COMP 242 Class Notes
Section 7: User Interface

1. USER INTERFACE

So far we have seen how an operating system provides resources and services to *processes*. We will now study how an operating system provides resources and services to *users*.

An operating system provides services to users through one or more (batch or interactive) programs which may be invoked by users. Some of these, called **command interpreters**, provide a **command language** which is used to express a user's request. We shall study these in detail. Others, invoked usually through a command interpreter, provide services such as manipulation of files and processes. (eg: 'ls', 'chmod', 'rm', 'ps' and 'kill' in Unix). Some of these such as the Windows Directory Explorer may even be interactive and provide an alternative to command interpreters. We shall discuss only command interpreters.

1.1 Command Interpreters

A command interpreter, through its command language, defines the operating system from the point of view of non-programmers.

1.1.1 *Built-In vs Separate Program.* In older systems a command interpreter was *built-in* to the system as a module of the kernel. Multics introduced (and Unix popularized) the idea of command interpreters that are separate from the kernel and invoked like utility programs.

Treating a command interpreter as a utility program gives a user the flexibility to replace an existing command interpreter with another program. This feature has encouraged the evolution of command interpreters in Unix. (Bourne shell, C shell, etc)

1.1.2 *Built-In vs User-defined Commands.* A command may be *built-in*, or *user-defined.* A user-defined command names a utility program that is executed to process the command. For instance the command 'cd' is built-in and processed entirely by the command interpreter, while the command 'ls' results in the execution of the 'ls' program. The arguments of user-defined commands are interpreted by the invoked programs.

For each user-defined command, the command interpreter asks the operating system to create a new process executing the specified program. The command interpreter may wait for the child process to terminate before accepting another command, in which case we say the child process has been created in **foreground**, or start the new process and accept the new command while the child process is executing, in which case we say the child process is executing in **background**. The user is given the option to specify whether a command should be executed in the background or foreground. For instance, in Unix, the command:

```
cc prog.c
```

executes the compiler in foreground, while the command

```
cc prog.c &
```

executes the compiler in background.

Why support both background and foreground processes? Background processes are useful for starting several activities, for instance editing and compiling, concurrently. Foreground processes are useful for waiting for an activity to start before beginning another. For instance, a user would want the following commands executed sequentially:

```
chmod 740 prog.c
ls
```

In systems that do not support windows, a user may want the interactive processes

```
mail
edit foo
```

to be executed sequentially, in order to enforce 'mutual exclusion' on the input and output streams.

How does a command interpreter create new processes and wait for their termination?

The following pseudo code illustrates how this is done in Unix.

```
loop
   GetCommand (object_file, parameters)
   (* execute the program specified in object_file} *)
   if fork() = 0 then (* child process *)
       exec (object_file, parameters)
   elsif foregroundDesired  then
       wait()
end
```

The *fork* call creates a child process that is identical to the calling process in the sense that it gets a copy of the core image and the open file table of the calling process. The call returns the process id of the child process in the parent process, and the id 0 in the child process. This return value is used by the two processes to do different activities: The child process executes the system call *exec (object_file)*, which replaces the core image of the calling process with the contents of *object_file*. The parent process either waits for the child process to terminate or directly processes the next command, depending on whether a foreground or background process is desired.

1.1.3 *Macro Substitution.* A command interpreter may do some useful preprocessing of operands of (user-defined and built-in) commands. This preprocessing normally consists of *macro substitution.* For instance a '*'may be replaced by a string containing the file names in the current directory. Macro processing promotes *automation* (since the user-defined program does not have to do the substitution), *uniformity* (the symbol '*' has the same meaning for different commands), and *ease of use* (most programs would probably not take the trouble of defining the '*' symbol on their own, macro substitution thus provides a user with a useful facility that would otherwise go unsupported).

1.1.4 *Input/Output Redirection.* The input source and output destination of a program must be *bound* at some point; the logical name used by the program must be associated with a physical name of a file or device. A simple, but inflexible, scheme is to do the binding at compile time. A more flexible scheme is to do so at invocation time, thus providing the user control over the binding. Some command interpreters provide such binding. For instance the Unix C shell processes the command:

```
sort
```

by binding the logical input and output to the input and output streams of the terminal, the command

```
sort < data.in
```

by binding the input to the file 'data.in' and the output to the terminal, and the command,

```
sort < data.in > data.out
```

by binding the input to the file 'data.in' and the output to 'data.out'.

How does the command interpreter do the binding of logical input and output? Let us first consider input. In Unix, the child process, before executing the desired object file, checks if the input is to be redirected to some file *in_file*. If so, it executes the following code:

```
i <- open (in_file);
dup2 (i, 0); (* 0 is standard input *)
close (i)
```

which opens *in_file*, then copies the contents of the *i*th entry in the processes' file table into the 0th entry, and finally removes the ith entry. After execution of this code, the descriptor for standard input points to *in_file*. Similarly, standard output can be redirected to some file.

1.1.5 *Pipes.* A command interpreter may also bind the standard input of a process to the standard output of another process. This facility is useful for creating new applications by combining together existing applications. Thus a user may execute the command:

```
ls | more
```

to get a paged listing of a directory. This command is more useful than the sequence of commands:

```
ls > temp
more < temp
rm temp
```

The latter scheme is more longwinded. Moreover, it results in the unnecessary creation of a temporary file on disk. Finally, it reduces the concurrency and does not allow lazy evaluation.

How are pipes set up? The example command

```
ls | more
```

illustrates the mechanism. The shell calls the procedure *pipe*, which creates a pipe and returns two descriptors *i* and *o* for reading from and writing to the pipe. It then forks two processes *p* and *q* for executing the *ls* and *more* programs respectively. Process *p*, before execing the file *ls* calls *dup2 (o, 1)* to duplicate the descriptor *o* onto its standard output descriptor, and then closes *i* and *o*. Process *q* does a similar set of steps to connect its standard input to *i*.

1.1.6 *Starting and Resuming a Process.* A command interpreter may allow a user to **stop** a process and then later **resume** it. This feature allows a user to multiplex between several activities. Thus a user can stop an editor, read mail, and then resume at the point it was stopped. (Is this feature useful in a system that allows several processes to run simultaneously in different windows?)

1.1.7 *Parameters.* A process may define several **parameters** that allow specification of options. Some of these are **local**, they are associated with a program and apply to all execution instances of it. An example of a local parameter is the '-l' parameter defined by the 'ls' program. Others are **global** and are defined for all processes. An example is the Unix **home directory**. Any process that inputs file names may define this parameter. (We shall, in this discussion, ignore non-local parameters that are non-global. An example of such a parameter may be the 'suppress warning' parameter shared by all compilers. We shall treat them as global parameters.)

A process may be associated with a large number of parameters. Therefore it is important that default mechanisms be provided to relieve the user from specifying all the parameters of a process. One popular mechanism is for a process to receive some parameters from an **initialization** file associated with its program. For instance the C shell program is associated with the '.cshrc' file and the mail program with the '.mailrc' file. Initialization files allow a user to specify parameter values once for each program instead of once for each process. They may be used to specify default settings for both local parameters and global ones. An initialization file may be considered a mechanism for supporting **IS-A** or **type inheritance**: it is associated with a program (class) and each instance of that program (i.e. each process that executes that program) reads the file and thus essentially inherits the parameters.

A method for specifying default global parameters involves **IS-PART-OF** or **structure inheritance**, wherein a child process inherits the global parameters of its parent, which can later be overriden. These parameters may be stored in the process table entry of a process, and the kernel, when it starts the process, makes a copy of the parent process' parameters. Alternatively, a process, (in particular a command interpreter) when it starts another, may tell the kernel the value of global parameters. These may then be copied in a well known location in the process. Unix supports the second method. In Unix, the set of global parameters and their values are called a process' **environment**. The *exec* call can take as an argument the desired environment. Unix also supports automatic copying of global parameters from a well known location in the parent process to a well know location in a child process, when the environment of the child process is not explicitly specified.

How does a process receive its inherited global parameters? In Unix, when a C program of the form:

```
main (argc, argv, env)
int argc;
char **argv, **env;
{ ... };
```

is executed, *env* points at an array of strings of the form (parameter name=parameter value).

So far we have seen how default parameters of a process are specified. How does a user change these default values? The answer depends on the process. The Unix C shell illustrates one approach. It allows global (environment) parameters to be changed via the **setenv** call. Thus to change the home directory, a user can execute the command:

```
setenv HOME /usr/joe
```

The command:

```
setenv p
```

defines a new global parameter *p*, which is inherited by all processes started by the shell (and can be changed like other parameters), and the command

```
setenv
```

prints all the global parameters and their values.

Unix allows local parameters to defined, changed, and examined via the **set** command. The command:

```
set q
```

defines a new local parameter *q*, the command

```
set q=5
```

gives it the value 5, and the command

```
set
```

prints all the current local parameters and their values.

What does it mean to let the user define a new parameter (which we said defined an option), if the shell does not know about this parameter and thus cannot use its value? If the parameter is an environment parameter, then even if the shell does not know about it, some other process started by the shell may be able to use its value. Moreover, the local and environment parameters defined by the shell also serve as **shell variables**, whose values can be used in the command line. A user thus create a new variable and change or use its value later, as shown below:

```
set d=/usr/joe/242/test.c
cc $d
ls $d
lpr $d
```

History

A command interpreter may allow a user to examine previous commands and allow them to be resubmitted with changes. For instance the C shell allows commands of the form:

```
!v
```

which says repeat the last command starting with a 'v', and the command

```
vi !$
```

which says use the previous operand. This history mechanism uses the principle of locality applied to user activity. In a phase, a user is interested in a small number of commands and operands. The C shell allows the user to specify the size of this "working set", that is, the number of previous commands it remembers at one time.

Studies have shown that the working set of a user does not change much over time. Bala Krishnamurthy at Purdue investigated command usage among a variety of users - faculty, industry, research, secretary and student - and found that the average number of commands executed over 85 percent of the time was 15. The most frequently used commands for faculty were ls, vi, cd, set, more, stty, echo, pg, fg, logout, mail, ps, rm, pwd, date, tset, jobs, hostname, dirs. For other classes of users, the set was not very different. This is a strong argument for Mac-style permanent menu bar containing these often used commands.

Control Flow

A command interpreter may provide constructs for control flow. For instance, the Unix C shell allows specification of the kind:

```
foreach i (*.c)
cp $i newdir
end
```

copies all C source files (ending with the suffix '.c') to directory "newdir".

Command Procedure

A sequence of commands may be stored permanently in a file called a **command script** or **command procedure**, which may later be executed to invoke these commands. For instance the command:

```
who | grep $1
```

may be stored in a command script called 'wg'. The '$1' stands for the 1st argument of the command script. Later the command script may be invoked as:

```
wg fred
```

which is equivalent to

```
who | grep fred
```

Control constructs and command procedures give the user a powerful facility to program a sequence of actions.

Command Languages vs Programming Languages

Notice the similarity between some of the constructs such as the foreach provided by command languages and those provided by programming languages. Given this similarity, an interesting issue, we will not study here, is whether we can support a unified command/programming language?