OPERATING SYSTEMS FOUNDATIONS
MSIT 5170

University of Massachusetts Lowell
Department of Computer Science
Spring Semester 2020

Weeks 1 and 2 Lesson

1. The material in this lesson follows the reading assignment for ch 1.1 through ch 1.4 from the Tanenbaum text
2. Please read the text before you read through this lesson.
3. The material in chapter 1 provides a broad view of the topics that we will visit throughout the semester. We will work through chapter 1 during the first 3 weeks of this term, looking at ch 1.1 through ch 1.4 for this first 2 week lesson.
4. There are several examples of system level tools that are given here, and I encourage students to experiment with these tools. Each of you will be provided with web access to various systems to complete class assignments, many of which will require tools such as those shown here.

What is an Operating System?

The hardware environment provided by a typical digital computer is a complex collection of devices that require a substantial amount of programming before they can be generally useful to an application. An application that wishes to write data to a simple file, for example, will eventually have to accomplish the correct programming of the processor it's running on, the chipset that exposes the system bus to the processor, the device controller that provides access to the persistent storage used by the file system, and even the target device itself.

These types of operations have to be done over and over again for any applications that have a need to store some data. Rather than require every application to take full control of the entire system to complete its mission, operating systems have been developed to manage the details of the system and provide a set of services that applications can share, thus avoiding the need for each application to provide such features on its own.

The first operating systems were little more than a collection of run-time routines that one application at a time could leverage to control the computing environment. As systems became more powerful, however, it became apparent that the concurrent execution of multiple applications would provide a much more cost-effective use of the expensive system resources.

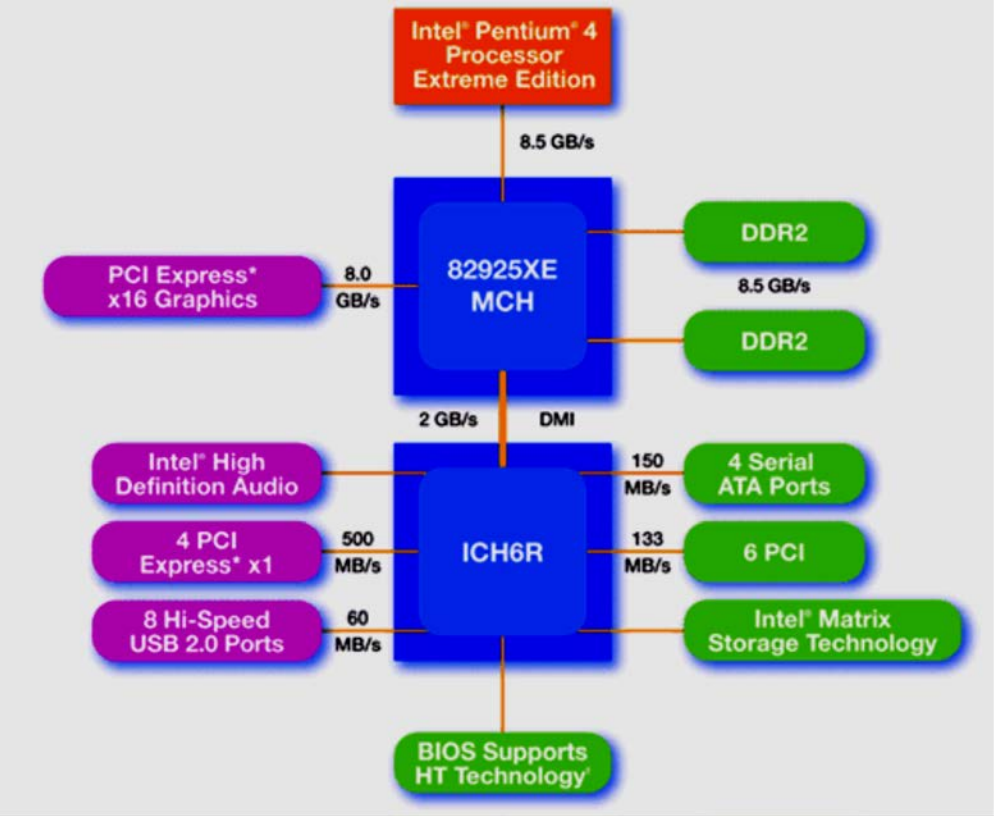
Operating systems have evolved into resource managers, taking control of the hardware environment they are booted into, and providing support to applications through a set of procedures that we commonly refer to as the **system call Application Programming Interface** (API). Managing resources is a complex business, and thus operating systems are among the most complex entities created by human beings. Enterprise grade operating systems like Solaris, Linux, Windows and others represent millions of man hours of development and testing, and yet for all their complexity, they are well organized and systematically deployed. Our job in

this course is to tackle this systematic organization, and expose the architecture of contemporary operating systems.

Computer hardware review:

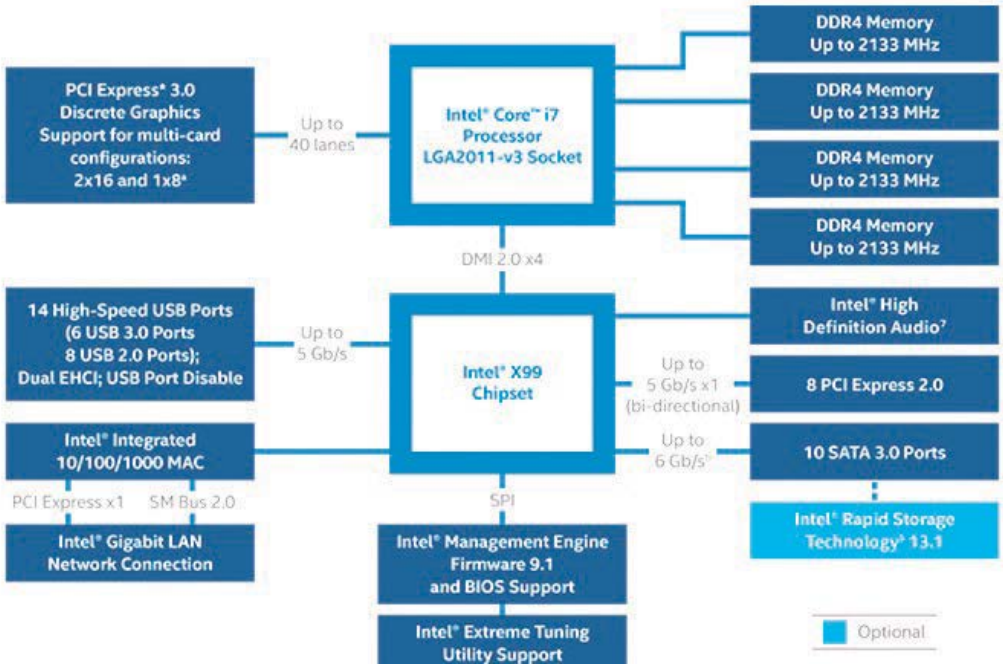
A quick review of the hardware environments of typical systems will help us build a foundation for our operating systems studies. While hardware components are fast moving targets, the generic architecture of low and mid-range systems has been fairly stable for the past two decades. We generally regard the components of these systems as:

- One or more processing elements which we refer to as Central Processing Units (CPUs), although the term “core” has come into recent use as well
- A memory channel or north bridge chip (shown below as an Intel MCH 82925XE) that exposes a system bus to the CPUs
    - This device typically provides one or more memory controllers (if the CPU chip does not have this functionality built in)
    - It also provides an IO bridge (usually some form of Peripheral Component Interconnect (PCI)) or related interconnect technology
- The north bridge chip typically interfaces to an IO hub or south bridge chip (shown below as the Intel ICH6R) that provides embedded controllers for common peripheral interfaces like USB devices, IDE devices, and legacy parallel and serial devices.



The picture above shows an Intel system from about 2009 that uses a memory channel hub (MCH) to provide access to DDR2 RAM, a 16x PCI-express interface for a graphics controller,

and a DMI bridge (also a form of PCI-express) to connect to an IO channel hub (ICH). The ICH provides integrated controllers for IDE devices, additional PCI-express capability to support USB controllers, an audio controller and PCIe slots, as well as a conventional PCI bridge to provide legacy parallel PCI support. Later versions of Intel processors known as the i-series (i3, i5, and i7), modify the above picture by relocating the memory controller (and possibly graphics) functionality directly into the CPU complex, leaving a Southbridge type chipset to provide the peripheral functionality used to connect the rest of the system.



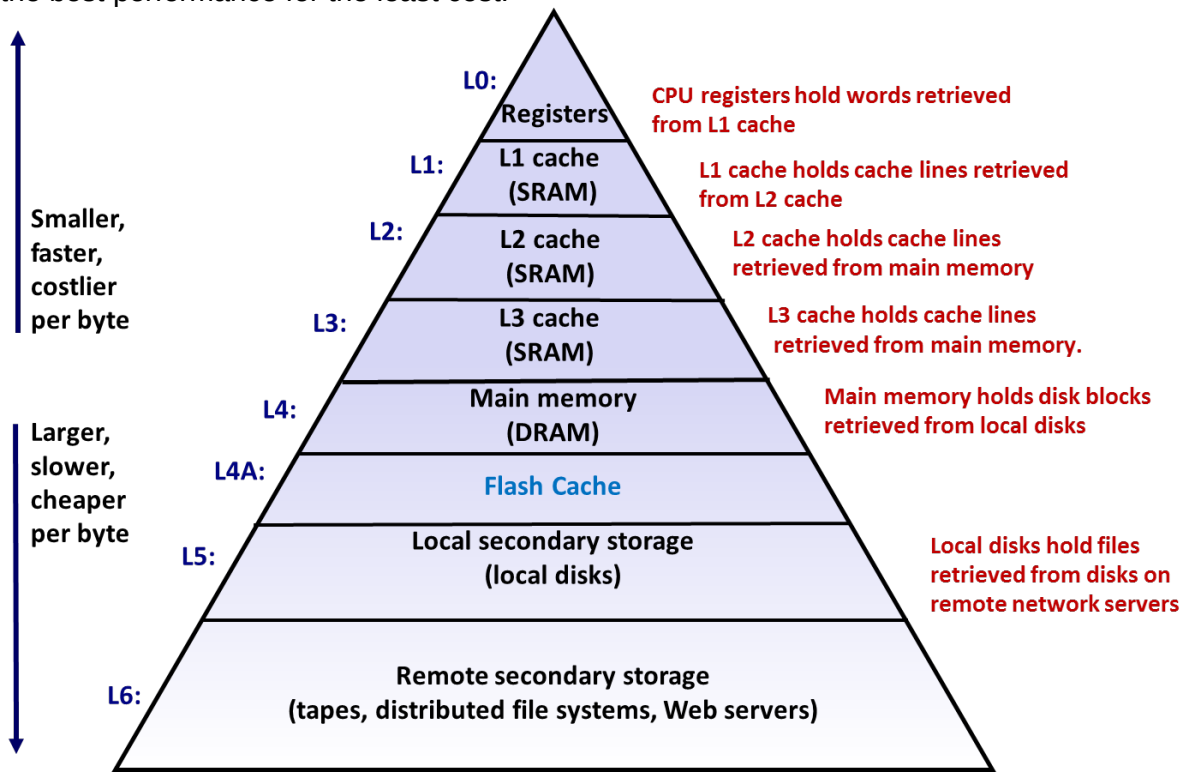
The picture above shows a newer Intel system from about 2016 that provides both memory and graphics support integrated into the CPU package. The Core i7 package includes 4 cores, 4 DDR4 memory controller channels and up to 40 Lanes of PCI Express 3.0 peripheral support (specifically multiple graphics adaptor cards). The X99 chipset supports additional devices such as USB (3.0 and 2.0 ports), SATA 3.0 ports, additional PCI Express 2.0 lanes, Ethernet, an audio controller and system management support.

Processors typically support a local L1 (level 1) cache per core, and either a local or shared L2 cache per core, with the Intel i-series providing a shared L3 cache among all the cores on a single processor package (chip). Processor caches provide an efficient interface to system RAM by leveraging the spatial and temporal locality that is typically found in virtually all software implementations. When a processor is looking for an instruction or a piece of data, the odds are very much in favor of the fact that an adjacent instruction or piece of data will be needed in the near future. This statistical fact motivates the cache strategy that basically says, “if I need something at location x, I will soon need something at location x +/- some local offset”. The processor’s memory interface responds to this logic by implementing a cache line load policy (cache lines are typically 64 bytes or greater), with the premise that, even though a current instruction requires only 4 bytes of data, reading into cache a full cache line of data is efficient, since the current executing thread will, with high probability, require something else from this cache line in the near future and reading 64 bytes in one 8 beat burst transaction is

much more efficient than reading the same 64 bytes in a series of 8 separate transactions (8 because the Intel front side bus is 8 bytes wide and 8X8 = 64).

The general organization of caches is based on the memory hierarchy shown below. The fastest memory components are registers, located in the processor, and providing source operands to the Arithmetic and Logic Unit (ALU) of the CPU. Registers, in turn, are supported by the LI cache components, which are fast and near to the processor's registers. The LI cache depends on the services of higher order caches, such as dedicated or shared L2, L3, etc. cache components.

When operands are found in the L1 cache, a processor (core) can complete an operation in a single clock cycle. Cores running at a typical 3 GHz clock rate can therefore complete ~3 billion machine operations per second with 100% L1 hit rates. An L1 miss extends the execution time to 5-10 clock cycles if the required operand is in L2 cache and 10-50 clock cycles if in L3 cache. The picture drops off dramatically if operands must be fetched from RAM and worse still if from secondary storage (SSD or HDD). High L1 hit rates are necessary to get the best performance from a processor, so cache sizes have to be sized in ways that provide the best performance for the least cost.



As mutli-core architectures begin to dominate the landscape, a move towards higher level cache tiers is inevitable. The latest Intel processors (i3, i5, i7) are using an L3 cache to share system bus components among the growing number of cores in a single chip (now at 10 cores for the high end i7 chips). Each core in the 10 core implementation (the Broadwell-E processor), has its own L1 and L2 local caches, and shares a large (up to 25 MB) L3 cache among all the cores.

Allowing multiple cores to share a common cache (L3) can minimize cache invalidation issues, and keep the collected cores away from the front side system bus as much as possible.

Buses

From the processor complex, the outside world is exposed by the memory channel hub (north bridge chip). As we've seen, this chip provides access to RAM across a synchronous memory channel, and access to peripheral devices across another synchronous bus called the PCI bus. These busses are synchronous in the sense that data transfer across them occurs at a fixed rate, using some clock generated pulses to enable each transfer.

Transfers across a memory channel often occur on both the rising and falling edges of the clock signal, and are thus referred to as double data rate (DDR) interfaces. PCI transfers, however, occur only on the top of the clock, and are thus considerably slower than memory channel transfers (beginning with a much slower clock, and performing only one transfer per clock cycle).

A variety of different controller devices are likely to be found on a PCI bus, some, such as a Small Computer Systems Interface (SCSI), can operate in both a synchronous or an asynchronous mode, and others such as a Fibre Channel (FC) interface or an Ethernet controller, operate only in a synchronous mode.

In essence, each of these controllers that can be found on a PCI bus, makes their own bus (synchronous or asynchronous) available to target devices like disks, network transceivers, printers or keyboards. The PCI bus they are attached to, concentrates their data flow, and provides a path into and out of the system RAM.

We view these collections of end point target devices as capillaries, flowing into larger vessels in the form of intermediate busses such as fibre channel or SCSI, which, in turn, are collected into larger conduits like a PCI bus that interfaces with the processor and the memory system through the memory channel hub (aka North Bridge).

Since the processor(s) and the PCI bridge are constantly contending for access to system RAM, and other parts of the physical address space, and since the system bus can only be accessed on discrete clock edges, the memory channel hub must provide arbitration among the contending parties to ensure efficient use of the physical address space. If a processor, for example, is trying to fetch an instruction from some address in RAM, and the PCI bridge has some buffered data from one of its attached devices that must be placed to some address in RAM, both the processor and the bridge must arbitrate for access to the bus. The winner will place its target RAM address on the bus and begin a data transfer on the next clock edge. The data component of the Intel x86 system bus, for example, is 64 bits (8 bytes) wide, so this is the extent of a single bus transaction. If more data has to be read or written by a bus master like the CPU or the PCI bridge, then additional bus cycles will have to be won in arbitration.

Accessing the system bus in random 8 byte chunks, requires an address component and a data component, and thus retrieving 32 bytes of data in a set of random 8 byte chunks would require 4 address and 4 data accesses to the system bus. The arbiter, however, recognizes a form of request called a burst transfer, in which a single address is provided, followed by multiple 8 byte data loads that will be transferred to/from contiguous locations along the system bus. This is the key to caching, providing a way to load or store a cache line with minimal overhead, and dealing with the entire line in only slightly more time than it would take to deal with a single

arbitrary location along the system bus. If other parts of the line will eventually be used (locality tells us that this is likely), then caching is a winning strategy.

Collecting Hardware Information

Most installed operating systems provide tools to examine, to one degree or another, the platform upon which the operating system is running. Throughout this course we will look at specific platform and third party tools for the Windows and Linux environments. In the case of the Windows world, for example, we are able to collect a significant amount of information about the system we're running on using the Device Manager applet from the Computer Management option of "My Computer" ("This PC" in Win10):

- From the Start menu, right click on the "My Computer" list element
- From this drop-down menu, select the "Manage" list element
- From the Computer Management window select the "Device Manager"
- From the "View" drop down menu, select "Devices by Connection"

A quick look at the resulting details shows that my system (an older core-2 duo system) has a dual core processor, and a Memory Channel Hub (Q35, north bridge) chip that provides PCI and PCI-express interfaces that consolidate most of the remaining components, several of which are implemented by the ICH9 (south bridge) chip. Of course there are many other tools available from both Microsoft and third parties that provide a much richer collection of hardware details than the device manager. One of the most useful freeware tools I've found for the Windows world is called HWiNFO Diagnostic Software, which can be downloaded from:

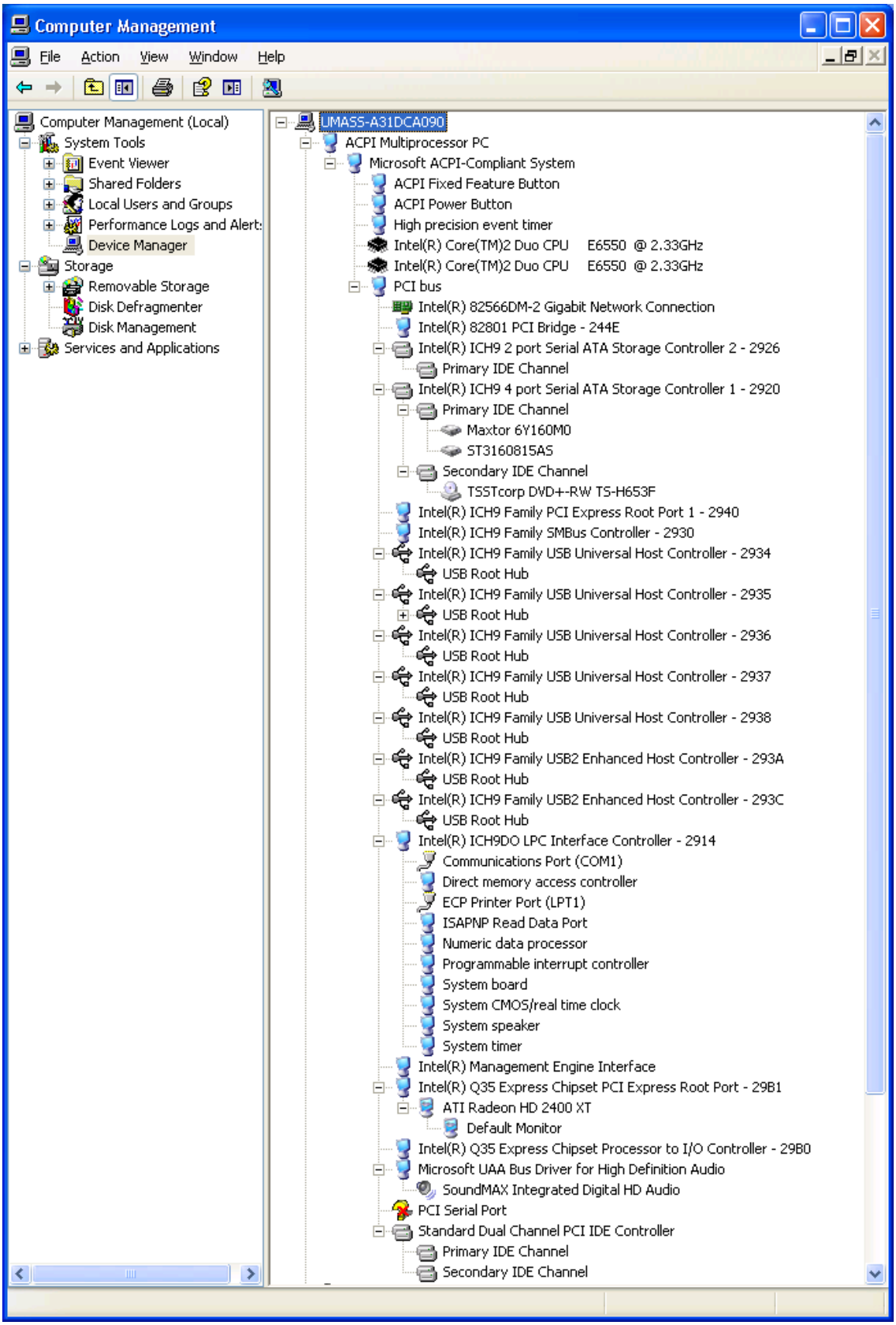
https://www.hwinfo.com/

HWiNFO provides detailed information about most system components, including the system processor(s), chipsets, memory, IO busses and attached controllers and target devices.

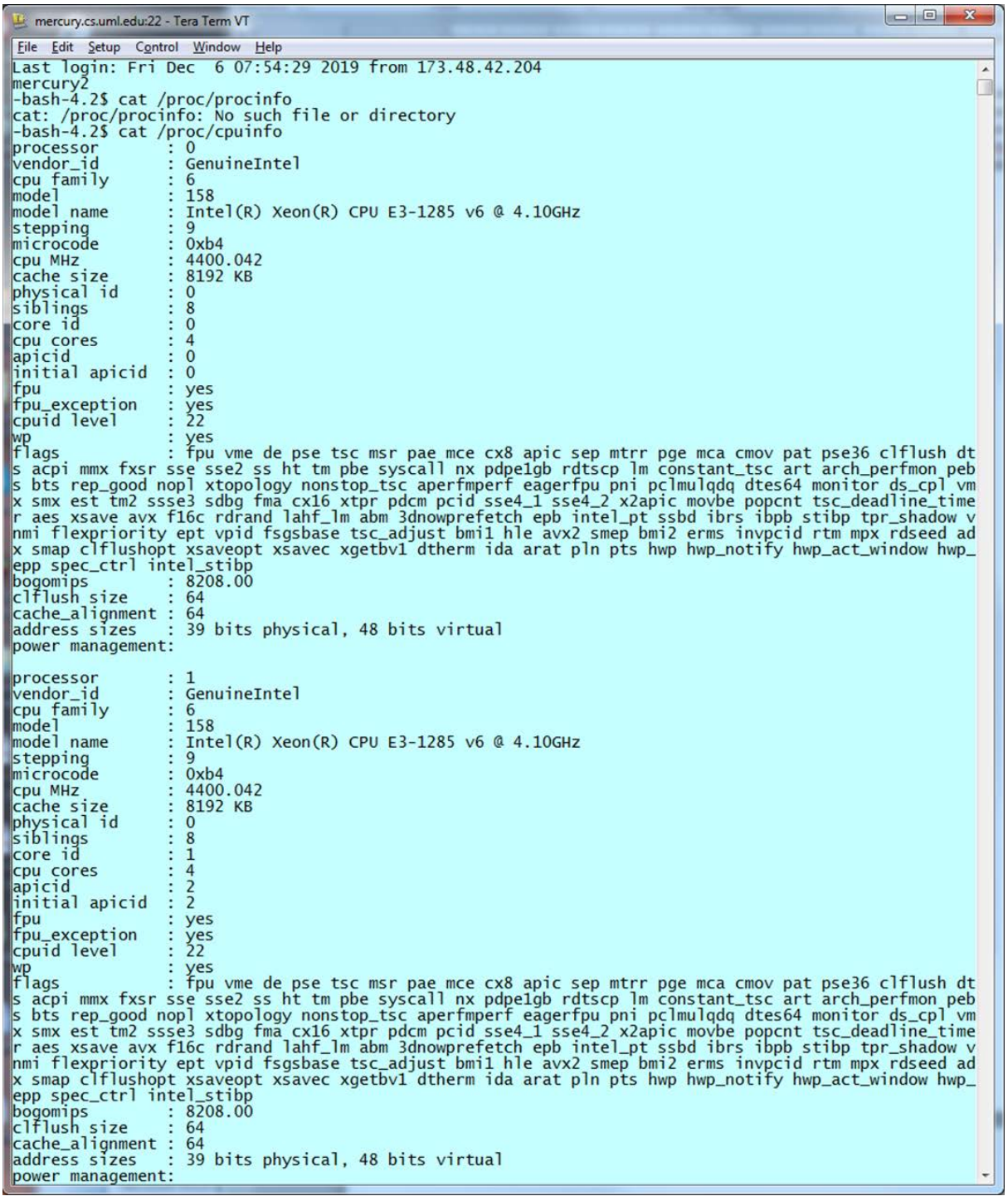
Another useful tool for mapping and analyzing your systems PCI connectivity can be found from PCI-Tree. PCI-Tree is another freeware application that can provide very low-level details about the busses, bridges and controllers comprising your system. You can find PCI-Tree at:

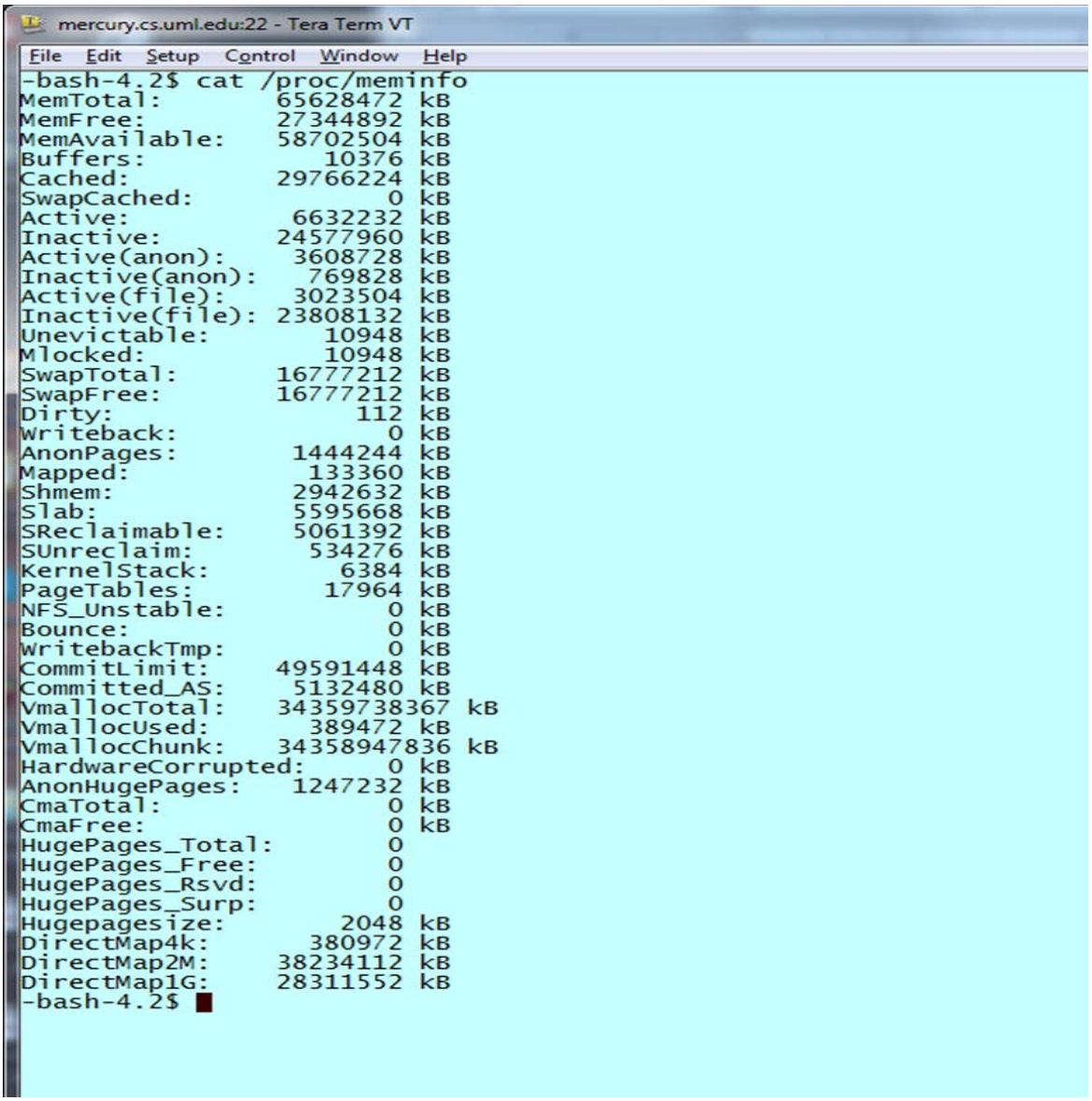
http://www.pcitree.de/index.html

The "Manage" option discussed above is easiest accessed from a Windows 10 system by pressing the start button and selecting the "File Explorer" button (above the "Settings" button). Within "File Explorer", find the "This PC" icon and right click it to get the same menu described above.


Computer Management
File Action View Window Help
Computer Management (Local)
System Tools
Event Viewer
Shared Folders
Local Users and Groups
Performance Logs and Alert:
Device Manager
Storage
Removable Storage
Disk Defragmenter
Disk Management
Services and Applications
UMASS-A31DCA090
ACPI Multiprocessor PC
Microsoft ACPI-Compliant System
ACPI Fixed Feature Button
ACPI Power Button
High precision event timer
Intel(R) Core(TM)2 Duo CPU E6550 @ 2.33GHz
Intel(R) Core(TM)2 Duo CPU E6550 @ 2.33GHz
PCI bus
Intel(R) 82566DM-2 Gigabit Network Connection
Intel(R) 82801 PCI Bridge - 244E
Intel(R) ICH9 2 port Serial ATA Storage Controller 2 - 2926
Primary IDE Channel
Intel(R) ICH9 4 port Serial ATA Storage Controller 1 - 2920
Primary IDE Channel
Maxtor 6Y160M0
ST3160815AS
Secondary IDE Channel
TSSTcorp DVD+-RW TS-H653F
Intel(R) ICH9 Family PCI Express Root Port 1 - 2940
Intel(R) ICH9 Family SMBus Controller - 2930
Intel(R) ICH9 Family USB Universal Host Controller - 2934
USB Root Hub
Intel(R) ICH9 Family USB Universal Host Controller - 2935
USB Root Hub
Intel(R) ICH9 Family USB Universal Host Controller - 2936
USB Root Hub
Intel(R) ICH9 Family USB Universal Host Controller - 2937
USB Root Hub
Intel(R) ICH9 Family USB Universal Host Controller - 2938
USB Root Hub
Intel(R) ICH9 Family USB2 Enhanced Host Controller - 293A
USB Root Hub
Intel(R) ICH9 Family USB2 Enhanced Host Controller - 293C
USB Root Hub
Intel(R) ICH9DO LPC Interface Controller - 2914
Communications Port (COM1)
Direct memory access controller
ECP Printer Port (LPT1)
ISAPNP Read Data Port
Numeric data processor
Programmable interrupt controller
System board
System CMOS/real time clock
System speaker
System timer
Intel(R) Management Engine Interface
Intel(R) Q35 Express Chipset PCI Express Root Port - 29B1
ATI Radeon HD 2400 XT
Default Monitor
Intel(R) Q35 Express Chipset Processor to I/O Controller - 29B0
Microsoft UAA Bus Driver for High Definition Audio
SoundMAX Integrated Digital HD Audio
PCI Serial Port
Standard Dual Channel PCI IDE Controller
Primary IDE Channel
Secondary IDE Channel

Collecting configuration information on a Linux system is generally a matter of visiting the /proc local file system. System configuration routines as well as driver and module components typically create and populate simple text files in the /proc RAM based file system during system initialization or module loading. For example, the files /proc/cpuinfo and /proc/meminfo provide considerable detail about a system's processor(s) and RAM, as can be seen below (the cat command in Linux/Unix systems will print the content of a file to the screen):





There are many sources of system information, and we will continue to consider them as we work our way through the course.

Operating System Flavors

At this point, hopefully we have some perspective on the basic hardware environment that our PCs deploy, and the need for some sort of operating system to manage this collection of iron. In fact, there are many different varieties of operating systems out there, each of which is typically designed to meet slightly different hardware and end user requirements. These software systems range from very hardware specific implementations such as mainframe operating systems, to the more general purpose multiple platform systems (Linux, Window, etc.) to the specialized light-weight systems (embedded systems) that we associate with everything from smart phones to smart cards.

In each case, we find a collection of software that is designed to make the underlying hardware platform accessible with a minimal amount of application effort. The APIs of these various systems provide the specific services that make sense for the platforms they are deployed on. These APIs allow applications to focus on a more logical view of the hardware topology, and not have to concern themselves with platform details. To leverage the services that a given system offers, however, one has to have an overview of how these services are organized, and that's really the essence of this course. We need to expose the basic architecture that a contemporary operating system embraces, beginning with the concept of a process, and exploring the components, construction and lifetime of these system management entities.

First Lesson (weeks 1 and 2) Summary:

Operating Systems have evolved from the simple collection of run-time routines that were shared by various applications on early machines, to the complex resource managers and service providers that we find in systems today. In a sense, a contemporary operating system provides a user accessible abstraction of an underlying collection of complex and user hostile devices.

Computing platforms come in many different shapes and sizes, and so too do the various operating systems that accompany them. We will see, however, that the principals of operating systems implementation are well understood, and the architecture of systems like Windows and Linux , which may seem to have little in common on the surface, are remarkably similar in their deployment.

An understanding of operating systems goes hand-in-hand with an understanding of the hardware environment that these systems run in. In particular, it is critically important to build an understanding of how information moves along various types of busses, how these busses behave, and how their use must be carefully coordinated to preserve the integrity of the data that they carry. There is nothing more fundamental to an operating system than to maintain the consistency of the resources such a system must manage. An operating system is a vast repository of the shared data that defines the state of a computing system at any given time. These data are constantly changing in response to events that occur at nanosecond frequencies, but the operating system must ensure that each new state has been coherently mapped from its predecessor, and, regardless of the degree of contention, all updates to shared data are atomic and consistent.

OPERATING SYSTEMS FOUNDATIONS
MSIT 5170

University of Massachusetts Lowell
Department of Computer Science
Spring Semester 2020

Week #3 Lesson

1. The material in this lesson follows the reading assignment for ch 1.5 through the end of chapter 1 from the Tanenbaum text
2. Please read the text before you read through this lesson.
3. The material in chapter 1 provides a broad view of the topics that we will visit throughout the semester.
4. We continue to look at system level tools, and I encourage students to experiment with these tools. You will each be emailed a Linux account this week to use with some of these tools in various assignments.

Operating Systems Infrastructure:

While operating systems come in wide variety of flavors, contemporary systems tend to have a common unit of management called a process. A process is a living entity that exists within an operating system, serving as a resource envelope. The operating system itself can be viewed as a collection of data and code that will be shared by all of the processes that exist in a system at a given time. Each process is viewed as a private object that will include among its resources, a program that will contain its own executable code and its own data, but the operating system will supply additional code and data that all process will share.

I find it helpful to think of a process like a classroom in a school building. Each classroom is a separate container, orthogonal to all other classrooms, but they all share a set of common resources, like the corridors, rest rooms and open areas in the building. The execution of activity in one classroom has no effect on other classrooms, but what happens in the common areas affects everyone, and has to be carefully synchronized to avoid conflicts. Let's' consider some of the attributes that we ascribe to a process:

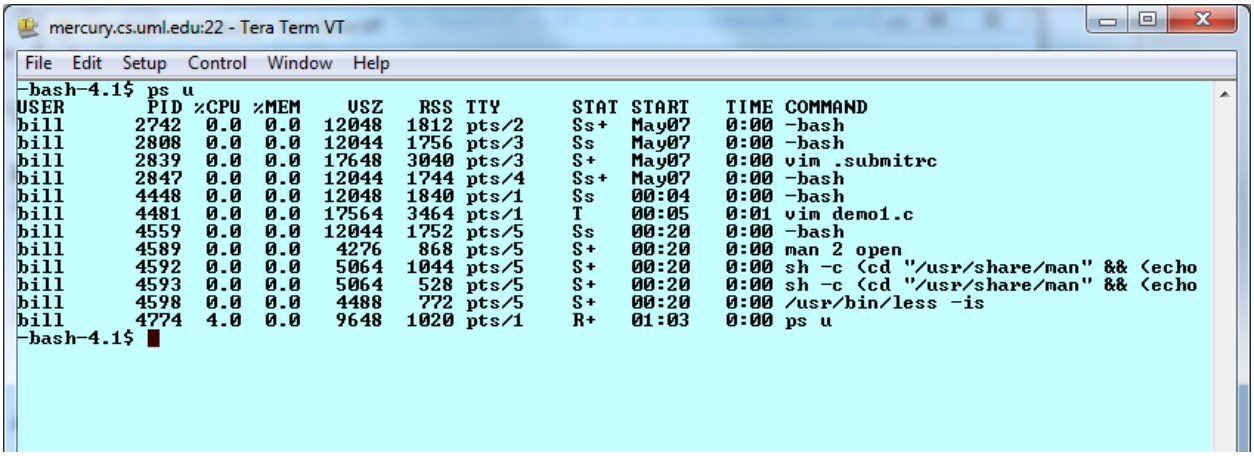
Process Attributes:

We find the concept of process in Window, Linux, Unix, etc., and, while some of the details vary from one platform to another, there are many common process attributes that we expect to find on all of these platforms.

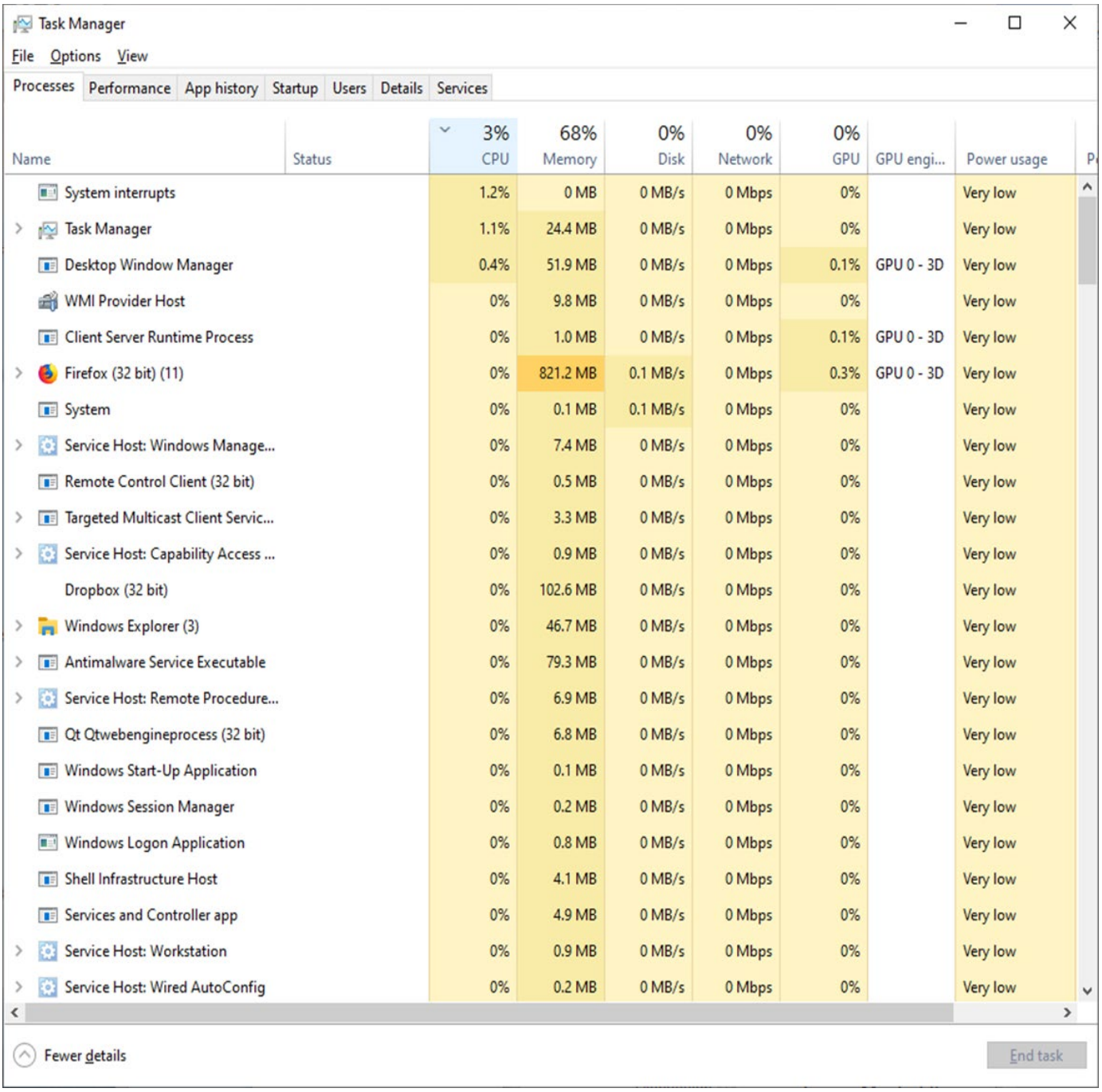
- A PID
    - A process has an identity that is normally expressed as a Process ID (PID). Each process in any of the systems mentioned above will have a unique PID during its lifetime. PIDs may be recycled, but at any given time each PID is unique.
- An Address Space

    - As in the school building analogy above, each process owns its own private address space. An address space is a contiguous collection of addresses, within which we will find the process' executable code, and its global and local data elements.
- Credentials
    - A process has identity credentials, which serve to mediate the access that the execution elements of the process have to various resources. Whether an execution element of a process has the right to write to a particular file, for example, will depend on the credentials of the process. The most important credential of a process is typically the account (user) that owns the process.
- One or more Threads
    - The execution elements of a process are commonly known as threads. Threads are the schedulable entities within a process, and only a thread can make computational progress. At any given time on a contemporary system (Windows, Linux/UNIX), there must be a thread running on every schedulable core in the system.

There are many more process attributes to consider, and we will revisit processes and threads in greater detail in chapter 2, but for now, the attributes itemized above will suffice. On a Linux system, the ps program can be run from a command line shell to collect some information about processes. Below I'm looking at just the processes that I currently have running on this system:



Similar information can be collected on a Windows system using the task manager application:



System Elements:

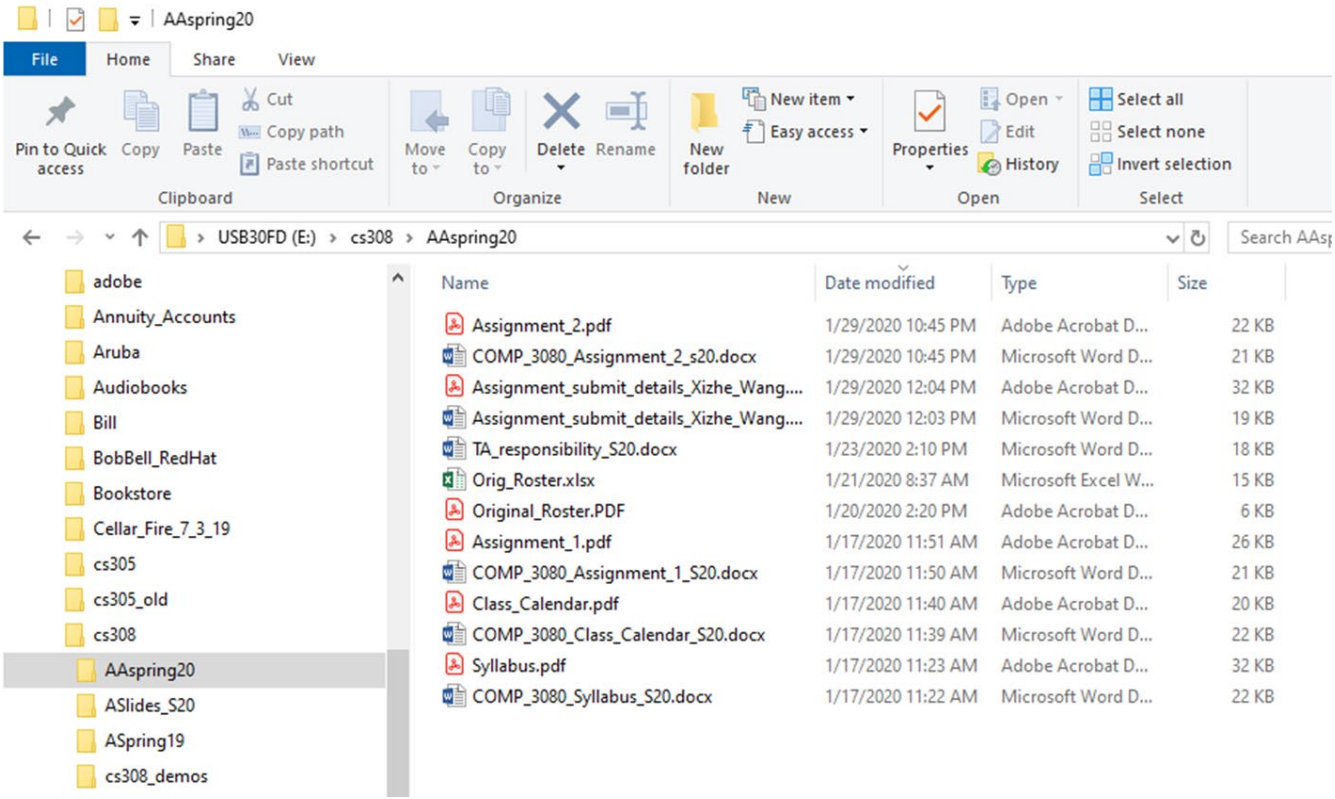
Last week we looked at some of the hardware components that make up a typical computing system. To complete the picture we must also look at some of the software components. A major software component is that of the process, described above, but we must also consider file objects with similar emphasis. File objects typically encapsulate those software components that we need to load into a system to make it useful. This includes the operating system itself, which is "booted" into the hardware from a file object at system startup time. File objects may

contain many different things, such as machine instructions, binary data, plain text data, etc., or may be used for their “name space” attributes only and have no actual content. In many respects, the ways in which the threads of a process can interact with various files available within the system define the system as a whole. Like processes, files have various attributes, and we want to consider a few here.

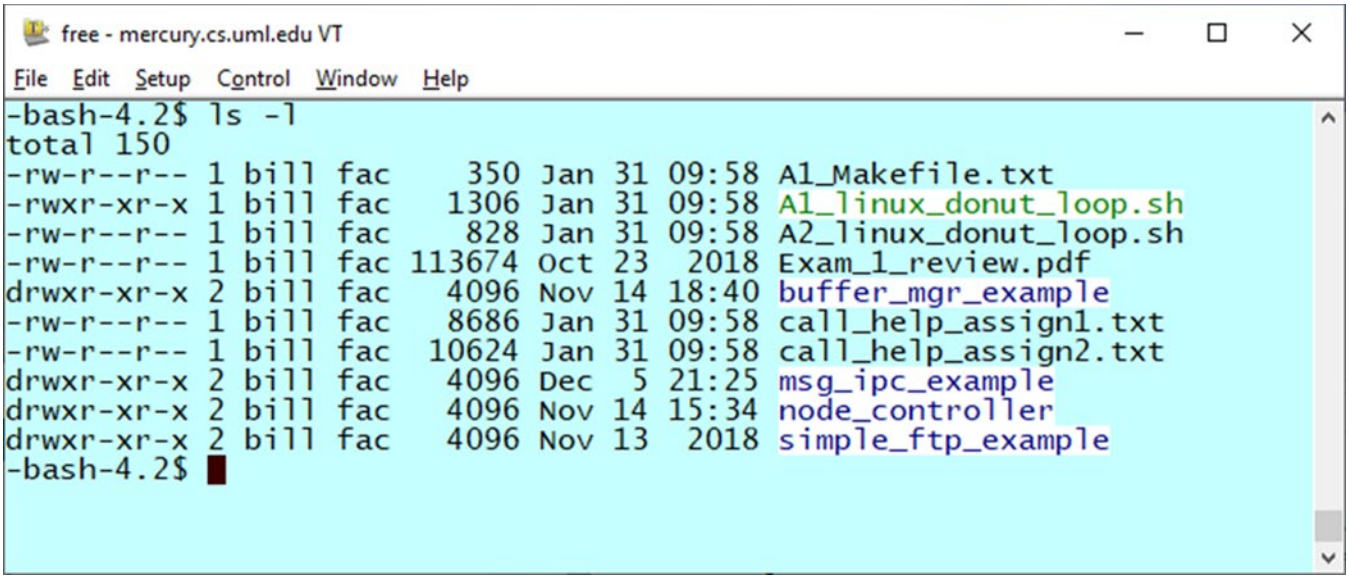
File Attributes:

- Object owner
  - Each file in a given system is normally owned by some account (user). The owner of a file object generally has control over what accounts in the system can access the object, and in what ways.
- Permissions
  - The access rights provided to various threads in the system. The threads are members of processes, and the credentials of a given process must be compared to the permissions of a given file to determine if a thread of said process has some desired access capability to the file object.
- Name
  - File objects typically have a human readable name, often expressed as part of a path name in environments that use hierarchical file systems. When a thread is interested in accessing an object, it must know the object name in order to establish a connection with the object.

Windows Explorer provides a view of some file attributes from a Windows7 system:



On Linux, the ls program provides a collection of file information for us:



System Access to Users:

We expect that an operating system will provide some set of mechanisms whereby a user can access and use the system's resources. This is typically done by processes that are created when the system is booted. These processes normally run what we call service programs (daemons) that monitor some potential entry port into the system. For example, at boot time one possible behavior of a Linux system is to create a process to run the KDE session manager, which monitors the system keyboard and mouse, and paints a login request message on the system console. Another possibility is for the creation of a process that runs the sshd program (secure shell daemon), and monitors port #22 in the TCP port space. In both cases, these daemon processes expect to communicate with a user who will provide credentials (username and password) for gaining access.

If the connecting user is able to succeed in authenticating with one of these daemons, the particular daemon will create a new process on behalf of the accessing user, and begin running a program often called a shell in this new process. The particular shell program may be a simple command line interpreter running in a real or emulated dumb terminal environment (the bash shell running in an ssh terminal emulation), or it may be a full featured GUI style shell (the KDE shell, using the graphical user interface components that the system hardware offers), but either way, it's a shell program being executed by one or more threads in a new process that was given your account credentials when it was created for you by some system daemon during your login activity. The shell program in this process provides an interface between you and the resources of the computing system.

Accessing System Services:

A shell program typically provides some kind of prompting mechanism, and awaits a user input operation. Once a user has responded to a prompt (by typing in a command, or clicking on an icon), the shell program attempts to fulfill the user's request. Some requests can be handled

directly by the shell process (eg. the pwd command when given to the Linux bash shell, asks the shell to list out the shell process' present working directory), but many requests must be fulfilled by creating a new process to run a separate program to do the work (eg. the ls –l command when given to the Linux bash shell, causes the shell to create a new process and load that new process with the ls executable program to produce the required output).

In either case, much of the information that the shell is likely to procure for a user is necessarily kept among the data of the operating system. As previously stated, the operating system is a collection of global data and code used to maintain system state, and includes the details of all processes, all active files and various other system resources. When you ask the shell to retrieve any of this information, the shell, like any other program, must interact with the operating system. Continuing with our school building analogy, this means that a thread of the shell process must leave the classroom it's in and enter the shared corridor region to find the data required, and bring a copy back into its private classroom. Such an excursion is the essence of a "system call".

System Calls:

A process lives in its own private address space, and its threads remain in this address space when they are executing the code owned by the program in the process. If a thread needs something kept in the operating system address space however, the thread must leave its own address space and begin executing code in the operating system address space (note that the calling thread cannot access the data directly, but must call operating system code to access the required data). This transition of address space is accomplished by a system call.

An operating system provides a set of functions that an application program can call when in need of access to the operating system's address space. Collectively, these functions are called the operating system API (application program interface). Since all applications need various system services (eg, the device driver code needed to read the keys from the keyboard I'm typing on right now is located in the operating system address space, not in my process address space), the system call API is a critical component of a system. When we gave the shell the ls –l command described above, a thread running shell code to process the command had to make a system call into the operating system address space to run the operating system code needed to create the new process that will eventually run the ls program.

We will have more to say about system calls later, but it is important to note, that an executing thread, regardless of which process it belongs to, is either executing user code in its own process address space, or is executing operating system code in the operating system's address space. We say that an executing thread is either in user mode (executing in its process address space) or in kernel mode (executing code in the operating system address space), and, while there is a separate and discrete address space for each process, there is only one kernel address space that is shared by all threads when system services are required.

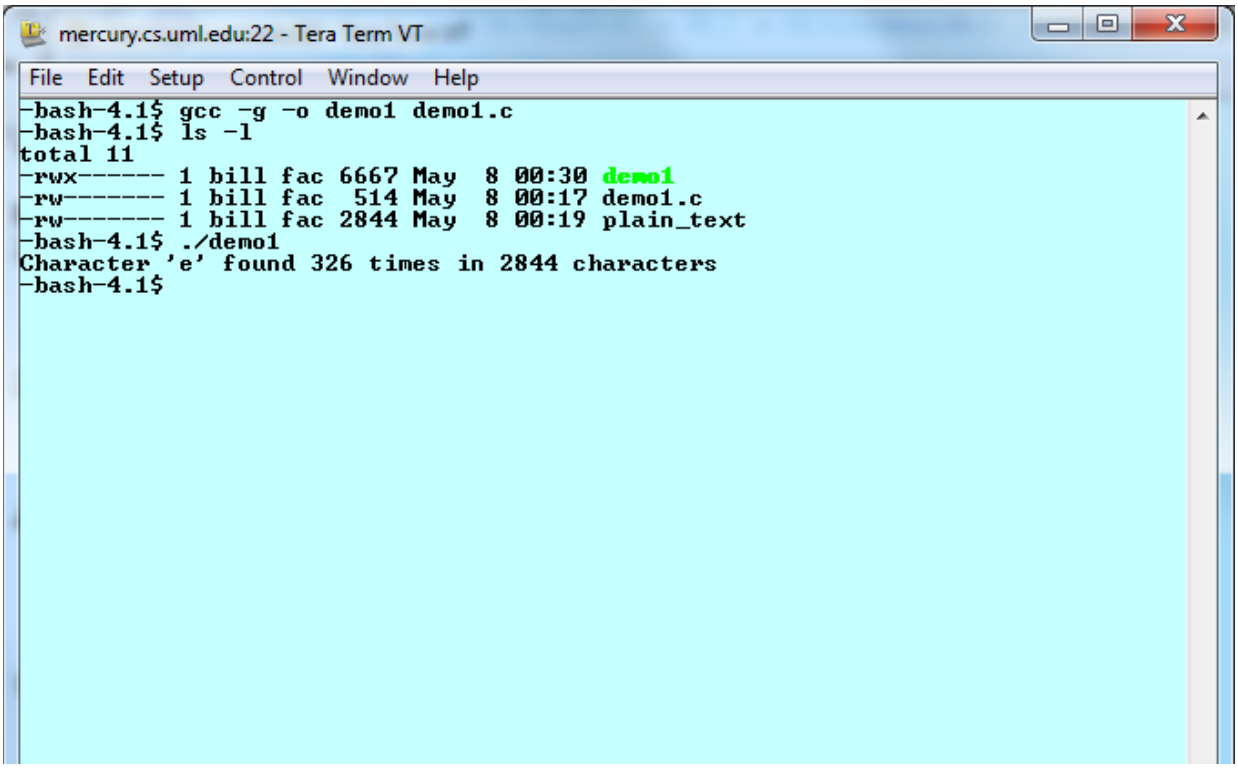
A C Program to Demonstrate:

The following simple program written in C provides an example of a program that will be executed in its own process by a single thread that will run the main() function. The main function in this example will make system calls to open a file that contains arbitrary text and then iteratively read the file content into a local buffer, attempting to retrieve 100 characters per iteration. Each time the system call to read from the file returns a non-zero count of bytes, the single thread will run user code to count the number of times it finds the letter 'e' among the

data. This single thread will run sometimes in user mode (when it is searching occurrences of the letter ‘e’), and sometimes in kernel mode (when it is opening and then reading from a data file).

```
// demo1.c:  mixed system and user code

#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

int main(void){
  char mybuf[100];
  int  esum=0, tot_count=0, count, chan, i;

  if((chan = open("./plain_text", O_RDONLY, 0)) == -1){
        perror("open failed ");
        exit(1);
  }

  while(count = read(chan, mybuf, 100)){
        for(i=0; i<count; ++i){
          if(mybuf[i] == 'e')++esum;
        }
        tot_count += count;
  }

  printf("Character 'e' found %d times in %d characters\n", esum, tot_count);
  return;
}
```

When the above code is run against a plain text data file that contains some of the text from this lesson, you can see the output produced below. Keep in mind that the bash shell that accepted this command created a new process to execute this code and when the code was complete, this new process terminated and control was returned to the bash shell process which posted a prompt for the next shell command. The single thread remains entirely in user space when it iterates through the for loop looking for the character ‘e’ in the character buffer called mybuf, but the thread must leave the local process space and move into the kernel space to run the open and read system calls.



Week #3 Summary:

Operating Systems come in many flavors, but most contemporary systems consider the process construct as the central system management element. A process is a resource container, whose attributes include a PID, identity credentials, a private address space, a program and one or more threads to run the program.

Files provide a source and destination for exchanging information between the outside world and various applications running under the control of some operating system. We envision the threads of various application processes making system calls to open, read and write files during the normal course of execution. How a thread is able to access and manipulate a file object is one of our principal concerns in this course.

The operating system is a collection of code and data that is shared among the threads of all processes, although it lives in its own single kernel address space. When a thread of some process requires operating system support, the thread makes a system call, using one of the functions provided by the platform in its system call API.

Each thread in a system belongs to some process, and when a thread is executing in user mode it is running code in the address space of its process. When a thread is executing in kernel mode, it has left the address space of its process and transitioned into the address space of the operating system. Whenever a thread is in kernel mode, it is executing operating system code to manipulate data that is kept in the operating system's address space.

The sample C code shows a singly threaded program that can be started from a shell prompt and will have its single thread run alternately in user mode and in kernel mode as it executes the main() function.

OPERATING SYSTEMS ORGANIZATION
MSIT 5170

University of Massachusetts Lowell
Department of Computer Science
Spring Semester 2020

Week #4 Lesson

1. The material in this lesson follows the reading assignment for ch 2.1 through the chapter 2.2 from the Tanenbaum text
2. Please read the text before you read through this lesson.
3. The material in chapter 2 will focus on process and thread details, with emphasis on process life cycles and thread states.
4. We continue to look at system level tools, and I encourage students to experiment with these tools. We will also build some simple applications in the Linux environment to explore some process and thread elements.

Processes and Threads:

Last week I introduced the construct of an operating system process, identifying it as the primary unit of organization within an operating system. We will now make a detailed examination of a process. Remember, a process is a living, dynamic entity that encapsulates a computation. Processes don't actually execute, but they gather the resources necessary for their resident program to be executed by one or more process threads. All threads belong to some process in the system, and it's always a thread that is dispatched to a CPU (central processing unit). Threads execute their machine instructions (the instructions forming part of their process address space), allowing the program within the process to make computational progress. A program is simply a passive collection of code and data, and to execute a program typically requires a process to be created, the program to be loaded into the process, and a thread component of the process to be dispatched to a CPU to begin executing the programs code.

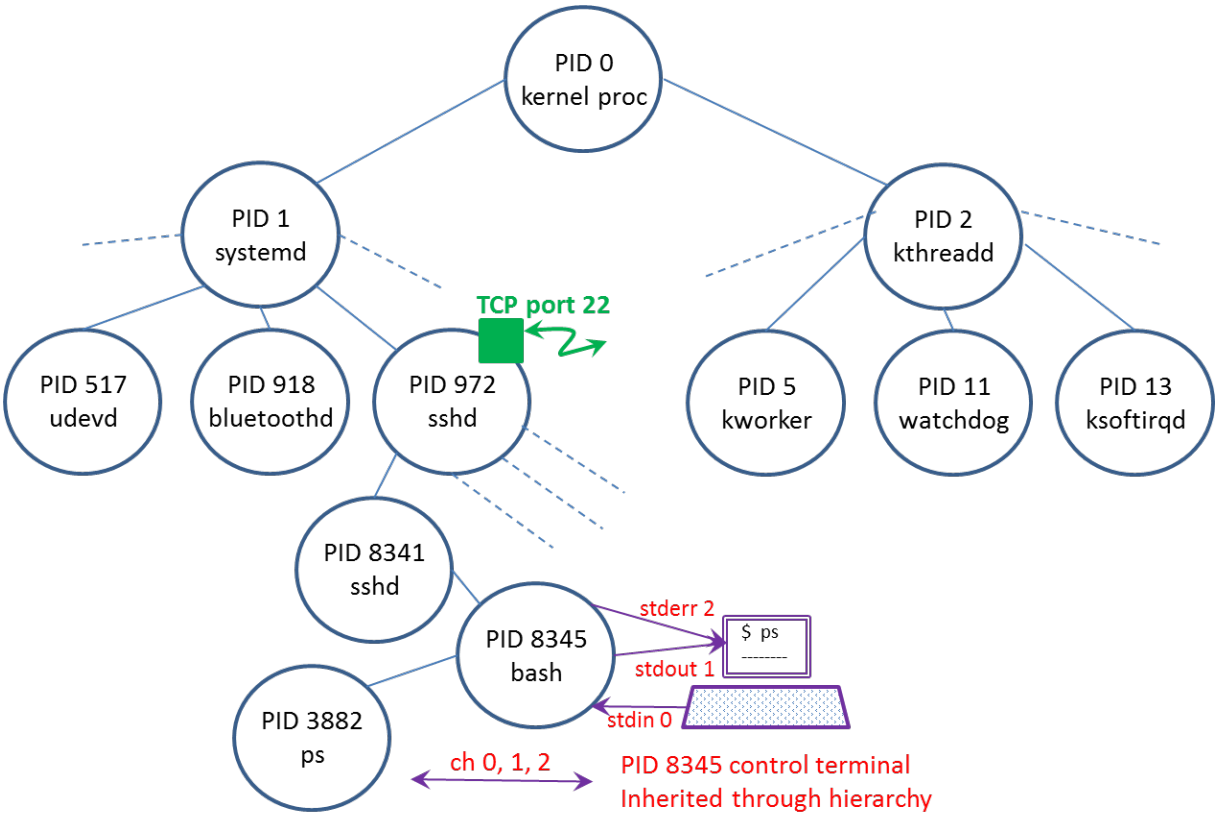
I will demonstrate much of our process specifics around the process model used in the Linux operating system environment, but you should understand that Windows processes are much more like Linux processes than they are different, and all of the important ideas that we look at through Linux glasses, apply in virtually the same way in the Windows world.

Operating System Startup:

The action of bringing an operating system to life from a cold start is typically called booting the operating system. Boot time activities vary from one system to another, but in Linux (and most UNIX based systems) the activity involves the only time during the systems life that code will run outside of the process construct. There is only monolithic boot code to execute while the system is configuring the environment to accommodate the operating system to come. At the end of the boot sequence, the monolithic boot code transforms itself into the first system process (in Linux this is a kernel process with no user scope), and this first process then creates

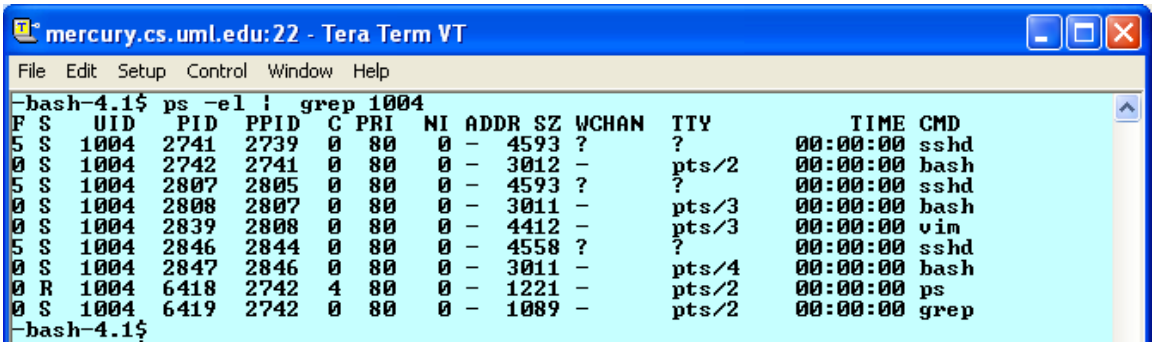
the first 2 user visible processes, the first with PID 1, running a program called systemd (init in older versions of Linux/UNIX) , and the second with PID 2 , running a program called kthreadd. These two initial user processes are immediate descendants of the boot time kernel process PID 0, and each of these processes will have a role in bringing the rest of the system to life. Once PID 0 is created, all other process creation will be done by some existing thread that, like all threads, belongs to some existing process. The main role of PIDs 1 and 2 is to create several more processes that have a part in maintaining the system and providing various services. These processes are often called daemons, based on the fact that they are started by the initial boot time user processes, and not by some user that has logged into the system.

Daemon processes are not really different from any other process, but their credentials often provide them with more privileges than processes that have been started from user login sessions. Many of these daemons have a UID (user ID) of 0, meaning that they are running with the credentials of the root user (superuser), and therefore have virtually unlimited access to all system components. The concept of a process with root credentials is similar to the concept of a process with administrator credentials in a Windows system. In the Linux system, all process have a parent child relationship, forming a tree with the kernel process PID 0 at the root. So every process is related in some way to every other process in the system, since they all have the same grand ancestor. Windows does not visibly maintain this sort of process tree, but it does keep track of all child-parent relationships in the kernel process data structures.



The diagram above shows a collection of processes that includes some of the early processes created at boot time from an actual Linux system. Most of the processes are running with root credentials and serve various daemon functions. I have expanded one of these daemons shown with PID 972 and running the sshd (secure shell daemon) program. A thread in this

process monitors TCP port 22 (the inbound port for ssh connections), and whenever a connection request shows up (from some outside client that is trying to log into the system over the network), PID 972 creates a child process to handle that request. Like its parent, the child (here shown as PID 8341), is running with root credentials, and has the ability to look into all system data in order to authenticate the requesting connector (validate the user name and password). While PID 8341 retains its root credentials, it spawns its own child to handle this user, changing the credentials of this child (here PID 8345) to those of the authenticated user, thereby limiting access through this process to only what the user credentials allow. PID 8341 checks this user's account data, and finds that the user wants to have a process created for it at login time that is set to run the bash shell program, so PID 8341 creates a child (here PID 8345) and loads that child process with the bash program, providing the user with a command line environment in a dumb terminal emulation. PID 8341 created the pseudo control terminal before creating the child that would eventually become the bash process.. The actual user, in this case, may be running an ssh program like putty or Tera Term from a Windows system. These are freeware programs for Windows that provide a dumb terminal emulation screen, and can be used to connect to a Linux host as described above. Here is a screen shot of my Tera Term session interacting with a process on a Linux host running the bash shell. I've asked the bash shell to create a child process to run the ps program (you can see the ps process (PID 6814) itself listed among the various processes that I currently own on this system. In this example, I have asked the bash process to actually create 2 children, the ps child and the grep child. The ps child process will generate a lot of data (there are currently 250+ processes running on this host), but that output is sent to the grep process for filtering. I've asked grep to only report lines that have my user ID on them (my UID is 1004). You can see the results, including the sshd login process shown in the diagram above (there are actually 3 sshd processes here because I've used ssh to log into this system from 3 different terminal emulation windows).

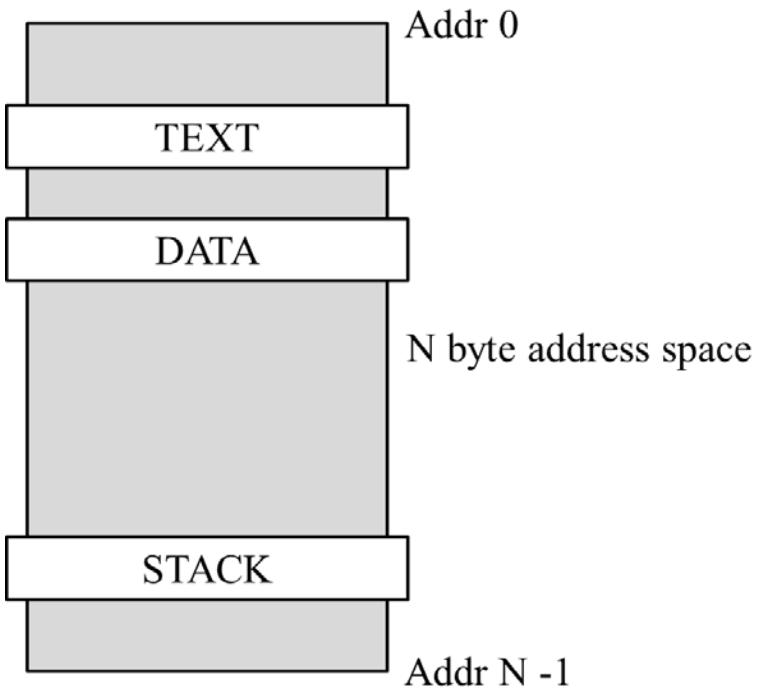


The output presented by ps includes my user ID (UID), the PID of each process shown, the PPID (parent PID) of each process, and the name of the program that each process is running in the far right column. Other fields provide additional per-process information that we will look at later. First, we want to revisit the elements that comprise a process, and examine more details.

Process Attributes:

On Linux platforms we can expect a process to be comprised of a number at attributes, some of which we have already seen, and some of which we will discuss in more detail below.

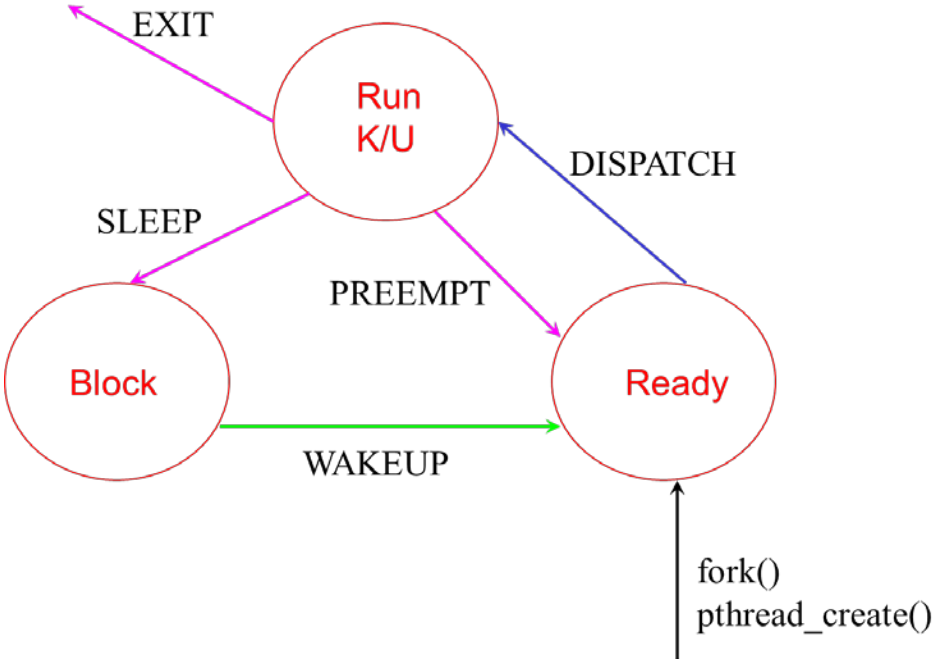
- A process ID (PID)
- A parent process ID (PPID)
    - Each process, with the exception of the initial kernel process PID 0, must have a parent PID. If a process should find that its parent has terminated before it has, then the process will be adopted by PID 1. Any process will either have a PPID of its original parent, or that of PID 1.
- An Address Space
    - Each process in a Linux system is given a private address space that consists of some collection of contiguous virtual linear addresses that the process may populate with memory objects. The size of this space is a function of the underlying hardware that the operating system is running on. A Linux system running in 32 bit mode on an x86 platform will normally provide each process in the system with a 3 GB address space.
    - A process populates its address space with memory objects, including:
        - A Text object that includes the program code for the program running in the process
        - A Data object that contains global variables and dynamic heap allocation space needed by the process
        - A Stack object that manages the local variables and activation records of a thread of the program (must have one stack per thread).
    - The memory objects described above are not the only memory objects that may be found in a process address space, but a process cannot exist unless at least these three are found in the address space.



- Credentials

- A process has identity credentials, which serve to mediate the access that the execution elements (threads) of the process have to various resources. These include the UID and GID (group ID) of the process, and are inherited from the creating parent process. Generally, these credentials are set in the user shell process during login, and all subsequent processes created in the hierarchy from this shell process down will share common credentials via inheritance.

- One or more Threads
  - The execution elements of a process are commonly known as threads. Threads are the schedulable entities within a process, and only a thread can make computations progress. Threads always have a specific state at any given instant, and can move from one state to another over time. The state transition diagram below depicts the states of thread in a Linux system:



  - When a thread is in the Run state, it is executing on a CPU. From the Run state it may choose to finish (EXIT), wait for some event like a read from disk to complete (SLEEP), or move back to the Ready state, yielding the CPU to some more worthy thread (PREEMPTION).
    - If a thread leaves the Run state, then another thread must be moved to the Run state via the DISPATCH arc immediately.
    - There must be a thread in the Run state on each CPU at all times for the system to function properly, so during the system boot activity, the PID 0 kernel process creates one thread per CPU called an IDLE thread. An IDLE thread typically has the lowest priority of all threads in the system, so is only dispatched to a CPU when there is no other thread in the system ready to run.
  - If a thread is in the Block state, we assume that some event will eventually occur that will allow it to be moved to the Ready state.
    - The type of event that we anticipate is something like an interrupt to a CPU from a controller that has just completed some task (ie., a disk

controller that just completed transferring data from a disk device into memory).
        - When the interrupt is raised against a CPU, that processor will save the state of the thread it is running (remember, a CPU must be running some thread at all times), and the running thread will be forced to enter the kernel (if it's not already there) and execute the appropriate exception routine to handle the event that just occurred.
        - The handler code will, among other things, determine if any thread was in the Block state waiting for this event, and if so, will transition that thread from the Block state back to the Ready state, following the WAKEUP arc.
        - When the thread has finished running the specific exception routine, it will go back to whatever it was doing before the event occurred. If it was running in user mode before the event it will return to user mode after the event is handled, and if it happened to be running in kernel mode when the event occurred, it will return to whatever it was doing in kernel mode before the event occurred.
    - You should see from this description, that the operating system code is run by arbitrary threads in one of two ways:
        - A thread specifically makes a system call, in which case the thread voluntarily moves from running in user mode to running in kernel more to fulfill the system call. This is known as in-context kernel entry.
        - A thread is snatched into the kernel to run an exception handler because that thread happened to be running on the CPU when the exception was presented to that CPU. This is known as out-of-context kernel entry.
    - You should also understand that since each CPU has an interval hardware timer that posts an interrupt to the CPU after some few milliseconds, no thread remains running in user mode for long, before it finds itself in the kernel running the exception handler for the interval timer. These interval clock interrupts provide a mechanism to ensure that no thread can monopolize a system CPU for any length of time.
    - A thread can only EXIT the system from the Run state, which assures the system that any thread that wants to finish what it's doing will have to run its way out of the system, cleaning up its resources as it leaves. In other words, threads must commit suicide, they cannot be killed by some other thread from a distance.
        - If we want to get a thread out of the system, we must first get it to the Run state, and then convince it to commit suicide.
        - This is accomplished by sending a software exception to a target process in the form of a SIGNAL (Linux has 32 different types of signals). The signal forces some thread in the target process to not only kill itself, but to kill the entire process and all of its threads.
        - When the last thread in a process moves through the exit code, it brings the entire process down with it. A process cannot exist without at least one living thread.
- Other process attributes include:
    - A working directory (shared by all threads).
    - A table of open channels (shared by all threads), connecting the process to files, and various inter-process communication features (IPC mechanisms like pipes, sockets, shared memory, etc.).
    - A table describing SIGNAL behavior (shared by all threads), with an entry for each of the 32 signals that can be used.

- A default set of scheduling parameters used by threads created within this process; unless they are overridden during the thread create activity.
    - A process defines a default scheduling policy, priority, CPU affinity, etc., used as thread attributes during the creation of threads.

Threads:

As we've seen above, threads comprise the executable elements of a process. A process must have at least one thread to exist, but a single process may have thousands of threads if needed. If an application logically has multiple concurrent activities that it must support, it may be deployed as a collection of singly threaded processes, or as a single multiply threaded process.

In either model, we will have threads running on CPUs to deploy the application, but the cost of switching a CPU from one thread to another is very dependent upon the threads involved in the context switching activity. The cost of a context switch (in terms of execution time and effort) depends on the address space manipulation that accompanies the context switch.

- When a thread from some process PID x leaves the Run state (yields the CPU to another thread), and the inbound thread (the thread being dispatched onto the CPU) lives in another process such as PID y, we say that we have a heavy weight context switch. Because each process has its own address space, when we switch between threads from different processes we have to discard the cached address space components of the outbound thread's process, and re-establish a new cached address space on the CPU for the inbound thread's process address space. This activity has a very negative effect on system performance.
- If we switch between threads that live in the same process, however, we can retain the cached elements of the process address space, and avoid the performance impact of a heavy weight context switch. This type of switching is known as a light weight context switch, and supports much greater system performance.

As the available hardware becomes more concurrent by way of multiple cores, embedded memory controllers and very high speed peripheral interconnects, applications built around multithreaded technology are increasingly in demand. Highly concurrent applications, however, bring with them the need to manage complex synchronization requirements. This need for synchronization has always been a core issue for operating systems developers, but it is now becoming a growing concern for application developers working in the world of multithreaded software deployment. We will take a look at these synchronization issues next week, as we continue to work our way through chapter 2.

Week #4 Summary:

Operating systems are organized around the process construct. A process is the unit of management for most contemporary operating systems, and is viewed as a resource container for some computation.

Processes have many attributes, but key among them is a private address space, a collection of credentials, a executable program, and one or more threads to execute that program on a CPU. During a system boot, the monolithic boot code brings the system up to a point where the first process can be created, and the operating system is established. All subsequent processes

which may come into existence during the life of the system are created by a thread from some exiting process.

We consider a running system as a collection of processes, each of which has one or more threads that may, from time to time, be dispatched onto a CPU. A critical requirement of such a system is that each CPU in the system must have a thread running on it at all times. During system boot, the initial kernel process addresses this issue by creating an IDLE thread for each of the CPUs in the system. IDLE threads have a very low priority, and so will only be dispatched to a CPU if there are no other threads in the system that want to run.
The need for a thread per CPU at all times is a critical part of how an operating system runs its code and manipulates its data. The operating system is just a passive collection of code and data that must be shared by the threads on a platform, and it's these individual threads that actually run the operating system code when it needs to be run. A thread that needs system services (like access to network packets that are arriving over the internet), can voluntarily place itself into the operating system address space and run the OS code necessary to drive the support required by making system calls. The code in the operating system that responds to system calls is referred to as base level code. There are other code routines in the kernel that must be run asynchronously when certain events occur in the system, however, and this code is executed by whatever thread is running on a given CPU at the time that the CPU in question receives an interrupt or some other form of exception. Since we can never know when an exception will posted against a CPU, it is critical for each CPU to be running some thread that can be immediately snatched into the kernel and coerced to run the exception code (thus the absolute need for the IDLE threads).

Because user threads are frequently executing in the kernel (as a result of a system call or exception), the system has to be carefully designed and deployed to avoid race conditions and data collisions among these potentially many concurrent threads. Using our school building analogy again, you're safe from collision with anyone if you're the only one running around your classroom (private address space), as is the case for a singly threaded process. If you leave your room to move into the corridor, however (as the result of making a system call or fielding an exception on your CPU), you must be more careful about your movements, because there are other active entities (threads) out there (in the kernel's address space) that you could run into. This need for thread synchronization will be our topic next week.

OPERATING SYSTEMS FOUNDATIONS
MSIT 5170

University of Massachusetts Lowell
Department of Computer Science
Spring Semester 2020

Week #5 Lesson

1. The material in this lesson follows the reading assignment for ch 2.3 through the end of chapter 2 from the Tanenbaum text
2. Please read the text before you read through this lesson.
3. The remaining material in chapter 2 will focus on synchronization, scheduling and inter-process communication issues (IPC).
4. We continue to look at system level tools, and I encourage students to experiment with these tools. We will also build some simple applications in the Linux environment to explore some process and thread elements.

System Synchronization Requirements:

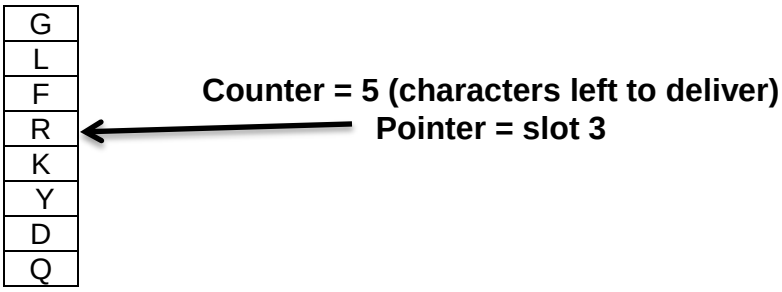
When we think about an operating system, we envision a collection of code and data that is required to maintain a coherent, stateful computing environment. Since the kernel's address space is visited by all of the threads in the system at one time or another and these threads are often interested in manipulating the kernel global data (with system calls like setpriority() or fork() for example), maintaining this data in a coherent form is a major challenge.

In this section of the book, we will examine the need for synchronization among threads that are accessing kernel code that will manipulate shared kernel global data. Remember, threads enter the kernel regularly, either voluntarily when making a system call or when coerced to handle some exception (like a clock or disk controller interrupt). In either case, the thread begins to execute kernel code that references or updates shared kernel data. Because we could have two different threads trying to run the same kernel code at the same time (if we had a system with two or more CPUs), the data these threads manipulate can become corrupted unless the kernel code is very careful about how the data is touched.

For example, suppose two threads from the same process both attempted to get the next character from an open file connection (channel). If we had a separate CPU for each of them to run on, then they could be running the exact same kernel code at the exact same instant, and this could lead to inconsistent results. The kernel code that performs a read system call and returns data to the caller depends on a kernel buffer that holds the requested data, and that buffer is controlled by a counter that specifies how many bytes are in the buffer, and a pointer that specifies where the next character will be taken out from the buffer. The kernel read code must check the counter to see if there are more characters in the buffer, must use the current buffer pointer to remove one character from the buffer to return to the caller, and then must update the pointer (to the next slot in the buffer) and decrement the character count (since a character has been logically removed). The pointer and the counter must be manipulated by only one thread at a time (an atomic update), or we could have erroneous results.

To understand the problem, you have to remember that to test or manipulate a data object, the object must first be brought from its RAM storage location into a CPU register (loaded from memory), where a CPU instruction can operate on it. If the data object is changed as a result of such an operation, then it must be moved back to RAM (stored into memory) after the change. The order of loads and stores on a computing system with multiple CPUs is non-deterministic. Each CPU needs a bus cycle to move an object between a CPU register and a memory location, but the CPUs can never be sure about what bus cycles they will get, since they are often asking at the same time, and the bus arbiter will assign the available bus cycles in a non-deterministic way. For the example above, 4 critical bus cycles will be needed for each thread to complete its read. A cycle will be needed by each thread to load the counter and check for more characters. If there are characters, then the counter value will be decremented, and the updated value will need a bus cycle to get back to memory. Each thread will then need to load the pointer value to remove a character from the buffer to return from the read call, and will then increment the pointer and use a bus cycle to store the updated value in memory.

A kernel buffer with characters to be returned to a read system call:

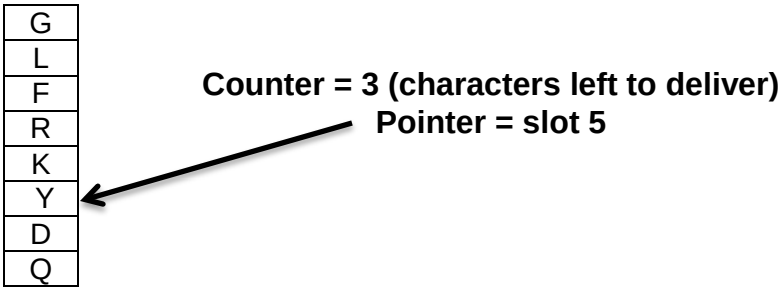


If the relative order of the bus cycles goes like this:

1. Thread A loads counter
2. Thread A decrements the counter (counter = 4) and stores it back to memory
3. Thread A loads the pointer
4. Thread A uses the pointer (return character R), increments the pointer to the next buffer slot and stores it back to memory (pointer now = 4, pointing at character K)

4. Thread B loads counter
5. Thread B decrements the counter (counter = 3) and stores it back to memory
6. Thread B loads the pointer
7. Thread B uses the pointer (return character K), increments the pointer to the next buffer slot and stores it back to memory (pointer now = 5, pointing at character Y)

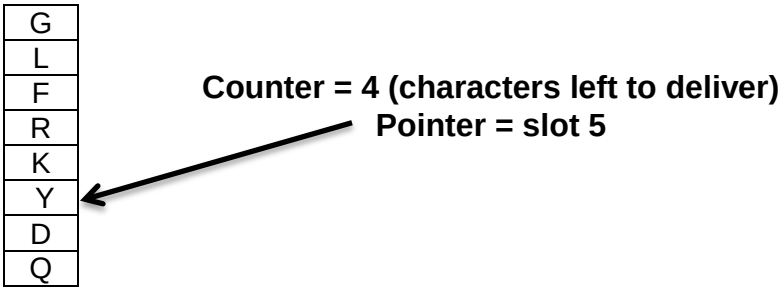
Then everything is fine, and each thread gets a discrete character from the buffer (thread A got R and thread B got K) and the buffer has been updated coherently as shown below:



But we could have had a different result if the bus cycles were given out in some other order:

1. Thread A loads counter
2. Thread B loads counter
3. Thread A decrements the counter (counter = 4) and stores it back to memory
4. Thread B decrements the counter (counter = 4) and stores it back to memory
   Counter has not been updated properly
5. Thread A loads the pointer
6. Thread A uses the pointer (return character R), increments the pointer to the next buffer slot and stores it back to memory (pointer now = 4)
7. Thread B loads the pointer
8. Thread B uses the pointer (return character K), increments the pointer to the next buffer slot and stores it back to memory (pointer now = 5)

The pointer update shows 3 more characters left (Y, D and Q), but the corrupted counter value thinks there are 4 characters left, which may lead to a buffer overflow, and the return of invalid data. Now the picture looks like this and is **inconsistent**:



Synchronization Mechanisms:

You might think that the previous example could only happen on multiple CPU systems which could have two or more threads running in true concurrent fashion, but this problem exists even in single CPU systems. When a thread is running on the only CPU in the system, it's reasonable to assume that any code it runs to update control data would happen atomically, but you must remember that a thread can be coerced to run an exception routine at any time (if the CPU it's running on just took an interrupt from a network controller for example), and if the exception code attempts to manipulate any of the data that the thread was working on at the time of the interrupt, that data could become corrupted in a fashion like that shown above.

In the case of the single CPU type system, the operating system uses a strategy of disabling interrupts before attempting to execute any code that might update data that an interrupt handler may also be interested in. If no exceptions can coerce the running thread to run contending exception code, then the thread will achieve the required atomic update to the potentially shared data. In single CPU systems, any code that has to run atomically begins by disabling interrupts, and finishes by restoring interrupts.

Spin Locks:

This approach does not solve the multi-CPU problem, since disabling interrupts is a CPU specific activity, and while a thread may be safe from contending exception code on its own CPU, we could find contending code running on some other CPU. In systems with multiple CPUs, we revert to using a locking mechanism on the data that is subject to contending update activity. The lowest level lock used in such systems is called a spin lock, and, as the book explains, spin locks can be implemented in simple load/store systems using software solutions like Peterson's algorithm. Even though software solutions are known to work, they require multiple machine instructions per execution and are thus very inefficient. Most hardware vendors have added a specialized machine instruction to their CPUs (often called a test-and-set instruction) to support the efficient implementation of spin locks. A test-and-set style instruction is specialized in the sense that it guarantees the CPU executing the instruction two back-to-back bus cycles that no contending CPU can get between. This means that such an instruction can load a memory location into the CPU on one cycle, and store a CPU register back into memory on the guaranteed next cycle. The book describes an Intel x86 version of this instruction called the exchange instruction:

XCHG Reg, Mem_location

The XCHG instruction guarantees the atomic exchange of the content of the CPU register **Reg** with the RAM location **Mem_location**, with two adjacent bus cycles and no possible intervention from any other bus master.

The spin lock depends on the Mem_location keeping the state of the lock. For example, let's say that Mem_location is **0** if unlocked, or **1** if locked. The exchange instruction is then used with a CPU register set to 1 against this Mem_location. After the instruction executes, the Mem_location will be 1, since the Reg content is deposited there, and it's the **value that is now in the Reg** that determines if we were successful in getting the lock. If the value in the Reg is still 1, then Mem_location was locked when we exchanged with it, and we will just do the entire operation over and over again (spin) until we see the Reg value become 0. If the value in the Reg is 0, then we now own the lock, since we know we put a 1 there no matter what, and if we got a 0 back, it means that the Mem_location was unlocked when we made out last exchange execution. The term spin lock lives up to its name, since a code path that needs such a lock will simply spin through try after try until it succeeds in getting the lock. When the holding code path is done doing its atomic updates, it releases the spin lock by simply storing a 0 value back into the Mem_location. Note that any other code path on a different CPU that needs the lock will be spinning until the current holder stores that 0 value back into the Mem_location.

Blocking Locks:

Spin locks are always at the base of any locking mechanism used on a multiple CPU platform, but they can only be used in limited ways. If a thread is attempting to get a busy spin lock, then that thread is burning off CPU cycles without getting any real work done. If the wait is expected

to be very short (a few hundred instructions worth of execution), then this may be reasonable, but when a wait is likely to last a long time, it may be better to force the waiting thread to give up its CPU and move from the **Run** state to the **Block** state until whatever it's waiting for happens. Spin locks leave the caller in the Run state, possibly wasting valuable CPU time as its spins waiting for the lock. Blocking locks force the calling thread to the Block state if they're not available, allowing the CPU to be used by some other productive thread. One of the most widely deployed types of blocking lock is an object called a semaphore. A semaphore is a data structure that contains two elements, a simple integer count field, and a queue. The count field is some value that ranges from 0 to some logical positive number (depending on the application). The queue is used to keep track of any threads that are currently blocked on the semaphore. There are two basic operations for a semaphore known as the wait and signal operations. The wait operation is a **conditional decrement** with the following semantics:

wait(sem) if **sem.count** != 0, then **decrement** sem.count and continue execution
else
add yourself to **sem.queue**, and block (yield the CPU to another thread)

the signal operation is a **conditional increment** with the following semantics:

signal(sem) if **sem.queue** is not empty, wake up a thread (move to **Ready** state)
else
increment the sem.count

As you can see, a simple lock uses a possible sem.count value of 0 or 1 (known as a binary semaphore), with the idea being that if the sem.count is currently 1, when a thread makes a **wait(sem)** call, that thread will **continue to execute**, but the sem.count has now become 0. A subsequent thread calling wait(sem) before the first thread calls signal(sem), will find a sem.count of 0 and will thus **block** on the **sem.queue.** When the first thread is finished with its atomic operations, it will call **signal(sem)** which, in this case, will wake up the thread on the sem.queue by moving it from the sem.queue (a Block state) to the **Ready** state. The semaphore components must be carefully synchronized themselves, and this accomplished (as with all synchronization primitives) using an underlying spin lock (remember, spin locks are at the bottom of all synchronization mechanisms for multi CPU systems). The **wait(sem)** function, for example, depends on a spin lock to access the semaphore safely, check its value and proceed to either decrement the sem.count and release the spin lock and allow the caller to continue execution, or, if the sem.count ==0, to safely place the calling thread on the semaphore queue, change its state to Block, unlock the spin lock and **context switch** the CPU to another thread.

Semaphores can also be used to protect multiple instances of a resource, by allowing the **sem.count** value to range between 0 and N, where N represents the instance count of some resource. The classic example of this type of use is a **ring buffer**, into which some producer thread places objects, and from which some consumer thread removes objects. Ring buffers are used extensively within an operating system to provide hysteresis between **producer and consumer** threads that may run at different speeds and at different times.

For example, a network interface controller (**NIC**) is connected to a network and listens for packets moving across the network that are addressed to it. When it discovers a packet on the network that carries its destination address, it copies the packet off of the network wire and into a packet buffer on the controller if that buffer is available (if not, the packet is simply discarded). Once the controller has captured the packet, it interrupts a CPU and coerces the current thread

that's running on that CPU to move into kernel space and run the **exception handler** for the NIC interrupt. This **interrupt handler code** will copy the packet from the controller's packet buffer into a slot in a ring buffer used by the driver to manage packets. Interrupt handlers must do the minimum amount of work required to service their device, and finish as quickly as possible, since they've basically hijacked some thread that has its own job to do, and must release this thread back to where it came from. This means that the interrupt handler will not process (rip) the packet (that's far too time consuming to do during an interrupt), but will clear the device buffer so the device can receive the next packet targeted to it, by attempting to move the just arrived packet from the controller's packet buffer into a slot in a memory based **ring buffer**. The interrupt handler (in conjunction with the NIC) is the ring buffer producer, and we now need a thread to remove the packet from the ruing buffer (a consumer) and extract its content to the intended endpoint. The consumer thread is usually a thread that is either processing a previously arrived packet, or is waiting in the Block state for the arrival of a packet to process. The producer and consumer threads synchronize their operation by using a ring buffer of N slots, a **producer semaphore** (Psem) whose **Psem.count** value can vary between 0 and N, a **consumer semaphore** (Csem)) whose **Csem.count** value can also vary between 0 and N and two pointers called **in_ptr** (used by the producer) and **out_ptr** (used by the consumer0).

```
#define N 100
some_object_type    ringbuffer[N];
int                 in_ptr=0, out_ptr=0;
sem_type            Psem = N, Csem = 0;  // Psem.count counts empty slots
                                         // Csem.count counts slots holding objects
```

PRODUCER CODE

```
void pro_funct (some_object_type *obj){
  wiat(Psem);    // wait for a space
  riingbuffer[in_ptr] = *obj;
  in_ptr = (in_ptr + 1) % N;
  signal(Csem);
  return;
}
```

CONSUMER CODE

```
some_object_type con_funct(void){
  some_object_type rtn_obj;
  wait(Csem);   // wait for an object
  rtn_obj = riingbuffer[out_ptr];
  out_ptr = (out_ptr + 1) % N;
  signal(Psem);
  return rtn_obj;
}
```

We start the producer semaphore at N so we can safely perform the **wait(Psem)** call N times before the calling producer would have to wait (out of empty slots). Each time the producer runs through its code, it calls **signal(Csem)**, to either awaken a waiting consumer thread, or increment the **Csem.count** of objects. We start the consumer semaphore at 0 (since there are no objects initially in the ring buffer), assuring that a consumer who arrives before any producer will have to wait on the **Csem.queue**. The semaphore implementation shown, guarantees that the number of producer executions can lead the number of consumer executions by no more than N, assuring that a producer cannot overflow the ring buffer (i.e. write into a slot that has not yet been consumed since the previous time it was written), nor can a consumer underflow the ring buffer (consume from a slot that has not been re-written since the last time it was consumed). The in and out pointers are incremented **MOD N** to insure their values can never be out of bounds, and must always remain between 0 and (N-1) (the valid subscripts to an N element array).

Other kinds of blocking synchronization primitives are discussed in the book (mutexes, monitors, message passing, etc.), but I will only test over semaphores on the first half exam.

Scheduling:

Now that we've looked at processes and threads in some detail, and we've looked at the states and state transitions of threads during their lifetimes, we need to consider the **Dispatch** arc in our state diagram in greater depth. Threads are dispatched to the **Run** state as a result of the **scheduling mechanisms** used by an operating system. Scheduling mechanisms are generally designed to meet various objectives, including:

- Maximizing system throughput
    - get the most work out of the system
    - keep as many system components productively executing as possible
    - schedule to get maximum cache utilization (affinity considerations)
- Minimizing execution latency
    - when an important thread wants to run, get it from the Ready state to the Run state quickly
- Providing predictability
    - limiting the variance in dispatching to promote fairness

A scheduling policy is a specific set of constraints that trades off one of these objectives to optimize for another. Most of the contemporary operating systems in use today divide the scheduling arena into two basic policies:

- Timesharing
    - generally focused on providing good response time and reasonable throughput
    - providing fairness by **adjusting thread priorities** based on thread behavior
    - providing predictability by limiting the amount of time a thread can hold a CPU before it's required to yield the CPU to a **Ready** peer
- Real Time
    - generally focused on minimizing latency
    - fairness is typically not of high importance, and real time thread priorities are generally **NOT** adjusted based on their behavior.
    - dispatching variance may be high with real time threads, since one thread may hold onto a CPU for a long time before yielding to a peer

Each thread is created with a scheduling **policy** and a scheduling **priority** within that policy. If the policy is timesharing, threads usually have a small range of relatively low priorities, and dispatching is done by selecting the highest priority thread in the **Ready** state to be the next thread to move to the **Run** state. Timesharing is also characterized by assigning a dispatched thread a **time-slice,** or **quantum** of execution time. When a thread in the **Run** state exhausts its **time-slice** it may have to yield the CPU to a peer, and wait for another turn back in the **Ready** state. One of the most common timesharing policies is known as Preemptive Highest Priority First/Round Robin (**PHPF/RR**), and operates with the following constraints:

- whenever a thread moves to the **Ready** state, its priority is checked against the currently executing thread
- if the newly arriving thread has a **higher priority** than the currently executing thread, then the currently executing thread will be **pre-empted,** and forced to **context switch** with the higher priority **Ready** thread
- when the currently executing thread has **exhausted its quantum**, it must look to the threads in the **Ready** state to see if there is one with the **same priority** as itself. If a peer exists, the running thread will be preempted back to the **Ready** state and its peer

will be moved to the **Run** state (**round robin** peer rotation). This forces threads with the same priority to share the CPU among themselves
- threads that are not getting much CPU time will **gradually have their priorities adjusted** upwards (**aging**), to help them compete with higher priority threads
- higher priority threads that are getting a lot of CPU time, will gradually have their priorities adjusted **downward** in an effort to provide more sharing (fairness)

In contrast to timesharing policies, consider the real time policy called **Real Time FIFO** (one of the policies available on Linux). This policy provides priorities that can be **much higher** than a timesharing priority, thereby assuring that when a thread with this policy is moved to the **Ready** state it will soon be running on a CPU since its high priority will force a preemption event with some lower priority timesharing thread currently using the CPU. Once a Real Time FIFO thread reaches a CPU, it will **hold** that CPU until it either finishes, blocks or is preempted by an even higher priority thread. It has no **timeslice** to force it to share with peers, and its priority is **NEVER** adjusted regardless of its behavior.

On single CPU systems, we generally consider **policy and priority** as the major factors driving scheduling decisions. We can envision a single **Ready** queue that's sorted by priority, and each time a **dispatch** operation must happen (each context switch), we simply select the thread at the head of this priority sorted ready queue and move it into the **Run** state. When we have a system with more than one CPU however, we must consider upon which CPU a thread **last executed**, and try and make sure that we run it on **this same CPU** again when it's next dispatched. When a thread runs on a given CPU, **it accumulates a cache footprint** there, and that footprint can provide a big **performance boost** if the thread runs in the presence of this footprint the next time it is dispatched. Giving preference to a specific CPU when we run a thread for a second, third, etc. time is called **affinity scheduling**, and is a growing consideration with most operating systems as the number of CPUs per system continues to increase.

Week #4 Summary:

Because operating systems include code and shared global data that are used and manipulated by all of the threads on a platform, there is a critical need for **synchronizing** access to these shared global data. If two threads are concurrently running the same code on separate CPUs, and that code manipulates shared data, the data may become **corrupted** if it is not treated **atomically** by each thread. Atomic execution can be assured by using an appropriate **synchronization primitive** such as a spin lock or semaphore

Spin locks are the lowest level primitive, and all higher level synchronization mechanisms (such as semaphores), use spin locks at the base of their implementation. Spin locks are ideal when there is little contention for the lock, and when no thread holds the lock for more than a very short time. If a thread needs to wait for a long period of time, **a blocking mechanism like a semaphore** is sensible. Semaphores may be used as simple locks (a binary semaphore), or as more elaborate multiple instance resource managers (counting semaphores).

Ring buffer sharing between a **producer** and **consumer** threads, provides a perfect application for counting semaphores. Since ring buffers are used extensively in virtually all operating systems (especially in the device drivers used in an operating system), it is important to understand how they are deployed, and how the semaphores used for keeping track of empty and full slots can avoid **overrun** and **underrun** by the contending threads.

Scheduling of threads to CPUs is done in many different ways from system to system, but we can generally categorize schedulers by their policy and priority range. The two main types of policies that we considered are **Timesharing** and **Real Time**, and it's important to understand the basic differences between them. While timesharing policies tend to put **fairness** high on the desirable attribute list, real time policies tend to focus on **low latency** dispatch and minimal system intervention (no dynamic priority adjustments).

OPERATING SYSTEMS FOUNDATIONS
MSIT 5170

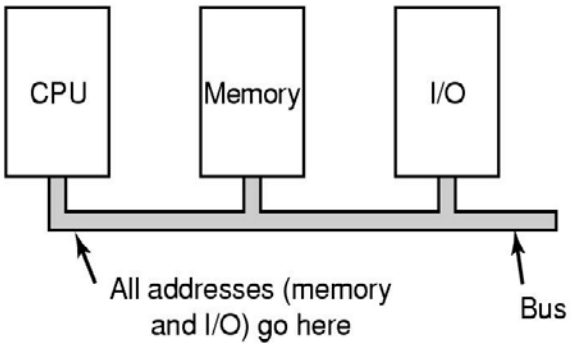
University of Massachusetts Lowell
Department of Computer Science
Spring Semester 2020

Week #6 Lesson

1. The material in this lesson follows the reading assignment for ch 3.1 through ch 3.4 from the Tanenbaum text
2. Please read the text before you read through this lesson.
3. The material in chapter 3 will focus on issues of memory management and address space.
4. We continue to look at system level tools, and I encourage students to experiment with these tools. We will also build some simple applications in the Linux environment to explore some of the dimensions of memory management.

The Need to Manage System Memory:

In our most fundamental view of a Von Neumann style computing system, we envision a processor (CPU), a working memory of some size (RAM), some sort of interface to the outside world (an I/O bridge), and a system bus to tie these basic components together.



Manipulation of data requires that the data be first brought into the CPU (registers) for processing. This is typically done by loading from memory (RAM) or directly from an IO device (like a UART register). Computed results are then generally moved from the CPU (registers) back into memory where they can be accessed either by the CPU again (for further processing) or sent to the outside world through the I/O interface. The machine instructions that a thread will execute to manipulate data must themselves be first moved into memory from the outside world (usually from an executable binary kept somewhere in a file system that is accessed via the I/O interface). Once in memory, the individual machine instructions are brought into the CPU during a fetch cycle, to be executed by the CPU's functional units. We see the CPU during

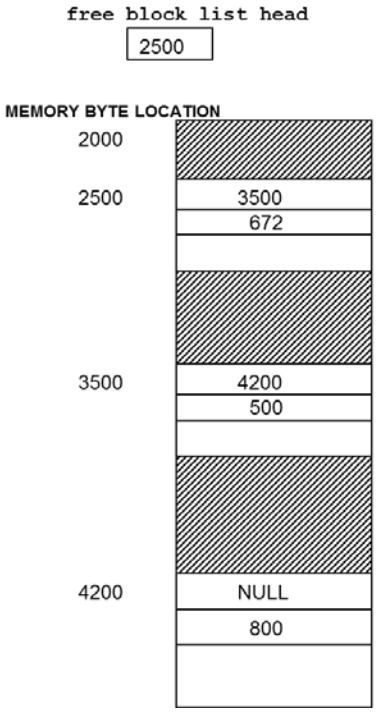
normal execution constantly performing load and store operations to move data objects between its registers and memory (or IO devices), and constantly performing fetch operations to move machine instructions from memory into the CPU instruction interpretation register (IR). Clearly, memory is a central and critical resource, and must be managed carefully in order to extract peak performance from a given system.

An operating system is generally concerned with three basic aspects of memory management:

- **When** to bring something into physical RAM
  - should entire programs be loaded into RAM when we want to run them, or should we only bring in what's demanded at any given time
- **Where** should we place something into physical memory
  - placement is critical to using the available space most efficiently
  - good placement is intended to limit fragmentation
- When and **what should be evicted** to make room for incoming code and data
  - removing things that have been "used up" and will not be demanded again for the life of an application is the goal
  - how can we know what has been "used up" if the owning application is still alive?
  - if we choose poorly, we may remove something that will be needed again soon, and will then have to evict something else to recover something we already had in memory once

Managing Free Space:

Placing objects into memory requires some strategy for knowing what parts of memory are available (free) to load into. Several strategies for managing free memory are discussed in the book, including **bit maps** (which are used extensively to manage free disk space in a file system, but not very often to manage RAM) and a number of variations of **linked list management** (linked lists being frequently used to manage RAM in many simple embedded systems). Embedded linked lists are very space efficient, since their control data (meta data) components are kept in the free (otherwise un-assigned) parts of memory.



In the picture shown above, a region of memory beginning at byte location 2000 is being managed by a linked list implementation. There is a head pointer with a value of 2500, indicating that the memory from 2000 to 2499 is in use, but that there is a block of free memory beginning at 2500 that is 672 bytes long. The linked list header element located at 2500 holds the address of the next free block (3500), along with the size of this free block (672 bytes). The rest of this block is free unused memory. Shaded areas have been allocated (as we mentioned for the space from 2000 to 2499), and the next free block is at 3500. The linked list header element located here identifies this block as being 500 bytes long, and points to the next free block at location 4200. The block at 4200 is the last free block in the managed memory area, spanning some 800 byes (its next pointer of NULL indicates the end of the list).

Using Linked Lists:

There are three basic algorithms used to find a free block in a linked list managed memory area:

- **First Fit**
  - knowing what size is required, a search begins at the head of the list and continues until the first block that is big enough to satisfy the allocation request is found
  - the starting address of this area will be returned to the caller, and unless the block is a perfect fit, the residual space (external fragment) will have a new header built into it and will be linked into the rest of the list
  - for example, if a request for 300 bytes was made from the system shown above:
    - the first free block is used since it is big enough
    - the returned address will be 2500
    - the space at 2500 + 300 = 2800 will have a new header created
    - the next pointer will still be 3500, but size field will now be 672 - 300 = 372
    - the head pointer will have to change to point to this new external fragment at 2800

- **Best Fit**
  - the search begins at the head of the list, but does not stop until it has checked each element on the list to see which element is closest in size to the request
  - the objective is to leave the **smallest external fragment possible**, hoping that this strategy will be more space efficient than first fit (i.e. for a given initial free area, will be able to satisfy more allocation requests before it runs out of memory)
  - using the same example, if a request for 300 bytes was made from the system shown above:
    - the free block containing 500 bytes will be selected, since it will leave the smallest external fragment
    - the returned address will be 3500
    - the space at 3500 + 300 = 3800 will have a new header created
    - the next pointer will still be 4200, but size field will now be 500 - 300 = 200
    - the previous block at 2500 will have its next pointer now set to 3800
- **Worst Fit**
  - the idea here is to leave the **largest external fragment**, assuming its large size will be usable to satisfy subsequent requests
  - as with best fit, the search must examine all elements before a choice can be made, and the hope is that this strategy will be more space efficient than the others (i.e. for a given initial free area, will be able to satisfy more allocation requests before it runs out of memory)
  - using the same example, if a request for 300 bytes was made from the system shown above:
    - the free block containing 800 bytes will be selected, since it will leave the largest external fragment
    - the returned address will be 4200
    - the space at 4200 + 300 = 4500 will have a new header created
    - the next pointer will still be NULL, but size field will now be 800 - 300 = 500
    - the previous block at 3500 will have its next pointer now set to 4500

Managing the list is further complicated by the need to deal with space that is returned by a user when the user has finished with it. Free space must be put back into the list, but it must be **coalesced** if appropriate with free neighbors above it or below it, or both.

Research has shown that **first fit** is, on average, as space efficient as either of the others, but that its average run time is half of the others, so first fit is most often the choice for linked list management.

Linked list mechanisms are appropriate when allocations are allowed for any arbitrary sized chunk of memory extracted from some contiguous range of free memory, in a fashion similar to the way in which the **malloc()** dynamic memory allocator routine works in a **C program** to manage the so-called heap space. This view of a contiguous range of free space, however, would be difficult to maintain if we were operating directly against physical memory. We would have to break up physical memory into varying sized chunks, which would cause a fragmentation nightmare. Since we need to project the flat, contiguous memory model (address space) to a program, and the process it runs in, we need a **low-level strategy** to deal with physical RAM efficiently. This underlying model provided by an operating system is called the **virtual memory model**.

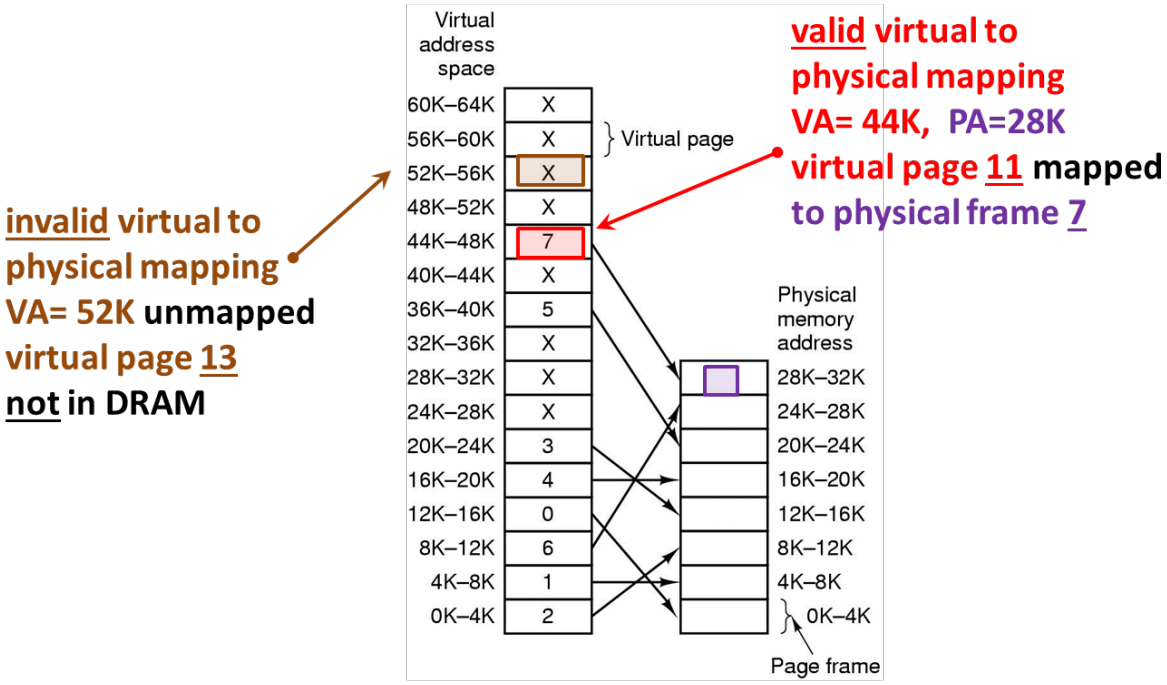
Virtual Memory:

While most contemporary processors view RAM as a contiguous array of byte addressable locations at their lowest level of operation, just about everything except the simplest CPUs support memory management functionality to model RAM as a collection of **fixed sized pages** for higher levels of operation. This paging model is built around a physical memory allocation policy of a fixed sized contiguous chunk of RAM called a **page**. In paged systems, physical RAM is always allocated in this fixed sized page unit.

The Intel x86 family of processors, for example, do not power-up in paged mode, but once they load and begin running an operating system like Windows or Linux, these operating systems quickly enable paged mode, and remain in paged mode as long as the system continues to run. Page sizes vary from processor to processor, and are adjustable in some processors. **The x86 family uses a 4 KiB** (2^{12} = 4096 byte) page size (along with some larger choices like 2 MiB and 1 GiB), while the HP Alpha uses an 8 KiB page and certain ARM processors can vary their page size from 1 KiB to 64 KiB. The obvious question to ask at this point is "how do we satisfy a request from a running thread for say 1 MiB of memory, if the operating system can only allocate 4 KiB at a time". The operating system could try and find enough 4 KiB free pages in contiguous order to make the allocation, but this may be impossible if different groups of pages have been given to other requests and no contiguous run of 1 MiB remains. There may be much more than 1 MiB of memory available, but not in contiguous order.

To solve this problem, operating systems have adopted a high level technique of memory management called **virtual memory management**. As we've seen in chapter 2, each process in a Linux or Windows system is provided a private address space. Now we can see that this address space is not immediately carved out of physical RAM, but is instead, an **abstraction** in the form of a virtual address space. The virtual address space represents a **contiguous region of byte addressable storage** to any thread running in it, but the operating system views it as a collection of virtual pages, any one of which can only be used if and when that virtual page is **connected to a physical RAM page**. Connecting a virtual page to an actual physical RAM page is called a **mapping operation**, and typically occurs to resolve what is known as a **page fault**. Page faults occur when a running thread attempts to reference a virtual address that has no physical mapping. The CPU hardware reacts to a memory reference (to fetch, load or store) that is not physically mapped by raising an **exception**. The exception handler, which is part of the resident operating system, then reacts to this event by determining if the memory reference is to a valid object, and if so, will find a **free physical page,** and place the appropriate content into this page (i.e. the exception handler loads needed code or data from the executable file for example), and then **completes the mapping of the virtual to physical address** to allow a retry of the faulting memory reference instruction. The faulting address will, of course, be a part of some page, and the page fault mechanism will handle these events one page at a time. The newly mapped page will resolve the current fault, and because of the **locality** attributes of programs, will hopefully serve to fulfill other references in near proximity.

The mapping of virtual pages to physical pages is managed for each process using a process specific data structure called a **page table**. The operating system maintains a **discrete page table** for each process in the system, and this table is a key component of the process address space we've previously discussed. The book provides a simple example of a page table mapping a collection of virtual pages to physical pages (page frames are 4 KiB):



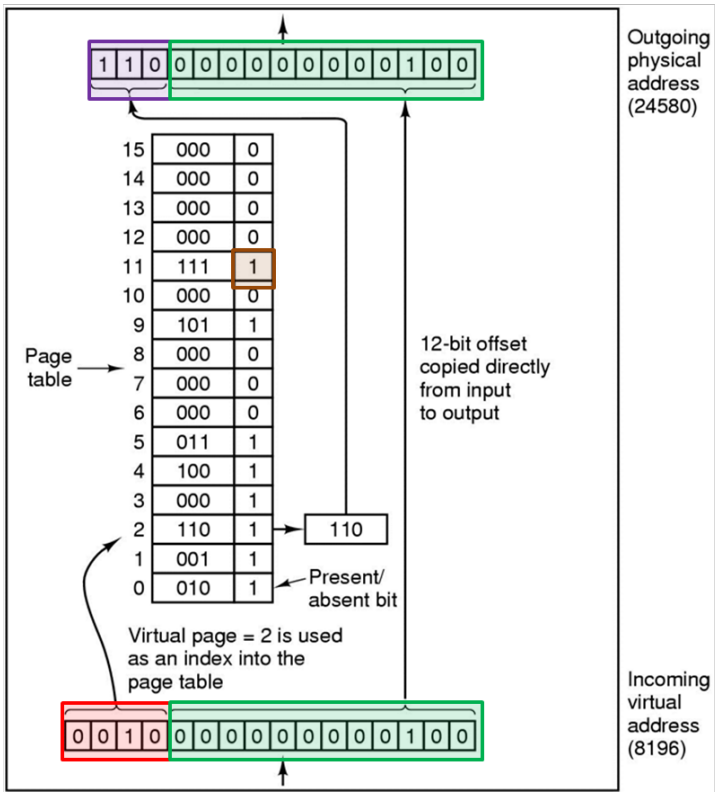
A Page Table is an array of Page Table Entries (PTEs) that either map a virtual page to a physical page if a mapping exits, or indicate that no current mapping exits

The virtual address space shown above consists of **16 virtual pages**, with a 4 KiB page size. Physical memory, as shown here, has only **8 physical pages**. The entire virtual address space could never fit into the physical address space at the same time in this example, but we can map as many as 8 virtual pages at any given time. The operating system will do its best to make sure that the **"best" 8 pages** are mapped at any time, meaning the 8 pages that a running thread will need to reference the most during a given time interval. Since each physical frame shown is currently mapped, if a thread running against the given virtual address space suddenly make a reference to an address in the range of 28 to 32 K (this range is shown un-mapped), a page fault would occur, and the operating system would have to select an existing mapping to evict, such that the freed frame could be used to satisfy the **current fault**.

Choosing a victim to **evict** when a page fault occurs against a full physical memory is one of the most critical parts of page fault management. If we evict a page that will be needed again soon, we will have wasted a lot of effort in removing the page, only to have to fault it right back in again. We need to select victims that are unlikely to be needed again to achieve optimal performance. Section 3.4 from the text examines some replacement algorithms.

The page table entries (PTEs) shown above include either a physical page number, if the virtual page is mapped, or the letter X, if the virtual page is not mapped. In a real system, PTEs normally include enough bits to point to any existing physical page, along with some **additional bits** to indicate a valid mapping and the **type of access permitted** to the target physical page. The number of additional bits is dictated by a given CPU, but there will always be at least one bit to indicate if a mapping is **valid** or not. This bit is called the valid (V) or present (P) bit, and may be accompanied by other bits that describe how the mapping can be used (read page only, read/write page, cache parts of page, page modified state, etc.). For the simple system shown above, the bit level view of the page table would look like:

An example of a **16 virtual page** address space mapped into an **8 frame physical DRAM space**. PTEs have a **"present or valid"** bit to show **map state**. The 12 bit **"offset"** portion of an address points to a byte on a **4KB** page/frame

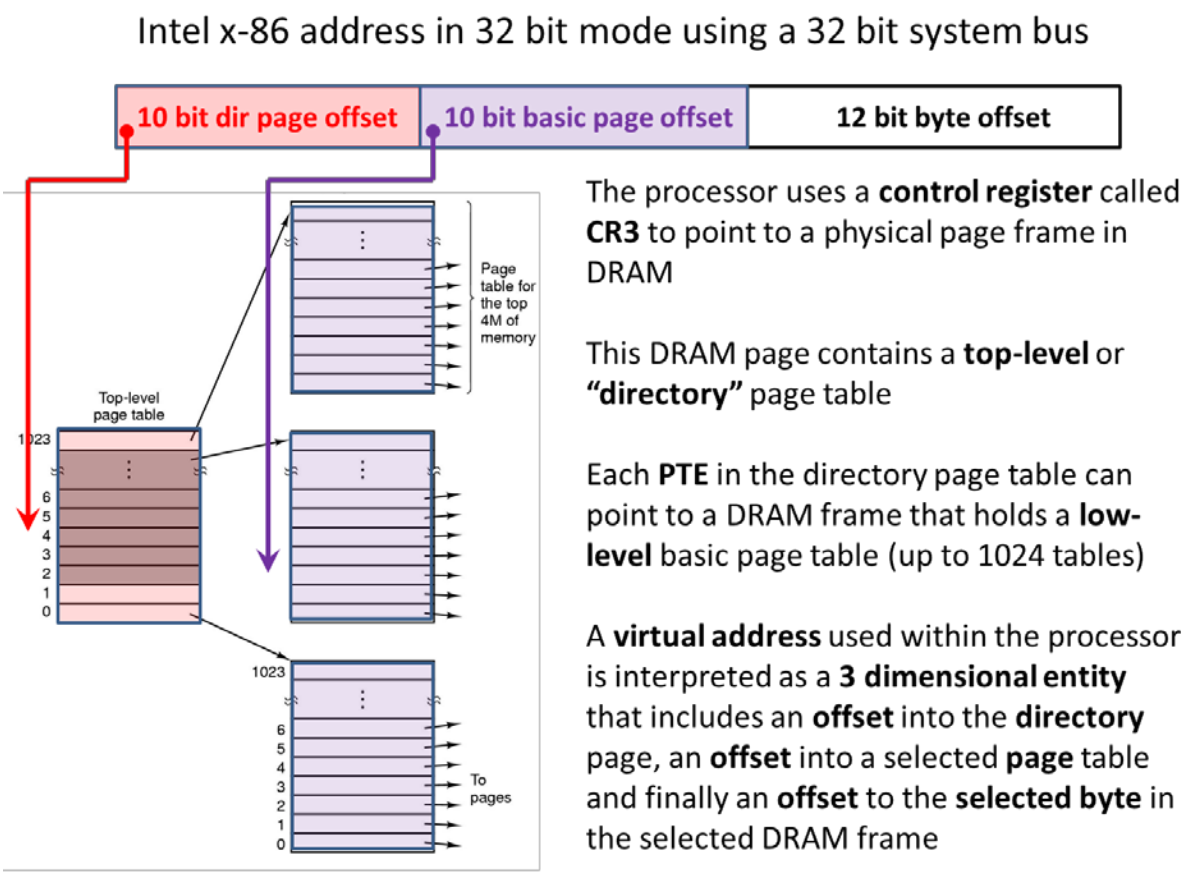


Notice how the incoming 16 bit virtual address of 8196_{10} ($0010\ 000000000100_2$) is **translated**. The full virtual address space discussed here is 2^{16} = 64 KiB (16 pages * 4 KB/page), so the given 16 bit virtual address is broken into a page component (the most significant 4 bits point to a virtual page, since 2^4 = 16) and a byte offset (the next 12 bits point to some byte location on a 4 KiB page, since 2^{12} = 4096, AKA 4K). The page bits bring us to a PTE (0010 send us to slot 2 in this case), and from there we can trade our virtual page bits (0010) for the corresponding physical page bits (here 110). The trade is allowed, because the **P** (present) bit is set to 1 (TRUE). If our inbound page bits were set to say 0111 (slot 7), we could not have made the trade for virtual to physical page bits because slot 7 has the **P** bit set to 0 (FALSE), and this would have resulted in an exception to the CPU (a page fault). This translation involves changing the original 16 bit virtual address into a 15 bit physical address (because physical memory is only 2^3 = 8 pages * 2^{12} = 4 KiB/page = 2^{15} = 32 KiB).

In general, the translation of a virtual to physical address involves **replacing the virtual page bits** in the inbound address with the appropriate **physical page bits** in the outbound address. The byte offset portion of the address just gets carried along, since the byte location is the same whether we consider the page to be virtual or physical. In some cases, a virtual address may be smaller than a physical address (e.g. the x86 architecture in 32 bit mode only allows a virtual address space per process of 2^{32} or 4 GiB, but the processor's physical bus supports at least 2^{36} or 64 GiB of physical real estate). In other cases, the virtual address space may be (much)

larger than the physical address space (e.g. the x86 architecture in 64 bit mode allows a virtual address space per process of 2^{48} or 256 TiB, but the largest physical bus presently available supports 2^{46} or 64 TiB of physical real estate).

Virtual memory translation requires a **page table per process** (address space), as we've seen. The larger the virtual address space, the larger these tables must become. Considering the x86 architecture in 32 bit mode, the 4 GiB virtual address space that the processor supports requires 1M (2^{20}) page table entries for full mapping (i.e. 2^{20} (1 M PTEs) * 2^{12} (4 KiB/page) = 2^{32}, or 4GiB). In this case, each PTE is 4 bytes long, meaning that the full mapping page table for just **one process address space** would be **4 MiB big**, and we know that we need such a map for each process that is alive in a system like Windows or Linux. It's not practical to require such a large table to be kept **resident** in RAM for each process (the problem is much worse for 64 bit systems as you can envision), so page table structures are generally **broken into two or more hierarchically related tables**, only the highest level of which is required to be memory resident at all times. The x86 processor has several special registers that manage memory, one of which is called **CR3** (control register #3). The CR3 register is one of the few registers in the CPU that actually has a **physical address** in it (once the operating system has started, all addresses used in the rest of the CPU for normal computations are virtual). The address in CR3 always points to a **physical 4 KiB page** that represents the **top level** (directory level) of the page table hierarchy of whatever process is currently using that CPU (remember, there must be a thread on every CPU running at all times, and every thread belongs to some process, and runs within that process address space). When the running thread presents the CPU with a reference to a virtual address (to fetch an instruction or load/store a piece of data), the CPU must translate that virtual address into it corresponding physical address. It does this by using **CR3** to find the **directory level page table**, and then locates a **PTE** in this table by using the most significant 10 bits of the given virtual address (since the table is in a 4 KiB page and each entry is 4 bytes, there are 4096/4 = 1024 = 2^{10} = 1K entries, and 10 bits can index 1024 entries). An entry in this directory level page, if valid, will contain the **physical address of a 4 KiB page acting as a low-level page table.** A low-level page table also has 1024 PTEs, so this table is indexed by the **next 10 bits** in the virtual address. The selected entry in this table, if valid, will have the **actual physical address** of the page that holds the referenced instruction or piece of data, and completes virtual to physical translation. This can be seen for the x86 32 bit deployment in the diagram below:



Notice that each entry in the top-level table can point to a physical page with 1024 entries to target physical pages. This two level strategy provides 1024 * 1024 PTEs, the required 1 M entries. The x86 CPU requires a top-level (directory) table to be **resident** in memory for **each living process** in the system, and each time that a **heavy weight context switch** occurs on a CPU, the CR3 register is reset to point to the directory page of the inbound thread's process directory table. Only the directory table is required to be memory resident, the low-level page table pages are only paged into memory **when they are needed**, lessening the burden of page tables on the physical memory system.

Page Replacement Algorithms:

Virtual memory implementations allow an application to see a contiguous range of (virtual) memory addresses, but allow the system to locate any one of the needed virtual pages wherever in memory there is an available free physical page. If an application needs 512 MiB of contiguous memory space to build a data structure in, it can get that space using a malloc() request. The malloc() routine returns a starting address for this region to the calling application, and the application is **assured** of having this **contiguous** space available. As the application references different parts of this address chunk, the operating system will **find physical pages** to map to the referenced virtual addresses. The connected physical pages will have **little or no contiguity** among one another, but the application will see its space as a contiguous range of addresses as required. Virtual memory systems allow us to use physical memory very efficiently, by **eliminating the need for allocating physically contiguous memory ranges**.

We can imagine that at boot time, the operating system has a large number of physical memory frames (4 KiB pages) that are not immediately in use (**the free list of pages**). As applications begin to run and **fault** pages into their address spaces, the **free list dwindles**. Since page faults happen frequently in a busy system, the operating system must keep a **supply of free pages** in order to handle faults when they occur. If the operating system detects that the size of the free page list is getting **too small,** it will run a thread called the **page-purger**, that will begin to **take pages away** from various processes that have previously faulted these pages into physical frames. This is called **page replacement**, and requires the page-purger to **decide** what pages to take away from which processes. In general, we would like the purger to **reclaim pages** that are **unlikely to be needed** any further by the processes that own them. Removing a page from a process address space involves finding the page table mapping for the target page and updating that entry to be **invalid** (turn off the valid or present bit). If the selected page frame is **dirty** (has been modified since the owning process first mapped it), then that page will have to be **backed up to disk somewhere,** so if the owning process ever needs it again we can retrieve its content.

Several algorithms are discussed in the text, but in practice, page replacement algorithms tend to be various approximations of the **Least Recently Used** (LRU) strategy. The **LRU** strategy attempts to keep a small amount of information about each physical page frame, regarding when it was **last referenced** (used). The assumption is that if a page has not been referenced for a long time, the owner of that page has probably finished using it and has moved on to other parts of its program, thus making its **reclamation efficient**. If, on the other hand, the owner of the reclaimed page demands it back right after we've taken it away, then our replacement algorithm will **not be very efficient**.

Whatever algorithm is deployed in a given operating system, you can see that what we need to make page replacement efficient is:

- an algorithm that can **quickly** make a decision about what page to reclaim
- an algorithm that **limits the overall system page fault rate** by making **"good"** decisions about what page to reclaim.

Evaluating the efficacy of a page replacement algorithm is often done by using a post analysis on a particular workload with the **Optimal Replacement Algorithm** (OPT). The page reference pattern of a particular workload (benchmark) is collected on an instrumented system. This data consists of recording every memory reference that occurs during the execution of the benchmark work load. The data are then analyzed with the **OPT** algorithm, which determines what the **minimum number of page faults** for this workload could possibly be for a **given sized physical memory**. OPT provides a lower bound on page faults for the benchmark, but it is a post analysis tool, and is of no value to a running system. Once the benchmark data has been analyzed, the benchmark is re-run under an operating system that is attempting to evaluate its page replacement algorithm. The number of faults in this real system is collected, and compared to the OPT result. For example, two different replacement algorithms might yield 230,000,000 and 210,000,000 faults respectively. The second would appear superior to the first, but if OPT analysis shows a lower bound of 90,000,000, then neither algorithm is really any good, and it's back to the drawing board.

We will consider some additional aspects of paging systems in the next lesson, but keep in mind that this paging mechanism is used by all but the simplest embedded operating systems to enable efficient memory management. When systems underperform, thrash or cause endless delays to users, it's very likely that system memory management is the underlying problem. Contemporary virtual memory implementations can utilize memory very efficiently, but even the

most efficient management of a resource will falter when the load simply outstrips the capacity. Adding more physical memory to a system is almost always a remedy for performance woes.

Week #6 Summary:

We think of memory as one of the critical resources of a computing system. A CPU cannot execute an instruction or manipulate a piece of data unless that object has first found its way into some kind of physical memory.

In contemporary systems, memory management occurs at different levels of abstraction, beginning with a view of memory as a contiguous range of byte sized storage elements. When presented with this view of memory, we need to consider management strategies that deal with the allocation and accounting of variable sized chunks of contiguous memory to be allocated on request. We looked at several **linked list approaches** to this type of allocation, including **first-fit, best-fit and worst-fit**, considering implementation and performance aspects of both allocating and then freeing contiguous memory.

Managing physical memory with contiguous allocation policies, however, leads to serious **fragmentation** issues, so this level of management needs to be layered on top of a more efficient mechanism. **Virtual memory** provides a solution by giving each process the illusion of a contiguous address space to work in, but that space is virtual, and requires a lower level management technique to work efficiently.

Below the virtual address space, is a **paging system** that manages physical memory in **fixed sized allocation units called pages.** When a running thread references a part of its address space that is not currently realized as physical, a **page fault exception** is delivered to the operating system, which then connects the virtual page that includes the referenced address to a physical page frame and allows the reference to proceed.

Because free pages are in constant demand on a busy system that is experiencing many page faults per second, the operating system maintains a thread called the **page-purger.** The purger is responsible for running the **page replacement algorithm** of the system, during which it **reclaims pages** from existing processes to add to the free list. The replacement algorithm is a critical element of overall system performance. A good replacement implementation will keep the system page fault rate as close to the theoretical minimum as possible for the least amount of execution time possible. In other words, it will make **good decisions** about what pages to reclaim, and will make those decisions **very quickly**.

OPERATING SYSTEMS FOUNDATIONS
MSIT 5170

University of Massachusetts Lowell
Department of Computer Science
Spring Semester 2020

Week #7 Lesson

1. The material in this lesson follows the reading assignment for ch 3.5 through the end of chapter 3 from the Tanenbaum text
2. Please read the text before you read through this lesson.
3. The material in chapter 3 will focus on issues of memory management and address space.
4. We continue to look at system level tools, and I encourage students to experiment with these tools. We will also build some simple applications in the Linux environment to explore some of the dimensions of memory management.
5. During this week you must complete the "First Half Exam", as described in the weekly agenda. The exam will be available to take as of Wednesday morning, March 4, but I advise you to wait at least until the Wednesday chat session is over before you consider taking the exam.

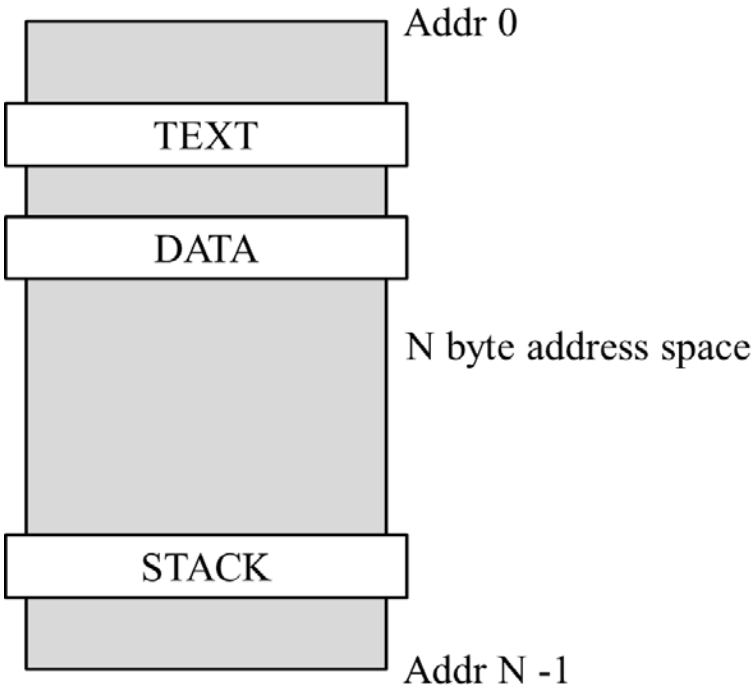
Design Issues for Paging Systems:

Thus far, the material in chapter 3 has considered the need for memory management, discussed some methods for achieving this with a physically contiguous address space, and considered a more practical approach using a contiguous virtual address space with an underlying physical paging mechanism. Some sort of paging mechanism is supported by most CPU implementations, and all but the simplest embedded operating systems typically exploit these hardware resources. Most well known operating systems such as Linux, OSX, Windows, VMS, etc., deploy a virtual, paged memory management implementation.

Page replacement algorithms play a central role in any virtual memory implementation, as we've seen, but there's more to consider on the backup side of replacement. If a page is selected for replacement, and that page has been modified since it was last paged in (the VM hardware support provides a hardware update to a bit in the mapping PTE called the dirty or modified bit, that is set anytime a store operation is performed on the PTE's target physical page), then the page-purger must first write out the modified version of the page to a safe disk location. This is the most expensive part of page replacement, because it requires an IO operation. The page-purger would much rather select a page for replacement that has not been modified since it was last paged in, since such a page can simply be added to the free list without need to write its content anywhere. There are times, however, when most of the likely replacement victims are all dirty, and there is little choice but to page-out modified pages.

To understand the mechanisms for paging out modified pages, we have to revisit our address space model. We know that a process address space must be populated by a minimum of 3 memory object, called the text, data and stack objects. Like any memory object, each of these

objects must occupy a contiguous range of virtual addresses, and are never allowed to overlap in the virtual address space. Recall our earlier address space diagram:



Each memory object in an address space (and there can be many more than 3) has 2 principal characteristics; whether the object is a shared or private object, and whether the object is backed up by the file system or the swap disk. When we designate an object as shared, we mean that we allow such an object to appear in several separate address spaces concurrently. The text object is in this category, meaning that if we have 5 processes in the system all running the bash shell, they will all share the same text object in their private address spaces. For a text object, this is sensible, since the object is usually mapped as read only, and the code within cannot be modified, so it's easy to share. A stack object, on the other hand, is a private object, and we expect that any given stack object can only be mapped into a single address space and used exclusively in just one process. When an object is backed up by the file system, we say it is a file based object, and when it is backed up by the swap disk we say it is an anonymous object. Here are the categorizations of the common memory objects found in both Windows and Linux:

- Text object usually shared, file based object
    - Can be linked private if we want to include any self-modifying code, but such pages can then only be backed up to the swap space, since we are not allowed to modify the original executable file.
- Data object private, anonymous
    - Although initialized global data is initially loaded in from an executable file just like text is, any modified global data must be written to swap if a data page of data is selected for replacement.
- Stack object private, anonymous
    - The entire content of the stack object is dynamically constructed from zeroed out physical pages when a process address space is created at process create time.
- Shared memory object shared, anonymous

    - A shared memory object can be mapped into the address space of many processes, but like a stack object, its initial content is constructed from zeroed out physical pages when the object Is first mapped.
- Memory mapped file object usually shared, file based object
    - An entire file or some range of a file can be directly mapped into an address space to allow file access using simple memory reference instructions. When the object is mapped shared, modified pages of the object are written directly back to the file itself, and multiple processes can share the object concurrently. If the object is mapped privately, then any modified pages of the file that are selected for replacement must be written to swap, since a private mapping cannot modify the original state of the mapped file.

The page-purger has a propensity for selecting private object pages for replacement, since they can only appear in one address space, and if a page from such an object is reclaimed for the free list by the page-purger, only one page table has to be adjusted to reflect the replacement. Private pages are also added to the free list whenever a process terminates, since they cannot be used in any other address space than the one that was just torn down when the owning process terminated. Notice that pages from a shared object cannot be automatically added to the free list when a process that has that object mapped terminates, since the pages may be in use by other processes that also have that shared object mapped into their address spaces.

Thrashing

Even the best page replacement algorithms will not be able to keep a system from thrashing under certain load conditions. Thrashing is a term used to describe a degenerate system that spends all of its CPU and IO cycles processing page–purger operations, with little to no resources applied to actually getting any application work done. Such a system looks very busy, with constant disk activity and context switching, but no thread ever gets to run for more than an instruction or two before it page faults and starts running fault handler code and the page-purger kernel thread. Thrashing occurs because none of the runnable threads on a given system can get enough physical page frames to hold what is known as their current “working set”. The current working set of a given process is the set of physical pages that its threads need during some phase of its operation to avoid page faulting. Consider a thread that is running through an iterative loop of code that spans 5 pages for one million iterations. If all 5 pages are memory resident, then each iteration can run very fast by avoiding any page faults. If the system is so short of pages however, that it can only come up with 4 of the needed 5 pages, than each iteration will lead to a page fault, with one of the 5 necessary pages being swapped out so the fifth page can be brought in to complete an iteration. This would lead to one million extra page faults than would have occurred if we could have just found one more physical frame to give to the thread. There are times, however, when just such a scenario can form in a system, leading to thrashing.

Load Control

Load control mechanisms are programmed into most memory management systems, to both detect and react to thrashing. The operating system collects various statistics about general system behavior, but one of the most important statistics is the rate of execution of the page-purger. The system wide page fault rate is also important, but we really don’t worry about a high fault rate if we have enough free pages to cover the faults. If we see the page-purger running frequently in an effort to reclaim in-use pages to build up the free list to handle page

faults, then we know that we're in trouble. Now we are "robbing Peter to pay Paul", and this leads us into thrashing. We need to either add more physical memory to our system (an expensive and disruptive endeavor), or we need to reduce the number of thread that are competing for page frames. Load control is all about limiting the number of threads that can reach the run state (and thereby fault pages) until the system reduces its page-purger run rate. There are various approaches to this, but they broadly fall into two camps. The swap camp uses an approach of selecting a process to "swap out" and then collects all of the currently paged in physical pages of the process and writes them out to disk. This collection of physical pages can now be distributed among the remaining processes to attempt to resolve the current thrashing situation. The swapped out process can be recovered some time later when there enough free pages available in the system to page the swapped process back into physical memory and resume their execution without thrashing. Selecting a process to swap out is based on different factors, not the least of which is the number of physical pages that we reclaim by giving a specific process the boot. The second approach (the one used by Linux) os to locate a process that has a large number of physical pages currently paged in and send that process a termination signal. When one of the threads in the process sees the termination signal, it is forced to leave the run state by the exit path and bring down the entire process with it. The pages of the now deceased process can then be put on the free list in the hopes of eliminating the thrashing.

There are several different topics discussed in the remainder of chapter 3 that you will visit as you read your way through, but the material included above and in the previous lesson is what we're primarily interested in and what I will test over in the first exam. Virtually every question on the first half exam will come from either some part of the notes that you've been looking at across these first 7 lessons, or some part of the assignment work you've been doing for the first 5 assignments, so these weekly notes should be your first resource in studying for the exam, followed by a review of the weekly assignments.