Justin Ciocoi

Oct. 27, 2023

CSCI 375 Textbook Notes

Chapter 1: Introduction to Operating Systems

1.1: What Operating Systems Do

- We will begin this discussion by breaking down a computer system into its more rudimentary components
- Specifically, this section will focus on *hardware*, an *operating system*, *application programs*, and the *end user*
- The **hardware** components, CPU, memory, and I/O Devices, provide the basic computing resources needed by the system
- The **application programs** define the different ways in which these resources can be used to solve an end user's computing problems
- An operating system, by itself, performs no useful functions but rather provides an environment in which other programs can do useful work
- **1.1.1: User View**
    - The user view can vary according to the computer experience, whether it is a traditional desktop, laptop, or smartphone.
    - In these cases, the operating system is designed mostly with *ease of use* in mind
- **1.1.2: System View**
    - From the system's point of view, the operating system is the program which is most intimately involved with the system hardware
    - Thus, in this context, the operating system acts as a *resource allocator* for the various computing resources present in system hardware
    - This context also tasks the operating system with acting as a *control program*

- A *control program* refers to a program which manages the execution of user programs to prevent errors and improper resource usage
    - It is especially concerned with operation and management of I/O devices

- **1.1.3: Defining Operating Systems**
    - The term *operating system* is a fairly broad one that encompasses many roles and functions present in a computer system
    - This is because although the history of the computer is relatively short, the technologies used in computing have evolved rapidly due to *Moore's Law*
    - *Moore's Law* refers to the prediction made by the founder of Intel, Gordon Moore, that the number of transistors on an integrated circuit will double roughly every 18 months
        - This idea was later amended to every 24 months, which has largely held true
    - In general, when trying to come to a definition of the term *operating system*, there is no single adequate definition that could describe everything that acts as an operating system
    - Operating systems main reason for existence comes from a need to make a computer system usable for a human being.
        - Thus, the fundamental role of operating systems is to solve that problem by executing programs and making solving user processes easier
    - The operating system does this by implementing most of the common functions found among different processes, and implementing them such that each process can better take advantage of computer resources
    - A more general definition for an operating system is the one program that remains running at all times on the computer
        - This program is usually called the **kernel**
    - Along with the kernel, are the application programs we mentioned earlier, as well as *system programs* which are associated with the operating system while not directly being a part of the kernel
    - In the modern era of personal computing, mobile operating systems include not only a core kernel, but also a variety of *middleware*

        - **Middleware** refers to a set of frameworks that provide additional services to application developers
        - For example, both Apples *iOS* and Google's *Android*, feature a core kernel as well as middleware which supports things like databases, multimedia, or graphics
    - Thus, for our purposes we can define operating systems as a core kernel which is always running, middleware frameworks that ease application development, and system programs that aid in managing the system while it is running

1.2: Computer-System Organization

- A modern *general-purpose* computer system consists of one or more CPUs and a number of chips which act as device controllers connected through a common *bus* which provides access between components and shared memory
- Each device controller is in charge of a specific type of device, whether that be *keyboards, displays*, or *video cards*
- Typically, operating systems have a **device driver** for each device controller which will understand the device controller and provide the rest of the operating system a uniform interface to that type of device
    - To ensure orderly access to the shared memory, a memory controller can synchronize access to that memory
- **1.2.1: Interrupts**
    - In order for a program to start an I/O operation, the device driver loads the appropriate registers in the device controller.
    - The controller will then examine the contents of those registers and determine which action to take
    - The controller begins the transfer of data, and once the transfer is complete, the controller informs the device driver that it has finished, and the driver returns control to other parts of the operating system
    - How can the controller inform the driver that it has finished its operation
        - This can be accomplished using an **interrupt**
    - At any time, hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus

        - While other buses do exist in a computer system, the system bus is the main communication path between components
    - Whenever the CPU is interrupted, it stops what it is doing and transfers execution to a fixed location
        - This fixed location generally contains the starting address where the service routine for the interrupt is located
        - The interrupt service routine executes, and upon its completion, the CPU is free to resume the interrupted computation
    - Each computer design has its own interrupt mechanism, but there are several functions which are common among them
    - Since interrupts must be handled quickly, a table of pointers is used to point to interrupt routines.
        - The interrupt routine can be called indirectly through the table with no intermediate routine needed
        - Generally, this table is stored in *low memory*, meaning the first few hundred memory locations
    - The interrupt architecture should also explicitly save the current state of the process it is interrupting such that the interrupted process can later resume as if it had never been interrupted
    - The basic interrupt mechanism works by utilizing a CPU hardware feature known as the *interrupt-request line*, which is a wire that the CPU will sense after executing every instruction
    - When the controller senses that a controller has asserted a signal on the interrupt-request line, it reads the interrupt number and uses the table of pointers to jump to the appropriate interrupt routine
    - We say that the device controller **raises** an interrupt by asserting a signal on an interrupt-request line, the CPU **catches** the interrupt, **dispatches** it to the interrupt handler, and allows the handler to **clear** the interrupt by servicing the device
    - Here is a visual representation of what is described above
    -

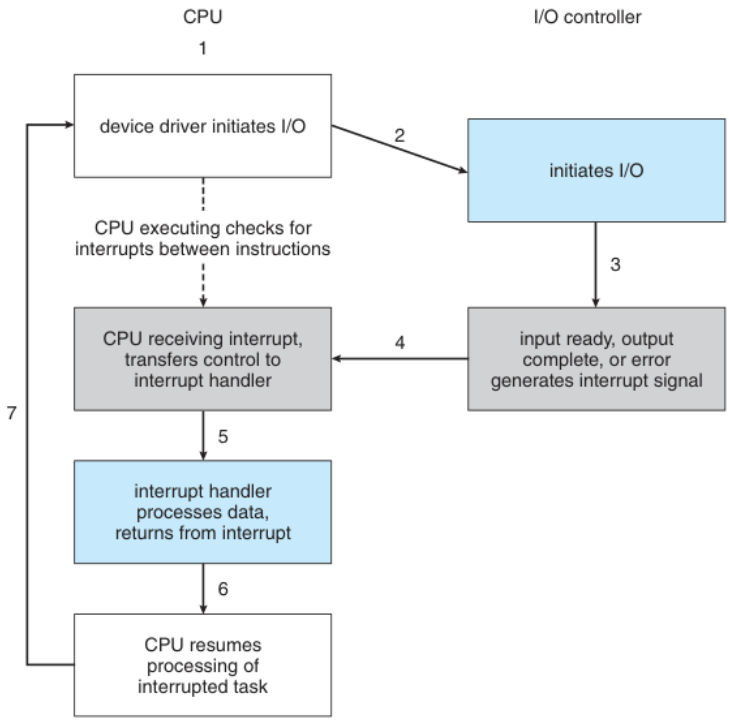


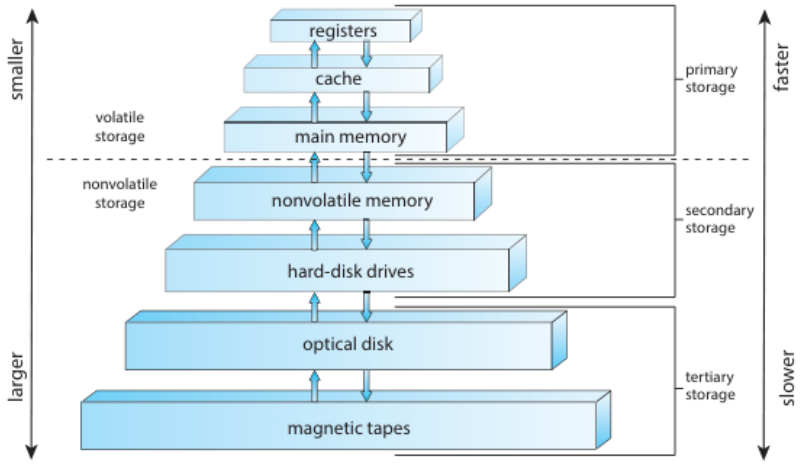
Figure 1.4 Interrupt-driven I/O cycle.

- This had been a fairly rudimentary explanation of interrupt architecture, but in modern operating systems we will need more sophisticated interrupt handling features
  - Interrupt handling must be able to be deferred during critical processing
  - Dispatching to the proper interrupt handler must be done efficiently
  - Multilevel interrupts are required such that the operating system can distinguish between higher and lower priority interrupts and respond with the appropriate degree of urgency

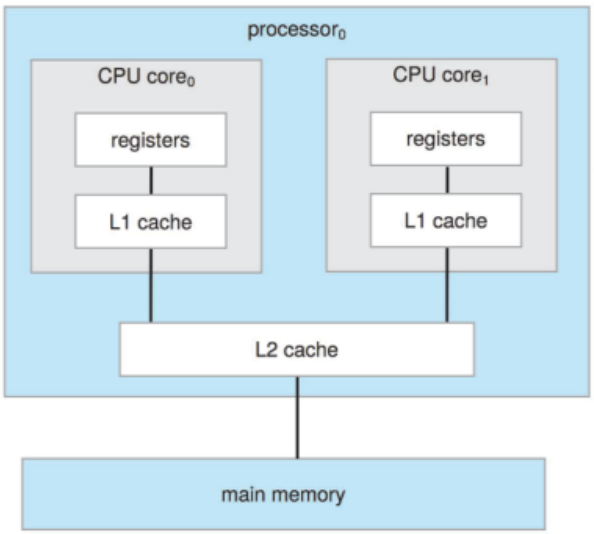
- **1.2.2: Storage Structure**
  - The CPU in a system can only load instructions from memory, so any programs must first be loaded into memory in order to run
  - General purpose computers run most of their programs from rewritable memory, called main memory, and also known as random access memory, or **RAM**
  - Computers also use other forms of memory
    - For instance, since the main memory is volatile, the first program to run on a computer, the *bootstrap program*, is instead stored in electrically erasable programmable read-only memory, or *EEPROM*, which is infrequently written to and *non-volatile* meaning it will not lose its state if it loses power

        - For example, the Apple iPhone uses EEPROM to store serial numbers and hardware information about the device, which should never change regardless of user activity on the device

- The *Von Neumann architecture* refers to the general computer architecture that separates the CPU from main memory
- A typical instruction-execution cycle, as executed on a system with a Von Neumann architecture, first fetches an instruction from memory and then stores that instruction in the *instruction register*
- The instruction will then be decoded and may cause operands to be fetched from memory and stored in some internal register
- Ideally, all needed data and programs would be permanently held on the main memory, but due to its volatile nature and usually small storage capacity, secondary storage systems are almost always used as an extension of main memory
    - The most common of these are *Hard-Disk Drives (HDDs)* and *Nonvolatile Memory (NVM)* devices, such as *Solid State Drives(SSDs)*
- Most programs and data will be stored in the secondary storage until they are loaded into the memory
- Secondary storage is therefore often both the source and destination of a process's processing
- Tertiary storage, such as optical discs or magnetic tapes also exist and mostly serve as backups for secondary storage on computer systems
- Below is a diagram illustrating a fundamental view of the storage hierarchy present in computer systems
-



1.3: Computer-System Architecture

- **1.3.1: Single-Processor Systems**
    - In the beginning of computing's history, most systems used a single processor containing one CPU with a single processing core
    - The *core* is the component that executes instructions and registers for storing data locally
    - In this architecture, the CPU is the only processor that can perform a general instruction set as well as instructions from processes, whereas other device-specific processors may exist in a single processor system
- **1.3.2: Multiprocessor Systems**
    - Nowadays, machines from mobile devices to data-center servers almost all use a *multiprocessor* architecture
    - Even more recently, *multicore* architecture where multiple CPU cores reside on a single chip has become more popular
        - Since on-chip communication is faster than between-chip communication, this increases throughput even more than a traditional multiprocessing system
    - Many multi core systems also implement a hierarchical cache system that behaves very similarly to the storage hierarchy pictured in the previous section
        - CPU cores will have private L_1 caches which are smaller than shared L_2 caches
        - This architecture is pictured below
    - 

1.4: Operating System Operations

- As mentioned before, the *bootstrap program* is the first program that will run on any computer
- In order to initialize the system, the bootstrap program must locate the operating-system kernel and load it into memory
- Some services are provided outside of the kernel by system programs which are loaded into memory at boot time to become *system daemons*, or processes which run the entire time the kernel is running
- Another form of the interrupt which was discussed earlier in this chapter is a *trap* or *exception* which is a software-generated interrupt caused either via an error or a specific request from a user program
    - These requests come in the form of a *system call*
- **1.4.1: Multiprogramming and Multitasking**
    - One of the most important aspects of an operating system is the ability to run multiple programs since a single program cannot usually keep either the CPU or I/O devices busy at all times
    - Therefore, *multiprogramming* or allowing multiple processes to run at once, allows CPU utilization to be maximized
    - In multiprogramming, a system keeps several processes in memory simultaneously, and chooses one to begin executing
        - Eventually, the process might have to wait for something, such as an I/O operation, and instead of allowing the CPU to remain idle while waiting, the CPU can begin to execute a different process
    - *Multitasking* comes as a logical extension of multiprogramming, where the switching between processes occur rapidly, which provides the user with a much faster *response time*
    - We have to also make sure that the operating system maintains control of the CPU and that a user program does not get stuck in an infinite loop
        - For this reason, we accomplish a timer whose duration represents the amount of time a user process will be allowed to execute until control of the CPU is passed back to the operating system

Justin Ciocoi

Oct. 27, 2023

CSCI 375 Textbook Notes

Chapter 2: Operating-System Structures

2.1 Operating System Services

- Operating services can vary in multiple ways across different systems and architectures, but we can identify a few common classes of services that will normally be implemented in a general purpose computer
- One set of operating system services provides functions that are helpful to the user
  - **User Interface**
    - Most common modern computers use a *graphical user interface (GUI)*
    - Another option is a *command-line interface (CLI)*
  - **Program Execution**, which allows users to load programs into memory and execute them
  - **I/O Operations**, such that the user has a reasonable and effective way to conduct I/O
  - **File System Manipulation**, such that a user is able to organize files into folders and create directories
  - **Communications**, which refer to communications between processes, either on the same computer, or on different computers which are connected by a network
    - Communications can be implement utilizing *shared memory* pr *message passing*, in which packets of predefined formats are moved between processes
  - **Error Detection**, for the many different types of error that might occur in a computer system
- There is another set of operating system services which exist not for the user directly, but rather for ensuring the efficient operation of the system itself
  - **Resource Allocation**
    - When multiple processes are executing at the same time, resources have to be appropriately allocated to each of them
    - The operating system manages many different types of resources (CPU cycles, main memory, file storage, etc.)
  - **Logging**, such that the computer will keep track of resource usage, which can be used either for financial accounting purposes, or simply for data retention purposes

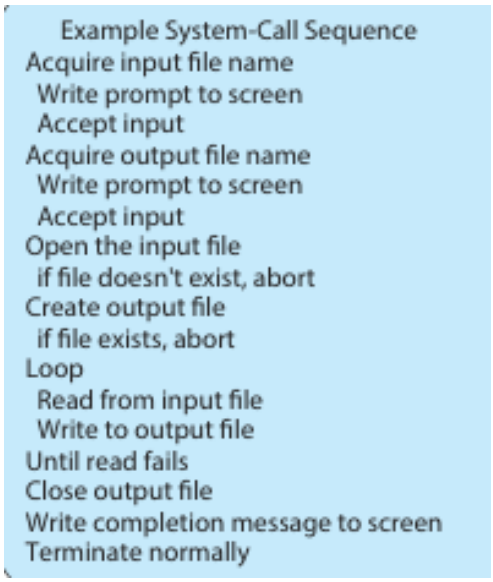
- **Protection and Security**
  - This refers to the ability to segment permissions across a multi-user system whether it is a single computer system or a file sharing system over a network
  - Such multi-user security implementation generally begins with the introduction of user password at computer startup and certain privileged actions

2.2: User and Operating-System Interface

- **2.2.1: Command Interpreters**
  - In most modern operating systems (Windows, MacOS, Linux)the command interpreter is treated as a special program that is running when a process is initiated or when a user first logs on
  - The command line will interpret text-based commands from the user, but there is a steep learning curve associated with using a command interpreter efficiently
- **2.2.2: Graphical User Interface**
  - A second, more popular strategy for user interface is through a user friendly graphical user interface
  - This is the interface with which most people are most accustomed to, featuring windows and menus, a moving mouse for selection, and icons representing different programs and files

2.3: System Calls

- *System calls* provide an interface to these operating-system services, and are usually functions written in C or C++, although certain low-level tasks may have to be implemented using assembly language
- **2.3.1: Example**
  - Let us consider the UNIX `cp` command in the context:
    - `cp in.txt out.txt`
  - This command will copy the input file, `in.txt` to the output file `out.txt`
  - So, once the two file names have been obtained, the program will open the input file, and create and open the output file
    - Each of these operations requires another system call

- The full list of system calls involved in such an operation is shown below
- 

- **2.3.2: Application Programming Interface**
  - On many occasions, systems will execute thousands of system calls per second
  - However, programmers never see this level of detail since typically developers design programs according to an *application programming interface (API)*
  - The API specifies a set of functions that are available to an application programmer, including 5the parameters that are passed to each function and the return values the programmer can expect
  - The system call names used throughout this text are generic ones, and each operating system has its own name for each system call
  - So now we ask the question of why a a programmer would prefer to program according to an API rather than invoking specific system calls
  - One reason is portability, since a developer can expect their program to compile and run on any system that supports the same API
  - Another reason is that actual system calls can often be more detailed and difficult to work with than APIs
- **2.3.3: Types of System Calls**

- Process control
  - create process, terminate process
  - load, execute
  - get process attributes, set process attributes
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device attributes
  - set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach or detach remote devices
- Protection
  - get file permissions
  - set file permissions

2.5: Linkers and Loaders

- Usually, a program will reside on a disk as a binary executable file
- In order to run on a CPU, the process must be brought into memory and then placed in the context of a process
- Source files of program scripts must first be compiled into object files which are designed to be loaded into any physical memory location
- Next, the *linker* will combine these object files into a single binary executable file
- During this linking phase, other object files or libraries, such as the standard C or math libraries
- A *loader* is then used to load the binary executable file into memory, where it is eligible to run on a CPU core

2.6: Why Applications are Operating-System Specific

- On a fundamental level, applications that are compiled on one operating system are not executable on other operating systems
- There are ways, however to make an application executable on multiple operating systems
    - The application can be written in an *interpreted language* such as Python, which has an interpreter available for multiple operating systems
    - The application can be written in a language that includes a virtual machine containing the running application, such as Java and the Java Virtual Machine
    - The application can be *ported* or translated by a developer to be able to run on multiple operating systems using a standard API
- The specificity of programs as far as operating systems comes down to three main factors
    - Each operating system has a binary format for applications that dictates the layout of thew header, instructions, and variables, which must all be in the proper location to ensure proper execution
    - Different CPUs have varying instruction sets, and only applications containing the appropriate instructions can execute correctly
    - System calls that are implemented directly in a program, or through the use of a standard API, vary from operating system to operating system

2.7: Operating-System Design and Implementation

- **2.7.1: Design Goals**
    - Design goals for an operating system can be fundamentally divided into *user goals* and *system goals*
    - There is no unique definition for what these goals should be since computers are used in such a wide variety of ways and such a wide variety of operating systems exists
- **2.7.2: Mechanisms and Policies**
    - One important principle in operating system design is the separation of *policy* from *mechanism*
    - *Mechanisms* determine how something will be done, whereas *policies* determine what will be done

    - For example, the timer concept is a mechanism, but the duration to which the timer is set for each user is a policy decision
    - Separating these two is an important concept when it comes to flexibility
    - Since policies are likely to change across time or place, it would be ideal for mechanisms to be able to adapt
    - The worst case scenario would be each policy change requiring an underlying mechanism change
    - A general mechanism, which is flexible enough to work across a range of policies is preferable
- **2.7.3: Implementation**
    - Once an operating system has been designed, it must be implemented
    - Because of the nature of operating systems and the different ways in which they are designed, it is difficult to make general statements about operating system implementation
    - Early operating systems were written entirely in assembly language, but nowadays most are written in higher level languages like C or C++, with small amounts of the system implemented using assembly language
    - Using higher level languages in operating system development provides many of the same advantages as it does in application development
        - Code is easier to write, understand, and debug
        - Code is far more compact
        - Code is easier to rework in order to run on various computer hardware
    - The only potential disadvantages that come with implementing an operating system using higher level languages are reduced speed and increased storage requirements due to the need to 'translate' or compile high-level languages to machine code

2.8: Operating System Structure

- This section will discuss how the common components of an operating system are interconnected and melded into a kernel
-

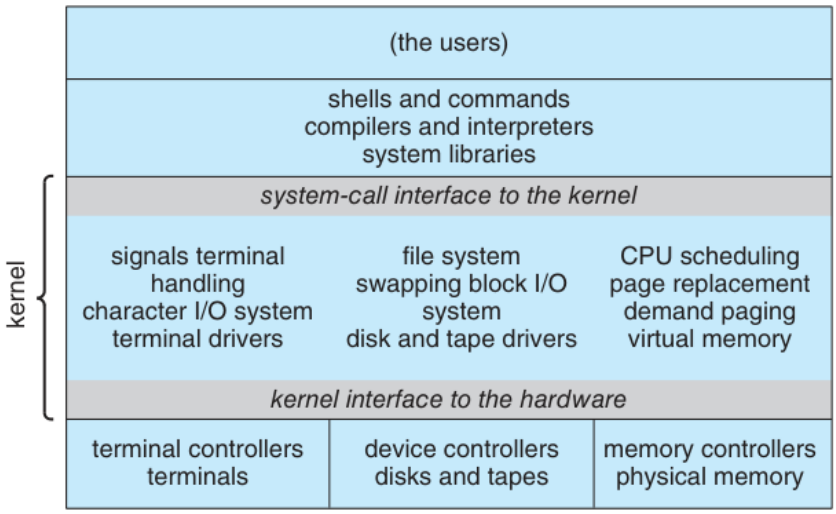


Figure 2.12 Traditional UNIX system structure.

- **2.8.1: Monolithic Structure**
  - A monolithic structure is the simplest structure an operating system can take, owing to the fact that a monolithic system has no structure at all
  - The entire functionality of the kernel is placed into a single, static binary file that runs in a single address space
  - This is a common technique for designing operating systems
  - Monolithic kernels are difficult to implement and extend, but they do have a distinct performance advantage since there is very little performance overhead in the system-call interface and communication with the kernel is very fast \
  - The monolithic approach is often known as a *tightly coupled* system since changes to one part of the system can have wide-ranging effects on other parts of the system
- **2.8.2: Layered Approach**
  - Alternatively, we could design a *loosely coupled system* such that the system is divided into separate, smaller components that have specific and limited functionality
  - One method for designing a loosely coupled system is the layered approach
  - The operating system is broken down into a number of layers, the lowest of which (layer 0) is the computer hardware, and the highest of which (layer n) is the user interface
  - The main advantage of the layered approach is the simplicity of construction and debugging

        - The layers are selected such that each uses functions and services of only lower-level layers

- **2.8.3: Microkernels**
    - As computing's history went on, the kernels that were being used became large and unwieldy as more and more functionality was added to operating systems
    - Thus, microkernels have evolved, or kernels from which all nonessential components are separated and implemented as user level programs that reside in separate address spaces
    - One large benefit of the microkernel structure is that it makes extending the operating system easier since new services are added to the user space and consequently do not require modifications to be made of the kernel
- **2.8.4: Modules**
    - Currently, the best methodology for operating-system design involves using *loadable kernel modules*, where the kernel has a set of core components and can also link in additional services using modules, either at boot time, or at run time
    - This is similar to a layered approach, but it is more flexible in that any module can call any other modules rather than layers only being able to call lower-level layers
- **2.8.5: Hybrid Systems**
    - In practice, there are very few operating systems which strictly adopt one of the previously examined structures
    - Instead, most modern operating systems combine different structures, which results in a variety of *hybrid systems*

2.9: Building and Booting an Operating System

- **2.9.1: Operating-System Generation**
    - Usually, when a computer is purchased, it has an operating system already installed
    - However, if you want to change operating systems, one option you have is *generating* an operating system, which will consist of the following steps
        - Write the operating system source code, or obtain previously written source code
        - Configure the operating system for the hardware on which it will run

        - Compile the operating system
        - Install the operating system
        - Boot the computer and its new operating system
    - You can also download an .iso file which contains a compiled version of general purpose operating systems, which can be flashed onto a USB drive or optical disc and installed onto a computer

- **2.9.2: System Boot**
    - The process of starting a computer by loading the kernel is known as *booting* the system
    - On most systems, the boot procedure is as follows
        - A small piece of code known as the *bootstrap program* or *boot loader* locates the kernel
        - The kernel is loaded into memory and started
        - The kernel initializes hardware
        - The root file system is mounted
    - Many modern computers use a *multistage* boot process
        - When the computer is first powered on, a small boot loader located in nonvolatile firmware known as *BIOS* is run
        - This initial boot loader does nothing more than load a second boot loader, which is located at a fixed disk location called the *boot block*
    - Recently, many computer systems have replaced the BIOS-based boot process with *UEFI* or *Unified Extensible Firmware Interface*, which provides more modern support, as well as the advantage of being a single, complete boot manager and therefore is faster than the multistage BIOS boot process

Justin Ciocoi

Oct. 29, 2023

CSCI 375 Textbook Notes

Chapter 3: Processes

3.1: Process Concept

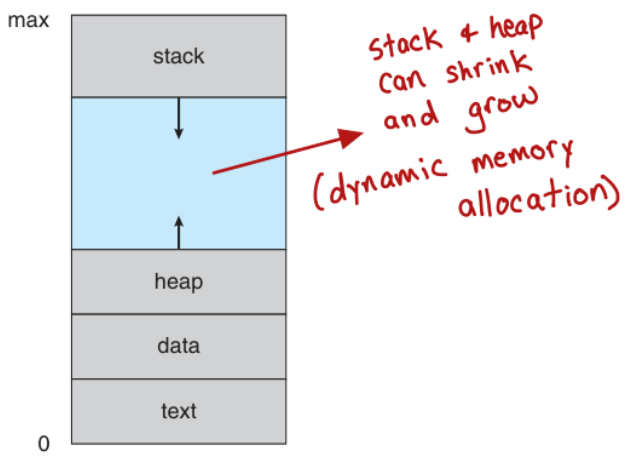
- **3.1.1: The Process**
  - Informally, a process is a program in execution
  - The status of a process's current activity is represented by the value of the *program counter* and the contents of the processor's registers
  - The memory layout of a process is typically divided into sections as shown below
  - 

    Figure 3.1 Layout of a process in memory.

  - The *text section* contains the executable code
  - The *data section* contains global variables
  - The *heap section* which is memory that is dynamically allocated during a program's run time
  - The *stack section* which is temporary data storage when invoking functions, such as function parameters, return addresses, and local variables

- Although the stack and heap sections of a process grow *towards* each other, the operating system must ensure that they do not overlap

- **3.1.2: Process State**
  - As a process executes, it changes its *state*
  - A process's state can be any one of the following
    - **New**, when the process is being created
    - **Running**, when instructions are being executed
    - **Waiting**, when the process is waiting for some event, such as I/O completion or signal reception, to occur
    - **Ready**, when a process is waiting to be assigned to a processor
    - **Terminated**, when a process has finished execution
  - These names are arbitrary and vary across operating systems, but the concepts presented here remain consistent

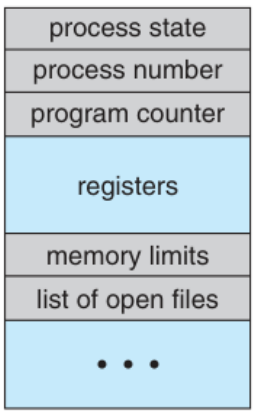
- **3.1.3: Process Control Block**
  - Each process is represented in the operating system by a *process control block (PCB)*, which is illustrated below
  - 

Figure 3.3 Process control block (PCB).

  - In brief, the PCB serves as the repository for all of the data needed to start or restart a process, along with some accounting data

- **3.1.4: Threads**

    - So far, the process model that has been described has implied that a process is a program that performs a single *thread* of execution
    - In most modern operating systems, the process concept has extended to allow a process to have multiple threads of execution and thus perform multiple tasks at a time

3.2: Process Scheduling

- The objective of multiprogramming is to have some process running on the CPU at all times such that the system is maximizing CPU utilization
- In order to meet this objective, the *process scheduler* selects a ready process for program execution on a core
- In general, most processes can be described as either *I/O bound* or *CPU bound*
    - I/O bound processes spend more time doing I/O than computations
    - CPU bound processes spend more time doing computations than I/O
- **3.2.1: Scheduling Queues**
    - As processes enter the system, they are put into a *ready queue*, where they are ready and waiting to execute on a CPU's core
    - Similarly, processes that in are in a waiting state are placed in a *waiting queue*
- **3.2.3: Context Switch**
    - As mentioned in chapter 1, interrupts case the operating system to change a CPU core from its current task and to run a kernel routine
    - When an interrupt occurs, the system must save the *context* of the currently running process so that it can restore that process when the interrupting process is done
    - Generically, we say that a system performs a *state save* of the current state of the CPU core, and then a *state restore* to resume operations
    - This operation is known as a *context switch*, and the duration of this operation is pure overhead since no useful work is done during a context switch
    - The duration of a context switch is highly dependent on hardware support

3.3: Operations on Processes

- **3.3.1: Process Creation**

- During the course of a process's execution, a process may create several new processes
  - Here, the creating process is called the *parent process* and the created process are called *child processes*
- Each of these processes may also create new processes, thus forming a *tree* or processes
- Most operating systems identify processes according to a unique *process identifier (PID)*, which is typically an integer number
- When a parent process creates a child process, that child process will need resources to accomplish its task
  - A child process might be able to obtain these resources directly from the operating system, or it might be constrained to a subset of the resources from the parent process
  - The parent process may partition its resources among child processes, or it may be able to share resources among several child processes
- In addition to providing resources to a child process, a parent process will also pass along initialization data, or input, to the child process
- When a process creates a new process, two possibilities for execution exist
  - The parent continues to execute concurrently with its children
  - The parent waits until some or all of its children have terminated

- **3.3.2: Process Termination**
  - A process will terminate when it finishes executing its final statement and asks the operating system to delete it by using the `exit()` system call
  - All of the resources of this process will be deallocated and reclaimed by the operating system
  - A parent process might terminate the execution of one of its child processes for a variety of reasons, such as
    - The child process has exceeded the usage of its resources which it has been allocated
    - The task assigned to the child process is no longer required
    - The parent is exiting, and the operating system does not allow a child process to continue if its parent terminates

    - When an operating system does not allow a child process to exist after its parent process has terminated, all child processes will also be terminated, which is known as *cascading termination*
    - A *zombie* process is one that has had its resources deallocated by the operating system, but its parent process has not yet called `wait()`
        - All processes transition into this state as they terminate, but generally they only spend a brief period of time as zombie processes
    - An *orphan* process is one that has had its resources deallocated by the operating system, and has had its parent process terminate before calling `wait()`

3.4: Inter-process Communication

- Processes which are executing concurrently in an operating system might be *independent processes*, which do not share data with any other processes, or *cooperating processes*, which can effect or be effected by other processes executing in the system
- There are a variety of different reasons why providing an environment that allows for process cooperation is advantageous
    - **Information sharing**, since multiple applications might be interested in the same piece of data, we should provide an environment to allow *concurrent* access to this data
    - **Computation speedup**, since we can increase the speed of computation by breaking a down into several smaller sub-tasks and executing them concurrently
        - This can only be achieved on a system with multiple computing cores
    - **Modularity**, which allows for more flexibility by dividing system functions into separate processes or threads
- Fundamentally, there are two models when it comes to the implementation of inter-process communication
    - *Shared memory*
    - *Message passing*
-

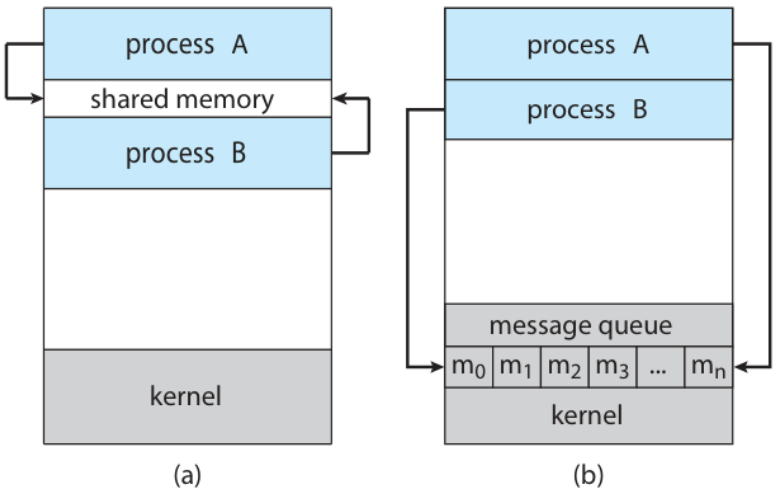


Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

- Both of these models are common in operating systems, and many systems implement both
- Message passing is useful for exchanging smaller amounts of data, since no conflicts need to be avoided, and it is also easier to implement
- Despite the need to synchronize concurrent access to shared memory, using this approach is generally faster than message passing, since message passing is generally implemented using system calls and requires more time-consuming kernel interventions

3.5: IPC in Shared-Memory Systems

- Using shared memory as the medium for inter-process communication requires communicating processes to establish a region of shared memory
- Normally, the operating system tries to prevent one process from accessing another process's memory, but shared memory requires that two or more processes agree to lift this restriction
- Here, the processes are also responsible for ensuring that they are not writing to the same location simultaneously
- In order to illustrate the concept of cooperating processes in a fundamental sense, we will consider the **producer-consumer problem**
- A *producer* process produces information that is consumed by a *consumer* process
- **Producer**

```
item next_produced;

while(true)
{
    //produce an item in next_produced

    while(((in+1)%BUFFER_SIZE)==out)
        ; //do nothing

    buffer[in] = next_produced;
    in = (in+1) % BUFFER_SIZE;
}
```

- **Consumer**

```
item next_consumed;

while(true)
{
    while(in==out)
        ; // do nothing

    next_consumed = buffer[out];
    out = (out+1) % BUFFER_SIZE;

    //consume the item in next_consumed
}
```

- One solution to this problem uses shared memory, where a *buffer* of items exist that can be filled by the producer, and emptied by the consumer
- **Buffer**

```
#define BUFFER_SIZE 10

typedef struct{
    .....
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- This *buffer* will reside in a region of memory that is shared by the producer and consumer processes

- A producer can produce one item *while* the consumer is consuming another item
- Two types of buffers can be used
    - The *unbounded* buffer places no practical limit on the size of the buffer
    - The *bounded* buffer assumes a fixed buffer size

3.6: IPC in Message-Passing Systems

- Another way to achieve the effects of inter-process communication described in the previous section is to implement *message passing*
- This allows processes a mechanism to communicate and synchronize their actions without sharing the same address space\
- A message passing *facility* or the mechanism by which it is achieved, must provide at least two operations
    - `send(message)`
    - `receive(message)`
- **3.6.1: Naming**
    - Processes which want to communicate with each other must have a way to refer to each other
    - They can communicate either directly or indirectly
    - Under *direct communication*, each process that wishes to communicate with another must explicitly name the recipient or sender of the communication
    - In this scheme, the `send()` and `receive()` primitives are defined as
        - `send(P, message)` - Send a message to process P
        - `receive(Q, message)` - Receive a message from process Q
    - A link will be established automatically between every pair of processes that want to communicate
        - A link is associated with exactly two processes, and between each pair of processes, there exists exactly one link
    - This is a *symmetric* scheme, whereas an *asymmetric* scheme exists where only the sender names the recipient, and the receive primitive is defined as follows
        - `receive(id, message)` - Receive a message from any process where the variable `id` is set to the name of the process with which communication has taken place
    - In an *indirect* communication scheme, messages are sent to, and received from *mailboxes*, or *ports*

- For example, POSIX message queues use an integer value to identify a mailbox
- A process can communicate with another process via a number of different mailboxes, but two processes may only communicate if they have a shared mailbox
- Here, the `send()` and `receive()` primitives are defined as follows
  - `send(A, message)` - Send a message to mailbox A
  - `receive(A, message)` - Receive a message from mailbox A
- In this system, a link is established between two processes only if both have a shared mailbox
- A link may be associated with more than two processes, and between communicating processes, a number of links might exist, with each link corresponding to one mailbox

- **3.6.2: Synchronization**
  - Message passing may be implemented in either a *synchronous* or *asynchronous* manner
    - *Synchronous send*, where the sending process is blocked until the message is received by the receiving process or mailbox
    - *Asynchronous send*, where the sending process sends the message and resumes operations
    - *Synchronous receive*, where the receiver blocks until a message is available
    - *Asynchronous receive*, where the receiver retrieves either a valid message or a null
  - Different combinations of these `send()` and `receive()` primitives can be used
  - **Message-Passing Producer**
    - ```
      message next_produced;

      while(true)
      {
          //produce an item in next_produced

          send(next_produced);
      }
      ```
  - **Message-Passing Consumer**
      ```

```
message next_consumed;

while(true)
{
    receive(next_consumed);

    //consume item in next_consumed
}
```

- 3.6.3: Buffering
  - Messages that are being exchanged by communicating processes reside in a temporary queue, which is typically implemented in one of three ways
    - **Zero capacity**, where the queue has a maximum length of zero, thus not allowing any messages in the queue, so the sender must block until the recipient receives this message
    - **Bounded capacity**, where the queue has a finite length n
    - **Unbounded capacity**, where the queue's length is potentially infinite

3.7: Examples of IPC Systems

- 3.7.1: POSIX Shared Memory
  - Several different IPC mechanisms are available for use in POSIX systems, including shared memory and message passing
  - In shared memory, a process must first create a shared-memory object using the `shm_open()` system call
    - `fd = shm_open(name, 0_CREAT | 0_RDWR, 0666);`
  - The first parameter here specifies the name of the shared-memory object
  - The subsequent parameters specify that the shared-memory object has yet to be created (`0_CREAT`) and that the object is open for reading and writing (`0_RDWR`)
  - The last parameter specifies the file-access permissions of the shared-memory object
  - A successful call to `shm_open()` returns an integer file descriptor for the shared-memory object

- Once the object is established, the `ftruncate()` function is used to configure the size of the object in bytes using the call
  - `ftruncate(fd, 4096)`
- This call sets the size of the object to 4096 bytes
- Finally, the `mmap()` function established a memory-mapped file containing the shared-memory object, as well as a pointer to this file used for accessing the shared-memory object

- **POSIX Shared-Memory Producer**

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main()
{
    //size, in bytes, of shared-memory object
    const int SIZE = 4096;
    //name of shared-memory object
    const char *name = "OS";
    //strings written to shared memory
    const char *message_0 = "Hello";
    const char *message_1 = "World!";

    //shared-memory file descriptors
    int fd;
    //pointer to shared-memory object
    char *ptr;

    //create shared memory object
    fd = shm_open(name, O_CREAT || O_RDWR, 0666);

    //configure sie of shared-memory object
    ftruncate(fd, SIZE);

    //memory map the shared-memory object
    ptr = (char*)
    mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    //write to shared-memory object
    sprintf(ptr, "%s", message_0);
    ptr += strlen(message_0);
    sprintf(ptr, "%s", message_1);
    ptr += strlen(message_1);

    return 0;
}
```

- POSIX Shared-Memory Consumer

- ```c
  #include <stdio.h>
  #include <stdlib.h>
  #include <fcntl.h>
  #include <sys/shm.h>
  #include <sys/stat.h>
  #include <sys/mman.h>

  int main()
  {
      //the size, in bytes, of the shared memory object
      const int SIZE = 4096;

      //name of the shared-memory object
      const char *name = "OS";

      //shared memory file descriptor
      int fd;

      //pointer to shared-memory object
      char *ptr;

      //open the shared-memory object
      fd = shm_open(name, O_RDONLY, 0666);

      //memory map the shared-memory object
      ptr = (char *);
      mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

      //read from the shared-memory object
      printf("%s", (char *)ptr);

      //remove shared-memory object
      shm_unlink(name);

      return 0;
  }
  ```
  ```

Justin Ciocoi

Oct. 29, 2023

CSCI 375 Textbook Notes

Chapter 4: Threads and Concurrency

4.1: Overview

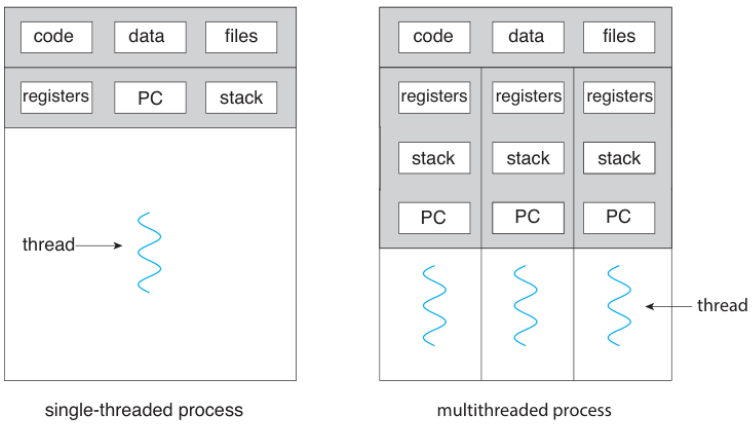
- A *thread* is a basic unit of CPU utilization, which is comprised of a *thread ID, program counter, register set*, and a *stack*
- A traditional process has a single thread of control, but if a process has multiple threads, it can perform more than one task at a time
- **4.1.1: Motivation**
  - Most applications that run on modern computers and mobile devices are multithreaded
  - Typically, an application is implemented as a separate process with several threads of control
  - 

    Figure 4.1 Single-threaded and multithreaded processes.

  - Additionally, applications can be designed in order to leverage processing capabilities on multicore systems, allowing them to perform several CPU-intensive tasks in parallel across multiple computing cores
  - Most operating system kernels are multithreaded, so during system boot, several kernel-level threads are created

- **4.1.2: Benefits**
    - The benefits of multithreaded programming can be broken down into four major categories
        - **Responsiveness**, since multiple operations may be ongoing and the user will not have to wait for each process to finish in serial
        - **Resource sharing**, since threads share, by default, the data of the process to which they belong
        - **Economy**, since there is less overhead required when dealing with multithreaded applications
        - **Scalability**, since a multithreaded program will benefit greatly as the number of computing cores increases

4.2: Multicore Programming

- As computers have evolved from single-CPU to multi-CPU systems, they have also evolved to place multiple CPU computing cores on a single processing chip, where each core appears as a separate CPU to the operating system
- We refer to such systems as *multicore* systems, and multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency
- **4.2.1: Programming Challenges**
    - The trend towards multicore systems continues to place pressure on system designers and application programmers to make better use of the multiple available computing cores
    - In general, five areas present challenges in programming for multicore systems
        - **Identifying tasks**, which involves examining applications to find areas that can be divided into separate, concurrent tasks, which are ideally independent of each other and can run in parallel on individual CPU cores
        - **Balance**, such that processes that are being run in parallel perform approximately equal work of equal value
        - **Data splitting**, where the data accessed and manipulated by tasks must be divided to run on separate cores

        - **Data dependency**, where data that is accessed by two tasks must be examined for any potential dependencies between the two tasks, and proper synchronization must be implemented to avoid data race conditions
        - **Testing and debugging**, which is inherently more difficult in multicore programming due to different possible execution paths across multiple CPU cores
    - **Amdahl's Law**
        - Amdahl's Law is a formula that describes the potential performance gains from adding additional computing cores to an application that has both parallel and serial components
        - If S is the percentage of computation that must be performed in serial on a system with N processing cores, the formula for Amdahl's Law is as follows

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

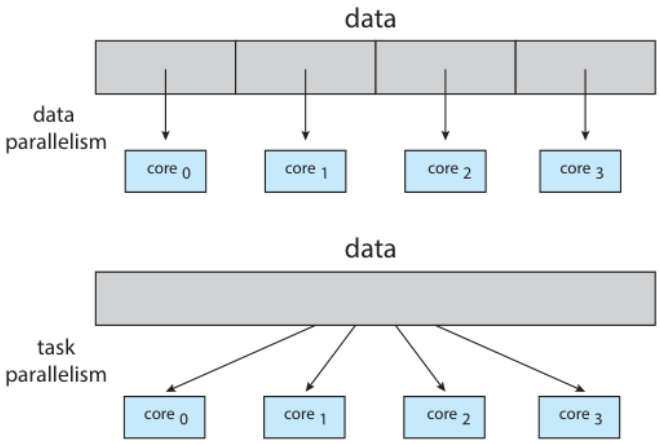
- **4.2.2: Types of Parallelism**
    - In general, there are two types of parallelism: *data parallelism* and *task parallelism*
    - *Data parallelism* focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core
    - *Task parallelism* focuses on distributing not data, but tasks, or threads, across multiple computing cores where each thread is performing a unique operation
    - 

Figure 4.5 Data and task parallelism.

4.3: Multithreading Models

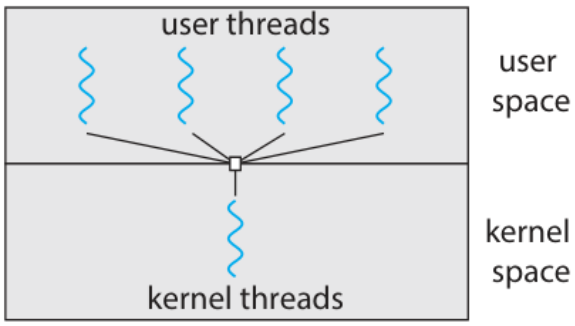
- So far, we have treated threads in a fairly generic sense, but support for threads may be provided either at the user level, for *user threads*, or at the kernel level, for *kernel threads*
- Virtually all contemporary operating systems support kernel threads
- In this section, we will differentiate between the different models that can be used to implement the relationship between user- and kernel-level threads
- **4.3.1: Many-to-One Model**
    - 

Figure 4.7 Many-to-one model.

    - This model maps many user-level threads to a single kernel-level thread
    - Here, thread management is done by the thread library in user space, so it is efficient
    - However, if a thread makes a blocking system call, the entire process will block
    - Additionally, since only one thread can access the kernel at a time in this scheme, multiple threads are unable to run in parallel on multicore systems
    - This scheme is used sparingly in the modern era due to its inability to take advantage of multiple processing cores, which has become standard on almost all computer systems
- **4.3.2: One-to-One Model**
    -

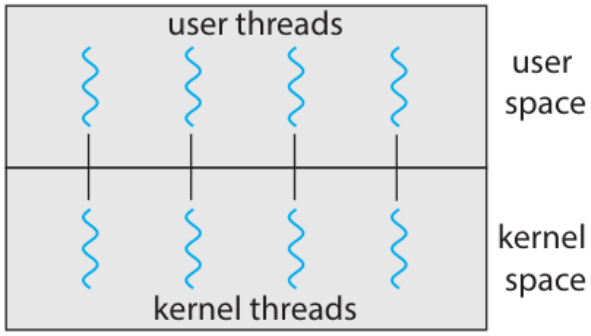


Figure 4.8 One-to-one model.

- The one-to-one model maps each user thread to a kernel thread
- This allows for more concurrency that the many-to-one model by allowing another thread to run when a thread makes a blocking system call
- It also allows multiple threads to run in parallel on multiprocessors
- The main drawback to this scheme is that the large number of kernel level threads may burden the performance of a system

- **4.3.3: Many-to-Many Model**
  -

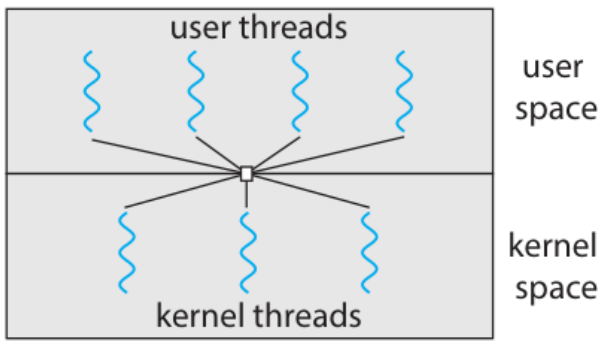


Figure 4.9 Many-to-many model.

- In the many-to-many model, user-level threads are multiplexed and passed to a smaller or equal number of kernel-level threads
- This model is free from the drawbacks from the previous two models since the developer can create as many user threads as necessary, and the kernel threads can run in parallel on a multiprocessor
- A variation on the many-to-many model, called the *two-level model*, allows this multiplexing and demultiplexing of threads, but also allows user level threads to be bound directly to kernel level threads

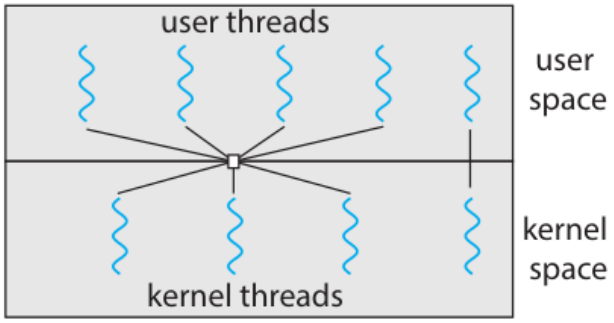


Figure 4.10 Two-level model.

4.4: Thread Libraries

- A *thread library* provides a programmer with an API for creating and managing threads
- There are two primary ways in which thread libraries are generally implemented
  - The first is to provide a library entirely in user space with no kernel support such that all code and data structures for the library exist in user space and invoking a library function results in a local function call rather than a system call
  - The second approach is to implement a kernel-level library supported directly by the operating system such that code and data structures for the library exist in kernel space and invoking a library function typically results in a system call
- The three main thread libraries that are in use today are *POSIX Pthreads, Windows*, and *Java*
- We will illustrate this in the POSIX API using the well known summation function

$$sum = \sum_{i=1}^{N} i$$

- Before we proceed, we must introduce the concepts of *synchronous* and *asynchronous* threading
  - In asynchronous threading, once the parent thread creates a child thread, the parent resumes its execution such that the parent and child execute concurrently and independently of one another
  - Synchronous threading occurs when the parent thread creates one or more child threads and then must wait for all of its child threads to terminate before it resumes
  -

- 4.4.1: Pthreads
    - *Pthreads* refers to the POSIX standard defining an API for thread creation and synchronization
    - Now, let us show a multithreaded C program using the POSIX Pthreads API
    - ```c
      #include <pthread.h>
      #include <stdio.h>
      #include <stdlib.h>

      //declare global sum variable shared by threads
      int sum;

      //declare function which threads will call
      void *runner(void *param);

      int main(int argc, char *argv[])
      {
          pthread_t tid; //thread id
          pthread_attr_t attr; //set of thread attributes

          //set attributes of thread
          pthread_attr_init(&attr);
          //create the thread
          pthread_create(&tid, &attr, runner, argv[1]);
          //wait for the thread to exit
          pthread_join(tid, NULL);

          printf("sum = %d\n", sum);
      }
      ```

```
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for(i=1; i<= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

4.5: Implicit Threading
```

- As multicore processing has grown over the years, applications containing hundreds, or even thousands, of threads are looming on the horizon
- In order to develop applications that effectively utilize such a framework, developers must invest more time and effort to program and debug than in a single-threaded application
- One way to address these difficulties and better support the design of concurrent and parallel applications is to transfer the management of threading from application developers to compilers and run-time libraries
- This strategy has been coined *implicit threading*, and it allows application designers and developers to more easily take advantage of multicore processing
- **4.5.1: Thread Pools**
    - A *thread pool* can be utilized such that a number of threads are created at startup and placed into a pool, where they idly sit and wait for work
    - For instance, when a server receives a request, rather than creating a new thread, it can submit the request to the thread pool and resumes waiting for additional requests
    - Once a thread completes its service, it will return to the pool to await more work
    - Thread pools offer the following benefits
        - Servicing a request with an already existing thread is usually faster than waiting to create a thread
        - A thread pool limits the total number of threads that exist at any one point, which prevents systems from becoming burdens due to too large a number of threads
        - Separating the actual task from the mechanics of creating ones allows for more flexibility when it comes to running tasks, such as allowing a task to execute after a time delay, or periodically
- **4.5.2: Fork Join**
    - The strategy of thread creation covered in section 4.4 is often called the *fork-join* model
    - In this method, the main parent thread creates (forks) one or more child threads, and then waits for the children to terminate and join with it, at which point it can retrieve and combine their results
    -

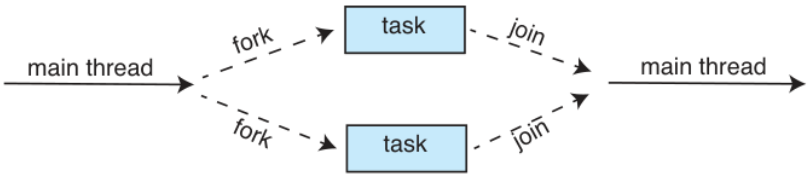


Figure 4.16 Fork-join parallelism.

- **4.5.3: OpenMP**
  - OpenMP is a set of compiler directives, as well as an API written for programs written in C, C++, or FORTRAN which provide support for parallel programming in shared-memory environments
  - OpenMP defines *parallel regions* as blocks of code that may run in parallel
  - Let us look at the following C program illustrating the use of OpenMP

```c
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    //sequential code

    #pragma omp parallel
    {
        printf("I am a parallel region.");
    }

    //sequential code

    return 0;
}
```

  - When OpenMP encounters the directive `#pragma omp parallel`, it creates as many threads as there are processing cores in the system
    - For a dual core system, two threads are created
    - For a quad core system, four threads are created
    - And so on
  - All the threads then simultaneously execute the parallel portion of the code and as each thread exits the parallel portion, it is terminated

4.6: Threading Issues

- **4.6.1: The `fork()` and `exec()` System Calls**
  - If one thread in a program calls `fork()`, does the new process duplicate all threads, or is the new process single-threaded
  - Some UNIX systems have two versions of `fork()`, one that duplicates all threads and one that duplicates only the thread that invoked the `fork()` system call
  - Which version of the `fork()` system call to use is decided on a case by case basis as a function of the application being used
- **4.6.2: Signal Handling**
  - A *signal* is used in UNIX systems to notify a process that a particular event has occurred
  - A signal can be received either synchronously or asynchronously, however all signals follow the same pattern
    - A signal is generated by the occurrence of a particular event
    - The signal is delivered to a process
    - Once delivered, the signal must be handled
  - A signal can be *handled* by one of two possible handlers
    - A default signal handler
    - A user-defined signal handler
  - Different signals are handled in different ways
    - Some may be ignored
    - Others might result in the immediate termination of an application, such as illegal memory access
  - For a system utilizing multithreaded applications, how does the system know where a signal should be delivered
  - In general, the following options exist
    - Deliver the signal to the thread to which the signal applies
    - Deliver the signal to every thread in the process

        - Deliver the signal to certain threads in the process
        - Assign a specific thread to receive all signals for the process
- **4.6.3: Thread Cancellation**
    - *Thread cancellation* involves terminating a thread before it has completed
    - A thread that is to be cancelled is often referred to as the *target thread*, and cancellation of a target thread may occur in two different scenarios
        - **Asynchronous cancellation**, where one thread immediately terminates the targeted thread
        - **Synchronous cancellation**, where the target thread periodically checks whether it should terminate, allowing itself an opportunity to terminate itself in an orderly fashion

Justin Ciocoi

Oct. 3, 2023

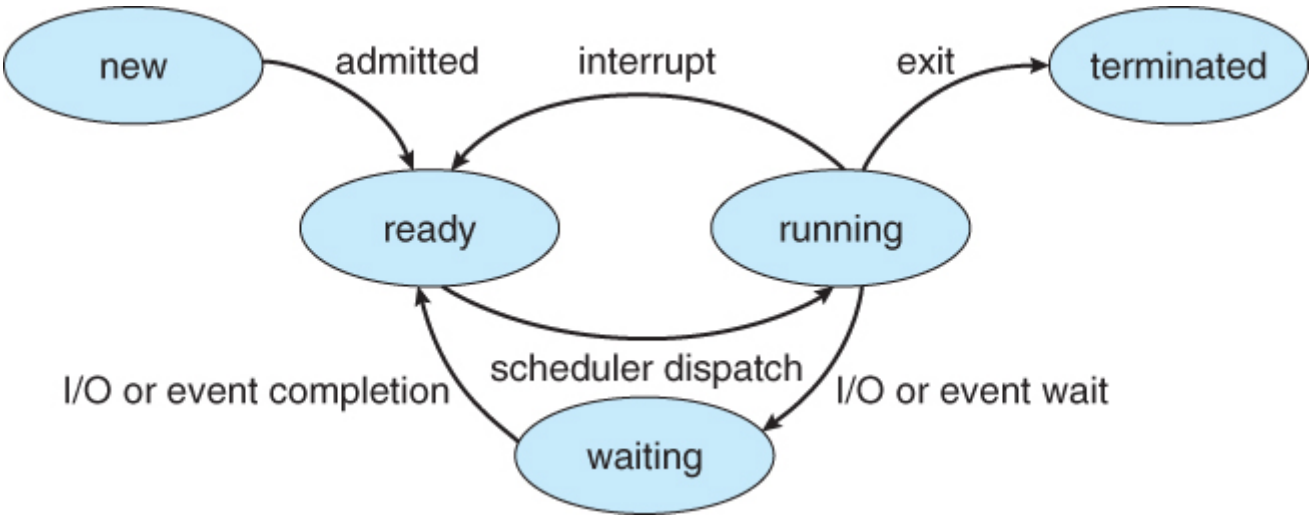
CSCI 375 Textbook Notes

Operating Systems

Chapter 5: CPU Scheduling

5.1: Basic Concepts

- The most basic idea of multiprogramming is to have some process running at all times such that CPU utilization is maximized
- CPU scheduling in order to divide CPU utilization is achieved through CPU scheduling
    - CPU scheduling is one of the **most fundamental** operating system functions
- **CPU-I/O Burst Cycle**
    - Process execution consists of a cycle between execution of instructions by the CPU and waiting for I/O operations
    - These "CPU Bursts" and "I/O Bursts" cycle between each other until the final CPU burst sends a system request to terminate execution
- **CPU Scheduler**
    - When the CPU does become idle, the CPU scheduler must choose one of the processes in the ready queue to be executed
    - The ready queue here is not necessarily a First-in First-out (FIFO) data structure, but can be implemented in a variety of ways
    - Below is a diagram representing the process state control mechanisms in an OS



- **Preemptive and Nonpreemptive Scheduling**
  - There are four conditions under which CPU scheduling decisions are made
    1. When a process switches from the running state to the waiting state
    2. When a process switches from the running state to the ready state
    3. When a process switches from the waiting state to the ready state
    4. When a process terminate
  - Under circumstances 1 and 4, there is no choice in terms of scheduling, and therefore we say that this scheduling scheme is nonpreemptive or *cooperative*
  - Otherwise, it is called *preemptive scheduling*
  - Preemptive scheduling can result in data race conditions leading to shared memory access inconsistencies
- **Dispatcher**
  - The dispatcher is a module involved in CPU scheduling which gives control of the CPU's core to the process selected by the scheduler
  - It involves the following processes
  - Switching contexts between processes
  - Switching to user mode
  - Jumping to the proper place in order to resume a program

    - The dispatcher should prioritize efficiency since it will be invoked for every context switch
    - **Dispatch latency** is the time it takes for the dispatcher to stop one process and start another
- How often do context switches occur?
    - You can use the Linux terminal command `vmstat` to find the number of context switches
    - First line is average number of context switches per second since boot
    - Next two lines are the average number of switches in the last two 1-second periods

5.2: Scheduling Criteria

- There are multiple different CPU scheduling algorithms and each can have different properties when it comes to selection of a new process
- The most substantial criteria when it comes to comparing CPU scheduling algorithms are as follows:
    - *CPU Utilization* which should be as close to 100% as possible
    - *Throughput* or the number of processes completed per time unit
    - *Turnaround time* or the amount of time it takes for a process to start and complete
    - *Waiting time* is the time spent by all processes in the waiting queue
    - *Response Time* often used instead of turnaround time as this is a more accurate measure of when a process is fully complete
- CPU Utilization and throughput should be maximized, whereas turnaround, waiting, and response time should be minimized

5.3: Scheduling Algorithms

- **First-Come, First-Served Scheduling**
    - This is by far the *simplest* form of CPU scheduling algorithm
    - Whichever process requests the CPU first is allocated the CPU first

    - This is inefficient as easy to complete processes might be stuck behind particularly complex or long-lasting processes
    - This algorithm is *nonpreemptive*, so it poses many issues in any sort of interactive system
- **Shortest-Job-First Scheduling**
    - This scheduling algorithm looks at the CPU burst time of the next CPU burst for all processes
    - It finds the shortest next CPU burst, and when the CPU is next available, that process is started
    - When there is a tie between two shortest bursts, the first-come first-served algorithm is used to break the tie
    - There is no way for the CPU scheduler to know the approximate length of the next CPU burst, so the next CPU burst is predicted using an exponentially weighted trailing average
        - Similar to the prediction of round-trip-time that TCP does
- **Round-Robin Scheduling**
    - This scheduling algorithm is similar to the first-come first-served algorithm but also introduces preemption to allow the system to switch between processes
    - Uses a defined amount of time known as a *time quantum* or *time slice*
    - After each time slice, a context switch occurs
    - Thus, the smaller the time slice, the more efficient the Round-Robin algorithm can be since more context switches will occur per unit time given there are multiple processes ready to run
- **Priority Scheduling**
    - In this algorithm, the system allocates priority levels to different processes
    - These priority levels can be defined either internally or externally
    - *Starvation* is a major problem of priority scheduling, as processes ready to run might be blocked due to having to low of a priority level while higher priority processes execute

- **Multilevel Queue Scheduling**
    - This algorithm combines priority and round-robin scheduling
    - When multiple processes are at the same priority level, the round robin algorithm is implemented to select between them
- **Multilevel Feedback Queue Scheduling**
    - This is similar to the previous algorithm, but allows processes to switch queues such as moving from the background to the foreground

5.4: Thread Scheduling

- On most modern operating systems, it is *kernel-level threads* - not processes - that are scheduled by the operating system
- *User-level threads* are managed by a user-level thread library of which the kernel is unaware
- In order to run on a CPU, user-level threads must be mapped to associated kernel-level threads through mapping that could be indirect or use a lightweight process, henceforth referred to as an LWP
- **Contention Scope**
    - Competition for the CPU takes place among different threads which belong to the same process
- **Pthread Scheduling**
    - Now we talk about the POSIX API that allows for the specification of PCS or SCS during thread creation
    - Pthreads identifies the following:
        - `PTHREAD_SCOPE_PROCESS` schedules threads using PCS scheduling
        - `PTHREAD_SCOPE_SYSTEM` schedules threads using SCS scheduling
    - For systems that implement the many-to-many multithreading model:
        - `PTHREAD_SCOPE_PROCESS` schedules user-level threads onto available LWPs
        - `PTHREAD_SCOPE_SYSTEM` will create and bind an LWP for each user-level thread, which essentially serves to make the many-to-many machine operate on a one-to-one thread mapping model

        - The Pthread IPC provides functions for setting and getting the contention scope policy
            - `pthread_attr_setscope(pthread_attr_t *attr, int scope)`
            - `pthread_attr_getscope(pthread_attr_t *attr, int *scope)`
            - The first parameter for both functions contains a pointer to the attribute set of the thread
    - The following code illustrates a Pthread scheduling API

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char* argv[])
  {

      int i, scope;

      pthread_t tid[NUM_THREADS];
      pthread_attr_t attr;

      //get default attributes
      pthread_attr_init(&attr);

      //inquire on current scope
      if(pthread_attr_getscope(&attr, &scope)!=0)
          fprintf(stderr, "Unable to get scheduling scope\n");
      else
      {

          if(scope == PTHREAD_SCOPE_PROCESS)

              printf("PTHREAD_SCOPE_PROCESS");

          else if(scope == PTHREAD_SCOPE_SYSTEM)

              printf("PTHREAD_SCOPE_SYSTEM");
          else
              fprintf(stderr, "Illegal scope value.n\")
      }
      //now, set the scheduling algorithm either to PCS or SCS
      for(i=0;i<NUM_THREADS;i++)
          pthread_create(&tid[i], &attr, runner, NULL);

      //now join on each thread
      for(i=0;i<NUM_THREADS;i++)
          pthread_join(tid[i], NULL);
  }

  //Now, each thread will begin control in the following function
  void *runner(void *param)
  {

      //do something....

      //......
      //......
      pthread_exit(0);
  }
```

5.5: Multi-Processor Scheduling

- So far, the discussion has centered mostly around systems that utilize only a single CPU core
- In a multi-core system, the issue of scheduling threads and processes becomes considerably more complex than in a single core system
- The term *multiprocessor* can refer to any of the following
    - Multicore CPU systems (most consumer systems)
    - Multithreaded Cores (most modern consumer systems)
    - NUMA systems
    - Heterogeneous multiprocessing
- **Approaches to multi-processor scheduling**
    - *Asymmetric* processing
        - Only one core access the system data structures
        - Introduces potential bottlenecks where system performance might be impacted
    - *Symmetric* processing
        - Each processor is self-scheduling
        - Each processor has a schedule that will examine the ready queue and select a thread to run based on the implemented scheduling algorithm
        - There are two possible thread organization structures when using symmetric multiprocessing
            - All threads can be stored in a common ready queue
            - Each processor can have its own private queue of threads
            - The first structure introduces potential data races and data dependencies, whereas the second structure is less efficient on average due to threads being limited to certain processors
- **Multicore Processors**

    - In modern CPUs, each core retains the architectural state of a traditional CPU and thus each core is recognized by the system as its own logical CPU
    - Because CPUs operate at a much higher speed than RAM, the CPU might wait significant amounts of time waiting for memory access to become available
        - This is known as a **memory stall**
        - It can also occur because of a cache miss
    - Multithreaded processing cores now contain two (or more) hardware threads such that if one hardware thread stalls while waiting for memory, another hardware thread can be used
    - The technique where a single physical CPU acts as many logical CPUs is known as chip multithreading (CMT)
    - Modern Intel processors use the term *hyper-threading* to describe this technique
        - The Intel i5-12400f, the CPU in my gaming PC, has 6 cores, with two threads per core, leading to 12 logical CPUs on the system
        - The Oracle Sparc M7 processor supports 8 threads per core, of which it has 8, so 64 logical CPUs
    - In general, there are two ways to multithread a processing core
        - *Coarse-grained* multithreading
            - A thread executes on a core until a long latency event (such as a memory stall)
            - Comes with a high cost of thread-switching
        - *Fine-grained* multithreading
            - Threads switch on a finer level of granularity, typically at the boundary of an instruction cycle
            - Thus, the cost of thread switching is considerably lower
- **Load Balancing**
    - In order to keep the utilization of a system near maximum, the balancing of loads between processing cores is an important topic

    - Load balancing is mostly dealt with in systems that use a private queue for each processing core, as a shared queue would make load balancing occur implicitly

- **Processor Affinity**
    - If a process must switch between threads, the original cache must be invalidated and the new cache must be repopulated such that the process can continue running
    - Because of the high overhead of switching CPU caches, most operating systems prefer to keep processes running on the same processor and take advantage of a *warm cache* which still contains all relevant data
    - This is known as **processor affinity**
        - This means that a process has an affinity for the processor on which it is currently running
    - If a system is using private ready queues for each thread, processor affinity is implicitly achieved as processes will not move between different processors
    - Multiple forms of processor affinity
        - *Soft affinity* where an operating system has a policy of attempting to keep a process running on the same processor
        - *Hard affinity* where an operating system has a policy of keeping a process running only on a specified subset of processors

- **Heterogeneous Multiprocessing**
    - In order to better manage power consumption and efficiency, heterogeneous multiprocessing allows different CPU cores tasked with similar instruction sets to operate at different clock speeds and power consumption levels
    - Think of Intel's *P-Cores* (Performance Cores) and *E-Cores* (Efficiency Cores)

5.6: Real-Time CPU Scheduling

- **Minimizing Latency**
    - Two types of latency
        - *Interrupt Latency* which refers to the period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt (i.e. the "reaction time" of the CPU)

        - *Dispatch Latency* refers to the amount of time required for the scheduling dispatcher to stop one process and start another
    - In dispatch latency, the conflict phase has two components
        - The preemption of any process running in the kernel
        - A transfer of resources from low-priority processes to higher-priority processes
- **Priority-Based Scheduling**
    - In a *real-time* operating system, the most important feature is an immediate response to a real-time process as soon as that process requires the CPU
- **Rate-Monotonic Scheduling**
    - This scheduling algorithm schedules periodic tasks using a static priority policy with preemption
    - If a low priority process is running, and a higher priority process becomes available to run, the first process will be preempted
    - The shorter the period of a task, the higher its priority
    - Rate-Monotonic scheduling assumes that time of each CPU burst is *identical*
- **Earliest Deadline First Scheduling**
    - This scheduling algorithm assigns priority levels to processes dynamically according to their deadline
        - The earlier the deadline of a process, the higher priority it will be given
    - Under this system, whenever a process becomes runnable, it must make its *deadline requirements* known to the system
- **Proportional Share Scheduling**
    - In a proportional share scheduler, T shares are allocated among all applications and an application can receive N shares of time
        - Each application will thus have $\frac{N}{T}$ of the total processor time
    - These schedulers must work together with an admission-control policy in order to guarantee that an application receives its allocated shares of time

- **POSIX Real-Time Scheduling**
  - Here, the POSIX API for scheduling real-time threads is covered
  - POSIX defines two scheduling classes for real-time threads
    - `SCHED_FIFO` which schedules threads according to a first-come first-served policy
    - `SCHED_RR` which schedules threads according to a round-robin policy
    - `SCHED_OTHER` is also provided by POSIX, but its implementation is undefined and it may behave differently on different systems
    - Similar to getting and setting the scope, POSIX API provides functions for setting and getting the scheduling policy, namely
      - `pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy)`
      - `pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy)`
  - The following is an implementation of POSIX real-time scheduling

```c
    #include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{

    int i, policy;

    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;

    //get default attributes
    pthread_attr_init(&attr);

    //get current scheduling policy
    if(pthread_attr_getschedpolicy(&attr, &policy)!=0)
        fprintf(stderr, "Unable to get policy\n");
    else
    {

        if(policy==SCHED_OTHER)

            printf("SCHED_OTHER\n");
        else if(policy==SCHED_RR)
            printf("SCHED_RR\n");
        else if(policy==SCHED_FIFO)
            printf("SCHED_FIFO\n");
    }

    //set the scheduling policy
    if(pthread_attr_getschedpolicy(&attr, SCHED_FIFO)!=0)
        fprintf(stderr, "Unable to set policy\n");

    //create threads
    for(i=0;i<NUM_THREADS;i++)
        pthread_create(&tid[i], NULL);

    //join on each thread
    for(i=0;i<NUM_THREADS;i++)
        pthread_join(tid[i], NULL);
}

//threads begin control in this function
void *runner(void *param)
{

    //do some work
```

```
    pthread_exit(0);
}
```

5.7: Operating System Examples

- **CPU Scheduling Examples in Linux**
  - Traditionally, Linux used the UNIX scheduling algorithm which was not designed with multiprocessing systems in mind
  - In Linux, scheduling is based on *scheduling classes* which are each assigned a different priority
    - Through the use of scheduling classes, Linux is able to implement a variety of different scheduling algorithms based on different situations
  - Rather than using strict priority values, Linux assigns a proportion of CPU processing time to each task
    - It does this using *nice values* which range from -20 to 19
    - It assigns a process a *targeted latency*, which is an interval of time during which every runnable task should run at least once
- **CPU Scheduling Examples in Windows**
  - The portion of the windows kernel that handles scheduling is called the *dispatcher*
  - Windows schedules use a priority-based, preemptive scheduling algorithm
  - The Windows API identifies the following six priority classes to which a process can belong
    - `IDLE_PRIORITY_CLASS`
    - `BELOW_NORMAL_PRIORITY_CLASS`
    - `NORMAL_PRIORITY_CLASS`
    - `ABOVE_NORMAL_PRIORITY_CLASS`
    - `HIGH_PRIORITY_CLASS`
    - `REALTIME_PRIORITY_CLASS`

    - Typically, processes are members of the `NORMAL_PRIORITY_CLASS` unless its parent was a part of the `IDLE_PRIORITY_CLASS` or if another class was specified when the process was created
    - In Windows, the priority class of a process, except `REALTIME_PROCESS_CLASS` is variable and can be changed using the `SetPriorityClass()` function in the Windows API
    - Within a priority class, a process can have a relative priority from the following values
        - `IDLE`
        - `LOWEST`
        - `BELOW_NORMAL`
        - `NORMAL`
        - `ABOVE_NORMAL`
        - `HIGHEST`
        - `TIME_CRITICAL`
    - Windows distinguishes between *foreground* and *background* processes and increases the scheduling quantum for foreground processes to assist user experience
    - Windows 7 introduced *user-mode scheduling* which allows applications to create and manage threads completely independently of the kernel

5.8: Algorithm Evaluation

- What are the most important criteria when it comes to selecting a CPU scheduling algorithm?
    - Maximizing CPU utilization
    - Maximizing throughput of a processor such that the average turnaround time is linearly proportional to the total execution time
- **Deterministic Modeling**
    - Deterministic modeling uses analytical evaluation using the given algorithm to produce some formula, equation, or number that describes or evaluates the performance of that algorithm
    - Deterministic modeling can be good for identifying trends across large data sets

- **Queuing Models**
    - Because the processes that will be run can be unpredictable on many end systems, no static set of processes can be used to do deterministic modeling
    - However, the distribution of CPU and I/O bursts can be measured and subsequently approximated
    - *Queuing network analysis* is the area of study that involves treating the CPU scheduling system as a network of connected devices
        - Let n be the average long-term queue length
        - Let λ be the average arrival rate for new processes in the queue
        - Let W be the average waiting time in a queue
        - **Little's Formula** $n = \lambda * W$
- **Simulation**
    - Simulations of different loads on different scheduling algorithms can be used to test their efficacy across different scenarios
- **Implementation**
    - Even a simulation might not be perfectly suited for testing, so implementations must also be put in place and tested in order to adequately select the correct scheduling algorithm for any given task

Justin Ciocoi

Oct. 20, 2023

CSCI 375 Operating Systems

Chapter 6 Synchronization Tools

Textbook Notes

Section 6.1: Background

- This chapter will largely focus on how concurrent or parallel execution can pose a threat to the integrity of data shared by several processes
- We can first return to the example of a bounded-size-buffer from chapter 3:

```
while(true)
{
    //produce an item in next_produced
    while(count==BUFFER_SIZE)
    {
    //do nothing
    }
    buffer[in]=next_produced;
    in=(in+1)%BUFFER_SIZE;
    count++;
}
```

- A corresponding consumer process is also similarly defined
- The consumer process will end with the line `count--`, which can lead to a data race for the shared variable count, since the `count++` and `count--` lines are executed as three lines in machine code and can be jumbled, resulting in incorrect or inconsistent output
    - This is called a *race condition*
- Because of issues such as these, much of this chapter will focus on *process synchronization* and *coordination*

6.2: The Critical Section Problem

- Consider a system consisting of n processes, $[P_0, P_1, ..., P_{n-1}]$
- In each process, there is a section of code called the *critical section* during which a process may be accessing or updating a data member that is shared with other processes
- The *critical section problem* refers to the problem which we will attempt to find solutions to in this section:
    - How can we design a *protocol* such that processes can synchronize their activity in order to *cooperatively share data*
- Generally, this will be implemented in a section preceding the section called an *entry section*
- The *critical section* can also be followed by an exit section that allows other processes access to shared data
- The *remainder section* contains the rest of the process code that does not need access to shared data

```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

- A solution to the critical section problem must satisfy the following three conditions
    - *Mutual Exclusion*
        - If a process, P_i, is executing in its critical section, then no other processes can be executing in their critical sections
    - *Progress*
        - If there is no process executing in its critical section, and some processes wish to enter the critical section, only processes not executing their remainder sections can participate in the decision making

        - The selection of the next process cannot be postponed indefinitely, thus progress is inevitable
    - *Bounded Waiting*
        - There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
- Generally speaking, there are two approaches used to handle critical sections in modern operating systems
    - Preemptive Kernel
    - Nonpreemptive kernel
- A *nonpreemptive* kernel is essentially free from any data race conditions since only one process is active in the kernel at a time
- While a nonpreemptive kernel does provide this advantage, it doesn't account for the possibility of a kernel process executing for an arbitrarily long time leading to low responsiveness on a system
- Therefore, most modern operating systems utilize preemptive kernel approaches on multi-processor systems despite the difficulty in implementation

---

6.3: Peterson's Solution

- There is one classic software-based solution to the critical section problem known as *Peterson's solution*
- Peterson's solution is limited to two processes, so in practice it is not very useful, but it presents a good look at the algorithmic fundamentals for solving the critical section problem
- For this section, we will refer to the two processes as P_0 and P_1
- Petersons solution requires processes to share *two* data items
    - ```
      int turn;
      bool flag[2];
      ```
- Below is the general structure of process P_i where $j = i - 1$
      ```

- ```
  while(true)
  {
      flag[i] = true;
      turn = j;
      while(flag[j] && turn == j)
      {
          //wait
      }

      /*Critical Section*/

      flag[i] = false;

      /*Remainder Section*/
  }
  ```

- Here, the variable `turn` indicates whose turn it is to enter the critical section
  - If `turn==i`, then process P_i is allowed to execute in its critical section
- The `flag[]` array is used to indicate if a process is *ready* to enter its critical section
  - if `flag[i]==true` then P_i is ready to enter its critical section
- In the above code snippet, process P_i first sets `flag[i] = true`, meaning process P_i is ready to enter its critical section and sets `turn = j` to allow process P_j to finish if it is in its critical section
  - `while(flag[j] == true && turn = j)` P_i will wait
- Once P_j has exited its critical section, it will set `flag[j] = false` and P_i will then proceed to its critical section
- As another example, consider the following data that is being shared between two threads
  - ```
    bool flag = false;
    int x = 0;
    ```
- Here, thread 1 performs the statements
  - ```
    while(!flag)
    {};
    print x;
    ```
    ```

- Thread 2 performs the statements
    - ```
      x = 100;
      flag = true;
      ```
- Here, the expected behavior would be that thread 1 outputs the value 100 for variable `x`
- However, machine level instructions may be broken down in an order that leads to *unresolved data dependencies* in the code leading to *inaccurate or inconsistent results* in a program

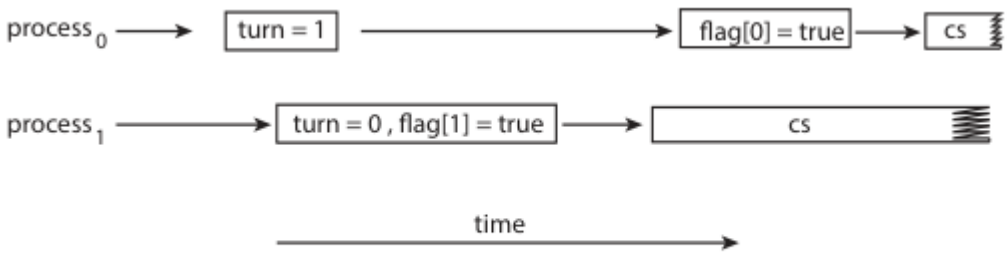


Figure 6.4 The effects of instruction reordering in Peterson's solution.

- In this case, `flag` is assigned a true value before the assignment of `x = 100`
    - Thus, thread 1 will output a value of 0 instead of the expected 100
- For a simple video explanation of Peterson's solution, click here

---

6.4: Hardware Support for Synchronization

- Peterson's solution is a software implementation, as it requires no additional hardware support to function
- This chapter will analyze hardware support for process synchronization
- **6.4.1: Memory Barriers**
    - In the previous section, we noted how a system may reorder machine level instructions, potentially leading to unreliable data states
    - The memory *guarantees* that a system will provide to an application program is determined primarily by a system's ***memory model***
    - Generally, a memory model will fall into one of two categories
        - *Strongly Ordered*, where a memory modification one on processor is immediately visible to all other processors
      ```

        - *Weakly Ordered*, where modifications to memory on one processor may not be immediately visible to other processors
    - To address issues in the communication of memory modifications, computer architectures provide instructions that can *force* any changes in memory to be propagated to all other processors, thereby ensuring that modifications are visible to threads running on other processes
    - These instructions are referred to as **memory barriers** or **memory fences**
        - Whenever a memory barrier instruction is performed, the system will ensure that all loads and stores will be completed before any subsequent load or store operations are performed

- **6.4.2: Hardware Instructions**

    - Many modern computer systems provide special hardware instructions that allow us either to test and modify the content of a word, or to swap the contents of two words *atomically*, meaning as one uninterruptible unit

    - Rather than discussing one specific instruction on one specific machine, this section will abstract these main concepts behind the `test_and_set()` and `compare_and_swap()`

    - First, let us define the `test_and_set()` instruction

        - ```
          bool test_and_set(bool *target)
          {
              bool rv = *target;
              *target = true;

              return rv;
          }
          ```

    - Now let us illustrate an implementation of mutual exclusion using the `test_and_set()` function
          ```

```
do{
    while(test_and_set(&lock))
    {
    ; //do nothing
    }

    // critical section

    lock = false;

    // remainder section
} while(true);
```

- Similar to the `test_and_set()` function, the `compare_and_swap()` function acts on a word atomically, but differs in its mechanism, which is based on swapping the content of two words
- First is the definition of the `compare_and_swap()` function

```
int compare_and_swap(int *value, int expected, int new_value)
{
    int temp = *value;

    if(*value==expected)
        *value = new_value;

    return temp;
}
```

- As we can see by looking at the function, the value of `*value` is replaced by `new_value` only if `*value==expected` is true
- However, the function will return the original value regardless
- Now, we look at an implementation of mutual exclusion using the `compare_and_swap()` function

```
while(true)
{
    while(compare_and_swap(&lock, 0, 1) != 0)
        ;//do nothing

    //critical section

    lock = 0;

    //remainder section
}
```

- In the above, a global variable, `lock`, was declared and initialized to 0
- The first process which invokes the `compare_and_swap()` function will set `lock` to 1
- Now, subsequent calls to the `compare_and_swap()` function will not succeed because `lock` is now not equal to the expected value of 0
- When the process exits its critical section, we can see that it sets the `lock` back to 0, thus allowing other processes to enter their critical sections
- While this code does satisfy the mutual exclusion and progress portions of the critical section problem, bounded waiting is not achieved
- Here, we will present another algorithm using the `compare_and_swap()` instruction that *does* achieve bounded waiting

```
while(true)
{
    waiting[i] = true;
    key=1;
    while(waiting[i] && key == 1)
        key = compare_and_swap(&lock, 0, 1);
    waiting[i] = false;

    // critical section

    j = (i+1) % n;
    while((j!=1) && !waiting[j])
        j = (j+1) % n;

    if(j == i)
        lock = 0;
    else
        waiting[j] = false;

    //remainder section
}
```

- In this algorithm, common data structures shared by processes P_i and P_j are
  - `bool waiting[n]`
  - `int lock`
- We can see that when a process exits its critical section, it scans the array waiting in the cyclic ordering, thus achieving bounded waiting

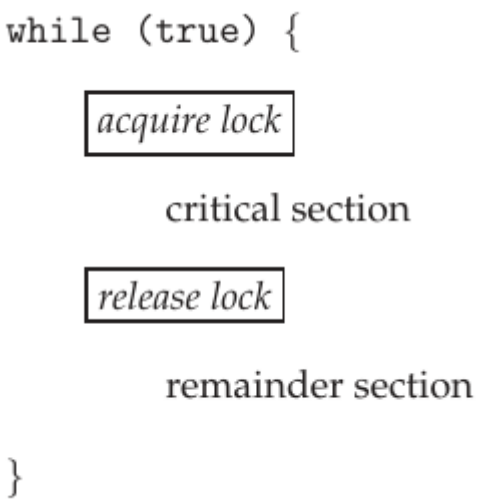
- **6.4.3: Atomic Variables**
  - Typically, the `compare_and_swap()` instruction is not used to directly provide a solution to the critical section problem, but rather to serve as a building block for other tools that aim to solve the critical section problem
  - One of these tools is known as an *atomic variable*, which provides atomic operations on basic data types such as integers and booleans
  - Atomic variables can be used to ensure mutual exclusion in situations where there may be a data race on a single variable while it is being updated, as when a counter is incremented.
  - Although atomic variables do provide atomic updates, they do not entirely solve race conditions in all circumstances

- Atomic variables are very commonly used in operating systems as well as in concurrent applications, although their use is usually only for single updates of shared data such as counters and sequence generators

---

6.5 Mutex Locks

- The hardware based solutions that were examined in section 6.4 are largely complicated and inaccessible methods that leave a lot to be desired in the world of application programming
- Instead of using these, operating-system designers use higher-level software tools to solve the critical section problem
- The simplest of these tools is the *mutex lock*, where *mutex* is short for *mut*ual *ex*clusion
- A process must first acquire a lock before entering its critical section, and then release the lock upon completion of its critical section



- If the lock is available, a call to acquire succeeds, and the lock is then considered unavailable
- A process that attempts to acquire an unavailable lock is blocked until that lock is released
- The definition of `acquire()` is as follows

```
acquire()
{
    while(!available)
        ; //busy wait
    available = false;
}
```

- The definition of `release()` is as follows

```
release()
{
    available = true;
}
```

- The calls to either acquire or release a mutex lock must be performed atomically, and are thus implemented using the `compare_and_swap()` function used in section 6.4

6.6: Semaphores

- Mutex locks are generally considered the simplest form of synchronization tools
- In this section, we can examine a more robust tool that can act in a similar fashion to the mutex lock, but can also provide even more sophisticated ways in which process may synchronize their activities
- A ***Semaphore*** `S` is an integer variable that - apart from initialization - can be accessed only through two standard atomic operations, `wait()` and `signal()`
- The definition of `wait()` is as follows

```
wait(S)
{
    while(S <= 0)
        ; // busy wait

    S--;
}
```

- The definition of `signal()` is as follows

```
signal(S)
{
    S++;
}
```

- Semaphore values can only be modified atomically, that is, by one process at any given time
- **6.6.1: Semaphore Usage**
  - Oftentimes, operating systems will distinguish between counting semaphores and binary semaphores
  - In a *counting* semaphore, the value of the semaphore can range over an unrestricted domain
  - In a *binary* semaphore, the value of the semaphore can range only between 0 and 1
    - This indicates that binary semaphores will operate in a manner fairly reminiscent of mutex locks
  - Now, let us consider two concurrently running processes, P_1 with a statement S_1 and P_2 with a statement S_2 and suppose that S_2 can be executed only *after* S_1 has completed
  - We could implement it in the following way
    - ```
      //Process P1
      {
          S1;
          signal(synch);
      }
      ```
    - ```
      //Process P2
      {
          wait(synch);
          S2;
      }
      ```
- **6.6.2: Semaphore Implementation**
  - The implementations of the mutex lock in section 6.5 suffers from busy waiting, and the above definitions of the `wait()` and `signal()` functions present the same problem

- To overcome this problem, we can redefine the `wait()` and `signal()` semaphore operations
  - ```
    wait(semaphore *S)
    {
        S->value--;
        if(S->value < 0)
        {
            add this process to S->list;
            sleep();
        }
    }
    ```
  - ```
    signal(semaphore *S)
    {
        S->value++;
        if(S->value <= 0)
        {
            remove process P from S->list;
            wakeup(P);
        }
    }
    ```

Justin Ciocoi

Nov. 8, 2023

CSCI 375 Slide Notes

Chapter 7: Synchronization Examples

- In the world of synchronization, there are a few classical problems which are often used to test newly-proposed synchronization schemes
- These include the following
  - *Bounded-Buffer Problem*
  - *Readers and Writers Problem*
  - *Dining Philosophers Problem*

Bounded-Buffer Problem

- In the bounded-buffer problem, there are n buffers, and each is capable of holding one item
  - The finite number of buffers here places a limit on how many items can be stored waiting for consumption
- Three semaphores are used
  - Semaphore `mutex` is initialized to 1
    - This is a *binary semaphore*, (0 or 1 *only*) which protects the buffer, ensuring mutual exclusion when producers or consumers access it to add or remove an item
  - Semaphore `full` is initialized to 0
    - This is a *counting semaphore*, which tracks the number of items in the buffer, ensuring that a consumer does not try to consume an empty buffer
  - Semaphore `empty` initialized to n
    - This is a *counting semaphore* which tracks the number of empty slots in the buffer, ensuring producers don't produce when the buffer is full

- The structure of a *writer process* is as follows

```
do
{
        ...
        //produce item in next produced
        ...
    wait(empty);
    wait(mutex);
        ...
        //add next produced to buffer
        ...
    signal(mutex);
    signal(full);
}
while(true);
```

  - First off, the producer will create the item to be placed in the buffer without accessing any shared data
  - `wait(empty)` waits for empty to be non-zero and then decrements it since it will be adding an item to the buffer, thus reducing the number of empty slots
  - Then, `wait(mutex)` ensures that the producer has exclusive access to the buffer before beginning to write to it
  - The writer process will then add an item to the buffer
  - `signal(mutex)` will allow other processes access to the buffer
  - `signal(full)` increments the full semaphore, signaling to all processes that the buffer is full

- The structure of a *consumer process* is as follows

- ```
  do
  {
      wait(full);
      wait(mutex);
          ...
      //remove an item from buffer to next consumed
          ...
      signal(mutex);
      signal(empty);
          ...
      //consume the item in next consumed
          ...
  }
  while(true);
  ```

  - Before it can remove an element from the buffer, the consumer process must first perform two wait operations
  - `wait(full)` waits for a producer process to call `signal(full)` indicating to consumers that there is a buffer ready to be consumed
  - `wait(mutex)` then ensures exclusive access to the buffer, after which the consumer will remove the item from the buffer
  - Once the item is removed, shared data access is no longer needed, so the consumer can call `signal(mutex)` and `signal(empty)`, allowing producers to produce into the buffer
- In the bounded-buffer problem, it is important to note that the `mutex` semaphore exists to provide exclusive access to the buffer, and the `full` and `empty` semaphores control the flow of production based on the state of the buffer (whether it is full or empty)

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - *Readers*, who only read the data set and do not perform any updates
  - *Writers*, who can both read *and* write to the shared data set
- The problem is allowing as many readers as want to access the shared data, but only allow one writer at a time
- There are several variations of how readers and writers are considered, and all involve some form of priority
  ```

- Shared Data
  - Data set
  - Semaphore `rw_mutex` initialized to 1
    - This semaphore ensures mutual exclusion for the writer processes by not allowing other writers or readers to access a dataset that is currently accessed by a writer
  - Semaphore `mutex` initialized to 1
    - This semaphore protects the `read_count` variable, which keeps track of the number of readers
  - Integer `read_count` initialized to 0
- The structure of a *writer process* is as follows
- ```
  do
  {
      wait(rw_mutex);
          ...
          //writing is performed
          ...
          signal(rw_mutex);
  }
  while(true);
  ```
  - `wait(rw_mutex);` waits for permission to write to the dataset, blocking if another writer is currently writing
  - Once `rw_mutex` is acquired the writer can safely write to the dataset
  - `signal(rw_mutex)` releases the `rw_mutex`, allowing other writers or readers to access the dataset
- The structure of a *reader process* is as follows
  ```

- ```
  do
  {
          wait(mutex);
          read_count++;
          if(read_count==1)
              wait(rw_mutex);
      signal(mutex);
          ...
          //Reading is performed
          ...
      wait(mutex);
          read_count--;
          if(read_count==0)
              signal(rw_mutex);
          signal(mutex);
  }
  while(true);
  ```

  - `wait(mutex)` waits for permission to modify the `read_count` variable
  - Once `mutex` is acquired, the reader process increments the read count
    - If this is the first reader, the process will also wait to acquire `rw_mutex` if it is not the first reader, it can assume the first reader had already waited for `rw_mutex` and subsequently blocked any writers until all readers have exited
  - Since `read_count` has now been incremented, *and* `rw_mutex` has been acquired, the reader can `signal(mutex)`, allowing other reader processes to modify the read count, and proceed to the critical section (reading)
  - Once reading is finished, the reader once again calls `wait(mutex)` such that it can safely decrement the `read_count` variable
  - If `read_count==0`, we know the last reader has exited the critical section, and can thus `signal(rw_mutex)` allowing writers to now enter their critical sections
- In this processes, the `while(true)` condition is used to indicate an infinite loop where readers and writers will indefinitely continue their operation
  - In real-world applications, this condition would likely be replaced with a more specific one, such as having the reader/writer exit after reading/writing a certain number of entries

Dining Philosophers Problem
  ```

- In this problem, we assume that philosophers (*processes*) alternate between eating (*executing critical section*) and thinking (*waiting for access to critical section*)
- These philosophers do *not* interact with their neighbors, but they will occasionally try to pick up two chopsticks (one at a time) to eat from the bowl (*shared data*)
    - They need both chopsticks to eat, and then will release both when they are done
- In the case of the 5 philosophers, there is the following shared data
    - Bowl of rice (*dataset*)
    - Semaphore `chopstick[5]` initialized to 1
- The structure of philosopher i is as follows:
- ```
  do
  {
      wait(chopstick[i]);
      wait(chopstick[(i+1)%]);

          //eat

      signal(chopstick[i]);
      signal(chopstick[(i+1)%5]);

          //think
  }
  while(true);
  ```
- However, there is a problem with this algorithm
    - Since philosophers pick up chopsticks one at a time, a deadlock can occur if all philosophers simultaneously pick up one chopstick
  ```

- There are a few ways to avoid the deadlock presented here
    - *Resource Hierarchy*, such that the philosophers must first acquire the lower numbered chopstick, followed by the higher numbered one
    - *Resource Allocation Limit*, such that the total number of philosophers trying to eat is limited
    - *Chopstick Timeout*, such that a philosopher picks up one chopstick, it will release it after a set amount of time if the other chopstick cannot be picked up
        - This is the method Professor Dogshit mentioned in class

Justin Ciocoi

Dec. 7, 2023

CSCI 375 Operating Systems

Chapter 8: Deadlocks

- In a multiprogramming environment, several threads might be competing for a finite amount of resources
- If a thread is waiting for a resource, but because of some activity, is never permitted to leave the waiting state, we say the thread has entered a *deadlock*

8.1: System Model

- Any system consists of a finite number of resources, and in most modern systems these resources must be distributed across a number of competing threads
- Each resource may accomplish a different task and have a different number of instances
- Any instance of a resource class should satisfy a call from a thread for that type of resources
    - If this is not the case, then the resource classes have been designed improperly by the operating system designer
- Mutex locks and semaphores can also be considered system resources, and on modern systems are the most likely source of a deadlock
- However, "lock" is not defined as a resource class since different locks will control access to different shared data, so each lock will be granted its own resource class
- This chapter will discuss deadlocks occurring with kernel resources, but it is also possible for a deadlock to occur between process such as in the case of ongoing interprocess communication
- Under the normal mode of operation, a thread may utilize a resource in only the following sequence
    1. **Request**: The thread requests the resource, which is either granted, or not, which places the thread into a waiting state
    2. **Use**: The thread can operate on the resource

3. **Release**: Once the thread has completed using the resource, it can release the resource such that other threads may now access it

8.3: Deadlock Characterization

- **8.3.1: Necessary Conditions**
    - A deadlock situation can arise in a system if and only if the following four properties hold simultaneously
        1. *Mutual Exclusion*
        2. *Hold and Wait*
        3. *No Preemption*
        4. *Circular Wait*
- **8.3.2: Resource Allocation Graph**
    - A system's *resource allocation graph* is a visual representation of a systems resources and the various different requesting threads in a system
    - Each graph has a set of vertices, V, and a set of edges, E
    - The set V is partitioned into two different types of nodes
        - $T = \{T_1, T_2, ..., T_n\}$, which is the set consisting of all active threads in the system
        - $R = \{R_1, R_2, ..., R_n\}$, which is the set consisting of all resource types in a system
    - An edge directed from a thread to a resource type denotes a request for that particular resource type
    - An edge directed from a resource type to a thread indicates that that thread currently is holding said resource type
    - If a resource allocation graph does not have a cycle, then the system is *not* in a deadlocked state
    - If there is a cycle, then the system *may or may not* be in a deadlocked state

8.4: Methods for Handling Deadlocks

- Generally speaking, there are three ways by which the problem of deadlocks can be dealt with
    - They can be ignored altogether and we can pretend that deadlocks will never occur in a system
    - We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state
    - We can allow the system to enter the deadlocked state, detect it, and then recover
- The first solution is the one that is most commonly used by modern operating systems including Windows and Linux
- Thus, it is usually up to kernel and application developers to write programs that handle deadlocks, typically using approaches outlined in the second solution
- In order to ensure deadlocks never occur, a system could use either a *deadlock prevention*, or *deadlock avoidance* scheme
- Deadlock prevention provides a set of methods by which a system will endure that at least one of the necessary conditions for a deadlock cannot hold
- Deadlock avoidance requires that the operating system be given additional information in advance concerning which resources a thread will request and use during its lifetime
    - With this knowledge, the operating system is capable of making threads wait or not in a manner that will altogether avoid deadlocks arising in the system

8.5: Deadlock Prevention

- Here we will examine different deadlock prevention schemes which aim to eliminate one of the four necessary conditions for deadlocks
- **8.5.1: Mutual Exclusion**
    - The mutual exclusion condition must hold, since there are certain system resources which are inherently non-shareable, such as a mutex lock
- **8.5.2: Hold and Wait**
    - In order to ensure that the hold and wait condition never occurs, we must guarantee that when a thread requests a resource, it does not already hold any other resources

    - We could allow only threads holding no resources to request resources, or make each thread request all resources at once, but either solution will cause low resource utilization and task starvation

- **8.5.3: No Preemption**
    - In order to make sure no preemption does not occur, we can simply create a protocol such that if a thread requests a resource and must wait, it will then release all resources\
    - Alternatively, we can first check whether the resources are available and then take action based upon that, granting the requested resources if they are available
- **8.5.4: Circular Wait**
    - The prior deadlock prevention schemes are largely impractical for a variety of reasons
    - However, the circular wait condition presents an opportunity for a much more practical solution
    - To illustrate this, we will let $R = \{R_1, R_2, ..., R_n\}$ be the set of resource types where each type has a unique integer
    - Formally, we will define a one-to-one function $F : R \to N$ where N is the set of natural numbers

$$F(\text{first\_mutex}) = 1$$
$$F(\text{second\_mutex}) = 5$$

    - Now imagine each thread can request resources only in an increasing order of enumeration
        - This means that a thread can request R_i, and then R_j only if $F(R_j) > F(R_i)$
    - For example, a thread that wants to use both first_mutex and second_mutex, must first request first_mutex, and then second_mutex
    - If these two protocols exist, then the circular wait cannot hold

8.6: Deadlock Avoidance

- An alternative method for ensuring that deadlocks to not occur in a system is deadlock avoidance

- In this method, the system will obtain additional information about resources, threads, and resource usage, and then be able to make scheduling decisions based on this information in a manner in which deadlocks will not occur
- **8.6.1: Safe State**
    - A state is *safe* if the system can allocate resources up to each thread's maximum in some order and still avoid a deadlock
    - More formally, we can say that a system is in a safe state if there exists a safe sequence
    - It is important to note that not all unsafe states are deadlocks, and instead the presence of an unsafe state merely implies the *possibility* of a deadlock

Justin Ciocoi

Dec. 16, 2023

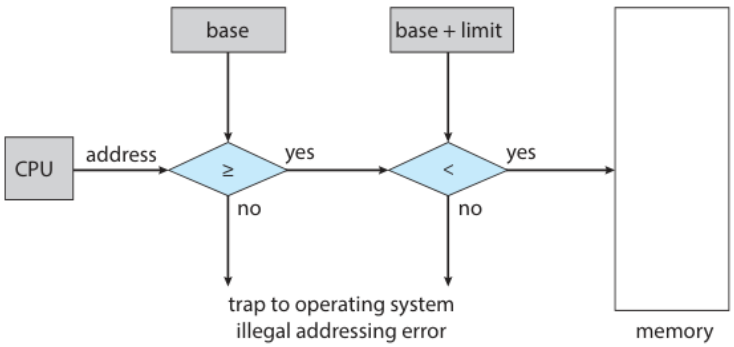
CSCI 375 Textbook Notes

Chapter 9: Main Memory

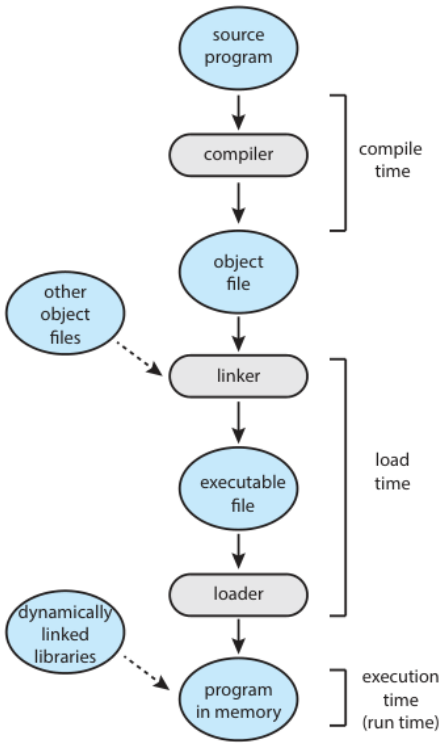
9.1: Background

- Main memory consists of a large array of bytes each with its own address
- A typical instruction0execution cycle will first fetch an instruction from memory, then decode the instruction, which may cause operands to be fetched from memory, then execute and possibly store results back in memory
- We can ignore how a program generates memory addresses, as from the point of view of a memory unit, we see only a continuous stream of memory addresses and are not necessarily interested in their methods of origin
- **9.1.1: Basic Hardware**
    - Main memory and the registers built into each processing core are the only general-purpose storage that the CPU can access directly
    - Therefore, if data is not in memory or in a CPU register, it must first be moved into memory before the CPU can execute on it
    - CPU registers are generally accessible once per every tick of the CPU clock, but completing memory accesses may take many cycles of the CPU clock, and in these cases, the processor must stall
        - In order to ease the issue of stalling, caches were introduced for CPUs such that more frequently used or likely to be used memory will be held in fast memory between the CPU and main memory
    - Speed is not the only consideration in the accessing of physical memory, as we must also ensure correct operation
    - We have to make sure that each process has a separate memory space

        - This protects processes from each other and is fundamental to having multiple processes loaded into memory for concurrent execution
    - To do this, we must first have a method by which to find the range f legal memory addresses which processes may enter
    - We can provide this function by using two registers, namely the *base register* and *limit register*
        - The base register will hold the smallest legal physical memory address
        - The limit register specifies the size of the range
    - Protection of the memory space is accomplished by having CPU hardware compare every address generated in user mode with the registers
    - This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users
    - The base and limit registers can be loaded only by the operating system, which prevents programs from changing the registers' contents

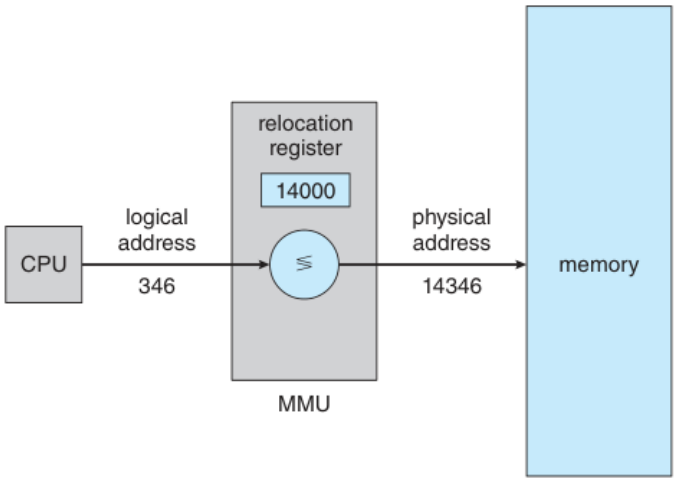


- **9.1.2: Address Binding**
    - In most cases, a program resides on a disk as a binary executable file
    - In order to run a program, the program must be brought into memory and placed within the context of a process where it becomes eligible for execution on an available CPU
    - As the process executes, it accesses instructions and data from memory, and once it eventually terminates its memory is reclaimed for use by other processes
    - Here we can see a visual representation of the multi-step processing of a user program



- **9.1.3: Logical Vs. Physical Address Space**
  - An address generated by the CPU is commonly referred to as a *logical address*
  - An address seen by the memory unit, that is, the one loaded into the memory-address register of the memory, is commonly referred to as a *physical address*
  - Binding addresses at either compile or load time generates identical logical and physical addresses
  - However, the execution-time address binding scheme results in differing logical and physical addresses
  - The set of all logical addresses generated by a program is called a logical address space, and the set of all physical addresses corresponding to these logical addresses is called a physical address space
  - The run-time mapping from virtual address to physical address is done by a hardware device called the *memory-management unit (MMU)*
  - Here, the base register is called a *relocation register* and the value in the relocation register is added to every address generated by a user process at the time the address is sent to memory
    - If a user program wants to access address location 5, and the relocation register is at 14000, the attempt to access is dynamically relocated to address location 14005

- So, we have two different types of addresses which can be thought of in the following manner
  - Logical addresses, in the range from 0 to *max*
  - Physical addresses, in the range from $R + 0$ to $R+$*max* for a base value R



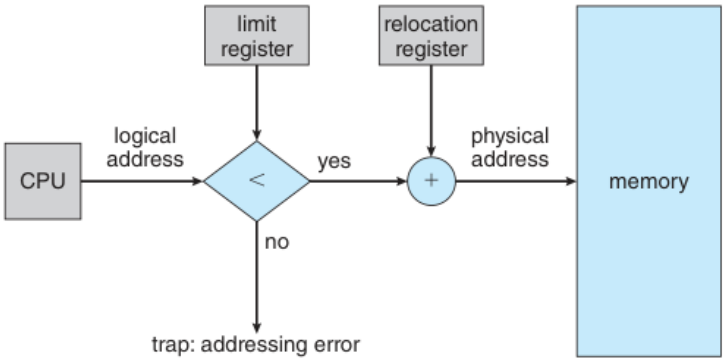
- **9.1.4: Dynamic Loading**
  - So far, we have assumed that it is necessary for an entire program and all data of a process to be in physical memory in order for the process to execute
  - Therefore, the size of a process will be limited to the size of physical memory
  - In order to obtain better utilization of the memory space, we can use *dynamic loading*
  - With dynamic loading, a routine is not loaded until it is called
  - All routines are kept on disk in a relocatable load format, and the main program is loaded into memory and executed
  - When a routine needs to call another routine, the calling routine first checks to see if the other routine has been loaded
  - If not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change
  - This provides the advantage of not having to always load infrequently used routines that consist of large amounts of code
- **9.1.5: Dynamic Linking and Shared Libraries**

    - *Dynamically Linked Libraries (DLLs)* are system libraries that are linked to user programs when the programs are run
    - Some operating systems support only static linking, in which system libraries are treated like any other object module and are combined by the loader into the binary program image
    - DLLs allow user programs to not need copies of certain system libraries such as the standard C language library
    - DLLs also can be shared among multiple processes such that only one copy of the DLL is in main memory
    - DLLs are used extensively in Windows and Linux systems
    - When a program references a routine that is in a dynamic library, the loader locates the DLL, loading it into memory if necessary
    - Dynamic linking and shared libraries generally require help from the operating system to function effectively, since processes in memory are protected from one another and would have trouble accessing shared memory

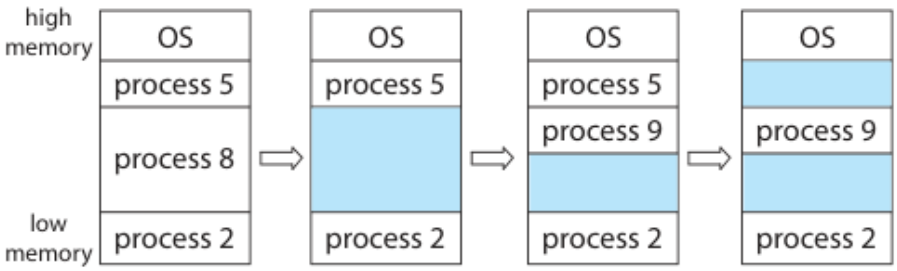
9.2: Contiguous Memory Allocation

- The main memory in a system must accommodate both the operating system and user processes
- Thus, we need to allocate main memory in the most efficient way possible
- Usually, memory is divided into two partitions, one for the operating system, and one for user processes, and usually operating systems are placed in high memory
- Generally, multiple user processes will reside in memory at the same time, so we therefore need to consider how to allocate available memory to the processes that are waiting to be brought into memory
- In *contiguous memory allocation*, each process is contained in a single section of memory which is contiguous to the section containing the next process
- **9.2.1: Memory Protection**
    - We must prevent a process from accessing memory it does not own by combining two previously discussed ideas

- In order to accomplish this goal, we can simply implement both a relocation register, and a limit register



- **9.2.2: Memory Allocation**
  - One of the simplest methods of allocating memory is to assign processes to variably sized partitions in memory, where each partition may contain exactly one process
  - In this scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied
  - Initially, all memory is available for user processes and is considered one large block of available memory, or a *hole*, and eventually the memory will contain a set of holes of various sizes
  - Here, we can see this variable partition scheme



  - What happens when there isn't sufficient memory to satisfy the demands of an arriving process?
    - One option is to simply reject the process and provide an appropriate error message
    - Alternatively, we can place these processes into a wait queue where they will be placed into memory when it becomes available

- When processes release their memory and form contiguous holes, they will merge to form one larger hole
- How do we decide which hole in which to place an arriving process if multiple different holes satisfy the memory requirements
- The *first-fit, best-fit, and worst-fit* strategies are the most commonly used
- In the *first-fit* method, the first hole that is big enough to accommodate the process will be used
- In the *best-fit* method, we will allocate the smallest hole which is large enough to accommodate the request
  - The entire list must be searched, unless the list is ordered by size
  - This strategy produces the smallest leftover hole
- In the *worst-fit* method, the largest hole will be allocated
  - Again, the entire list must be searched unless it is ordered by size
  - This strategy produces the largest leftover hole, which may be more useful than the smallest leftover hole produced in the best-fit method
- Simulations have shown that first-fit and best-fit are better than worst-fit in terms of decreasing time and storage utilization
- Between the two, neither is clearly better in terms of storage utilization, but first fit is generally faster

- **9.2.3: Fragmentation**
  - Both the first- and best-fit strategies for memory allocation suffer from *external fragmentation*
  - External fragmentation exists when there is enough total memory space to satisfy a request, but the available spaces are not contiguous
  - No matter which memory allocation algorithm we use, external fragmentation will pose an issue
  - Statistical analysis of the first-fit method reveals that given N allocated blocks, $0.5 * N$ blocks will be lost to fragmentation

- This idea, known as the *50-percent rule*, means that one third of memory may be unusable
- Memory fragmentation can be internal as well as external
- In *internal fragmentation* memory blocks are allocated only in a specific size, and are thus sometimes given to processes that require less than one full block
- Here, the extra memory allocated to a process which will not be used is considered to be lost to internal fragmentation
- One solution to the problem of external fragmentation is *compaction*, where the goal is to shuffle memory contents such that free memory is all placed together in one large block
- Compaction is not always possible, since it requires relocation to be dynamic and be done at execution time
- We can also allow processes to have a logical address space which is noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available
- This is the general approach used in *paging*, which is the most common memory-management technique for computer systems

9.3: Paging

- So, far memory management discussions have centered around schemes that require the physical address space of a process to be contiguous
- Now, we introduce *paging*, which is a scheme by which we allow a process's physical address space to be non contiguous
- **9.3.1: Basic Method**
  - The basic method for paging implementation involves breaking physical memory into fixed-size blocks called *frames*, and breaking logical memory into blocks of the same size called *pages*
  - When a process is executed, its pages are loaded into any available memory frames from their source
  - The source is also divided into fixed size blocks which are the size of individual memory frames or clusters of memory frames

- Every address which is generated by the CPU is divided into two parts, namely a *page number (p)* and a *page offset (d)*
- The page number is used as an index into a per-process *page table*
- The page table contains the base address of each frame in physical memory, and the offset is the location in the frame being referenced
- The following outlines the steps taken by the memory management unit in order to translate a logical address generated by the CPU to a physical address
    1. Extract the page number, p, and use it as an index into the page table
    2. Extract the corresponding frame number, f from the page table
    3. Replace the page number, p, in the logical address with the frame number f
- Like the frame size, the page size is determined by the hardware, and page sizes are typically powers of two ranging from 4 KB to 1 GB per page
- If the size of the logical address space is 2^m, and a page size is 2^n, then the high order $m - n$ bits of a logical address designate the page number, and the n low order bits designate the page offset
- Thus the logical address consists of p and d where p is an index into the page table and d is the displacement within the page
- If process size is completely independent of page size, we can expect internal fragmentation to average approximately one-half page per process
- This would suggest that smaller page sizes are more advantageous, but it is important also to note that a smaller page size will increase the total number of pages, and thus also increase system overhead
- Today, pages are generally either 4 KB or 8 KB in size, and some systems support larger page sizes
- Some systems even support multiple different page sizes
- When a process arrives in the system to be executed, its size, expressed in pages, is examined by the system
- If a process requires n pages, then at least n frames must be available in memory, and if they are, will be allocated to this arriving process

    - Since the operating system is responsible for managing physical memory, it must be aware of the physical memory's allocation details
    - This information is generally kept in a single, system-wide data structure called a *frame table*, which has one entry for each physical page frame, indicating whether the latter is free or allocated
- **9.3.2: Hardware Support**
    - As page tables are per-process data structures, a pointer to the page table is stored in the process control block (PCB) of each process
    - In the simplest case, the page table is implemented as a set of dedicated high-speed hardware registers, making the page-address translation a very efficient operation
    - **Translation Look-Aside Buffer**
        - Storing the page table in main memory may yield faster context switch times, but it also will result in slower memory access times
        - It will effectively double the memory access time since each memory access requires an additional memory access for the page-address translation step
        - This delay is considered intolerable on many systems, and the standard solution is to use a special, small, fast-lookup hardware cache called a *Translation Look-Aside Buffer (TLB)*
        - Each entry in the TLB consists of two parts
            - A key
            - A value
        - When the associative memory is presented with an item, the item is compared with all keys simultaneously, and if the item is found, the corresponding value field will be returned
        - The TLB contains only a few of the page-table entries
        - When a logical address is generated by the CPU, the MMU first checks if its page number is present in the TLB, and if it is, then the frame number is immediately available and is then used to access memory
        - If a TLB is full, then an existing entry must be selected for replacement
        - Policies for removal range from least-recently-used to round-robin or even random
        - Some CPU architectures also allow certain entries to be *wired down* meaning that they can not be removed from the TLB
        - The percentage of times that the page number of interest is found in the TLB is called the *hit ratio*

        - Effective memory access time can be calculated by utilizing the hit ratio, since hits will take one memory access time and misses will take two, thus allowing us to make a good approximation using the hit ratio

- **9.3.3: Protection**
    - Memory protection in a paged environment is accomplished by protection bits associated with each frame, which are normally kept in the page table
    - One bit can define a page to be read-write or read-only
    - When logical addresses are translated to physical addresses, these bits are also checked to verify that the protections are being followed
    - One additional bit is also generally attached to each entry in the page table, known as a *valid-invalid bit*
    - When this bit is set to valid, the associated page is in the process's logical address space and is thus a legal page
    - Otherwise, the page is not in the process's logical address space

- **9.3.4: Shared Pages**
    - One advantage of paging is the possibility of sharing common code, something that is of particular importance in an environment with multiple processes

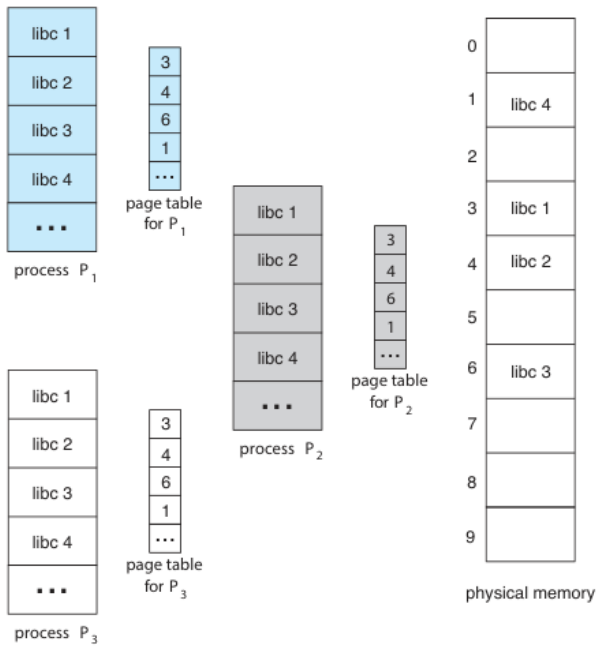


Figure 9.14 Sharing of standard C library in a paging environment.

9.4: Structure of the Page Table

- **9.4.1: Hierarchical Paging**

- In modern environments where systems support large logical address spaces, the page table itself will become excessively large
- In order to remedy this issue, we can implement a two-tiered paging system, where one page table pages a second page table, which is illustrated below

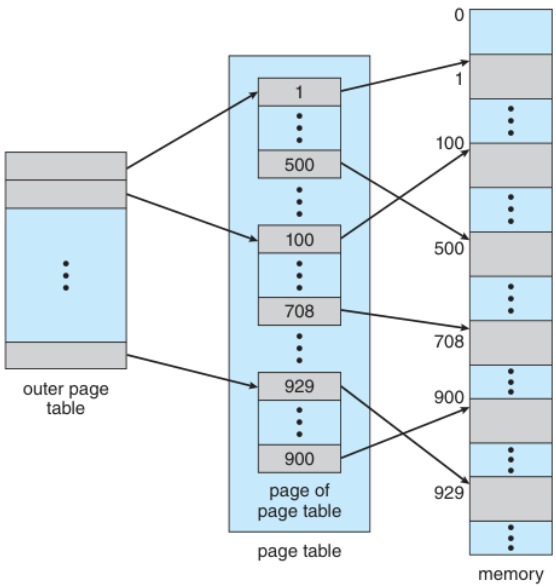
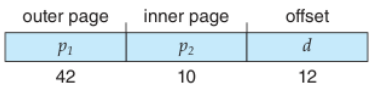
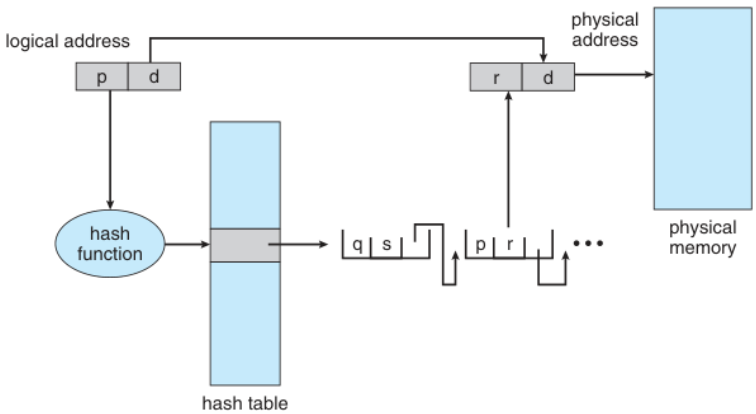


Figure 9.15 A two-level page-table scheme.



- **9.4.2: Hashed Page Tables**
  - Another approach for handling address spaces larger than 32 bits is to use a *hashed page table*



Justin Ciocoi

Dec. 17, 2023

CSCI 375 Textbook Notes

Chapter 10: Virtual Memory

- Virtual memory is a technique by which we can allow the execution of processes that are not completely in memory
- One large advantage of this scheme is that programs can be larger than physical memory

10.1: Background

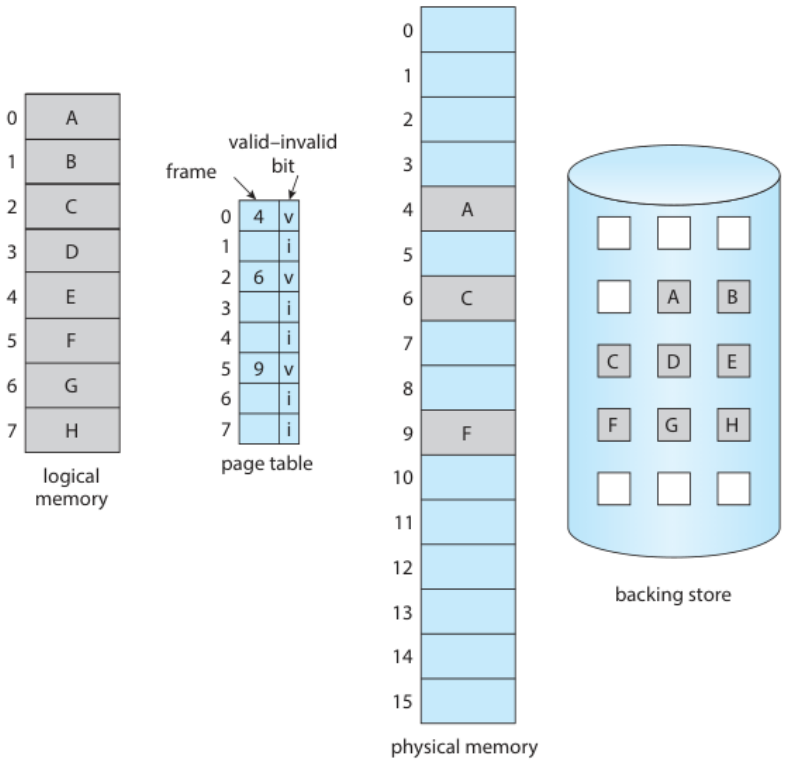
- The memory management algorithms which are outlined in chapter 9 are necessary because of the basic requirement that the instructions being executed must be in physical memory
- The first approach to meeting this requirement is to place the entire logical address space in physical memory
    - Dynamic linking can help to ease this restriction, but it generally requires special precautions and extra work by a programmer
- This requirement seems necessary and reasonable, but it unfortunately limits the size of a program to the size of physical memory in a system
- However, in many cases, the entire program is not needed, such as in the following cases
    - Programs that have code meant to handle unusual error conditions which will seldom, if ever, occur
    - Arrays, lists, and tables are often allocated more memory than they actually need
    - Certain options and features of a program may be used rarely
- Running a program not entirely in physical memory would allow users not only to have a far more efficient multiprocessing environment, but it also allows programmers to design programs for an extremely large virtual address space rather than a limited physical space
- *Virtual memory* involves the separation of logical memory as perceived by developers from physical memory

- This separation then allows for an extremely large virtual memory to be provided to application programmers even if only a limited subset of physical memory is available
- The *virtual address space* of a process refers to the logical view of how a process is stored in memory
    - Typically, this view is that a process begins at a certain logical address, and exists in contiguous memory
- However, this is not always the case as programs sometimes reside in non-contiguous memory
- For any process, we allow its stack and heap to grow towards each other
    - The holes in between the two are useful since they can be used when one of the two wants to grow such as when a library is dynamically linked
- In addition to separating logical memory from physical memory, virtual memory allows files and memory to be shared by two ore more processes through page sharing, leading to the following benefits
    - System libraries such as the standard C library can be shared by several processes through mapping of the shared object into a virtual address space
    - Similarly, memory can be shared, which allows processes to create regions of shared memory where each process has its own virtual copy, but they share the actual physical copy

10.2: Demand Paging

- How is an executable program loaded from secondary storage into memory?
- One option is to load the entire program into physical memory at program execution time
- However, we might not initially need the entire program to be in physical memory
- An alternative strategy is to load pages only as they are needed, which is a technique known as *demand paging*
- With demand-paged virtual memory, pages are loaded only when they are demanded during program execution
- **10.2.1: Basic Concepts**

- The general concept behind demand paging is to load a page in memory only when it is needed
- As a result, throughout the duration of a process's execution, some pages will be in memory, and some will be in secondary storage
  - We therefore need some form of hardware support to distinguish between the two
  - We can use the valid-invalid bit scheme to accomplish this goal



- When the bit is set to valid, the associated page is both legal and in memory, but if the bit is set to invalid, then the page is either not valid, or is valid but is currently in secondary storage
- What happens if a process tries to access a page that was not brought into memory?
  - Access to a page which is marked invalid causes a *page fault*
- The procedure for handling this page fault is fairly straightforward
  1. We check an internal table, which is usually kept with the process control block, in order to determine whether the reference was a valid or invalid memory access
  2. If the reference was invalid, we terminate the process, and if it was valid but we have not yet brought in the page, we now page it in

3. We find a free frame
4. We schedule a secondary storage operation to read the desired page into the newly allocated frame
5. When the storage read is complete, we modify the internal kept with the process and the page table to indicate that the page is now in memory
6. Now, we restart the instruction that was interrupted and the process can access the page as though it had always been in memory

- In the extreme case, we can start executing a process with no pages in memory
  - The process will immediately incur a page fault on its first instruction, and subsequently fault as necessary until every page that it needs is in memory
- This scheme is known as *pure demand paging*
- Theoretically, some programs could access several new pages of memory with each instruction execution, meaning multiple page faults per instruction are possible
- However, this behavior is exceedingly unlikely since programs tend to have a *locality of reference* which results in reasonable performance from demand paging
- The hardware support for demand paging is the same as hardware support for paging and swapping and is as follows
  - *Page table*, which has the ability to mark an entry invalid through a valid-invalid bit or a special value of protection bits
  - *Secondary memory*, which holds pages that are not present in main memory, and is known as the *swap device* which uses the *swap space* to accomplish this goal
- One crucial requirement for demand paging is the ability to restart any instruction after a page fault
- Because we save the state of the interrupted process when the page fault occurs, we must be able to restart the process in *exactly* the same place and state, except for the fact that the requested page is now in memory and is accessible by the program
- As a worst-case example, let us consider a three-address instruction such as adding the content of A to B and placing the result in C
- The following are the steps to execute the above series of instructions

1. Fetch and decode the instruction
2. Fetch A
3. Fetch B
4. Add A and B
5. Store the sum in C

- Even if a page fault occurs when storing the sum in C, the entire list of operations here will be done again, but the repetition comprises of less than one complete instruction and is necessary only whenever a page fault occurs

- **10.2.2: Free-Frame List**
  - When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory
  - To resolve page faults, most operating systems maintain a *free-frame list*, which is a pool of free frames meant to satisfy such requests
  - Operating systems will usually allocate free frames using a technique known as *zero-fill-on-demand*
  - These frames are "zeroed-out" before being allocated, which erases their previous contents
  - When a system starts, all available memory is placed on the free-frame list and as free frames are requested, the size of the list shrinks
- **10.2.3: Performance of Demand Paging**
  - Demand paging can significantly impact the performance of a computer system, and to see why we will compute the effective access time for demand-paged memory

$$\text{effective access time} = (1 - p) * ma + p * \text{page fault time}$$

  - In order to compute the effective access time, we must know how much time it takes to service a page fault, which causes the following sequence to occur
    1. Trap to the operating system
    2. Save the registers and process state
    3. Determine that the interrupt was a page fault

4. Check that the page reference was legal and determine the location of the page in secondary storage
5. Issue a read from the storage to a free frame
    1. Wait in a queue until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer if a page to a free frame
6. While waiting, allocate the CPU core to some other process
7. Receive an interrupt from the storage I/O subsystem that the I/O has completed
8. Save the register and process state for the other process, only if step 6 is executed
9. Determine that the interrupt was from the secondary storage device
10. Correct the page table and other tables to show that the desired page is now in memory
11. Wait for the CPU core to be allocated to this process again
12. Restore the registers, process state, and the new page table, and then resume the interrupted instruction

- These are not all always necessary, but in any case, there are three major task components of the page-fault service time
    1. Service the page-fault interrupt
    2. Read in the page
    3. Restart the process
- By looking at the aforementioned formula, we can see that the effective access time for memory is directly proportional to the rate of page-faults
- Thus, in order to keep the slowdown due to paging at a reasonable level, we must keep the probability of a page fault at $p < 0.0000025$ or less than one in every $399,990$ memory accesses

10.3: Copy-on-Write

- Process creation that uses the `fork()` system call may initially bypass the need for demand paging by using a technique which is similar to page sharing
- Recall that `fork()` works by creating a child process that is a duplicate of its parent
- Considering that many child processes invoke the `exec()` system call immediately after creation, the copying of the parent process's address space may be unnecessary
- In these cases, we can use a technique known as *copy-on-write*, which works by allowing the parent and child processes to initially share the same pages
- These pages are marked such that if either process writes to a shared page, then a copy will be created
    - This means as long as the processes are capable of sharing a single copy of a page, they will

10.4: Page Replacement

- Earlier, we assumed that each page faults at most once, when it is first referenced, but this is not strictly accurate
- Over-allocation of memory manifests itself as follows
    - While a process is executing, a page fault occurs
    - The operating system determined where the desired page is residing on secondary storage, but then finds that there are no free frames on the free-frame list, meaning all memory is in use
- The operating system, at this point, has several options, such as attempting to terminate the process
- However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput by not allocating unnecessary memory
- Most operating systems combine swapping pages with *page replacement*, which will be described in detail throughout the remainder of this section
- **10.4.1: Basic Page Replacement**
    - Page replacement fundamentally takes the following approach
        - If no frame is free, we find one that is not currently being used and free it

        - We can free a frame by writing its contents to swap space and changing the page table to indicate that the page is no longer in memory
        - We can now use the freed frame to hold the page for which the process faulted
    - We can modify the page-fault service routine to include page replacement in the following way
        1. Find the location of the desired page on secondary storage
        2. Find a free frame
            1. If there is a free frame, use it
            2. If there is no free frame, us a page-replacement algorithm to select a *victim frame*
            3. Write the victim frame to secondary storage and change the page and frame tables accordingly
        3. Read the desired page into the newly freed frame and change the page and frame tables accordingly
        4. Continue the process from where the page fault occurred
    - If no frames are free, two page transfers are required, which effectively doubles the page-fault service time and increases the effective access time accordingly
    - We can reduce this overhead by using a *modify bit* or *dirty bit*
    - When this scheme is used, each page or frame has a dirty bit associated with it in the hardware
    - Whenever any byte in the page is written into, the dirty bit is set indicating that the page has been modified
    - When we examine a page for page replacement, we can look at its dirty bit, and if it is set, we know that the page has been modified since it was read in from secondary storage
    - If the dirty bit is not set, however, we know that the page has not been modified and does not need to be written back into secondary storage before being replaced
    - In order to implement demand paging, we have to solve two major problems
        - Develop a *frame-allocation algorithm*

        - Develop a *page-replacement algorithm*
    - In general, we will select a page-replacement algorithm by choosing to use the one that exhibits the lowest page-fault rate, since page faults have an enormous impact on system performance
    - When looking at different page-replacement algorithms, we will use the following *reference string*, or string of memory references

$$7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1$$

- **10.4.2: FIFO Page Replacement**

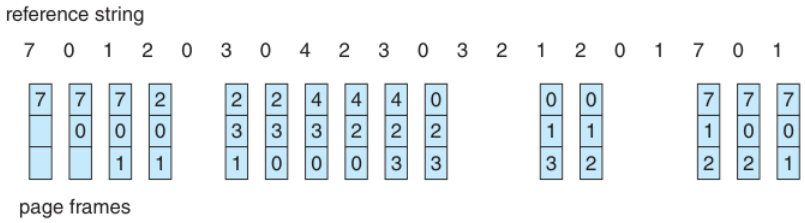
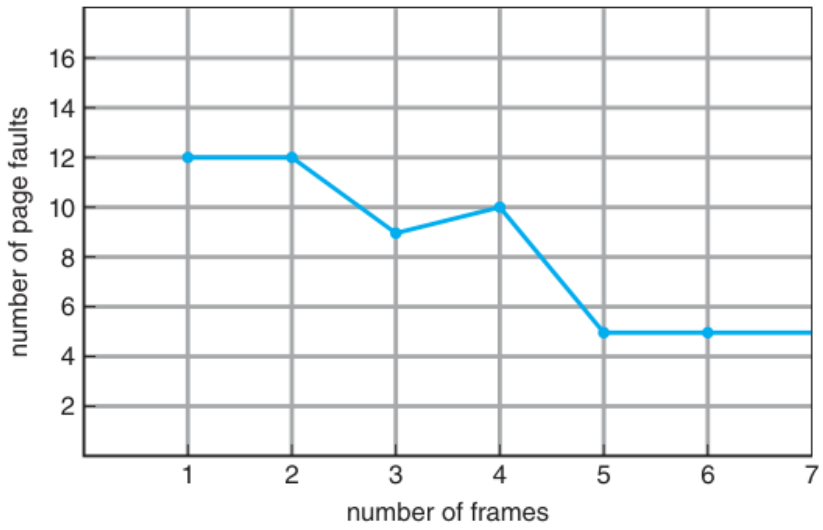


Figure 10.12 FIFO page-replacement algorithm.

    - This is the simplest page-replacement algorithm and associates with each page the time when that page was brought into memory
    - When a page must be replaced, the oldest page is chosen as the victim page
    - It is not strictly necessary to record the time each page is brought in, as we can instead opt to implement this as a FIFO queue which holds all pages in memory
    - Sometimes, the number of frames used for a page-replacement algorithm does not correlate directly with the rate of page faults
    - This is known as *Belady's Anomaly* and can be seen from the below graph



- **10.4.3: Optimal Page Replacement**

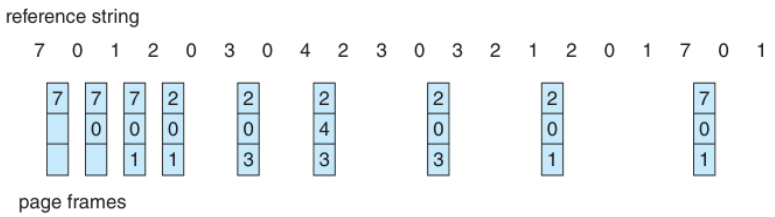


Figure 10.14 Optimal page-replacement algorithm.

  - As a result of the discovery of the above anomaly, the search for an optimal page-replacement algorithm ensued
  - This algorithm is purely theoretical and will opt to replace the page that will not be used for the longest period of time
  - However, because this algorithm requires future knowledge of the reference string, there is no perfect way to implement it, but we can compare other functional algorithms against this one as the gold standard of page-replacement algorithms

- **10.4.4: LRU Page Replacement**

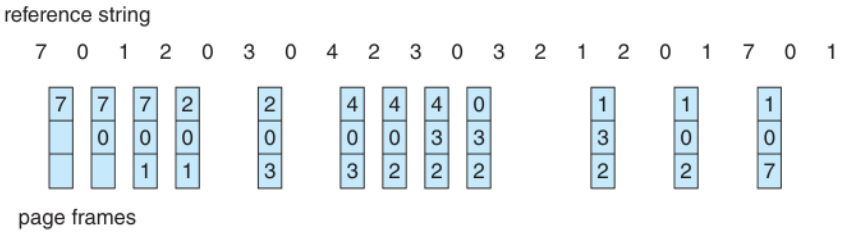


Figure 10.15 LRU page-replacement algorithm.

  - If we use the recent past as an approximation for the future, we can arrive at an algorithm that approximates the optimal one

    - We can do this, and then replace the page that has not been used for the longest amount of time
    - LRU replacement associates with each page the time of that page's last use
    - When a page must be replaced, LRU will choose the page that has not been used for the longest period of time
    - This policy is often used and is generally considered to be good, but the general problem is *how* to implement LRU replacement
    - Two implementations are feasible
        - *Counters*
            - In the simplest case, we can associate a time-of-use field with each page-table entry and add to the CPU a logical clock or counter, which will be incremented for every memory reference
            - When a reference to a page is made, the contents of the clock register are copied to the time-of-use field
            - In this way, we always have the time of the last reference to each page
        - *Stack*
            - Another approach is to keep a stack of page numbers
            - Whenever a page is referenced, it is removed from the stack and put on the top
            - In this way, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom of the stack
            - Since entries are removed from the bottom of the stack, it is best to implement this stack using a doubly linked list with a head and tail pointer
    - LRU does not experience Belady's Anomaly, as it is a *stack algorithm*, meaning it can be shown that the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with $n + 1$ frames

- **10.4.5: LRU-Approximation Page Replacement**

    - While the LRU policy seems very advantageous, not many computer systems provide sufficient hardware support for true LRU implementation

    - Hardware support usually comes in the form of reference bits which is set by the hardware whenever a page is referenced

    - Initially, all bits are cleared to 0 by the operating system, and as a process executes, the pit associated with each page is set to 1 by the hardware

    - Thus, we can see which pages have been used, but we do not know the order of use\

- **10.4.5.1: Additional-Reference-Bits Algorithm**
  - Instead of using a single reference bit, we can use a reference byte, consisting of 8 bits
  - When a page is used, the leftmost bit will be set to 1, shifting all other bits right
  - If we interpret these 8 bit bytes as unsigned integers, the page with the lowest number is the LRU page and will be selected by this algorithm
  - This algorithm is also called the *second-chance page-replacement algorithm*
- **10.4.5.2: Second-Chance Algorithm**
  - In essence, this algorithm is a FIFO page-replacement algorithm
  - When a page has been selected, we inspect its reference bit, and if it is set to 1, we give this page a *second chance*
  - A page that is given a second chance has its reference bit cleared and its arrival time updated to the current time
  - Thus, a page given a second chance will not be replaced until all other pages have either been replaced or given a second chance
- **10.4.5.3: Enhanced Second-Chance Algorithm**
  - The second-chance algorithm can be further enhanced through the use of a second reference bit rather than just one
  - This leaves us with 4 possibilities rather than 2
    1. $(0, 0)$: neither used nor modified - best page to replace
    2. $(0, 1)$: not recently used, but modified - not as good, since the page needs to be written out before replacement
    3. $(1, 0)$: recently used but clean - probably will be used again soon
    4. $(1, 1)$: recently used and modified - will probably be used again and needs to be written out to secondary storage before it is replaced
- **10.4.6: Counting-Based Page Replacement**
  - We can also keep a counter to the number of references that have been made to each page, and using these counters develop the following two schemes

        - LFU, or *least frequently used* page replacement will replace the page that has the smallest count
            - However, a page that was heavily used at startup, but wont be for the remainder of a program's life could lead to undesired results when using this algorithm
        - MFU, or *most frequently used* follows a similar idea, but is based on the argument that the page with the smallest count was probably just brought in and has yet to be used

10.5: Allocation of Frames

- How do we allocate the fixed amount of free memory we have in a computer system among various processes
- **10.5.1: Minimum Number of Frames**
    - There exists both a maximum and minimum for frame allocation operations
    - One reason for the existence of a minimum is performance, since a smaller minimum would increase the rate of page-faults
    - The minimum is usually determined by system architecture, whereas the maximum is determined by the total amount of available physical memory
- **10.5.2: Allocation Algorithms**
    - The easiest way to split m frames among n processes is to give each process an equal share of frames
        - This scheme is known as *equal allocation*
    - An alternative is to use *proportional allocation* where a process is allocated a number of free frames proportional to its size as determined by

      $$a_i = \frac{s_i}{S} * m$$

      where $\frac{s_i}{S}$ is equal to the proportion of size taken by this program and m is the number of free frames
- **10.5.3: Global vs. Local Allocation**

    - Another important factor in the way frames are allocated to various processes is page replacement
    - Since multiple processes are competing for frames, we can classify page-replacement algorithms into two categories
        - *Global*
        - *Local*
    - Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process
    - Local replacement requires that each process select only from its own set of allocated frames
    - Consider high- and low-priority processes and how higher priority processes might be more needing of global replacement algorithms whereas lower priority processes are not
    - Global replacement leads to varying process execution time, but generally increased throughput, whereas local replacement might under-utilize memory, but results in more consistent per-process performance
- **10.5.4: Non-Uniform Memory Access**
    - On NUMA systems with multiple CPUs, not all main memory is created equal, and thus a given CPU can access some sections of main memory faster than it can access others
    - Here, optimal performance comes from allocating memory which is architecturally close to the CPU on which the thread is scheduled

10.6: Thrashing

- What might happen if a process does not have the minimum number of frames it needs to support pages in the working set?
- The process will very quickly incur a page-fault, at which point it must select a page to replace
    - However, since all of its pages are in use, it will immediately incur another page-fault, and another, and another
- This repeated paging is called *thrashing* and a process is thrashing when it is spending more time paging than it is executing

- Thrashing results in severe performance problems
- **10.6.1: Cause of Thrashing**
    - Consider the situation where the operating system monitors CPU utilization, and if utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system
    - A global page replacement algorithm is used, and it replaces pages without regard to which process they belong
    - Now suppose a process enters a new phase of execution and needs more frames, at which point it will start faulting and taking pages away from other processes
    - This results in increased paging as those other processes will immediately page fault as well
    - The operating system will now see the CPU utilization decrease as a result of high page-fault rates, and will thus increase the degree of multiprogramming once more, compounding the issue
    - We can limit the effects of thrashing by implementing a *local replacement algorithm*
    - By doing this, we can ensure than if one process starts thrashing, it cannot steal frames from another process and cause that process to thrash as well
    - However, an individual process can still thrash and thus the problem is not yet solved
    - In order to prevent thrashing, we must provide a process with as many frames as it needs, but the issue is knowing how many frames a process needs
    - The *locality model* states that as a process executes it moves from locality to locality, which might overlap
    - If we do not allocate enough frames to a process to accommodate the size of the current locality, the process will thrash since it cannot keep in memory all of the pages which it is actively using
- **10.6.2: Working-Set Model**
    - The *working-set model* is based on the assumption of locality
    - This model uses a parameter, Δ, to define the working set window
    - The general idea is to examine the most recent Δ page references

    - The set of pages in this set is the working set
    - If a page is in use, it will be in the working set and if a page is no longer being used, it will be removed from the working set
    - $WSS_i =$ the total number of pages referenced in the most recent Δ
        - If Δ is to small, it will not encompass the entire locality
        - If Δ is too large, it will encompass several localities
        - If $\Delta = \infty \rightarrow$ will encompass entire program
    - $D = \sum WSS_i =$total demand frames
    - If $D > m$, there is thrashing
    - So, we can implement the policy that if $D > m$, we will suspend one of the processes
    - We can keep track of the working set by using an interval timer as well as a reference bit, but this not completely accurate schemes becomes more and more accurate the more reference bits there are and the shorter the timer interval is
- **10.6.3: Page-Fault Frequency**
    - The working-set model is successful, but it seems like a clumsy way to control thrashing
    - A strategy that uses the page-fault frequency takes a more direct approach
    - If a process has a low page fault rate, it may have too many frames, and if it has a high page fault rate it may not have enough
    - The operating will then make frame allocation decisions based on upper and lower bounds
- **10.6.4: Current Practice**
    - Currently, the best practice in implementing a computer system is to include enough physical memory such that thrashing and swapping is avoided whenever possible

---