

# Assignment: The Yash Shell

CS230/330 - Operating Systems (Winter 2001).

Due : Friday, January 12, 11:59 p.m.

## Overview

In this assignment, you will implement a simple Unix command shell known as Yash (Yet Another Shell). Your shell will have to support I/O redirection, pipes, signals, and background processes. The primary goal of this assignment is to familiarize yourself with the basic functionality of the Unix operating system including processes, files, and interprocess communication. In addition, you will be learning how to use some of the development tools needed for the class project.

## Get an account

Your first step is to get a CS account. Afterwards, stop by my office in Ryerson 257B to introduce yourself and to get an account on the class machines. All of your work in this class must take place on these machines as they have been configured for the OS project and provide some additional stability when compared to other machines in the department. Please note, the following machines are available:

- stonecrusher.cs.uchicago.edu
- sunspot.cs.uchicago.edu
- sidewinder.cs.uchicago.edu
- stimulation.cs.uchicago.edu
- wildcat.cs.uchicago.edu

## Description

The input to your shell is a sequence of commands, each provided on a separate line of input text typed interactively at the keyboard. The following commands must be supported:

```
progname [args]
```

Runs the program `progname` with the given, possibly optional, arguments. For example:

```
yash % ls
foo.c
bar.c
yash % cp foo.c foo1.c
yash % rm -f foo.c
```

```
exit
```

Forces the shell to exit.

## I/O Redirection

In addition to the above commands, your shell must support I/O redirection. I/O redirection is specified using the `<` and `>` operators at the end of a command line. For example:

```
progname [args] >file.out
```

Directs the standard output of `progname` to the file `file.out`.

```
progname [args] <file.in
```

Uses the contents of the file `file.in` as the standard input to program `progname`.

Both input and output redirection may be specified for a single command so your shell will have to check for both.

## Pipes

Your shell also needs to support pipes. A pipe is nothing more than a way of hooking up the standard output of one program to the standard input to another. A pipe is indicated using the `|` operator as follows:

```
progname1 [args] | progname2 [args]
```

Pipes the output of program `progname1` to the input of program `progname2`. For example:

```
yash % ls -l | wc
      7      56     366
yash %
```

More than one pipe may be included in a command line and pipes may be combined with I/O redirection. For example :

```
yash % foo <infile | bar | spam >outfile
```

## Background Jobs

When a program runs, it normally blocks you from performing any other operations until it has completed. However, you can put a program into the background using the `&` operator. For example:

```
progname [args] &
```

Detaches the program `progname` and runs it in the background. Control is immediately returned to the command shell where additional commands can be executed. Background jobs should continue to run even if you quit the shell before they have finished.

## Signals

Finally, your shell needs to catch a single signal, `SIGINT`. This signal is generated when a user presses Control-C on the keyboard. When received, your shell should pass it to the currently running program. In other words, Control-C should not cause your shell to terminate--rather, it should terminate the program the shell is currently running. Control-C has no effect on background jobs.

## Exit codes

When programs terminate, they will return an integer exit code to your shell. If this code is non-zero, your shell should print the returned value. For example:

```
yash % cp foo bar
cp: cannot access foo
[ Program returned exit code 1 ]
yash %
```

## Graduate Credit

Modify your shell with the following features:

- Pressing 'Control-Z' stops the currently running process and returns to the `yash %` prompt.
- Typing 'resume' resumes the last stopped process.

- Typing 'bg' places the last stopped process in the background and returns to the `yash %` prompt.

The implementation of these features will require you to investigate aspects of Unix job control, process groups, and signals. In addition, you may find yourself using systems calls such as `tcsetpgrp()`, `setsid()`, and `setpgrp()`.

## How to get started

Make sure you understand the assignment before beginning any work. Now, consider the following steps as a rough guide.

### Step 1 : Command line parsing

Write a function that takes a line of input text and parses it into some sort command structure containing information about the program name, arguments, and options for I/O redirection, pipes, and background jobs. If it helps, the syntax for the the shell is as follows (optional fields are in brackets) :

```
command      :  program
              |  program | program
              |  "exit"
              ;

program      :  identifier [ arglist ] [ <infile ] [ >outfile ] [ & ]
```

Tokens and arguments are separated by white space. To simplify parsing, Your shell does **NOT** need to support quoted strings such as the following:

```
yash % foobar "This is a quoted argument"
```

Furthermore, you can assume that no whitespace separates the < and > operators from the filename that follows.

After you've got your command line parser working, write an infinite loop that does nothing but print the shell prompt ("`yash %` "), read a line of input, and pass it to your command line parser. Check the data returned from the parsing function to make sure it looks reasonable.

Note: writing the command parser should be easy. Just use the `strtok()` function from the C library. There is no need to write a lex/yacc based parser or anything of comparable complexity (your parser should only be around 50 lines of code).

### Step 2 : Make your shell run programs

Once you're satisfied with the parser, modify the command loop to execute programs. You will need to use the `fork()` and `exec()` system calls to do this. While running, the shell process should wait for the program to complete by calling `wait()`. The shell should also check the exit code returned by the program and print a message if it is nonzero. Note : the exit code is placed into the lower 8-bits of the status code set by `wait()`. Your code will look roughly like this:

```
while (1) {
    read a line of input
    cmd = parse command line
    pid = fork();
    if (pid == 0) {
        extract the program name from cmd
        ...
        exec( ... args ...); /* Execute the command */
    } else {
        wait(&status); /* Wait for command termination */
        check return code placed in status;
    }
}
```

At this point you should have a working shell. Try it out by running some of your favorite Unix commands such as "ls", "cp", "cat" and so forth. If it doesn't work, you have done something wrong.

### Step 3 : Add I/O direction

To add I/O redirection, modify the child process created by `fork()` by adding some code to open the input and output files specified on the command line. This should be done using the `open()` system call. Next, use the `dup2()` system call to replace the standard input or standard output streams with the appropriate file that was just opened. Finally, call `exec()` to run the program.

### Step 4 : Add Background Jobs

This is a little more tricky. When a job is put into the background, the shell just starts it and forgets about it (the shell should return to the command prompt and allow more commands to be typed). However, this presents two problems. First, the background job should keep running even if the shell terminates. Thus, this means that the background job can't be a child of the shell process. Second, when the background job finishes, it needs to have its exit code collected--otherwise it turns into a zombie.

Modify your shell to run background jobs in a way that solves both of these problems. Hint : the solution involves the `fork()` function.

### Step 5 : Add pipes

To support pipes, you need to execute two separate programs and play some funny games with I/O to make the output of one program go to the input of the other program. To do this, you'll need to use the `pipe()` and the `dup2()` system calls.

### Step 6 : Make control-C work

Modify your shell so that the SIGINT signal causes the currently running program to terminate while the shell continues to run (i.e., your shell should ignore SIGINT).

### Step 7 (GRADUATE ONLY) : Implement job control.

Consider this part to be a challenge.

### Step 8 : Sit back and relax.

By now, you should be ready for the kernel project. "Ha, bring it on!", you say.

## Other Odds and Ends

### Header files

You will probably need to use the following header files in your solution.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
```

### Getting Help

Since this is an upper division/graduate computer science course, you are expected to do your own research

regarding the usage of various system calls, header files, and libraries. Information is readily available in the man pages, Unix reference books, or on the web.

Otherwise, do not hesitate to ask a question if you are unclear about how some part of the assignment is supposed to work.

## Handin Procedure

### Step 1: Organize your files

Your solution should minimally consist of the following files:

- `Makefile`. We should be able to compile your shell by typing 'make'. The executable for your shell should be called "yash".
- `yash.c`. The source code for your shell.

Put all of your files into a directory called 'yash'. Thus, the structure should look like this:

```
yash/  
  Makefile  
  yash.c
```

### Step 2: Import into CVS

Import the 'yash' directory into your CVS repository using the 'cvs import' command like this:

```
unix % ls  
yash  
unix % cd yash  
unix % ls  
Makefile          yash.c  
unix % cvs import -m "Yash Shell" yash yourname start
```

**Note: the 'cvs import' command must be executed from the same directory that contains yash.c and Makefile files!**

### Step 3: Move the directory

Leave the 'yash' directory and rename it to something else. For example:

```
% cd ..  
% mv yash old_yash  
%
```

### Step 4: Verify the checkin

Type the following information to check out your repository:

```
% cvs checkout yash
```

Go into the newly created 'yash' directory and make sure all of your files are there.

### Step 5: Create a README file

Within the 'yash' directory created in Step 4, create a README file that contains your name and any other pertinent information about your solution that we should know about.

## Step 6: Add your README file to the repository

Add the README file to the CVS repository as follows:

```
% cvs add README
% cvs commit -m "" README
```

## Step 7: You're done

Once your assignment is in the CVS repository, you are done. Your assignment will automatically be collected from the repository once the due-date has passed. **If you have not committed your program files to the repository before they are collected, you will receive no credit!**

To make sure the most up to date versions of your files are added to the repository, make sure you do all of your work from the 'yash' directory created by the 'cvs checkout' command. Also, make sure you do a 'cvs commit yash' on the entire directory to make sure all modifications are checked into the repository.

To verify your checkin, create a scratch directory someplace and do the following:

```
% mkdir junk
% cd junk
% cvs checkout yash
% cd yash
% make
% yash
Welcome to the yash shell
yash % ls -l
...
yash % exit
%
```

## Grading

Your shell will primarily be graded for correctness. An automated test program will be used to exercise various features of the shell. Because of this, you should **NOT** modify the `yash %` prompt or deviate from the specifications described in this handout (i.e., don't change the name of the commands or the shell syntax). Your grade will consist of the following:

- Correctness. 80%
- Programming style and efficiency. 20%.

**No late handins are accepted!**