

Still more UNIX

S-73

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)

S-74

Executables vs. Built-ins

- Most UNIX commands invoke utility programs that are stored as executable files in the directory hierarchy
- Shells also contain 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`

S-75

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

S-76

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 (i.e. no list variables ... compare shell variable `$PATH` to environment variable `$PATH`)

S-77

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

S-78

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`

S-79

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

S-80

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`
```

S-81

UNIX Systems Programming

S-82

System Calls

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

S-83

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 never affects the current value of **errno**
- An unsuccessful system call always overwrites the current value of **errno**

S-84

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)

S-85

perror() example

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

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

S-86

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**

S-87

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**

S-88

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 (if it doesn’t have any child processes)
- More details in a few weeks, when we cover Chapter 11 of Wang

S-89

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

S-90

Race conditions

- A **race condition** 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

S-91

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&
```

S-92

Miscellaneous

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

S-93