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:

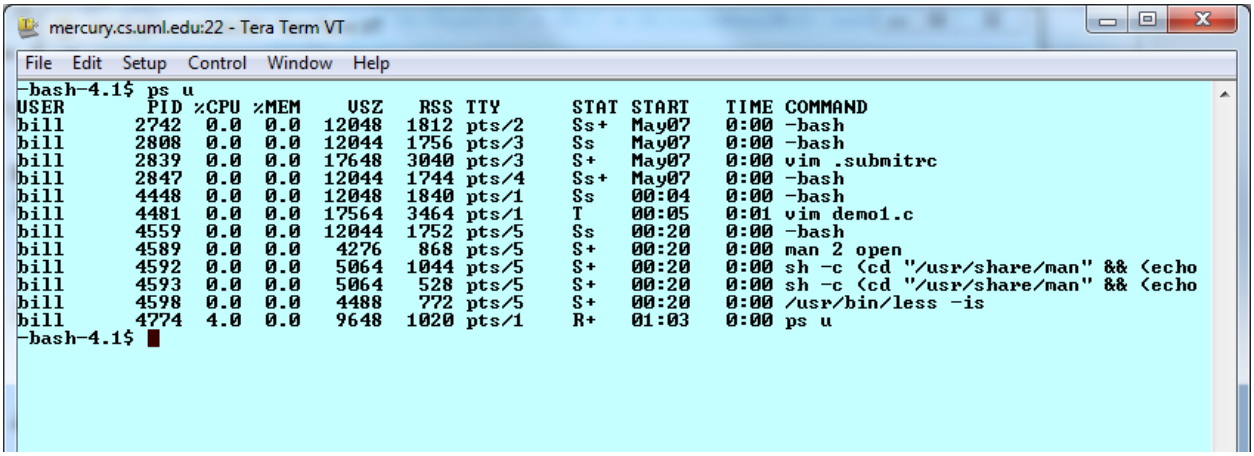
Process Attributes:

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.

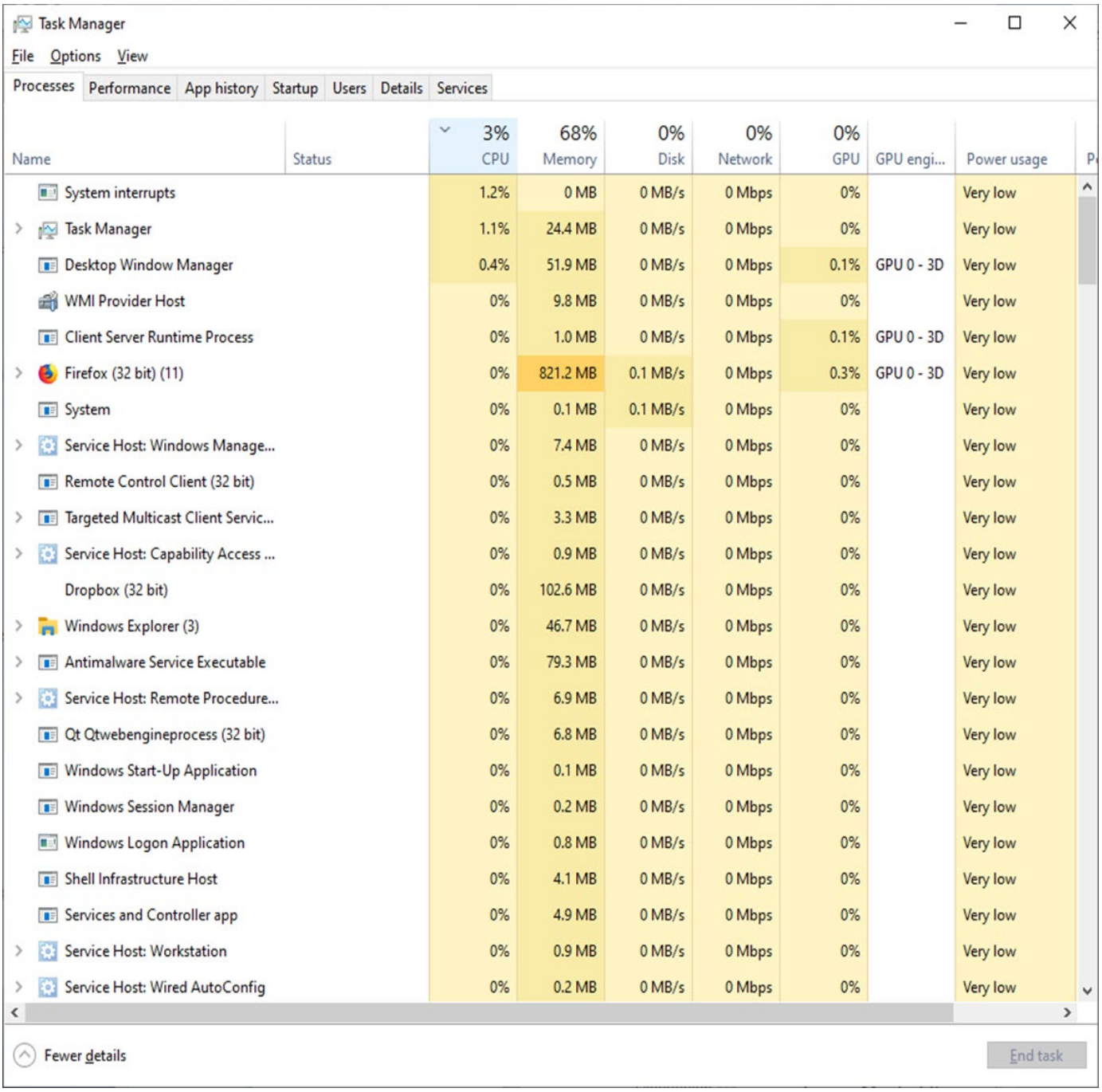
- A PID
    - A process has an identity that is normally expressed as a Process ID (PID). Each process in any of the systems mentioned above will have a unique PID during its lifetime. PIDs may be recycled, but at any given time each PID is unique.
- An Address Space

    - 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:

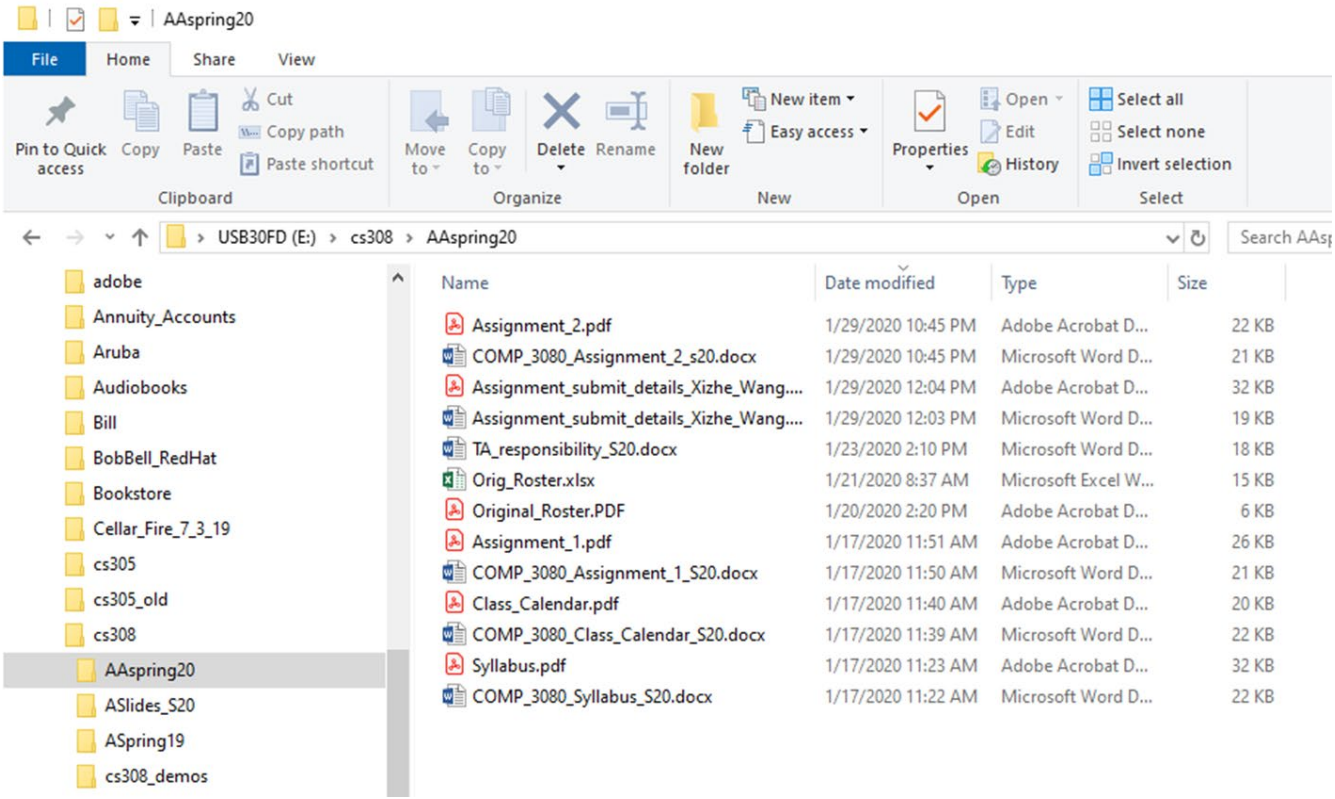
Last week we looked at some of the hardware components that make up a typical computing system. To complete the picture we must also look at some of the software components. A major software component is that of the process, described above, but we must also consider file objects with similar emphasis. 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.

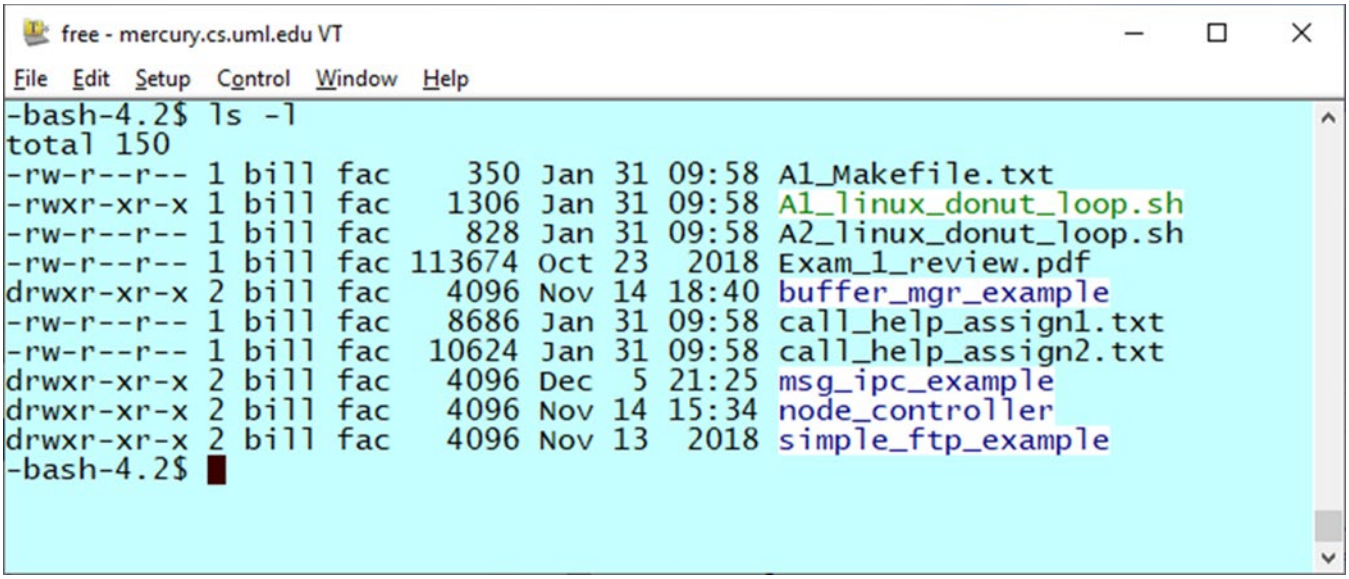
File Attributes:

- 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.

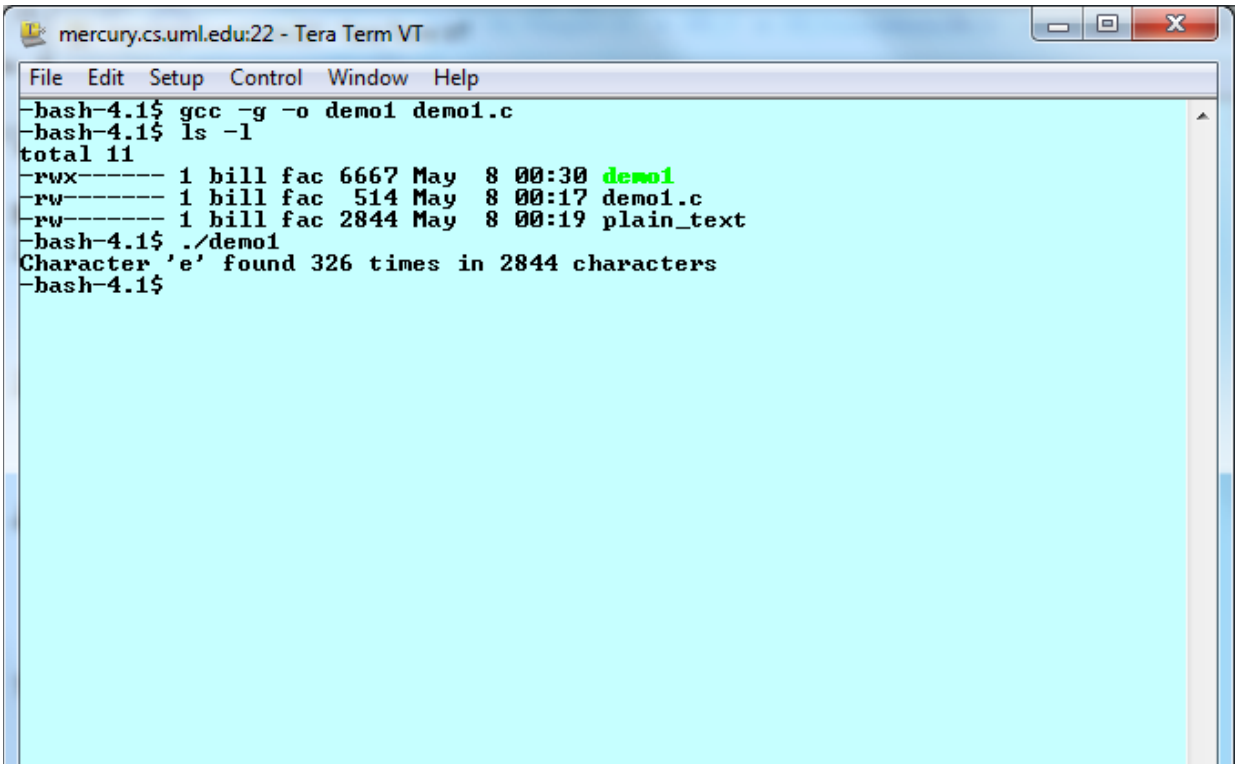
A C Program to Demonstrate:

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.