Still more UNIX

Core Functionality of Shells

- built-in commands (1.13, 6.1)
- variables (6.6, 6.7)
- wildcards (file name expansion, 6.5)
- background processing
- scripts
- redirection
- pipes
- subshells
- command substitution (6.5)

Executables vs. Built-ins

- Most UNIX commands invoke utility programs that are stored as executable files in the directory hierarchy
- Shells also contains several built-in commands, which it executes internally
- Type **`man shell_builtins`** for a partial listing
- Built-in commands execute as subroutines, and do not spawn a child-shell via `fork()`
  - Expect built-in (*e.g.* cd) to be faster than external (*e.g.* ls)

Built-In:

cd, echo, jobs, fg, bg

Non-Built-In:

ls, cp, more

Variables (6.6-7)

- Two kinds of variables:
  - local
  - environment
- Both hold data in a string format
- Main difference: when a shell invokes another shell, the child shell gets a copy of its parent’s environment variables, but not its local shell variables
- Any local shell variables which have corresponding environment variables (**`term`**, **`path`**, **`user`**, etc.) are automatically inherited by subshells

Variables (cont.)

- Local (shell) variables:
  - Simple variable: holds one value
  - List variable: holds one or more values
  - Use **`set`** and **`unset`** to define, delete, and list values
- Environment variables:
  - Use **`setenv`** and **`printenv`** to set and list values
  - All environment variables are simple (ie: no list variables … compare shell variable **`$path`** to enviroment variable **`$PATH`**)

Startup Files (6.9)

- Every time **`csh`** is invoked, **`$HOME/.cshrc`** is read, and contents of the file are executed

- If a given **`csh`** invocation is the login shell, **`$HOME/.login`** will also be read and its contents executed
- **csh -f** starts a shell without reading initialization files
- opening a new **xterm** under X-windows will (by default) open a new login shell

Sourcing files (6.5)

- Assume you create a file called “*my_aliases*”
- Typing **`csh my_aliases`** executes the lines in this file, but it occurs in the forked csh, so it will have no lasting effect on the interactive parent shell
- Correct method is to use the *source* command:

  ```
  source my_aliases
  ```

- Common setup:
  - put all aliases in a file called **`$HOME/.alias`**
  - add the line “source .alias” to the last line of **`$HOME/.cshrc`**

Input Processing (6.5)

- When a input is typed, it is processed as follows:
  - *history* substitution
  - *alias* substitution
  - *variable* substitution
  - *command* substitution
  - *file name* expansion

Command Substitution (6.5)

- Can substitute the output from a command into the text string of a command

```
set dir = `pwd`
set name = `pwd`/test.c
set x = `/bin/ls -l $file`
```

UNIX
Systems Programming

System Calls

- System calls:
  - perform a subroutine call directly to the UNIX kernel
- 3 main categories:
  - file management
  - process management
  - error handling

Error Handling

- All system calls return -1 if an error occurs
- **`errno`**:
  - global variable that holds the numeric code of the last system call
- **`perror()`**:
  - a subroutine that describes system call errors
- Every process has **errno** initialized to zero at process creation time
- When a system call error occurs, **`errno`** is set
- See **`/usr/include/sys/errno.h`**
- A successful system call <u>never</u> affects the current value of **`errno`**
- An unsuccessful system call <u>always</u> overwrites the current value of **`errno`**

`perror()`

- Library routine:

  `void perror( char *str )`
- **`perror`** displays **`str`**, then a colon (**`:`**), then an english description of the last system call error, as defined in the header file

  `/usr/include/sys/errno.h`
- Protocol:
  - check system calls for a return value of -1
  - call **`perror()`** for an error description during debugging (*see example on next slide*)

perror() example

```
#include <stdio.h>
#include <errno.h>

int main( void )
{
   int returnVal;
   printf( "x2 before the execlp, pid=%d\n", getpid() );
   returnVal = execlp( "nonexistent_file", (char *) 0 );
   if( returnVal == -1 )
      perror( "x2 failed" );
   return( 1 );
}
```

Processes Termination

- Orphan process
  - a process whose parent is the init process (pid 1) because its original parent died before it did
- Terminating a process: **`exit()`**
- System call:

```
int exit( int status )
```

- Every normal process is a child of some parent, a terminating process sends its parent a **`SIGCHLD`** signal and waits for its termination code status to be accepted
- The C shell stores the termination code of the last command in the local shell variable **`status`**

Zombies

- Zombie process:
  - a process that is “waiting” for its parent to accept its return code
  - a parent accepts a child’s return code by executing **`wait()`**
  - shows up with 'Z' in `ps -a`
- A terminating process may be a (multiple) parent; the kernel ensures all of its children are orphaned and adopted by **`init`**

`wait()`

- Waiting for a child: system call is

```
int wait( int *status )
```

- A process that calls **`wait()`** can:
  - block (if all of its children are still running)
  - return immediately with the termination status of a child (if a child has terminated and is waiting for its termination status to be fetched)
  - return immediately with an error (it it doesn’t have any child processes)
- More details in a few weeks, when we cover Chapter 11 of Wang

Signals

- Unexpected/unpredictable events:
  - floating point error
  - interval timer expiration (alarm clock)
  - death of a child
  - control-C (termination request)
  - control-Z (suspend request)
- Events are called interrupts
- When the kernel recognizes such an event, it sends the corresponding process a signal
- Normal processes may send other processes a signal, with permission (useful for synchronization)
- Again, we’ll cover this in much more detail in a few weeks

Race conditions

- A <u>race condition</u> occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run
- This is a situation when using forks: if any code after the fork explicitly or implicitly depends on whether or not the parent or child runs first after the fork
- A parent process can call **`wait()`** for a child to terminate (*may block*)
- A child process can wait for the parent to terminate by *polling* it (wasteful)
- Standard solution is to use signals

Example: Race Condition

```
#!/usr/bin/csh -f
set count = 0
while( $count < 50 )
   set sharedData = `cat shareVal`
   @ sharedData++
   echo $sharedData >! shareVal
   @ count++
end
```

- Create two identical copies, “**`a`**” and “**`b`**”
- Run as: **`./a&; ./b&`**

Miscellaneous

- From Wang:
  - **`rlogin (9.3)`**
  - **`rsh (9.3)`**
  - **`rcp (9.3)`**
  - **`finger (1.9, 4.6)`**
  - **`telnet (9.3)`**
  - **`ftp (9.4)`**