Laboratory Exercise 1 Solutions

cpe 453 Spring 2010

```
A pipe gives a wise man time to think and a fool something
to stick in his mouth.
```

— /usr/games/fortune

Due online to the `Lab01` directory of `pn-cs453` at (or before) 11:59pm, Wednesday, April 7th.
The Laboratory Exercises may be done with a partner.

Problems

There are no written exercises this week. Stay tuned for next week.

Laboratory Exercises

Two small lab exercises: one for my benefit, one for yours.

1. **Photograph**

   To help me with sorting out who's who in the class (and three years from now when you write me out of the blue asking for a recommendation) I am asking you to submit a digital photo of yourself to the `Lab01` directory of the `pn-cs453` account. I will not be sharing these with anybody so it doesn't have to be glamorous, but it does need to be recognizable. That is, that great photo from last Halloween of you in your mummy costume really isn't the thing for this lab.

   Please make the root of the file name your login name. E.g., mine would be `pnico.jpg`.

2. **Program: `pipeit`**

   This quick progam is intended to be a quick review of systems programming concepts, including `fork(2)`, `exec(2)`, pipes, and `make(1)`.

   Your task is to write a program that launches the pipeline "`ls | sort -r > outfile`" and waits for it to terminate. When it is finished, `outfile` should consist of a listing of the files in the current directory in reverse alphabetical order.

   For this lab, it is acceptable to hard-code all the steps necessary.

   Your program must:

   - create a pipe for interprocess communication;
   - `fork()` two children, one for each program;
   - set up each child's file descriptors appropriately;
   - `exec()` the appropriate program in each child;
   - terminate (the parent) with zero status on success, nonzero on failure;
   - do appropriate error checking;
   - check the exit status of the child processes and terminate with nonzero status in case of error, and;
   - be appropriately documented.

Tricks and Tools

This is not a large program, but it is designed to be a quick review of many systems programming concepts in preparation for studying Operating Systems. For reference, you may want to consider the functions listed in Table 1.

`fork(2)`	create a new process that is an exact copy of the current one.

`execl(3)`
`execlp(3)`
`execle(3)`
`execv(3)`
`execvp(3)`
`execve(2)`
	loads a new program into the current process

`dup2(2)`
`open(2)`
`close(2)`
	standard UNIX unbuffered IO system calls

Table 1: Some potentially useful system calls and library functions

Coding Standards and Make

See the pages on coding standards and make on the cpe 453 class web page.

What to turn in

For the Laboratory Exercises: Submit, via handin, to the `Lab1` directory on the CSL machines:

- Your photograph named after yourself.
- Your reasonably well-documented source files.
- A makefile (called `Makefile`) that will build your program with "`make pipeit`".
- A README file (called "README", no extension) that contains:
  - Your name(s) (with login name(s) in parentheses, e.g. "(pnico)").
  - Any special instructions for running your program.
  - Any other thing you want me to know while I am grading it.

  The README file should be **plain text,** i.e, **not a Word document**, and should be named "README", all capitals with no extension.

Sample runs

```
% ls
Makefile  pipeit   pipeit.c  pipeit.o
% ./pipeit
% echo $status
0
% cat outfile
pipeit.o
pipeit.c
pipeit
Makefile
% rm outfile
% chmod u-w .
% ./pipeit
outfile: Permission denied
% echo $status
1
%
```

(Aside: `$status` is the way to get the exit status of the previous command in tcsh. In bash, it would be "`echo $?`")

Solution:

File	**Where**
Makefile	p.4
pipeit.c	p.5

```
CC     = gcc

CFLAGS  = -Wall -g

PROG    = pipeit

OBJS    = pipeit.o

SRCS    = pipeit.c

HDRS    =

EXTRACLEAN =

all:     $(PROG)

allclean: clean
        @rm −f $(EXTRACLEAN)

clean:
        rm −f $(OBJS) *~ TAGS

$(PROG): $(OBJS)
        $(CC) $(CFLAGS) −o $(PROG) $(OBJS)

depend:
        @echo Regenerating local dependencies.
        @makedepend −Y $(SRCS) $(HDRS)

tags : $(SRCS) $(HDRS)
        etags $(SRCS) $(HDRS)

test:    $(PROG)
        $(PROG)
```

```
/*
 * pipeit: a simple pipeline demonstration program.
 *         Everything is hard-coded to launch "ls | sort -r > outfile",
 *
 *         Note that probably 3/4 of this program is error checking.
 *         such is the life of a systems programmer...
 *
 * $Log:$
 *
 */

#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<sys/fcntl.h>
#include <sys/wait.h>

#define READ_END  0
#define WRITE_END 1

#define OUTFILE "outfile"

#define PERMS ( S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH )

int main(int argc, char *argv[]) {
  int line[2];
  int fdout;
  int status;
  int err;
  pid_t child;

  err = 0;                /* so far, so good... */

  /* First create pipe */
  if ( pipe(line) ) {
    perror("pipe");         /* pipe creation failed. */
    exit(-1);
  }

  /* now, fork() off the children, and exec() appropriately */
  if ( -1 == (child=fork()) ) {
    perror("fork()");        /* fork() failed.... */
    exit(-1);
  }
  if ( !child ) {
    /* child one */

    /* dup file descriptors appropriately */
    if ( -1 == dup2(line[WRITE_END],STDOUT_FILENO) ) {
      perror("dup2");        /* dup2() failed */
      exit(-1);
    }

    /* clean up before exec()*/
    if ( close(line[READ_END])  ||
         close(line[WRITE_END]) ) {
      perror("close");       /* We should probably check each one, but */
      exit(-1);              /* if close fails we're in deep trouble */
    }                        /* any way you look at it... */

    /* do the exec */
    execlp("ls","ls",NULL);

    /* exec failed (if we're still here...) */
    perror("ls");
    exit(-1);
  }

  if ( -1 == (child=fork()) ) {
    perror("fork()");        /* fork() failed.... */
    exit(−1);
  }
  if ( !child ) {
    /* child two */

    /* Now open the output file */
    fdout = open (OUTFILE,O_WRONLY | O_TRUNC | O_CREAT, PERMS);
    if ( fdout == −1 ) {
      perror(OUTFILE);            /* outfile opening failed. */
      exit(−1);
    }

    /* dup file descriptors appropriately */
    if ( −1 == dup2(line[READ_END],STDIN_FILENO) ) {
      perror("dup2");           /* dup2() failed */
      exit(−1);
    }
    if ( −1 == dup2(fdout,STDOUT_FILENO) ) {
      perror("dup2");           /* dup2() failed */
      exit(−1);
    }

    /* clean up before exec()*/
    /* clean up before exec()*/
    if ( close(line[READ_END])  ||
         close(line[WRITE_END]) ||
         close(fdout) ) {
      perror("close");          /* as above... */
      exit(−1);
    }

    /* do the exec */
    execlp("sort","sort","-r",NULL);

    /* exec failed (if we're still here...) */
    perror("sort");
    exit(−1);
  }

  /* only parent gets here */

  /* clean up before wait()ing */
  if ( close(line[READ_END])  ||
       close(line[WRITE_END]) ) {
    perror("close");          /* We should probably check each one, but */
    exit(−1);                 /* if close fails we're in deep trouble */
  }                           /* any way you look at it... */

  /* wait for the children to terminate */

  /* one */
  if ( −1 == wait(&status) ) {
    err++;
  } else {
    err += !WIFEXITED(status) || (WEXITSTATUS(status)!=0);
  }

  /* the other... */
  if ( −1 == wait(&status) ) {
    err++;
  } else {
    err += !WIFEXITED(status) || (WEXITSTATUS(status)!=0);
  }

  return err;                 /* nonzero if children failed */
}
```