CS 111 - LECTURE 1 - APRIL 4, 2005
PROF: KOHLER, SP05

Operating System Principles!

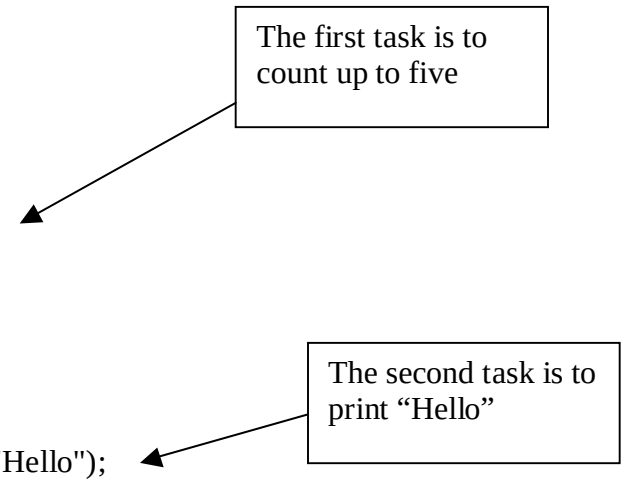
What it an operating system?

One may suggest that an operating system is a program that constantly runs on the computer. Although a general feature, this is not a defining characteristic of an operating system. There are many processes besides the operating system, such as iexplore, that may constantly run on the computer. It is also important to note that the earlier day computers did not require an operating system. In other words, operating systems are not essential to the functionality of computers, but as it will become more clear, they have become a critical part of modern day computers and facilitate many important operations.

As an alternative, an operating system may be defined as a program that enables the user to run multiple tasks simultaneously on the computer. Once again, this is not an exclusive characteristic of operating systems. Note the following example:

EX: The following example carries out two tasks simultaneously.

```
int main (int c, char *v[ ])
{
        int state = 0;
        int x=0;
        int printed =0;
        while (1)
        {
                if (states % 2 == 0)
                {
                        if (x <5)
                        x++;
                }else {
                        if (!printed)
                        {
                                printf ("Hello");
                                printed = 1;
                        }
                }
                state++;
        }
}
<end sample code>
```



The sample code above is an example of implementation of *cooperative multithreading*: carrying out more than one task simultaneously.

Another proposed definition may describe the operating system as an *abstract* system environment or a program that creates an interface between the hardware and many other software applications. Accessing data on a disk is an example that justifies the need for such a program. The disk can be considered as a linear array of bytes, and without the use of an operating system, applications may directly address an array cell location to read a certain byte of data, e.g. the 4000^{th} byte. This operation can certainly be simplified if a program (such as the operating system) would "abstract" the hardware (e.g. the disk) into a more usable form (e.g. …/work/temp). However, although interfacing with the hardware is indeed an essential role of operating systems, the given definition is still far from including all critical aspects of an operating system. These features are listed below:

The most fundamental characteristics of an operating system:

* Protecting hardware access

    In a computer with an operating system, only the operating system has the privilege to access the hardware directly; all other applications must access the hardware through the operating system. This is the most important functionality of an operating system as incorrect handling of hardware access may have catastrophic consequences. For example, the "HCF" (halt and catch fire!) opcode in IBM360 could cause the processor to toggle a subset of the bus lines as rapidly as it could; this could even cause the system to catch fire (HCF is not the actual instruction name, just a humorous reference to such instructions). One can imagine how improper handling of hardware by the operating system may have truly undesired consequences. This may be an extreme case, yet it demonstrates the importance of the operating system functionality in managing hardware access. *

* Synchronizing hardware access.

    In order to allow more than one process to run simultaneously on a computer (and access hardware while doing so), it is crucial to maintain the correct state of the architecture. For example, two processes may attempt to update a single register, or one may attempt to read a register content before another has finished updating its value. Keeping track of such interactions among hardware and multiple applications is another important role of operating systems in modern computers.

* Allowing safe sharing of resources

    Computer resources may be in demand by more than one process, and their sharing needs to be regulated and controlled based on availability and/or other factors. Unrestricted access to computer resources in such conditions is simply not an option, as the operating system needs to create a balance between demand and availability of resources while application are being executed.

GOALS of this course:
* understanding abstraction and interfaces
    - how to choose them
    - how to use them (performances, ease, safety)
    - how to improvise
    - what's under the hood.
* understanding protection, management (lower level), and policy (higher level)
* achieving an appreciation for system complexity:
    - managing and fixing the bugs/complexities of large programs.

READING DATA:

Let's take a look at a couple of potential operating system interfaces for reading data from a file. We'll start with a couple of potential *system call* interfaces (system call = interface between the operating system and applications). Each interface will impose some constraints on both the OS and the applications. It is important to know why the current OS interface is the way it is. In the following examples, a “file descriptor” (fd) will be defined and used as an object that represents a data file. For now, the details of what exactly a file descriptor is (a pointer, etc.) will be ignored, as this subject will be returned to later in the course.

Here are some different methods to read a line of data from a file:
1) Read the data into a data array that contains the files contents.
data[0] = 1st byte of file
data[1] = 2nd byte...
data[2] = 3rd byte... etc

2) As in higher order languages, such as Perl, one can simply put the contents of a file into a variable, (example: $DATA = <fd>);

3) Read the file using buffered I/O.
char buf[1024];
int r = read(fd, buf, 1024);
< NOTE > #2 compiles into code like #3

How is #2 implemented by the compiler? Below, some different approaches are considered and evaluated.

Approach 1: To have the OS determine the length of a line and form a buffer of appropriate length prior to reading the data into the buffer.
Determine the length of a line.
1. Form a buffer of appropriate length.
2. Read the data into the buffer.

In order to check the length of the line, one can implement a system call length_of_line(fd), with the parameter fd representing the file descriptor.

Example:

```
readline()
{
        int l = length_of_line(fd);
        char c, *s;
        s = malloc(l+2);
        seek (fd,0);
        read (fd,s,l+1);
}
```

Concerns with implementation suggested above: Such design would mandate certain policies to the OS:
1.) The files are organized in lines.
2.) A line needs to be defined.
3.) File descriptor needs to be defined.

There is also another major problem with this design: with a system call to determine the length of a line, the kernel is given an unbounded amount of work. This is not a good practice since kernel may fail to finish the unbounded task, and the entire system, not only the specific application, will crash.

Approach 2: To inspect the file one character at a time and look for the end of line character. This approach would solve the problem of unbounded work for the OS, since the read calls are finite and the kernel can switch off from the task.

Example:

```
readline()
{
        int l = 0;
        char c, *s;
        while ((read (fd, &c, l)), c! = '\n')
                l++;
        s = malloc(l+2);
        seek (fd,0);
        read (fd,s,l+1);
}
```

Policy and Mechanism:

Prior to evaluating the previous approach to reading a line of data, it is important to learn about policy and mechanism (policy was already referred to in the first approach). Policy is a particular high-level goal defined for an operating system. For example, the definition of a line is a policy (end of line character, maximum length, etc.). Mechanism is a concrete method to achieve a policy. Reading one character at a time and checking for an end of line character to read in a line of data is an example of a mechanism. In general, policies change frequently, and it is a bad idea to hard-code any

particular policy into an operating system. Mechanisms, however, are more general; one can implement many line-end policies with a read-one-character mechanism. Therefore, an operating system aims to provide mechanisms that are flexible enough to implement any reasonable policy.

While some of the problems of the first approach may be solved by the second, the new approach creates another major problem. There are l+2 to l+3system calls in the readline() function above. System calls by the kernel are expensive in terms of time, and therefore this implementation would be too slow to be practical.

Approach 3: The application may send a request to the operating system and read in more than one bit at a time. Then, the application checks whether the end of line character ('\n') is within the characters read into the buffer. If the end of line character does not appear in the buffer, another read request is sent to the operating system with a larger buffer. The following C code implements this operation starting with a character buffer of size 1024.

Example:

```
char* readline()
{
        char *buf = malloc (1024);
        int siz = 1024;
        int pos = 0;
        while (1)
        {
                read (fd , buf , siz);
                for (pos = 0 ; pos<siz ; pos++ )
                        if ( buf[pos] == '\n' )
                                return buf;
                free (buf);
                siz *= 2;
                buf = malloc (siz);
                seek (fd,0);
        }
}
```

Such an application may be regarded as another level of abstraction between the main operating system and the user application. By reading many characters at a time, the application avoids excessive interruptions of the operating system, while it does not mandate unnecessary policies to the operating system (such as having a certain length for a line,...)

There are three main problems with the sample code above:
A) Data is read multiple times from the file.
B) As the code executes, the file descriptor points at random locations.
C) There is no error checking for the following possible failures:

I) Failure to allocate memory for a larger buffer size.
II) Seek command failure.
III) Read command failure.
IV) siz*2 may overflow.

Approach 4: With some minor modifications to the previous sample code, one can avoid reading the same data bits more than once. Using a second buffer, called *nbuf, the example below reads in a series of bits into *buf only once. If the end of line character does not appear in the characters read, what is already read into the buffer will be kept aside and another read request to the operating system will be made.

Modified example:

```
char* readline ()
{
        char *buf = malloc (1024) , *nbuf;
        int siz = 1024;
        int pos = 0;
        while (1)
        {
                read (fd , buf+pos , siz-pos);
                for ( ; pos<siz ; pos++ )
                        if ( buf[pos] == '\n' )
                                return buf;
                nbuf = malloc (siz*2);
                memcpy (nbuf , buf , siz);
                free (buf);
                buf = nbuf;
        }
}
```

This approach avoids reading repetitive data bits into the buffer; however, it can still be improved with additional error checking mechanisms. For example, the memcpy or the read command may fail, and their failures need to be handled properly. In addition, following the read command, the application must check whether there were enough characters returned by the operating system to fill in the buffer.

Approach 5: Sharing a file with the operating system with memory mapping:
This approach will be addressed in later chapters.

CLOSING THOUGHTS

The exercise above has hopefully shown some of the consequences of choosing different interfaces. Choosing the right interface is an important lesson to learn whether one programs an operating system or any other application. Learning about operating systems, however, creates an exceptional opportunity to learn this lesson since interfaces find such great importance in the context of operating systems. A bad operating system

interface can affect every program written for that operating system. Whenever designing an interface, such as the read() interface, one must keep the following in mind:
1) what the interface will force the users to do
2) what the interface will force the implementer to do
3) safety characteristics
4) performance characteristics
5) generality
6) limitations
A good interface designer will trade off between different goals, just like the OS interface traded off simplicity (the “length_of_line()” system call) for safety and generality (the “read(N characters)” system call).

* HCF note taken from http://www.catb.org/~esr/jargon/html/H/HCF.html