Today's Lecture

1. `stdin`, `stdout`, `stderr`
2. Shell Programming
   (a) Motivation
   (b) History
   (c) Shell Commonalities
   (d) **KornShell Programming** (`ksh`)
3. Error handling: `errno`

Standard Input, Output, and Error

There are three predefined "special files" in Unix (similarly in Windows NT/95). They refer to an input character stream, output stream, and error stream.

In shells and system calls, file descriptor

- 0 refers to the *standard input*
- 1 refers to the *standard output*
- 2 refers to the *standard error*

For buffered IO (e.g. `fopen`), the symbol

- `stdin` refers to the *standard input*
- `stdout` refers to the *standard output*
- `stderr` refers to the *standard error*

Symbols for buffered IO are defined in `/usr/include/stdio.h` as, e.g.:

```
#define stdin   (&_iob[0])
#define stdout  (&_iob[1])
#define stderr  (&_iob[2])
```

Key point: IO can be *redirected*

Shells

What is a shell?

- A shell is a program (e.g. `/bin/csh` or `/bin/ksh`); it can be replaced by another
- All interactive processes run some type of shell
- Shell is a command interpreter
- Shell provides an environment and means of customizing this environment
- Shell is a fairly high-level programming language
- Alternative to an interactive shell is a windowing system (does not exclude shells within windows)
- MS-DOS prompt is the "shell" in Windows NT/95

Shells

What are shells good for?

- Shells let you issue commands
- Provide many useful features such as paths, filename expansion
  (e.g. `~cs333`)
- Glue => shells let you build new programs by combining existing programs
- Handle file-system operations well
- Handle string operations well
- Fairly easy to program => repetitive editing tasks can often be done by a throw-away shell program, for example
- Shell programs are often used for software installation, archival, system administration, *etc.*

Shells

Some disadvantages:

- Little support for programming in the large,
- Slow (compared to C)
- Arithmetic is cumbersome
- Many shells, slightly different syntaxes

Some Shell History

date	shell	src size (bytes)	
1976	Bourne shell	70583	
1979	csh	245885	rewrite of 6th edition (history mechanism, jobs)
1981	ksh	424495	Bourne + some csh
1983	v8 sh	122754	rewrite of Bourne
1985	tcsh	514438	editor
1989	bash	751835	GNU shell (csh rewrite) (ksh and tcsh influences)

Some Commonalities

All shells

- Have at least two modes:
  - interactive shell prompt (e.g., `$`)
  - executable shell scripts (MS-DOS `.bat` files); hence you must, for example, `$ chmod +x myscript.ksh`
- Provide a variable-based environment
- Admit some form of IO redirection

`ksh` References

Morris I. Bolsky and David G. Korn, *The KornShell Command and Programming Language*, Prentice Hall, 1989. ISBN 0-13-516972-0

Anatole Olcazak, *The Korn Shell user and programming manual*, Addison-Wesley, 1992. ISBN 0-201-56548-X

Programming with ksh

Lecture Topics:

1. Environment variables
2. IO redirection
3. Pipelines, Combining Commands
4. Quoting
5. Jobs
6. Scripts & Programming Language
7. Further Study

ksh: Environment Variables

Set variable x to value v by the command:

$ x=v

Examples:

```
$ FILE_NAME=myfile.c
$ X=1
```

Retrieve value of variable x:

$ $\cdots$ \$$x$ $\cdots$

Examples:

```
$ print $FILE_NAME
myfile.c
$

$ print $Y

$
```

Many built-in environment variables: `PATH`, `PPID`, `PWD`, ...

ksh: Exporting Variables

Use `export` when subsequent processes must inherit an environment variable:

```
$ FILE_NAME=myfile.c
$ X=1
$ print $FILE_NAME
myfile.c
$ print $X
1
$                      # ok, that works
$ ksh
$ print $FILE_NAME

$ print $X

$                      # what's going on?
$ exit
$                      # need an export...
$ export FILE_NAME X
$ ksh
$ print $FILE_NAME
myfile.c
$ print $X
1
$
```


ksh: Variable Attributes

`typeset` has many options for inspecting and setting variable attributes:

```
$ typeset
export VISUAL
export PATH
function ERRNO
export MAIL
function SECONDS
integer PPID
export PS1
PS3
export PS2
export TERMCAP
OPTARG
export USER
export SHELL
export FILE_NAME
export X
export HOME
export TERM
export PWD
integer MAILCHECK
$
```

ksh: IO Redirection

Basic redirection mechanisms:

<*file*
redirects standard input from *file*

>*file*
redirects standard output to *file*

>>*file*
redirects standard output to append to *file*

Redirection operators may be prefixed with a file descriptor. How would you redirect error messages to a file?

More powerful mechanisms:

<<*some-symbol*
A *here-document.* Uses subsequent lines as standard input, stopping when a line containing only *some-symbol*, or EOF, is found

<>*file*
use *file* as input and output

others
duplicate input/output streams, variants of here-document, *etc.*

ksh: IO Redirection Example


```
$ print "an example" >example
$ cat >>example <<DONE
 this is a here-document.  Everything entered
 here until the word DONE is found alone on
 a line is given as stdin to cat
 DONE
$ cat example
an example
this is a here-document.  Everything entered
here until the word DONE is found alone on
a line is given as stdin to cat
$
```


ksh: Pipelines & Combining

The output of a command may be *piped* as the input to another command:

```
$ ls | wc
       9        9       67
$
```

or

```
$ ls | wc | lpr
$
```

The ; separator is used to form a simple list of pipelines, executed sequentially from left to right:

```
$ ls | wc >tmp-file ; cat tmp-file
       9        9       67
$
```


ksh: Combining, Return Values

Commands return values; the integer `n` returned from the `exit(int n)` syscall, for example. However, in `ksh` zero (0) means *true* and non-zero means *false*!

The operators `&&` and `||` act like C's short-circuit boolean operators.

```
$ cd sub 1>/dev/null || print "dir exists"
$ cd subdir 1>/dev/null && print "dir doesn't exist"
directory does not exist
$
```

ksh: Quoting

Quoting is a shell mechanism to:

- override the shell's interpretation of special characters
- group separate tokens into a single entity (token)
- evaluate a command within a command

ksh: Quoting (cont.)

Here are some common types of **ksh** quoting:

- *c* escapes character *c*:

```
$ print \< \\
< \
$
```

- *newline* to continue current line:

```
$ print this is all on \
the same logical line
this is all on the same logical line
$
```


ksh: Quoting (cont.)

- Single quotes remove special meaning of all enclosed characters:

```
$ print '\$X is $X'
\$X is $X
$
```

- Double quotes remove special meaning of all enclosed characters except `$`, `` ` ``, `"` and `\`

```
$ print "\$X is $X"
$X is 1
$
```

- `$(...)` introduces a new command to be evaluated:

```
$ USER=$(whoami)
$ print $USER
cs333
$
```

  This type of quoting is called "New Command Substitution" and is extremely powerful.

ksh: Jobs

Interactively, **ksh** allows one to suspend (`^Z`) and resume a job either in the foreground (`fg`) or background (`bg`).

Every job has a number. To refer to a job, preface its number *n* with a `%` character. For example, `fg %n` places job *n* in the foreground.

`jobs` lists the current jobs:

```
$ jobs
[3] +  Running                    xdvi l2
[2] -  Running                    xdvi assignment1.dvi
$
```

`kill %`*n* terminates job *n*:

```
$ kill %2
[2] - Terminated                  xdvi assignment1.dvi
$
```

`kill` can also be given process ids;
`kill -9` will almost always remove the job (process).

Appending `&` to a command runs the command without waiting for it to complete.

Appending `|&` to a command runs the command as a co-process; the spawning **ksh** process can communicate with the new co-process via standard input and output.

ksh: Writing Scripts

A `ksh` script is a text file that starts with the line:

```
#!/bin/ksh
```

Note: comments also start with `#`; Is this good software design?

A simple script to clean out some non-essential files in the current directory:

```
#!/bin/ksh
# clean.ksh

rm -f tmp *.o a.out core
```

A script to periodically print the date and time to stdout:

```
#!/bin/ksh
# saydate.ksh
# run this in the background with &

HOUR_IN_SECONDS=360
date
sleep $HOUR_IN_SECONDS
exec saydate.ksh
# end of saydate.ksh
```


ksh: Script Parameters

A script can access its parameters in a manner similar to variable expansion: \$$n$ refers to the nth parameter, starting with 1.
\${n} is equivalent to \$$n$

For example,

```
#!/bin/ksh
# invert.ksh
# reverses the arguments to a two-argument command

CMD=$1
ARG_A=$2
ARG_B=$3
$CMD $ARG_B $ARG_A
# end of invert.ksh
```

Exercise: write a general (n parameter) version of `reverse`

ksh: Special Parameters

Special parameter

`$*` refers to all positional parameters

`$#` refers to the number of positional parameters

`$?` refers to the return value of the last command

Example:

```
#!/bin/ksh
# demo-args.ksh
print "there are $# args to this script"
print "they are \'$*\'"
print "the above \'print\' returned $?"
# end of demo-args.ksh
```

```
$ demo-args.ksh first second third
there are 3 args to this script
they are 'first second third'
the above 'print' returned 0
$
```

Other special parameters: `$@`, `$-`, `$$`, `$!`

ksh: Further Useful Features

- Familiar control-flow constructs: `if`, `while`, `for`, `case` (switch)
- Expressions can be used in, for example, conditionals
- Command return values can be used as expressions
- `test` can query many file-system related properties, for example:

  `test -r` *file*
  true if *file* exists and is readable

  `test -w` *file*
  true if *file* exists and is writable

  `test -x` *file*
  true if *file* exists and is executable

  `test -d` *file*
  true if *file* exists and is a directory

  `test -s` *file*
  true if *file* exists and has a size > 0

  many more...
- Pattern matching facility
- History mechanism

ksh: More Scripts


```
#!\bin\ksh
# watch-for.ksh

PATH=/bin:/usr/bin
SECONDS_PER_MINUTE=60

case $# in
0)      echo "usage: watch-for.ksh person"
        exit 1
esac
until who | egrep $1
do
        sleep $SECONDS_PER_MINUTE
done
# end of watch-for.ksh
```

```
#!\bin\ksh
# renumber.ksh
# renames all files with 4 character names to integer names
# dangerous!

integer count=1
for i in ????
do
    mv $i $count
    count=count+1
done
# end of renumber.ksh
```


Syscall Errors: errno, perror


```
PERROR(3)               C LIBRARY FUNCTIONS              PERROR(3)

NAME
     perror, errno - system error messages

SYNOPSIS
     void perror(s)
     char *s;

     #include <errno.h>

     int sys_nerr;
     char *sys_errlist[];
     int errno;

DESCRIPTION
     perror() produces a short  error  message  on  the  standard
     error describing the last error encountered during a call to
     a system or library function.  If s is not  a  NULL  pointer
     and does not point to a null string, the string it points to
     is printed, followed by a colon, followed by a  space,  fol-
     lowed  by  the message and a NEWLINE. If s is a NULL pointer
     or points to a null string, just  the  message  is  printed,
     followed  by  a  NEWLINE.  To  be  of most use, the argument
     string should include the name of the program that  incurred
     the  error.   The  error  number  is taken from the external
     variable errno (see intro(2)),  which  is  set  when  errors
     occur but not cleared when non-erroneous calls are made.
```

Example:

```
/* sayerror - print error message s, prefixed by t */
sayerror(s, t)
char *s, *t;
{
char buf[MAXLINE];

if (errno > 0) sprintf(buf, "%s%s (%d)\n", t, s, errno);
else sprintf(buf, "%s%s\n", t, s);
for (s = buf; *s; s++) ;
write(2, buf, s - buf);
errno = 0;
}
```


Next Lecture

- Study `~cs333/src/sh.c`
- Read `man` pages for:

  `execve`

  `wait` (section 2)

  `pipe`

  `dupe`