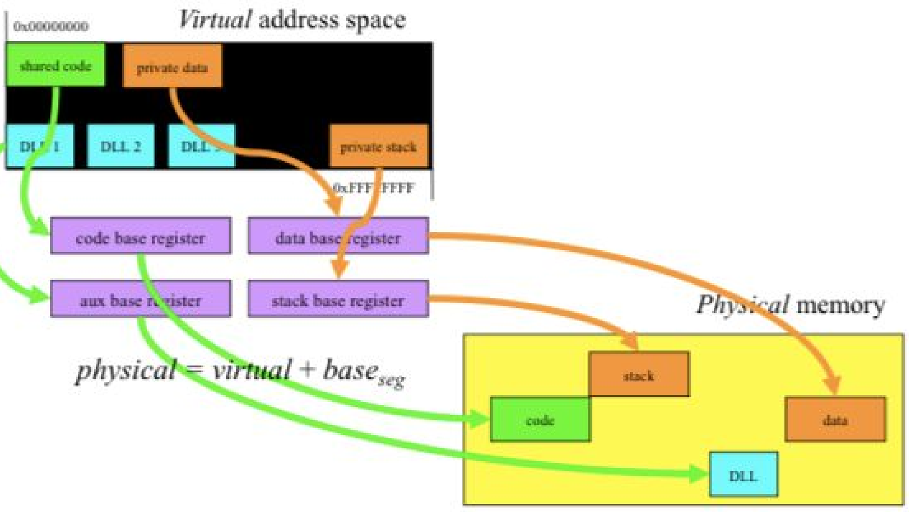
- Generally **not feasible** to relocate a process
    - Could **relocate references to code** because we know what it is at load time

- But **relocating references to data is challenging**
  - Pointer values may be changed
  - New pointeres may have been added
- Can't find and fix all address references
- Instead, **can we make processes location independent?**
  - Make processes location independent
  - We can use the **virtual address space and physical address space** to do this

Memory Segment Relocation

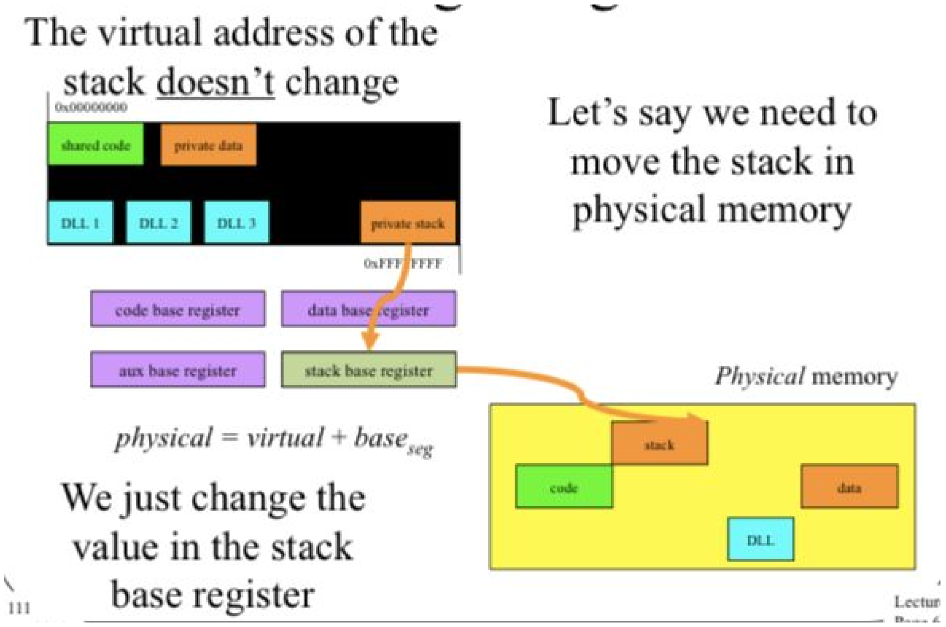
- Natural model:
  - Process address space is made of multiple segments
  - Segments are the main unit of relocation
- Allow relocation of segments by use of a translation unit
- The computer has **special relocation registers** built into the hardware
  - Called **segment base registers**
  - Point to the **start (in physical memory) of each segment**
  - CPU automatically adds a base register to every address
- OS uses the **segment base registers** to perform virtual address translation
  - We can set the base register to the start of the region when the program is loaded
  - **If the program is moved, reset base register to the new location**
  - Allows program wo work no matter where its segments are located
- We only need to change the base registers to move processes and alter the translation

Segment Relocation



- Relocation registres: code base registers, data base registers, aux base registers, stack based registers
- Each points to different words in physical memory

- Offsets are preserved → in the middle of the private data, it may map to the middle of the translated data



- Changing the **value in the stack base registe**r allows the program to still work because the translation still works
- We can change the location of the stack in physical memory and it will still work

Relocation and Safety

- Relocation mechanism (base registers) is good
    - Solves relocation problem
    - Enables us to move segments in physical memory
    - Insufficient
- Need protection
    - We need to know the end of the segment
    - Prevent processes from reaching outside of its allocated memory
- Segments also need a **length (or limit) register**
    - Maximum legal offset
    - Any address greater than this is illegal
    - CPU reports it as a segmentation fault

How Much of Our Problem Does Relocation Solve?

- Variable sized partitions cut down on internal fragmentation
- Move partitions around
    - Helps coalescing be more effective
- Still require continuity/contiguous chunks - external fragmentation is still a problem

Reading

Arpaci-Dusseau Ch. 13 - The Abstraction: Address Spaces

13.1 Early Systems

- Not much abstraction
- Physical memory addresses were used by programs

13.2 Multiprogramming & Time Sharing

- Multiprogramming - multiple processes ready to run at a given time
    - OS switched between them
    - Increased virtualization of the CPU
    - Increased efficiency
- Timesharing emerged and interactivity became important
- Want to run multiple processes and have them all stay in memory
- Also want to ensure protection

13.3 The Address Space

- **Address space**: abstraction of memory created by the OS
    - A running program's view of memory
- **Address space of a process**: contains all of the memory state of the running program
    - Including the code, stack, and heap
    - Code: static, easy to place
    - Heap & stack: **may grow and shrink** as the program runs, so, put them **both at opposite ends and have them grow in opposite directions**
- Each process is loaded at a different address (OS is abstracting memory)
- Processes share memory

13.4 Goals

- Job of the OS is to virtualize memory
- Goals of virtual memory
    1. Transparency: invisible to running programs
    2. Efficiency: virtualization should be as efficient as possible (both time and space)
    3. Protection: protect processes form each other and protect the OS itself, we want to provide the process with isolation

Arpaci-Dusseau Ch. 14 - Interlude: Memory API

- Memory allocation systems in Unix

14.1 Types of Memory

- Two types allocated in C:
    - Stack - managed by the compiler (automatic)
    - Heap - managed by the programmer

int *x = (int *) malloc(sizeof(int));

- Two allocations happening here
  - Stack allocation of a pointer
  - Heap allocation of an integer

14.2 The malloc() call

```
#include <stdlib.h>
void* malloc (size_t size);
```

14.3 The free() Call

```
int* x = malloc(10*sizeof(int));
...
free(x);
```

Arpaci Dusseau Ch. 17 - Free Space Management

- Issues surrounding free space management
  - Difficult when you have user memory allocation and in OS's using segmentation to implement virtual memory
- External fragmentation: free space gets chopped up into little pieces of different sizes and is fragmented
  - Results in a lack of single contiguous memory spaces

17.1 Assumptions

- Assume basic malloc() and free(0 interface from the memory allocation library
- Library manages the heap - which is managed by a free list that contains references to all free chunks of memory
- Our focus: external fragmentation
- Assume: once memory is given to a client it cannot be relocated and no compaction is available
- Allocator is assumed to manage a contiguous region of a fixed size

17.2 Low level Mechanisms

1. Splitting and coalescing
2. Tracking size of allocated regions
3. Building a simple list inside of free space

→ Splitting and Coalescing

- We can split regions of memory that satisfy the memory request
- We can coalesce the free space when memory is free
  - When we return a free chunk, we check if it sits near any other free space, then we merge

→ Tracking the Size of Allocated Regions

- free() doesn't take any size parameters because it can detect the size of a region
- The allocator stores size information in a **header** clock in memory, usually right before the allocated region

- free() frees the allocated region and the header
    - Allocated region is N bytes
    - **free() will release N + headerSize bytes**

→ Embedding a Free List
- We can build this inside the space itself

→ Growing the Heap
- What do we do when the heap is full?
    - Fail
    - Or use sbrk to **request more heap**

17.3 Basic Strategies
- Basic strategies for managing free space

→ Best Fit
- Search through the free list and find chunks >= than requested
- Return one that is smallest in that group (**best fit chunk**/smallest fit chunk)
- Reduces the wasted space
- Cost: heavy performance penalty by searching

→ Worst Fit
- Find the largest chunk and return the requested size
    - Keep the remaining large chunk in the list
- Tries to leave big chunks instead of small ones
- Cost: full search required, excess fragmentation

→ First Fit
- Finds the first block that fits
- Speed advantage
- Pollutes beginning of free list with small chunks
- Ordering issue
- Address-based ordering - list ordered by address

→ Next Fit
- Adaptation of **first fit**
- Keeps pointer at location **where you were last looking**
- Spread the search out through the list - creates a uniform allocation
- Similar in performance to first fit

17.4 Other Approaches

→ Segregated Lists
- If a particular application has one (or a few) popular size requests, we can keep a **separate list to manage objects of that size**
- Benefits: fragmentation is less of a concern, faster allocation for certain sizes
- Challenges: how much memory to allocate to that special pool

    → Slab Allocator
    - Object caches: segregated free lists
        - Request slabs of memory when it is running low

        - These slabs are reclaimed when the reference count of objects in a **slab** all go to 0
    - Avoids frequent initialization and destruction per object

→ Buddy Allocation

- "Binary buddy allocation"
- Makes coalescing simple
- **Binary buddy allocator** - space is 2^N bytes large
- Retrieves memory by recursively dividing by 2 until a large enough space is formed
- **Internal fragmentation happens often**, allocating too much memory is possibly
- **Coalescing is easy** - Say, when 8KB is free, we can check if the "buffy 8KB" block is free and coalesce them into 16KB
    - This can continue up the tree
- Still causes **external fragmentation** if the buddy is not free

→ Other Ideas

- Major problem is scaling: a list is too slow

Kampe: Garbage Collection and Defragmentation

Introduction

- Resources → addressing serially reusable resources
1. Garbage collection: recycling unused resources
2. Defragmentation: reassigning and relocating resources to eliminate external fragmentation

Garbage Collection

- How do allocated resources return?
    - Freed explicitly/implicitly by the client
    - Resource manager references count for each resource
        - We increment when obtained
        - Decrement when released
        - Free when the count is 0
- Not always practical
    - Os may not know about some resources returning
    - Resources may be kept track of automatically
    - Some resources are never freed and are automatically recycled
    - allocation/release → so frequent such that there is overhead

Garbage Collection in a System

- Resources allocated but not explicitly freed
- When resources are scarce, we can initiate resource collection
    1. List all resources originally in existence
    2. Find all reachable resources
    3. If it is reachable, remove it
    4. At the end of the scan, free unused resources

5. Resume program
- Works only if we can identify all of the active resources
- Cost: garbage collection may stop system, this must be synchronized

<u>Defragmentation</u>

- External fragmentation
- Changes which allocated are allocated
- Moving items around to create more contiguous free space
- Swapping

Swapping, Paging, and Virtual Memory

Lecture 6: Swapping, Paging, and Virtual Memory

Outline

- Swapping
- Paging
- Virtual memory

Swapping

- What we can do if we **don’t have enough RAM**
    - To handle all process’ memory needs
    - Or even to handle one process
- Keep the memory somewhere other than RAM, perhaps on disk
- But we cannot directly use code or data on the disk

Swapping to Disk

- Strategy to **increase effective memory size**
- When a process yields, we can **copy its memory to disk**
- When it is scheduled, **copy it back**
- This helps save memory
- Relocation hardware will help put the memory in different RAM locations
- Processes can then see a memory space **as big as total RAM**
- No need to share, processes can have all the RAM while running because of swapping

Downsides to Simple Swapping

- Expensive to move the entire program into the disk
- Cost of context switches become **very high**, we would need to:
    - Copy all of the RAM out to disk
    - Copy all of the stuff from disk into RAM
    - Before the process can even run
- Still **limiting processes** to the **amount of RAM we actually have**
    - Need to meet requirements that can be greater than what we have

Paging

- What we actually do in modern systems
- What is paging?
    - Problems it solves
    - How does it solve these problems
- Paged address translations
- Paging and fragmentation
- Paging memory management units (hardware)
    - **Translation lookaside buffer (TLB)**

Segmentation Revisited

- Segmentation relocation: solved relocation problem
    - Uses **base registers** to computer a physical address from a virtual address
    - Allowed us to move data around in physical memory
    - Only needed to update the base register
- This did **nothing about external fragmentation**
    - Segments still required to be contiguous
- **We need to eliminate "contiguity requirement"**
- Solve a lot of fragmentation problems by allowing us to put things where it fits → PAGING

Segmentation vs. Paging

- Segmentation
    - Arbitrarily sized parts of address space
    - Ranges can be used for different purposes
- Paging
    - **Memory is divided up into pages of similar sizes**
    - Allow more flexibility and relocation
- Segmentation and paging can exist in the same OS
    - Segmentation can indicate whether an address is legal or not
        - Helps you manage which addresses o memory are legal
    - Paging can allow us to map the pages in virtual memory to pages in physical memory

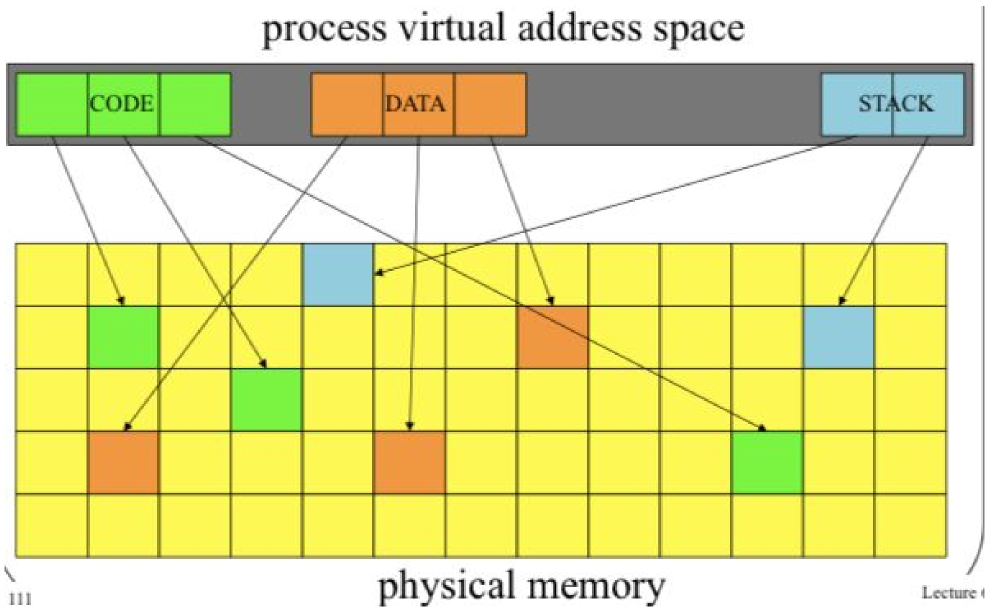
      → we can have **segments of memory** that **contain pages**

The Paging Approach

- Divide physical memory into **units of a fixed size**
    - Typically pretty small, 1-4 KB bytes, or words
    - Typically called a **page frame**
        - division/units called page frames (**page frames are part of physical memory**)
        - We place **pages** (**virtual memory has pages**)in them

  → virtual memory stores **pages** into **page frames** in physical memory
- The virtual address space is treated the same way
- Each virtual address space page has its data stored in a single **physical address page frame**
- Use a per-page translation mechanism to convert virtual → physical pages
- Translate bytes to pages
- Every **virtual page is stored in a physical page frame**

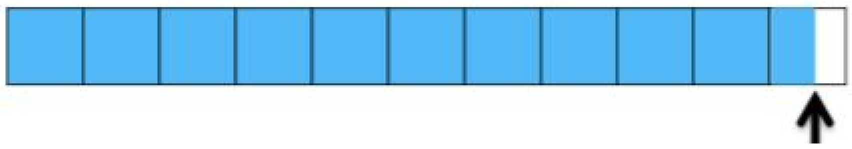
Paged Address Translation



- We can place the pages anywhere we want
- Pages can be scattered in physical memory

Paging and Fragmentation

- A **segment** is implemented as a **set of virtual pages**

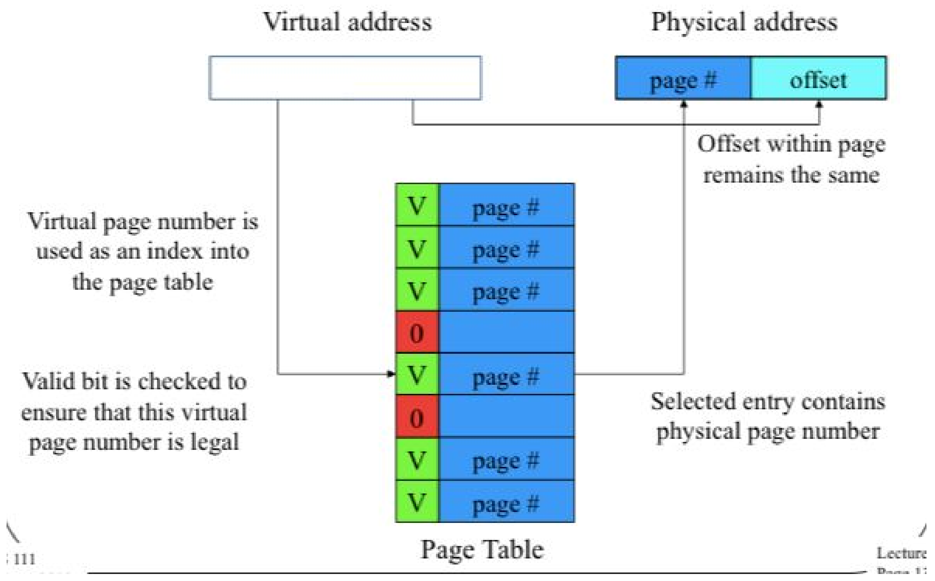


- Internal fragmentation: only averages about ½ a page
    - Sometimes we get an extra page that we requested
    - **Fixed partition (page) allocation** will ALWAYS cause internal fragmentation
    - Here, only the LAST page suffers
- External fragmentation: non-existent
    - We never carve up page
- In segmentation, we had **50% waste**
- Here, we only have **1.5% waste**
    - This is not true all the time
    - In reality, it is much less

Providing Magic Translation Mechanism

- Need to **change a virtual address to a physical address** on a per page bases
- Need to be **fast** – use hardware
- We use the **Memory Management Unit (MMU)**
    - Designed to be very quick
    - Translation and virtual memory placement is done here → faster
    - Contains the TLB

Paging and MMUs


Virtual address
Physical address
page #
offset
Offset within page remains the same
Virtual page number is used as an index into the page table
Valid bit is checked to ensure that this virtual page number is legal
Selected entry contains physical page number
V
0
page #
Page Table
111
Lecture

- The **valid bit** ensures that the virtual page number is legal
  - Valid bit = 0 means the page is **not in memory**
  - For invalid pages, the valid bit is marked to 0
  - **PAGE FAULT** when you try to access a page with an invalid bit
- Offset: tells you which word you want to access from the address
- The virtual address contains:

Page #	Offset

- The **page table** is loaded into the MMU
  - It tells whether a virtual address is valid and it tells you the page number in the physical address
- This all happens automatically by the hardware
  - The software knows nothing about this
  - Only the OS and the hardware know about this
- We cannot issue physical page frame address

The MMU Hardware

- MMUs **used to sit between CPU and bus**
  - TODAY, **integrated into the CPU**
- Page Tables
  - Originally implemented in special fast registers
  - Too expensive
  - Now, we need to have something better: **a cache**

Handling Big Page Tables

- Now, we store big page tables in **normal memory**
  - We now have to get entries for RAM
- But we cannot afford the 2 bus cycles it will require to access this memory
  - One to look up page table entry
  - One to get actual data

    - We would have to:
        - Go into RAM
        - Get the entry
        - Bring back the translatio
        - Do the translation
        - Figure out where you want to go
- We have a **very fast set of MMU registers used as a cache** called the **TRANSLATION LOOKASIDE BUFFER (TLB)**
    - We need to worry about hit ratios, cache invalidation, and other issues
- We have a new set of problems: **cache management**

The MMU and Multiple Processes

- Several processes running and each need a set of pages
- We can place these pages anywhere
- What if we need more pages than we have physically?
    - We can put them in secondary storage but they cannot be used there
        - They become invalid entries = not in RAM but on secondary storage
    - If the OS issues an instruction, we need to go get that page sitting in disk and bring it back into RAM

Ongoing MMU Operations

- Current process adds/removes pages
    - Update active page table
    - Flush stale cached entries
- Switch processes
    - Separate page tables for each process
    - Privileged instruction loads pointers to a new page table
    - Reload instruction flushes (stale) cached entries
- Share pages between processes
    - Page table point to the same physical page
    - Read-only or read/write sharing

Demand Paging

- What is demand paging?
    - Bringing pages from disk into RAM **only when we need it**
- Locality of reference
- Page faults and performance issues

What is Demand Paging?

- Processes don't need **all pages** in memory to run
- Only needs the ones that **it actually references**
- We don't need to load up all pages when we run a process
    - Instead, load only the necessary pages
- Don't need to get rid of all pages when we yield either (swapping)
- We can **move pages onto and off of disk "on demand"**
    - Run what we happen to have in memory
    - If we have everything - OK

    - If we need a page on disk, we can retrieve it at this point

How to Make Demand Paging Work?

- Pages are **either in page frames** (RAM) or **on disk somewhere**
- MMU needs to support "**not present**" pages
    - Generate a **fault/trap** when they are referenced (cause a big performance hit)
    - The OS can **bring in the page** and **retry** the faulted reference
- Entire process doesn't have to be in memory to run
    - Start process with a **subset of its pages**
    - **Load additional pages** as need be
- **BIG CHALLENGE: Performance**
    - A process cannot run when we are retrieving its pages from disk

Achieving Good Performance for Demand Paging

- Disk: slow, we should rarely go here for good performance
- Demand paging will **perform poorly** if most memory reference require disk access
- How do we ensure that we don't access the disk too often?
    - Ensure that **the page holding the next memory reference is in memory**

Demand Paging and Locality of Reference

- We need to predict what pages we need in memory
- We rely on **locality of reference** with respect to virtual memory
    - What you ask for in the future is similar to the requests you've made in the past
    - Chances are, you will use the same page again in the future

Why is Locality of Reference Usually Present?

- **Code** usually is sequential, instructions are typically nearby
    - Code can be placed on one page
    - Branches tend to be short
- We typically need access to things in the current or past **stack frame**
    - Stack shows good locality of reference
    - Stack frame changes only when you enter or leave a subroutine
- **Heap references** to recently allocated structures
    - This depends on the program
- All three types of memory are **likely** to show locality of reference

Page Faults

- Bad for performance but OK for correctness
- The OS **did not keep a page** that you needed in your page frame → **it is in disk → the invalid bit is set on the page table**
    - But, the OS cannot say no to a program
    - Cause a **page fault**
    - The program cannot do anything until the OS gets the address
- Page tables may not contain points of pages to RAM
- Some pages are not in RAM at the moment
- If you request an address at that page, we generate a **page fault**

- Tell the OS to get the page

Handling a Page Fault

- Initialize all page table entries to be **“not present”/invalid**
- How do we know we have a page fault?
    - Is the page is “not present” and it tells you where it is on the disk
- The **CPU will fault** if the “not present” page is referenced
    - The fault enters the kernel, like a regular trap
    - It is forwarded to the **page fault handler**
    - Determines **which page** and **where the page** resides
    - Schedule an I/O to **fetch it** and then **block the process**
    - The page table will now point to the newly read-in page
    - We no can back up the user-mode PC and **retry the failed instruction** later
    - We go back to user-mode and try again
- During this time, other processes can run instead

Page Faults Don’t Impact Correctness

- You will never crash because you get a page fault
- Only **slow a process down**
- After the fault is handled, the page is now in RAM
- The process can use it after it starts again
- Programmer has a **limited ability to control a page fault** it is based on the OS

Pages and Secondary Storage

- Pages live in **secondary storage** when they are not in memory
    - Typically in disk on the **swap space**
        - Where we keep the pages that aren’t currently in use
- How do we manage the swap space?
    - Pool of partitions?
    - Random collection?
    - File system?

Demand Paging Performance

- Page faults - can block processes
- Overhead (fault handling, paging in and out)
    - Processes are **blocked** while they are reading in pages
    - Delay execution and consume cycles
    - Directly proportional to the number of page faults
- The key: having the **“right” pages in memory**
    - Right pages → fewer faults, less paging activity
    - Wrong pages → many faults, more paging
- We **cannot control** which pages are read in
    - Instead, we can **choose which pages to kick out**
    - We need a good algorithm to decide which page gets rejected

Virtual Memory

- Abstraction provided to programs, they think they have all of the memory
- Generalization of what demand paging allows
- Form of memory, system provide abstraction:
    - Large quantity of memory
    - Each process gets its own set of addresses (entire set)
    - All addresses are accessible to the process
    - The speed is **approaching that of actual RAM**

The Basic Concept

- Give each process an **address space of immense size**
- Allow processes to **request segments** in that space
- Then use **dynamic paging and swapping** (because we don't have that much actual RAM) to support the abstraction
- Key issue: how do we **create** the abstraction when we don't have that much real memory?

The Key VM Technology: Replacement Algorithms

- Goal: have each page already in memory when a process accesses it
- But, we can't know what a process needs ahead of time
- Instead, rely on **locality of access**
    - To determine what pages to move out when page frames are full
- We can make wise choices

Candidate Replacement Algorithms

- Random, FIFI - BAD
- Least Frequently Used - sounds better, but isn't
- **Least Recently Used** - assert that the near future is like the past
    - How do we implement this?

Naive LRU

- Each time a page is accessed, **record the time**
- When we eject a page, we **look at all timestamps** for pages in memory and choose the one with the **oldest timestamp** to eject
- Issues:
    - Need to **store timestamps**
    - Need to **search all timestamps**
    - We need to **update the timestamp** at every instruction

Maintaining Information for LRU

- Where would we keep the information necessary for true LRU?
- Keeping it in the MMU → slows down the MMU
    - Translation needs to be fast, this is not good
    - At best, MMU will manage a read/write bit per page
- Maintain this information in software
    - We would need to cause a page fault to update the time slice
    - NOT GOOD
- We need a **cheap software surrogate** for LRU
    - We don't want extra page faults

    - Don't want to look at all page frames when we want to eject, this is expensive

Clock Algorithms

- A **surrogate** for LRU, not true LRU
- Organize all pages into a circular list
- MMU sets a **reference bit** for the page on access
- **Scan** the list whenever we need another page
    - We can **start at a particular point** (point we last stopped our search) using a **GUESS POINTER**
    - At each page, ask MMU if the page has been referenced
    - If so, reset the bit and skip this page
    - If not, consider this page to be the LRU → clear it out
    - **The next search starts here** (guess pointer)
- The position in scan is used as a surrogate for age
- No extra page faults because we usually only scan a few page
- This can be **built into the hardware**
- Performance:
    - Same as genuine LRU in example

Comparing True LRU to Clock Algorithm

- Same number of loads and replacements
- Both are approximations to the optimal
- LRU is suboptimal, clock is another suboptimal version of LRU

Page Replacement and Multiprogramming

- We don't want to clear out all page frames on a context switch
    - Some pages may be shared among multiple processes
- What about **sharing page frames**
    - Among all possible processes
- Possible choices:
    - Single global pool (clock on everybody)
    - Fixed allocation of page frames per process
    - **Working set-based page frame allocation**
        - What we actually do

Single Global Page Frame Pool

- Treat the entire set of page frames as a **shared resource**
- We can approximate the LRU for the entire set
    - Replace whichever process' page is LRU
- MISTAKE:
    - Bad interaction with round-robin scheduling
    - Lots of page faults can happen

Per-Process Page Frame Pools

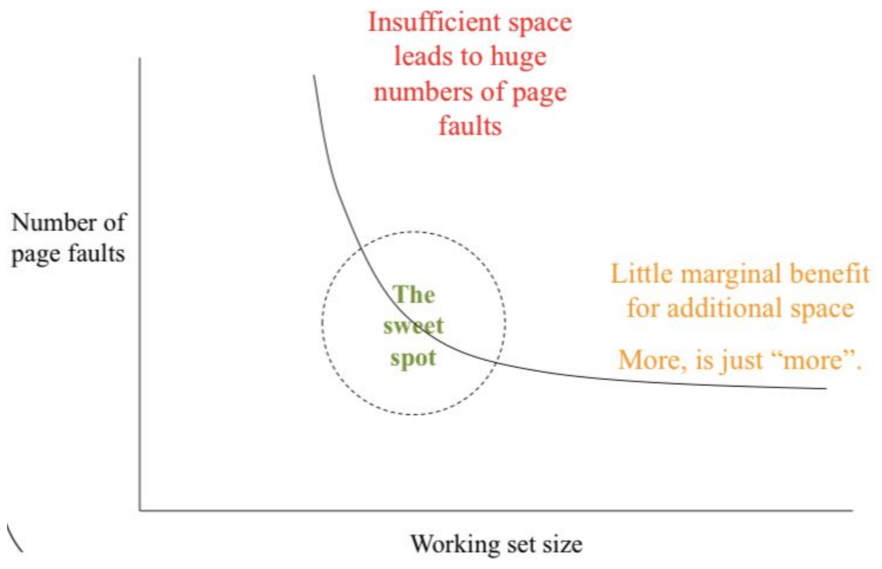
- Set aside a specific number of page frames for each running process
- Each process gets its own set
    - Use an LRU approximation for each set

- How many page frames do we give?
- Fixed number of pages/process is **bad**
    - Different processes have different locality behavior
        - Which pages are needed changes over time
        - Number of pages needed changes over time
    - Fixed allocation can give too many/too few pages
    - Small sets - not enough pages
    - Large sets - too many page faults
- We need **dynamic customized allocation**

Working Sets

- Only pages that **we actually need** in the page frame
    - Not the entire set, a subset of the pages
- We can give each running process **an allocation of page frames** matched to its needs
- We can use **working sets** to determine these needs
- **Working set:** set of pages used by a process in a fixed length sampling window in the immediate past
- We can allocate **enough page frames** to hold each process' working set
- And each process will **run page replacement** within its own set

The Natural Working Set Size



- Sweet spot: not too many page faults but you don't waste too much memory
- Pushing out to the right means that another process has **lost** pages, the number of page faults will not decrease

Optimal Working Set

- What is the optimal working set?
    - The number of pages needed during the next time slice
- Fewer pages → pages will replace one another continuously
    - Don't get to finish time slice before you get a page fault

    - Process will run slowly
- We can **observe process behavior** to determine set size (similar to MLFQ)
    - Give a process a number
    - Observe what happens:
        - No faults: at least big enough
        - Faults: need a bigger set
- Which pages should be in the set?
    - No need to guess, the process **will fault for them**

Implementing Working Set Size

- **Manage the working set size**
    - Assign page frames to processes
    - Processes page against themselves
    - Observe the behavior (faults/time)
    - We can then adjust the number of page frames accordingly
- **Page stealing algorithms**
    - Usually there are not many free pages
    - We need to take the pages from another process' working set
    - EX: working set-clock
        - Track last use time for each page for the owning process
        - Find the page LRU by its owner
        - Process that need more pages tend to get more
        - Processes that don't use their pages tend to lose them

Thrashing

- Thrashing: when there are not enough page frames to support all processes you are trying to run
- The sum of working set sizes > number of actual page frames
- If we don't have enough memory:
    - Nobody will have enough pages in memory
    - Whenever anything runs, it will steal a page from someone
        - Who also lacks memory
    - Page fault as soon as they start running
- **Thrashing**: everybody steals so nobody runs to the end of their time slice
    - Results in many page faults
    - Everybody has too few pages
- **Thrashing slows down entire system**
- Continues until the system takes action

Preventing Thrashing

- Two ways:
    1. Kill
    2. Swap
- We cannot reduce working set sizes
    - Will cause thrashing
- We can **reduce number of competing processes**

    - Set aside some processes and take their frames and spread them out
    - Swap out ready processes and ensure that there is enough memory for other processes
- Swapped out processes may not be able to run for quite a while
    - We need to build up other working sets to prevent page faults and thrashing
- We can **round-robin** which processes are swapped in and which are out

Un-Swapping a Process

- When a swapped process comes back from disk? There may be no pages left
- **Pure swapping** (too much memory)
    - Bring in **all pages before process is run**, no page faults
- **Pure demand paging** (slow startup time)
    - Pages only brought in as needed
    - This causes page faults
    - Fewer pages per process allows more process in memory
- We can **pre-load** the last working set
    - Keep this working set
    - Copy it in when the process is copied back in
    - Far fewer pages to be rad in when swapping
    - Probably the same disk reads as pure demand paging
    - Far fewer initial page faults than pure demand paging

Clean vs. Dirty Pages

- **"Clean" page**: a page whose in-memory copy has not been modified
    - Copy in disk is still valid
    - Can be replaced without writing back to disk
- **"Dirty" page:** a page whose in-memory copy has been modified
    - Copy on disk is not invalid
    - The in-memory copy is "dirty"
    - Must be written back to disk if it is swapped out
- We **want to get rid of "clean" pages** when we page fault
    - Page out clean pages
    - Don't need to write the disk version and update it
    - Dirty pages need to be updated in disk
- We may try to **skip over dirty pages** when chooses pages to replace

Dirty Pages and Page Replacement

- Clean pages can be replaced at any time
    - **We want to choose clean pages**
- Dirty pages need to be written to disk before the frame can be reused
    - Slow operation
- If we only kick out clean pages
    - Limits flexibility
    - We may up eventually swapping out **all clean pages** (that may or may not need to be used often)

Pre-Emptive Page Laundering

- Clean pages provide the memory scheduler with **more flexibility**
- We can increase flexibility by **converting dirty pages to clean ones** (update the disk version of a dirty page)
- Ongoing **background** write-out of dirty pages
    - Find and write out all **dirty, non-running pages**
    - Assumption: we eventually will have to page out
    - If we make them clean again, it will be easier to page them out
        - We will have more clean pages to page out
    - Done during a less busy time for the disk drive
- Outgoing equivalent of pre-loading

Paging and Shared Segments

- Some memory is shared between processes
- Page frame containing these segments is used by more than 1 process
    - Not really part of a single process' working set
- Don't fit the working set model
- Instead, use **global LRU** on all shared pages

Reading

Arpaci-Dusseau Ch. 18 - Paging: Introduction

- Two approaches to solving the space-management problems
    1. Segmentation → leads to fragmentation
    2. Paging
- Both involve breaking things up
- Paging: dividing the virtual address space into pages
    - Physical memory becomes an **array of page frames**
    - Each page frame maps to a virtual memory page

18.1 Overview

- Advantages of paging:
    - Flexibility: address space abstraction is supported
    - Simplicity of free space management: we can keep a free list of pages
- **Page table:** per process data structure that records where each virtual page of the address space is in physical memory
    - Stores the address translations of virtual pages
- A virtual address is split into the **virtual page number (VPN)** and the **offset**
- We can use the VPN to look up the physical page number/physical frame number (PFN)
- The physical address is split into the **physical frame number (PFN)** and the **offset** which remains the same from the virtual address

18.2 Where are Page Tables Stored?

- In memory
- Managed by the OS

18.3 What's Actually in a Page Table

- Data structure that maps virtual address/page numbers to physical addresses/page numbers
- Simplest form is a linear page table (array)
    - Indexed by the VPN
    - It looks up the page-table entry (PTE) by index
- Each PTR contains bits:
    - Valid bit: if a particular translation is valid
        - (the unused space between the stack and other data is **invalid**)
        - Supports sparse address space → we do not need to allocate frames for invalid addresses
    - Protection bits: a page can be read/write/execute
    - Present bit: tells whether **the page is in physical memory or in secondary storage**
    - Dirty bit: page has been modified?
    - Reference bit: page accessed → determines popularity

18.4 Paging: Also Too Slow

- Page tables can become very large, cause **slow performance**
- Summary of the process:
    1. Extract VPN from the virtual address
    2. Find address of the PTE
    3. Fetch the PTE (extra instruction — lots of work)
        a. Extra memory reference required to get the page table translation
    4. Check if the process can access the page
- Page tables using lots of memory are very slow

18.5 A Memory Trace

- Each instruction has two memory references
    - Page table to find the physical frame for the instruction
    - The instruction itself to fetch for the CPU

Arpaci-Dusseau Ch. 19 - Paging: Faster Translations (TLBs)

- Speed up the address translation with hardware
- We can add a **translation-lookaside buffer (TLB)** as part of the **memory management unit (MMU)**
    - Cache of popular virtual → physical address translations
- On a virtual translation request, we first check the TLB
- TLB - one TLB for an entire machine
    - Shared by all cores, accessible by all processes
    - On **context switch** bring the new process' TLB translations into the TLB

19.1 TLB Algorithm

- Algorithm for hardware
1. Extract the VPN from the virtual address
2. Check if the TLB holds the translation
    a. If YES, **TLB hit**
        i. Extract the PFN and concatenate it onto the offset
        ii. Access the memory
    b. If NO, **TLB miss**
        i. Access the page table
        ii. Update the TLB
            1. This is only done **if the translation is valid AKA the valid bit is set**
        iii. Retry the instruction

- The TLB assumes that common translations are in the cache
    - Thus, there is little overhead
- Misses incur a high paging cost
- Relies on both **spatial locality** (elements being close together) and **temporal locality** (quick references of memory in time)

19.3 Who Handles TLB Misses?

- Two possibilities: software (OS) or hardware
- TODAY, the software manages the TLB
    - On a miss, the hardware raises an exception
    - The trap handler looks up translations in the page table and update the TLB
- Return-from-trap instruction is modified
    - When we return from a TLB miss, we resume execution **from the instruction that caused the trap** instead of **after** like in a system call trap
    - Need to retry the instruction so we can now get a TLB hit
- OS needs to be careful to prevent an infinite chain of TLB misses
- Advantage of software-managed TLB is **flexibility** (the OS can implement the page table)

19.4 TLB Contents: What's in There?

- TLB: fully associative
    - Given translation anywhere, and TLB is searched in its entirety
- May contain VPN | PFN | some other bits
    - Valid bit - if there is a valid translation
    - Protection bits
    - Address-space identifier
    - Dirty bit

19.5 TLB Issue: Context Switches

- Issues when switching between processes
    - The TLB is **valid only for the current process**
    - We need to ensure that the new process **doesn't use the old process' translations**

- Solutions:
    - Flush the TLB on context switches (overhead)
        - Set all valid bits to 0
        - Each time a program runs it will **incur TLB misses**
    - Provide **address space identifier (ASIO)** field in the TLB like a process identifier
        - TLB can hold translations from different processes

19.6 Issue: Replacement Policy

- Cache replacement issue: which entry do we replace when the TLB is full and when it wants to add a new word?
- Follow LRU replacement policy
    - Locality advantages
- Random replacement
    - Simplicity avoids corner cases

Arpaci-Dusseau Ch. 21 - Beyond Physical Memory: Mechanisms

- OS needs to make a large, slower device appear like it has a large virtual address space
- Additional layer in the memory hierarchy to store the unused part of the address space
    - Put pages that **currently are not in high demand**
- Swap space: allows for the illusion of a large virtual memory

2.1 Swap Space

- **Swap space**: space on the disk reserved for moving pages back and forth
    - Swap pages into and out of memory in this area
- Os can read/write to the swap space
    - And needs to **remember the disk addresses of the pages**
- The size of the swap pages determines the maximum number of memory pages

2.2 The Present Bit

- Swap space is on disk
- Need the machinery to support swapping
- Add machinery (more than initially for memory reference)
    - When hardware looks at the PTE → detect that it is not PRESENT in physical memory (determined by a present bit)
- Accessing a page not in physical memory (**present bit = 0**) causes a **page fault**
    - Serviced by a **page fault handler**

21.3 The Page Fault

- OS runs a page fault handler on a page fault
    - The page is not present and is swapped to disk
        - OS swaps the page back into memory
        - How does the OS know where to swap?
            - Using the PTE to find the disk address

- The bits used in the PTE normally to store the page frame number of the page can be **instead used for the DISK ADDRESS** of the page that is not in memory
    - After the disk I/O, the OS updates the page table **to mark the page as present**
- At the next attempt, **there will be a TLB miss and then the page will be serviced and updated**
- **At the last try (3rd), the entry will be in the TLB**
- During the I/O, the process is blocked
    - We can have **overlap** meaning another process can execute during this time

21.4 What is Memory is Full?

- Page in a page from swap space when memory is free
- **When memory is full,** page out pages to make room for new pages
    - Picking a page to kick out is part of the **page-replacement policy**

21.5 Page Fault Control Flow

- Hardware during translation **(in the TLB miss case)**
    1. Page present and valid → TLB miss handler
        a. Handler grabs the PFN (page frame number) from the PTE (page table entry) and retries
    2. Page is valid but page fault because PTE is not present → page fault handler
        a. Valid, but not present
    3. Invalid page → bug in program, hardware and OS traps → trap handler runs
- OS handling the page fault
    1. Find physical frame for the soon-to-be-faulted-in page
        a. Wait for a replacement and kick out pages if there are none
    2. Handler issues an I/O request (SLOW)
        a. Read in page from swap space
    3. OS updates the page table
    4. Retry → TLB miss → retry → TLB hit

21.6 When Replacements Really Occur

- OS wants to keep portion of memory free proactively
- There are **high-watermark (HW)** and **low-watermark (LW)** indications to help decide when to begin evicting pages from memory
    - When LW pages < HW pages
        - Thread that frees memory runs
        - Thread evicts pages until HW pages are available
        - Called the **swap daemon/page daemon**
    - cluster/group pages - we can write them all out at once
    - Background paging thread - modifies the CF to include checking for available pages instead of replacing
        - If no, inform the daemon

Arpaci-Dusseau Ch. 22 - Beyond Physical Memory: Policies

- Less memory is free → memory pressure makes the OS page out often, choosing pages to evict using the **replacement policy**

<u>22.1 Cache Management</u>

- Our problem:
    - Main memory is a cache for virtual memory
    - Goal for the replacement policy: **minimize cache misses (times we need to fetch a page from disk) OR maximize cache hits**
- Average memory access time (AMAT)
    - AMAT = T(Access) + Percentage(miss) * T(Disk Access)
- Small hit rate → dominates as T(Disk Access) >>

<u>22.2 The Optimal Replacement Policy</u>

- Belady’s Optimal Policy (MIN)
    - **Replaces page accessed furthest in the future**
    - Fewest possible cache misses
- **cold-start/compulsory miss in the beginning**: accesses that result in cache misses at the beginning because the cache is empty
- We can’t build this, the future is knot known
- Optimal: serves as a reference point for us

<u>22.3 A SImple Policy: FIFO</u>

- First-in-first-out replacement: pages enter a queue
- Simple, but bad performance

<u>22.3 Another Simple Policy: Random</u>

- Picks a random page to replace
- Performance depends on luck, better than FIFO but worse than optimal

<u>22.4 Using History: LRU</u>

- FIFO and Random may kick out an important page
- Using history/previous accesses:
    - Frequency (less common)
    - Recency of access (more common)
- Family of policies based on temporal locality:
    - Least Frequently Used (LFU)
    - **Least Recently Used (LRU)**
- LRU does well, matches optimal
- Opposites of algorithms exist (MFU, MRU) but they are bad

<u>22.8 Approximating LRU</u>

- More feasible and requires hardware support
    - A **use bit/reference bit**
    - Set to 1 when the page is referenced, reset by the OS
- **CLock Algorithm**
    - Pages arranged in a circular list

    - Clock hand points to a page
    - When replacement is needed, clock checks if the use bit is 1 or 0
        - **1: recently used, set to 0 and incremented**
        - **Clock is incremented until a 0 use bit is found**
- Not as good as LRU but still works
    - Suboptimal replacement for LRU

22.9 Considering Dirty Pages

- Modification of clock algorithm can include whether a page has been modified
- If a page is modified/dirty
    - **Must be written back to disk before evicting it - EXPENSIVE**
- Hardware includes modified/dirty bit

22.10 Other VM Policies

- **Page selection:** OS decides when to bring pages into memory
- **Demand paging:** OS brings in pages to memory when accessed
    - Guessing ahead is **prefetching**
- **Writing out to disk**: clustering many pending writes

22.11 Thrashing

- **Thrashing:** system constantly paging
    - Happens when memory is oversubscribed
    - Memory demands > physical memory available
- Approaches:
    - Historical: system had a working set for processes
        - Can decide whether or not to run a set of processes
        - **Admission control**
    - Today: **out-of-memory killer**
        - Daemon chooses intensive process and kills it

Kampe: Working Sets

LRU is Not Enough

- LRU works because of spatial and temporal locality
- Locality is true about single processes
- In Round Robin, separate processes can rarey access the same page
- global LRU: bad in absence of locality
- LRU + RR: bad together
- **Instead, have a per-process LRU**

The Concept of a Working Set

- Avoid page faults due to a lack of memory
- **Thrashing: degradation in system performance because of a lack of adequate memory to run ready processes**
- If we increase page frames → little difference in performance
- If we decrease page frames → dramatic decrease in performance
- **Working set:** the number of page frames given to a process

How Large is a Working Set?

- Depends on the program
- Inferred by looking at the behavior
    - Many page faults, working set > memory allocated
    - No page faults, memory allocated may be **too much**
- Goal: limit page faults and maximize throughput

Implementing Working Set Replacement

- Similar to global LRU - clock-like scan
- Clock hand - **surrogate for age**
    - Recently examined pages are **behind the hand**
    - Pages exampled least recently are **immediately in front of the hand**
- Maintain more information
    - Each page frame associated to an **owning process**
    - Each process has an **accumulated CPU time**
    - Each page frame has a **last referenced time**
    - The **target age** parameter is maintained in the memory
- Age decisions are made based on accumulated CPU time
- Full scan without finding a page to take → replace the oldest page in memory

Dynamic Equilibrium

- **Page stealing algorithm**: process that needs a page steals from a process that doesn't need it
    - Every process is losing pages
    - Every process is stealing pages
    - More references mean a larger working set
    - Fewer references mean a smaller working set
    - Changes in behavior → the working set will adjust
- Manages memory to minimize page faults and thrashing

Threads & Interprocess Communication

Lecture 7: Threads, IPC, and Synchronization

Outline

- Threads
- Interprocess Communications (IPC)
- Synchronization
    - Critical sections
    - Asynchronous event completions

Threads

- Abstraction of code execution
- Why not just have processes?
- What is a thread?
- How does the OS deal with threads?

Why Not Just Processes?

- Processes are **very expensive**
    - To create: they have their own resources
        - You need to create page tables, and page frames
    - To dispatch: they have address spaces
        - Swapping between processes means changing address spaces
- Different processes are **very distinct**
    - **Cannot share** the same address spaces or resources
    - Not all programs require this distinction
- Not all programs require **strong separation**
    - Multiple activities **working for a single goal cooperatively**
        - Need something that should share
    - **Mutually trusting elements** of a system
        - Divided system means that there is no need for a strong system
        - No need for heavyweight processes

What is a Thread?

- A unit of execution/scheduling
    - Each has its own **stack, Program Counter, and registers**
    - But the **other resources (address space) are shared** among other threads in the same process
- Multiple threads can run in a process
    - Share the **same code and data space**
    - Have **access to the same resources**
    - As a result, they are **cheaper to create and run** because you do not need to create a new address space
- Multiple threads can share the CPU among themselves

- Two models:
    - **User-level threads** (voluntary yielding)
        - The **OS does not know** about these threads
        - You need to use a library that knows about and keeps track of and also schedules the threads
        - Implication: **can't run these threads on more than one core** because cores are scheduled by the OS
    - **Scheduled system threads** (preemption)
        - OS **is aware of these threads**
        - Abstraction provided by the OS
        - OS can schedule these threads on different cores if necessary because the OS knows about the stack, the context, etc.

When to Use Processes?

- Run **multiple, distinct programs**
- **Creation/destruction** are rare
- Need to have **distinct privileges**
- When there are **limited interactions and shared resources**
    - Low overhead for these things
- To **prevent interference** between other processes
    - Ex: malicious programs
- Create a **firewall** between the failures of one thread and the other threads
    - Don't want crashes to take down the entire process

When to Use Threads?

- For **parallel activities** in a single program
- If there is **frequent creation/destruction**
- When all of the threads can **run with the same privileges**
- When there is a **sharing of resources**
- When you need to **exchange messages and signals**
- If there is **no need for protection**

Processes vs. Threads - Trade Offs

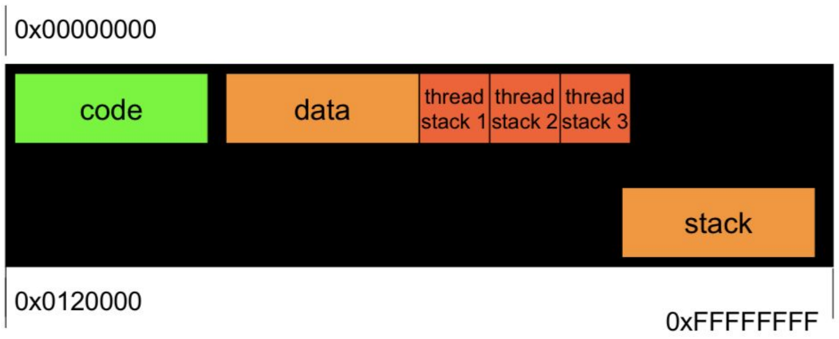
- Multiple processes
    - Overhead: program may be slower
    - Sharing resources is difficult
- Multiple threads:
    - You have to **create and manage them**
        - This may be inconvenient depending on whether it is a user or a system level thread
    - You have to **serialize resource use**
        - Synchronization issues
    - Programs will **be more complex to write**
        - Multithreaded programs are difficult to write

Thread State and Thread Stacks

- Each thread has **its own register, PS, PC**

  - Shared address space but nothing else from the execution environment is shared
- Threads **have their own stack area** - where can we store this in our address space?
- A **maximum stack size** is specified when a thread is created
  - Processes can contain many threads
  - But, **these threads cannot all grow towards a single hole** like the process stack and data segments can
  - The thread creator must know a **max required stack size**
  - And **stack space needs to be reclaimed** when the thread exits
- Procedure linkage conventions remain unchanged

Thread Stack Allocation



- Thread stacks are allocated near the data segment
- Each thread gets its own stack space
- They are **allocated next to each other**
  - Specify a size for each stack that is a **fixed size**
- If you run out of stack space, **it is bad**
  - The best you can hope for is an error message

User Level Threads vs. Kernel Threads

- Kernel threads:
  - Abstraction provided by the OS
    - Need a **system call** so they are **slow**
  - Share one address space
  - But scheduled by the kernel
    - Multiple threads can use multiple cores
- User Level Threads
  - The kernel knows nothing about them (no OS benefits)
  - Provided and managed via a user-level library
    - You have to voluntarily yield
  - Scheduled by library, not by kernel
  - Not true parallelism

Interprocess Communication (IPC)

- Mechanisms that facilitate the exchange of information between processes

- Processes cannot “touch” each other - Os needs to do it

Goals for IPC Mechanisms

- In an IPC mechanism, we want:
    - Simplicity
    - Convenience
    - Generality
    - Efficiency
    - Robustness and reliability
- Some contradict each other

OS Support for IPC

- Usually through **system calls**
    - We need to **talk to the OS** because the **OS can talk to everybody**
- Requires activity from both communicating processes
    - We **cannot force** another process to perform IPC
    - It must be a **cooperative** activity
    - Both processes have to want to communicate/be receptive
- Each step is **mediated** by the OS who facilitates communication between processes
    - This **protects both processes**
    - And **ensure correct behavior**

IPC: Synchronous and Asynchronous

- Synchronous IPC
    - **Writes block** until a message is sent/delivered/received
    - **Reads block** until a new message is available
    - Easy for programmers
    - We know exactly what happens
    - **Safer, but worse performance**
- Asynchronous IPC
    - **Writes return** when the system accepts the message
        - No confirmation of transmission/delivery/reception
        - Requires auxiliary mechanism to figure out errors
    - **Reads return** promptly if no message is available
        - Requires auxiliary mechanism to learn of new messages
        - Often involves “wait for any of these” operation
    - No guarantee that information has been received or sent
    - But is **much more efficient** in some circumstances
        - Processes can continue without blocking and waiting
        - But if there is an error, we need to check
        - We may need to do some remedial action
    - More complicated but **better performance**

Typical IPC Operations

- create/destroy a channel
- write/send/put: insert data into the channel

- read/receive/get: extract data from the channel
- Channel content query: check how much data is in the channel
- Connection establishment and query: control the connection from one end to the other of a channel
    - Info: endpoints and status

IPC: Messages vs. Streams

- **Fundamental dichotomy** in IPC mechanisms
- **Streams**
    - Continuous stream of bytes
    - read/write a few or many bytes at a time
    - write/read buffer sizes are unrelated
    - Stream may contain app-specific record delimiters
- **Messages** (aka datagrams)
    - Sequence of distinct messages, chunks of integral pieces of data
    - Messages have lengths subject to limits
    - Each message is typically read/written as a unit
    - Deliver is typically all-or-nothing
- Styles are suited for different types of interactions

IPC and Flow Control

- **Flow control**: making sure a fast sender doesn't **overwhelm** a slow receiver
- Separate streams of control:
    - creator/sender
    - consumer/receiver
- Queued messages **consume system resources** because the producer produces too much
    - They are **buffered in the OS** until the receiver asks for them
    - The producer has produced too much
- This may **increase buffer space**
    - Ex: fast sender, non-responsive receiver
- We need to **limit required buffer space**
    - Sender side: block sender or refuse the message
    - Receiving side: stifle the sender, flush old messages
        - Can tell the sender to be slower
    - This is handled by **network protocols**
- Mechanisms for feedback to the sender

IPC Reliability and Robustness

- OS won't accidentally "lose" data **within a single machine**
- Requests and responses **can be lost across a network**
- Even on a single machine, a sent message **may not be processed**
    - For instance, if you send to a crashed process
    - Receiver is invalid, dead, or not responding
- How long should the OS be responsible for IPC data?

    - If the OS can’t deliver it, how long should the OS hold the message in the buffer

Reliability Options

- When do we tell the sender that the message is being processed in the OS?
- When do we tell the sender “OK”?
    - Queued locally?
    - Added to receiver inputs queue?
    - Receiver has read it?
    - Receiver has explicitly acknowledged it?
- How persistently should the system attempt deliver?
    - **Especially across a network**
    - Retransmissions? How many?
    - Different routes or alternate servers?
    - How often should the OS resend a copy of the message?
- Do channels/contents survive receiver restarts?
    - New servers, are they able to pick up where the old one left off?
    - Can we restart?

Some Styles of IPC

- Pipelines
- Sockets
- Mailboxes and named pipes
- Shared memory

Pipelines

- Data **flows** through a series of programs
- Data is a **simple byte stream**
    - Buffered in the OS
    - No need for intermediate temporary files
- **No security/privacy/trust** issues
    - Under control of a single user
    - Only one user is using the pipe
- Error conditions
    - Input: EOF reached
    - Output: next program failed
- **Simple, but limited**

Sockets

- Connections between addresses/ports
    - connect/listen/accept
    - Lookup: registry, DNS, service discovery protocols
- Many data options
    - Reliable to best effort datagrams (messages)
    - Streams, messages, remote procedure calls, etc.
- Complex flow control and error handling
- trust/security/privacy/integrity

    - We cannot guarantee this
    - We might be communicating with processes on different machines
- **Very general but also more complex**

Mailboxes and Named Pipes

- Compromise between sockets and pipes
- client/server **rendezvous point**
    - Client and server go here to send/receive messages
    - **Name** corresponds to a server
    - Server **awaits client connections**
    - Once open, as simple as a pipe
    - **OS may authenticate** the sender
- Limited fail-over capability
    - If a server dies, another can replace it
    - In-progress requests may be loss
- Client and server **must be on the same system**
- **Some disadvantages/advantages**
- ***More complex than pipes but simpler than sockets***

Shared Memory

- OS arranges for **process to share read/write memory segments**
    - These are **pages of memory available to BOTH processes**
        - Processes have pointer to the **same page frame** that they can read/write
    - Mapped into multiple process' address spaces
    - Applications **provide their own control of sharing**
    - The OS is **not involved in data transfer**, it only arranges the sharing
        - Memory simply reads and writes
        - Very fast
- **Simple** in some ways
    - read/write shared page frame means that two pages can instantly see changes
        - Very fast, but very dangerous
    - It can be treated as a simple data structure
    - But also very complicated
        - Cooperating processes **must achieve the synchronization/consistency** they want
- Only works **on a local machine**
- Efficient
    - It is as if the memory was in your own address space
- BUT, **another process can mess it up** by writing bad stuff to it

Synchronization

- Want to make things happen **in the "right" order/a specific order**
- This is easy if only **one** set of things is happening
- This is easy if the simultaneously occurring things occur **independently**

- Very complicated otherwise
- One solution: avoid synchronization
    - This **will not work**, in the modern day, we need parallelism to get the performance we want

The Benefits of Parallelism

- Throughput: blocking on activity does not stop others
- Modularity: we can separate complex tasks into smaller pieces
- Robustness: failure of one does not stop failure of others
- Better suited for emerging paradigms: client server, web services, modern computing is all about parallelism
- If we have to have parallelism **we have to have synchronization**

Why is There a Problem?

- Sequential program execution is easy
    - We know exactly what will happen
    - The execution order is **obvious and deterministic**
- Independent parallel programs are easy
    - We only need to worry about multiplexing the resources
    - This is easy to do correctly
    - We've already solved this problem in previous lectures
- **Cooperating parallel programs** are difficult
    - Two execution streams **must be synchronized**
        - Results will depend on the order of instruction
        - Parallelism will make execution order **non-deterministic**
            - May work once, may not work another time
        - Results will be **combinatorially intractable**
        - We can not be sure of the order of the instructions
            - These may be interleaved
            - **Different results may occur because of different instruction orderings**

Synchronization Problems

- Race conditions
- Non-deterministic execution

Race Conditions

- **Only occurs with parallelism**
- **Race condition**: when you have different possible orderings of events
- This occurs when the result depends on the execution order of processes/threads in parallel
- Some streams of control may reach a critical point
    - Different orderings of the stream arriving may cause different results that **may be incorrect**
- Some race conditions affect **correctness**
    - Conflicting updates (mutual exclusion)
    - check/act races (Sleep/wakeup problem)

    - Multi-object updates (all-or-none transactions)
    - Distributed decisions based on inconsistent views
- Each can be **managed**
    - Once we recognize the race condition and the danger
    - We can ensure we get the results we want
    - This requires **mechanism** and **proper use** of these mechanisms

Non-Deterministic Execution

- Parallel execution **reduces predictability of process behavior**
    - Process block for I/O or resources
    - Time-slice end preemption
    - Interrupt service routines
    - Unsynchronized execution on another core
    - Queueing delays
    - Time required to perform I/O operations
    - Message transmission/delivery time
- All of these things **affect the stream of instructions** that can **prevent you from predicting things**
- Leads to problems
- We don't know what will occur and what the order will be in which these things occur

Synchronization

- True parallelism is too difficult for us to fully understand
    - Pseudo-parallelism - we may have a chance
- Two interdependent problems:
    - **Critical section serialization**
    - **Notification of asynchronous completion**

The Critical Section Problem

- **Critical section**: resource that is shared by multiple threads
    - Concurrent threads, processes, or CPUs
    - Interrupted code and interrupt handler
- User of the resource **changes its state**
- And the **correctness depends on execution order**

Critical Section Examples

- Updating a shared file
- Re-entrant signals
- Multi-threaded banking code

Single Instruction

- Even a single instruction can be a critical section
  count = count + 1
- Is actually **three instruction**
- These three instructions can be interrupted
- **One single machine language instruction cannot be interrupted**
- **>1 machine language instruction can be interrupted**

Interleavings *Seem* Unlikely

- Usually not very likely
- BUT, we are executing a billion instructions/second
- Low probabilities → can still result in high frequency
- At the speed at which we are operating can **cause rare events to occur very frequently**

Critical Sections and Mutual Exclusion

- Critical sections cause issues when **more than one thread executes them at a time**
- We can prevent this by ensuring that **only one thread can execute a critical section at a time**
- We want **mutual exclusion**
    - If A gets to use it, B can't use it and vice versa
    - Only **one thread/process** can access the resource at a time
- Need to ensure mutual exclusion **just during the critical section** to prevent improper parallelism

One Solution: Interrupt Disables

- Turning off interrupts during critical sections
- Works only on **single core machines**
- We can **temporarily block some or all interrupts** during critical sections
    - Using a privileged instruction
    - We do it by **reloading a new Processor Status Word** which contains information about interrupts
- Whatever is running first gets to complete the critical section without being interrupted
- Abilities:
    - Prevent **time-slice end** (timer interrupts)
    - Prevent re-entry of **device driver code**
    - **Exclusive access to all resource**
- Dangers:
    - Delays important operations
    - Bugs may leave the program permanently disabled
    - Won't solve all synchronization problems on multicore machines
        - Turning off interrupts still leaves the original say, 4 cores, running in parallel

What Happens During an Interrupt?

- **Hardware traps** to stop whatever is executing
- **Trap table** is consulted
- An **Interrupt Service Routine (ISR)** is consulted
- The ISR **handles the interrupt and restores the CPU**

Downsides of Disabling Interrupts

- Must be done in **privileged mode**
    - Requires a privileged instruction

- **Dangerous**
    - Could disable preemptive scheduling, disk I/O
- Delays system response to **important interrupts**
    - We may miss something important
    - Received data isn't processed until the interrupt is serviced
    - Or it may even be lost
    - Device is idle until the next operation is initiated
- May prevent **safe concurrency**
    - Reduces performance

Interrupts and Resource Allocation

- Interrupt handlers **not allowed to block**
    - Scheduled process/threads can block
    - Interrupts are disabled until the call completes
- They should never need to wait
    - Needed resources should already be allocated
    - Operations are implemented with lock-free code
- Brief spins may be OK
    - Wait for hardware
    - Wait for co-processor to release a lock

Interrupts - When to Disable Them

- When you are using **shared resources**
    - Used by synchronous and interrupt code
- **Non-atomic updates**
    - For single machine language instructions, we don't need to wait
    - Operations that require multiple instructions require you to wait
- Need to **disable interrupts** in these sections
    - But you should do it as **seldom** and as **briefly** as possible

Be Careful With Interrupts

- You should be **very sparing** in your use of interrupts
    - Increasing interrupt service time is **very costly**
        - Scheduled processes have been preempted
        - Devices may be idle
        - System will be less responsive
        - We may lost information
        - Hardware may lost information due to a queueing limitation
    - Disable **as few** interrupts as possible
    - Leave them disabled for **as briefly** as possible
- Interrupt routines **cannot block or yield the CPU**
    - They are not a scheduled thread
    - They don't do resource allocation
    - And they cannot do **synchronization operations**

Evaluating Interrupt Disables

- effectiveness/correctness

- Effective in single processors
    - Ineffective against multiprocessor/device parallelism
    - Must be in kernel mode code
- Progress
    - Deadlock risk if the handler can block
    - Run the risk of preventing things from progressing
- Fairness
    - Good if interrupt disable is brief
- Performance
    - One instruction, **much cheaper than a system call**
        - Only one machine instruction needed to reload the Processor Status Word
    - But **long disables may hurt** performance

Other Possible Solutions

- **Avoid shared data**
- **Eliminate critical section** with atomic instructions
    - Atomic operations
    - 1-8 contiguous bytes
    - Simple instructions
    - Some complex instructions: test-and-set, exchange, compare-and-swap
- We can use these complex atomic instructions to **implement locks**

Reading

Arpaci-Dusseau Ch. 26 - Concurrency: An Introduction

- **Thread**: abstraction for a running program
- Multithreaded program: program with multiple points of execution
- Thread are similar to separate processes but **they share the same address space**
- State of a single thread:
    - Program counter
    - Private set of registers
- Differences in context switch from a process: address spaces stay the same
    - State saved to a **thread control block**
- Stack (threads vs. pointers)
    - Each thread gets **its own stack space**
    - Stack, allocated variables, and etc. are part of thread-local storage

26.1 Why Use Threads?

- 2 major reasons:
    - **Parallelism:** transform a single-threaded program into a program that does work on several CPUs
        - Parallelization makes a program run faster

        - **Avoid blocking program progress due to slow I/O**
            - Allows the CPU to perform computations on other tasks while you are waiting for I/O
            - Enables the overlap of I/O with **other activities within a single program** (this was not possible with programs)
                - Like multiprogramming for processes across different programs
- Also, we are able to **share multiple processes** but **threads are better** for when you need to **share data**
    - Processes are better for when **little sharing is needed**

<u>26.2 EX: Thread Creation</u>

- When created, they run **independently of the caller**
    - They may return **before or after** the caller
    - This depends on the scheduler
- Can run in different orders depending on decisions made by the scheduler

<u>26.3 Why it Gets Worse: Shared Data</u>

- Two threads updating a local variable → **non deterministic results**

<u>26.4 The HEart of the PRoblem: Uncontrolled Scheduling</u>

- **Race condition**: results depend on the timing execution of the code
    - Bad timing for context switches can lead to an incorrect result
- **Critical section:** piece of code that accesses a **shared variable** (or shared resource) and cannot be concurrently executed by more than one thread
- We want **mutual exclusion**: ensures that only one thread is executing within the critical section and all others are prevented from doing so

<u>26.5 The Wish for Atomicity</u>

- One solution: have instructions that did what we needed **in a single instruction/step**
    - Allows instructions to execute **atomically**: they won't be interrupted mid-instruction
    - **"All or none" execution behavior**
- Writing a single instruction to be atomic is difficult
- Instead, we have **synchronization primitives** that come from the hardware

<u>26.2 One More Problem: Waiting for Another</u>

- Another issue: one thread may need to wait for another to complete an action before it continues
- Later: **condition variables**

Kampe: User-Mode Thread Implementation

<u>Introduction</u>

- Problems with processes:
    - Expensive to create and dispatch
        - Due to virtual address space unique ownership

    - Have their own address space so they cannot share resources
- Threads:
    - Independently schedulable unit of execution
    - Runs within the address space of a process
    - Accesses all system resources owned by a process
    - Has its own general registers
    - Has its own stack

A Simple Threads Library

- Threads **first implemented in a user-mode library**
- Basic model:
    - Each time a new thread is allocated
        - Allocated memory for a fixed-size thread-private stack from the heap
        - Create a new thread descriptor
            - Contains:
                - Identification info, scheduling info, and a pointer to the stack
        - Add the new thread to the ready queue
    - If a thread calls yield() or sleep()
        - Save the general registers onto its stack and select the next ready thread
    - Dispatching new threads means:
        - Restore the saved registers and return from the call causing the yield
    - sleep() - remove thread from the ready queue
        - Place it back when it is awakened
    - When a thread exits, free its stack and thread descriptor
- People later wanted preemptive scheduling
    - Linux can create a SIGALARM timer signal
    - We can schedule it and then send the signal if it is taking too long
    - Forces the thread to yield and the next thread can run
- Preemption created **critical section** issues that required atomicity and serialization

Kernel Implemented Threads

- Fundamental problems with user mode threads:
    - **System call blocks**
        - Processes are blocked until the system call is completed
        - All threads stop execution
        - The OS does not know that other threads are runnable
    - **Multi-processor exploitation (or lack thereof)**
        - Os is not aware of multithreading → the threads cannot execute in parallel
- These issues are solved if threads are implemented by the OS

Performance Implications

- For non-preemptive scheduling: user-mode threads need to be yielded with sleep(), yield() model
    - This is **more efficient in non-preemptive schedulers**
    - Today: lightweight thread implementations
- For preemptive scheduling: **cost setting alarms and servicing them is greater than allowing the OS to do scheduling**
    - **Kernel threads are more efficient**
- Threads running in **parallel or on multi-processors**
    - **Throughput from the parallel execution > efficiency losses associated with context switches**
    - The OS can maximize cache-line sharing
- Overhead for signal disabling and reenabling for mutex or a condition variable may be **higher than kernel-mode implementation**
- **BUT, you can get best of both worlds:**
    - User-mode implementation that uses an atomic instruction to lock but calls the OS if that fails

Kampe: Interprocess Communication

Introduction

- Process communication comes in two categories
    1. Coordination of operations with other processes
        a. Synchronization
        b. Signal exchange
        c. Control of operations
    2. **Exchange of data**
        a. Uni-directional data processing pipelines
        b. Bi-directional interactions

Simple Uni-Directional Byte Streams

- Easy, trivial
- Pipe: opened by parent and inherited by a child
- Characteristics:
    - Accepts a **byte-stream input** and produces **byte-stream output**
    - **Programs in pipeline are independent**
    - Byte streams are **unstructured**
    - Ensuring output is suitable input to the next program is the **responsibility of the agent who creates the pipeline**
    - Same as creating a temporary file
- **Pipe**: temporary file with special features
- They recognize the differences between a file and an interprocess stream:
    - If the reader exhausts all the data, the file descriptor is still open
        - An EOF is not sent

        - Reader is blocked until more data is sent or the write side is closed
    - Available buffer capacity is limited
        - If the writer is too far ahead of the reader, the OS may block the writer until the reader catches up, **flow control**
    - Writing to a closed pipe is illegal (SIGPIPE)
    - File deleted when read/write fd are both closed
- Pipeline privacy:
    - This is a closed system
    - Protection on initial input and final output
    - But there is no authentication or encryption

Named Pipes & Mailboxes

- Named pipe (fifo(7)): explicit connections
- Persistent pipe: reader/writer can open it by name
- **Named pipe: rendezvous point for unrelated processes**
- Not as simple as other pipes
- characteristics/limitations:
    - No authentication
    - Writes from multiple writers are interspersed
    - No clean fail-over
    - **Reader/Writer must be on the same node**
- **Mailboxes: general inter-process communication mechanism**
- Common features of mailboxes:
    - Data not a byte stream, instead stored as a distinct message
    - Writes are accompanied about ID information about the sender
    - Unprocessed messages remain in mailbox after the death of one reader and can be retrieved by another
- **Single node/single operating system restrictions** still placed on mailboxes

General Network Connections

- Standard set of network communication APIs
- Linux APIs:
    - socket(2) - interprocess communication endpoint
    - bind(2) - associate a socket with a local network address
    - connect(2) - establish a connection to a remote network address
    - listen(2) - **await** incoming connection request
    - accept(2) - **accept** incoming connection request
    - send(2) - send a message over a socket
    - recv(2) - receive a message from a socket
- Range of communication options:
    - Byte-streams
    - Best effort datagrams
- Foundation for higher level communication/service models
- General network models → interacting globally introduces complexity

    - Need interoperability between different OS's
    - Security issues
    - Discovering addresses over servers
    - Detecting and recovering from connection and node failure
- Limited throughput, high latency

Shared Memory

- Performance sometimes more important than generality
- High performance interprocess communication means:
    - Efficiency: low cost to transfer
    - Throughput: max # of bytes transferred
    - Latency: minimal delay
- If we want this **buffering data in the OS is NOT the way to go**
- Efficiency → **shared memory**
    - Create a file for communication
    - Each process **maps that file into its virtual space**
    - This shared segments **can be locked down so it isn't paged out**
    - Processes agree on a data structure in the shared segment
    - Anything written to the shared data is **immediately visible to all**
- **OS plays no role in data exchange** meaning that this is **fast**
- Price:
    - Processes need to be on the same memory bus
    - Bug can destroy the data structure
    - No authentication

Network Connections & Out-of-Band Signals

- Problems in FIFO - delay if first messages are longer
- We want **some signals (important messages) to go to the front of the queue**
- For local connections, we can send a signal to flush out data ("out-of-band" communication)
    - Signal travels over different part than the buffered data
- We can do the same thing for network based communications by opening multiple channels
    - One heavily used channel for normal requests
    - One reserved channel for **out-of-band requests - special requests**
- The server can poll for out-of-band requests before normal requests → little overhead

Linux Manual - Named Pipes

6.3 Named Pipes (FIFOs - First In First Out)

6.3.1 Basic Concepts

- Like a regular pipe with some differences
    - Device special file in the file system
    - Processes of different ancestry can share this

    - Remains after I/O is done for later use

6.3.2 Creating a FIFO

- In shell:

```
mknod MYFIFO p           (need to call chmod)
mkfifo a=rw MYFIFO       can change the permissions
```

- Identified by:

```
prw-r--r–...                   ____  MYFIFO|
```

- In C:
    - Use mknod() system call

```
int mknod(char* pathname, mode_t mode, dev_t dev);
```

6.3.3 FIFO Operations

- Same as normal pipes
- Need to use “open” system call to open up channel to pipe

6.3.5 SIGPIPE Signal

- SIGPIPE: sent to kernel if the process writes to the pipe with no reader

Asynchronous Completion & Mutual Exclusion

Lecture 8: Mutual Exclusion and Asynchronous Completion

Outline

- Mutual exclusion
- Asynchronous completions

Mutual Exclusion

- Critical sections cause problems when **more than one thread executes them at a time**
- We can prevent this
    - Need to ensure that **only one thread can execute a critical section at a time**
- We want **mutual exclusion** in the critical section
    - Only one thread/process/machine at a time is allowed to use a critical section

Critical Sections in Applications

- Common in **multithreaded applications** that share data structures
- Can also happen with **processes** that share OS resources
- We can avoid them if we **don't share resources of any kind**
    - Not feasible

Recognizing Critical Sections

- Usually involve **updates to an object's state**
    - If everybody is simply reading, there are no critical sections
    - If anybody writes, **there could be a critical section**
    - Updates to a single object
    - Related updates to multiple objects
- Multi-step operations
    - Object state is inconsistent until operation finishes
    - Preemption will **compromise the object or the operation**
- Correct operation requires **mutual exclusion** to ensure that a multi-step application occurs **atomically**
    - Only one thread at a time has access
    - Client 1 completes before Client 2 starts

Critical Sections and **Atomicity**

- We can achieve **atomicity** using mutual exclusion
    - We want this property in our critical sections
- **Atomicity:** things happen in one unit
- Two aspects:
    1. **Before or After** atomicity: two parties that need the same resource, no overlap

2. **All or None** atomicity: it all happens or nothing happens
    - Updates that start will complete
    - Uncompleted update has no effect
- Correctness requires both

Options for Protecting Critical Sections
- **Turn off interrupts**
    - Can prevent concurrency (which is a good thing in this case)
- **Avoid shared data** when possible
    - To get rid of critical sections
- Protect critical sections using **hardware mutual exclusion**
    - Figure out how to do everything you want in a hardware instruction (particular, atomic CPU instructions)
    - **This would work with multiprocessors**
- **Software locking**
    - What we use most of the time

Avoiding Shared Data
- Good design choice, if feasible
- But **not always an option**
- May lead to **inefficient resource use**
    - May be more complicated and not worth the trouble
- Sharing **read only data** will also avoid problems
    - Nobody writes → the order of reads will not matter

Atomic Instructions
- CPU instructions cannot be interrupted
- Some capabilities:
    - read/modify/write
    - Can be applied to 1-8 contiguous bytes
    - Simple instructions
    - Complex instructions: test-and-set, exchange, compare-and-swap
- We can try to do the **entire critical section** using one atomic instruction
- Or we can use atomic instructions to **implement locks**
    - Need mutual exclusion in the lock
    - BASE: **implement the lock using atomic instructions**

Atomic Instructions - Test and Set

```
bool TS(char* p) {
        bool rc;
        rc = *p;            //note the current value
        *p = TRUE;          //set the value to TRUE
        return rc;          //return the value before we had set it
}

if !(TS(flag))
        //we have control of the critical section
```

- FALSE: nobody has the critical section
    - Means that you acquire the lock
- TRUE: someone else has the critical section
- Eventually, you need to **clear the flag**

Atomic Instructions - Compare and Swap

```
bool compare_and_swap(int* p, int old, int new) {
        if (*p == old) {        //see if value has been changed
                *p = new;       //if not, set it to the new value
                return TRUE;    //tell caller he succeeded in changing
        }else
                return FALSE;//tell caller he failed
}

if (compare_and_swap(flag, UNUSED, IN_USE))
        //obtained the critical section
else
        //did not obtain the critical section
```

Preventing Concurrency via Atomic Instructions

- CPU instructions are **hardware-atomic**
    - We can try to squeeze a critical section into one instruction and that will solve our problems
        - This is hard, some operations are too complex
- What can we do with a single instruction?
        - Simple instructions
        - Some slightly more complex ones
        - We can implement data structures
- **Limitations**
    - Unusable for complex sections
    - Unusable as a waiting mechanism

Locking (software based)

- We can **protect critical sections** with a data structure implemented using atomic instructions
- **Locks:**
    - Party holding lock can access the critical section
    - Parties not holding the lock cannot access it
- Party that needs to use the critical section **tries to acquire the lock**
- If nobody holds the lock, the next party that asks will get it
- If somebody holds the lock, the party has to wait, they cannot get the lock
- When finished, **release the lock**
    - It is the programmer's responsibility to release the lock

Using Locks for Mutual Exclusion

```
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);
```

```
...
if (pthread_mutex_lock(&lock) == 0) {
      c = c + 1;
      pthread_mutex_unlock(&lock);
}
```

- Now it is mutually exclusive IF **every thread checks for the lock**

How Do We Build Locks?

- **Everybody has to agree** on what the critical section is
- Locking and unlocking a lock is a critical section
- We can solve the problem with **hardware assistance**
    - We can have the lock be built with one instruction

Single Instruction Locks

- **Core operation of acquiring a free lock**
    1. Check that no one else has it
    2. Change something so others know we have it
- We need to do two things at once
    - We can with the **test-and-set** and **compare-and-swap** instructions

Building Locks From Single Instructions

- Requires complex atomic instructions
    - Test-and-set
    - Compare-and-swap
    - These two happen **automatically**, we can use them to get control of the lock
- Instruction must atomically:
    - **Determine** if someone else has the lock
    - **Grant it** if nobody else has it
    - **Return something** that lets the caller know what happened
- The **caller must honor** the lock

What Happens When You Don't Get the Lock?

- You can **give up**
- But usually you **try again** in a loop (**spin lock**)

Spin Waiting

- Check if the event occured
- If not, check again
- And again
- And …
- We **continue checking**
    - Running instructions to **check and loop**
- This is **not incorrect but hurts performance**

Spin Locks: Pros and Cons

- Pros:
    - **Correct**, enforces access to critical sections
    - **Simple** to program

- Conds:
    - **Wasteful**
        - Spinning uses processor cycles
        - You are executing instructions that aren't useful
        - Somebody else could have executed instructions during this time
    - Likely to **delay freeing of resources**
        - Spinning uses processor cycles
        - The one holding the lock may be halted because you are checking
    - Bug may lead to **infinite spin-waits**

The Asynchronous Completion Problem

- Parallel activities move at **different speeds**
- One activity may need to wait for another to complete
    - producer/consumer relationship
- **Asynchronous completion problem**: how to perform waits **without killing performance**
- Examples:
    - Waiting for I/O
    - Waiting for response to network requests
    - Delaying execution for a fixed period of time
- Easy to do with spin-locks
    - But this is wasteful
- We want to tell the consumer that something is ready
- We can try to **send something to notify the consumer**

Spinning Sometimes Makes Sense

1. When an awaited operation proceeds in **parallel**
2. When awaited operation is guaranteed **to be soon**
    a. Spinning is cheaper than sleep/wakeup which uses system calls
3. When spinning does **not delay awaited operation**
4. When contention is **expected to be rare**
    a. Not a lot of different threads waiting

Yield and Spin

- Check if event occurred
- Check a few more times
- **Then yield if event has not yet occured**
- Sooner or later, **you get rescheduled**
- And then you can check again
- **Repeat** until your event is ready

Problems with Yield and Spin

- Extra **context switches** - expensive
- Still will **waste cycles** if you spin for entire scheduled time
- You may not even get scheduled to check **long after the event has completed**
- Works **poorly with multiple waiters**
    - May not get fairness

    - Someone may **never be scheduled** and have to wait a long time

How Can We Wait?

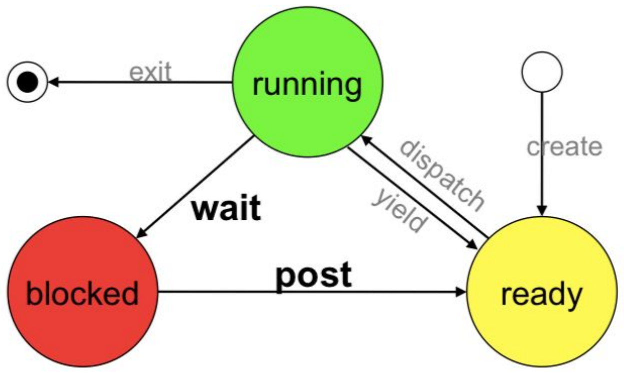
- Spin locking/busy waiting
- Yield and spin
- Either spin option requires **mutual exclusion**
    - And time spent spinning is wasted
- Instead, have **completion events**
    - Supported by the OS
    - Ask to be notified when something is done
    - OS needs to notify the thread/process

Completion Events

- Block if you can't get the lock
    - Then ask the OS to wake you when the lock is available
- Similar to other things you want to wait for
- We can implement this with **condition variables**

Condition Variables

- Create a **synchronization object associated with a resource or request**
    - May be a critical section or simply just a flag
    - Request **blocks** awaiting event on that object
    - Upon completion, the event is **"posted"**
    - Posting the event to object **unblocks the waiter**
- You can't proceed until something is done
- We want to be told when this thing is done



Reading

Arpaci-Dusseau Ch. 28 - Locks

- Locks: can ensure critical sections of code act atomically

28.1 Locks: The Basic Idea

```
lock_t mutex;//globally allocated lock
lock(&mutex);
```

```
    balance  = balance + 1;
    unlock(&mutex);
```

- Lock: variable
    - Holds the state of the lock at any time
    - Either **available (unlocked/free)** or **acquired(locked/held)**
- lock() - acquires the lock (if it is free)
    - Calling thread holding the lock is the owner
    - When another thread calls lock() → and the owner does not return → the other thread is **prevented from executing**
- unlock() - lock is **available again**
    - State is free is no other thread calls lock
    - Of the waiting threads, **one will take the lock**
- Locks: guarantee that only one thread is executing a specific region of code

<u>28.2 Pthread Locks</u>

- POSIX lock: mutex
    - Provide mutual exclusion between threads
- Different locks will protect different variables
    - Increase concurrency
- One large lock for all sections ("**coarse-grained**")
    - Inefficient
- Small locks protecting different code ("**fine-grained**")
    - More threads can be locked at once

<u>28.3 Building a Thread</u>

- Need to use both hardware and OS
- Efficient locks provide mutual exclusion at a low cost
- The OS will help build the locking library

<u>28.4 Evaluating Locks</u>

- **Goals of lock implementations**
    1. Provide mutual exclusion
    2. Fairness
        - Each thread should have a fair shot at acquiring the lock
        - Ask: do any threads starve?
    3. Performance
        - Time overhead of the lock
        - **Evaluate different cases:**
            - No contention: thread is simply grabbing the lock
            - Multiple threads: are there performance issues
            - Multiple CPUs with threads on each

<u>28.5 Controlling Interrupts</u>

- Disable interrupts for critical sections
    - This works for **single processor systems**
- When you lock, **disable the interrupts** and when you unlock, **enable the interrupts**

- Ensure that the code will **not be interrupted**
- PROS:
    - Simple
- CONS:
    - Threads need to perform a **privileged operation** (turning interrupts off/on) - we have to trust the threads
        - OS may never regain control
        - Problem is there are greedy programs that can manipulate lock() or lock() is stuck in an infinite loop
    - Doesn't work on multiple processors
        - Thread running on other processors can enter the event **even if interrupts are disabled**
    - **Interrupts can be lost** - you can miss I/O calls
    - This is **inefficient**
- This is used in **limited context** as a mutual exclusion **primitive**
- OS - interrupt masking
    - Guarantee atomicity when accessing its own data structures
    - OS can do this because the OS trusts itself

28.5 A Failed Attempt: Just Using Loads/Stores

- Rely on CPU hardware only
- We can use a variable (flag) to determine if a thread has a lock
- Threads entering the lock() can check if the flag is set
    - If not, we can set the flag to 1 meaning that the **thread holds the lock now**
- The thread will call unlock() to set the flag to 0 and clear it
- If another thread tries to call lock, it will **spin-wait** until the other thread calls unlock()
    - Once the owning thread calls unlock, the waiting thread owns the lock now
- Problems:
    - Correctness
        - Two threads can all lock and set the flag to 1 at the same time
        - Both threads will enter the critical section
    - Performance
        - **A thread "spin waiting" endlessly to check the flag value**
        - This wastes clock cycles
        - Bad performance, especially on a uniprocessor

28.7 Building Working Spin Locks with Test-and-Set

- **Hardware support for locking**
- Test-and-set instruction (**aka Atomic Exchange**) is atomic

```
int test_and_set(int *old_ptr, int new) {
        int old = *old_ptr;
        *old+ptr = new;
        return old;
    }
```

- All three lines are performed atomically
- Enables you to **test the old value** and **set the memory location** to the new value
- This can help you **build a "spin lock"**
- Test the old value and set the new value is an atomic operation
    - Only one thread can acquire the lock
- Spin lock - spins using CPU cycles until the lock is available
- **Requires a preemptive scheduler** on a single CPU because otherwise, the spinning thread will **never relinquish** the CPU

26.8 Evaluating Spin Locks

- Does well on correctness
- Performs poorly on fairness
    - **Spin locks do not provide fairness**
    - Can cause starvation
    - We cannot guarantee that a waiting thread will enter
- Performs poorly on performance
    - **On a single CPU, observe bad performance**
        - Threads can spin for an entire time slice
    - **Multiple CPUS, we will observe OK performance if the number of threads is similar to the number of CPUs**
        - Doesn't waste too many clock cycles
        - A & B may grab and A will get it but B will spin on a different CPU

28.9 Compare-and-Swap

- Hardware primitive like test-and-set
- **Compare-and-swap instruction**

```
int compare_and_swap(int* ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
}
```

- We can build a spin lock similar to test-and-set
    - Check if the fag is 0, if so, automatically swap to 1, acquiring
- **More powerful, lock-free synchronization**
    - we can replace to lock with:

```
void lock(lock_t *lock) {
    while (compare_and_swap(&lock->flag, 0, 1) == 1)
        ; //spin
}
```

- Identical behavior for now

28.10 Load-Linked and Store-Conditional

- Instructions that work together to build critical sections
- **Load-linked & store-conditional**
    - Can build locks and other structures
- **Load-linked**: like a load instruction
    - Fetches value from memory and places it in a register
- **Store-conditional**: succeeds (in updating value stored at the address just load-linked form) if no intervening store has taken place
    - **Success**: return 1, the value is updated
    - **Failed**: return 0, the value is not updated
- Two threads load link
    - One store conditionals first
        - So the other must spin and wait to try again
- Only one thread is able to update at a time

28.11 Fetch-and-Add

- Hardware primitive
- Atomically increments a value and returns the old value
- We can build a **ticket lock**: uses ticket and turn variable in combination to build a lock
    - When thread wants a lock, it can do a fetch-and-add on a ticket value
    - Value can indicate the thread's **turn**
    - When myTurn==turn it is the threads turn to enter the critical section
- Threads are assigned a ticket value and **eventually are scheduled**
    - **No other attempt has guaranteed this**

28.12 Too Much Spinning: What Now?

- Hardware-based locks are simple and effective but inefficient
- When threads are spinning
    - They spend an entire time slice checking a value
- We need OS support

28.13 A Simple Approach: Yielding

- Threads can spin endlessly, instead, try something else
- First try: give up CPU instead of spinning
- Assume an OS primitive called yield() that moves the caller from running → ready
    - Essentially, the caller **deschedules itself**
- Costly - **context switches occur when the thread yields**
- Starvation is not yet addressed
    - There can be threads that endlessly yield

28.14 Using Queues: Sleeping Instead of Spinning

- Problems with previous approaches: they were too random
    - Scheduler is the one who determines whether a tread must spin or yield
        - This **is wasteful** and causes a **potential for starvation** to occur
- Explicit control is needed
- We can **queue to keep track of threads waiting**

- Using things like **park() and unpark()** from the SOLARIS library
    - Build a lock that puts a caller to sleep when it tries to acquire a held lock
    - Wakes it up when the lock is free
    - Problem: **sleep/wakeup race**
        - One thread might be about to park but the OS switches to another thread who unparks
            - The thread that parked may potentially **sleep forever**
        - We can solve it with a **setpark()** instruction that indicates that a thread is **about to park itself**
- We can combine test-and-set with an explicit queue
- Queue of lock waiters and to **control who gets the lock next**
- Problem: wakeup/waiting race

28.15 Different OS/Different Support

- Linux uses the futex
    - More kernel functionality

28.16 Two-Phase Locks

- **Two-phase locks:** realizes spinning is useful when the lock is about to be released
- First-phase: lock spin hoping that it can acquire it
- Second-phase: caller put to sleep if lock is not acquired yet
- Hybrid approach

Arpaci-Dusseau Ch. 30 - Condition Variables

- threads may want to check a **condition before continuing execution**
- Want parent to sleep while waiting for a condition to be true

30.1 Definition & Routines

- **Condition variable:** explicit queue threads put themselves in when waiting for some condition that currently is not as desired
- Other threads can wake up some of these threads by signaling on the condition
- Declare condition variable: **pthread_cond_t c**;
    - Two operations:
        - wait(): thread puts itself to sleep
        - signal(): thread changed something and wants to wake a sleeping thread
- wait(): release the lock and puts calling thread to sleep
    - Needs to reacquire the lock when the thread wakes up
- Use while loop for waiting
- Complicated examples: producer/consumer or bounded-buffer problems