CSC209F Midterm (L0101)

Fall 1999

University of Toronto

Department of Computer Science

Date: October 26, 1999
Time: 1:10 pm
Duration: 50 minutes

Notes:

1. This is a closed book test, no aids are allowed.
2. Print your name, student number, and cdf username in the space provided below.
3. There are a total of 40 marks.
4. All shell questions assume `csh`.
5. This test is worth 20% of your final course mark.
6. This test comprises 6 pages. Do not detach pages, and answer questions in the blank spaces provided. If you need more space, answer on the back of the pages (clearly identify which question in this case).
7. **Read all questions before starting. Not all questions are worth the same number of marks, so budget your time accordingly. Answer questions in any order you desire.**

Name:	
Student Number:	**MARKING SCHEME**
CDF Username:	

Marks:

Q1	/10
Q2	/8
Q3	/10
Q4	/12
Total	

Q1: [10 marks] (1 mark each unless otherwise indicated)

1. (3 marks) What are the legal return values for `fork()`. Explain (briefly) what each one means.

   -1 : `fork()` unable to successfully create new process
   0 : normal return as seen by child
   PID > 0 : normal return as seen by parent

2. Show how to store the output from the command `ls -l` in a shell variable.

   ```
   set x = `ls -l`
   ```

3. What is the difference between a *program* and a *process*?

   A program is an executable file (script or binary) stored in the filesystem, whereas a process is an executing instance of a program.

4. What is the difference between an absolute and a relative pathname?

   An absolute path name starts with a '/', whereas a relative path name does not.

5. Show how to use I/O re-direction to append the output from `ls` to the file '`myData`'.

   ls >> myData

6. How can you overide aliasing on a one-time only basis. (E.g. "`rm -i`" is aliased to "`rm`", and you wish to execute "`rm`" without the "`-i`" option.)

   Place '\' in front of the command, *e.g.* "`\rm`".

7. What is the difference between a *local shell variable* and an *environment variable*.

   A local shell variable is not passed to `exec()`'d processes, whereas environment variables are.

Q2: [8 marks]

Write a csh shell script to compute the total size of all non-directory files in a directory. The script takes one (optional) parameter, the name of the directory for which the total is to be computed. If no parameter is supplied, the total is computed for the current working directory.

```
#!/usr/bin/csh -f # 1 mark for correct header (-f optional)

if ( $#argv > 1 ) then
  echo Usage: $0 <directory>
  exit 1
end if

if ( $#argv == 1 ) then # 1 mark for dealing with parameter
  set dir = $argv[1]
else
  set dir = "."
end if

# 2 marks for call to ls
set dirInfo = "`ls -alF $dir | grep -v /`"
shift dirInfo # gets rid of header line from ls data

set total = 0
while ( $#dirInfo ) # 2 marks for loop of some sort
  set temp = ( $dirInfo[1] )
  @ total = $total + $temp[4] # 1 mark for doing sum
  shift dirInfo
end

# 1 mark for output of result
echo Total \(non-directory\) bytes in $dir is $total
```

There are different ways to do this. For example, you could pass thr output from grep to get the size field only in `dirInfo`, and just do the sum that way. I don't care which field is extracted from the "`ls -l`" data, so long as the idea of extracting some field is present. Also, most will not have used the "`…`" technique to assign whole lines to each array index.

Q3: [10 marks]

Write a C program named `numDirs.c` to count the number of subdirectories within a given directory. The program takes one optional parameter, the name of the directory for which subdirectories are to be counted. If the parameter is not specified, the subdirectories in the current working directory are counted. The program should ignore "." and ".." while doing its count. Your solution must check the return status of all system calls for errors.

You did not need to specify include files, as stated in class.

```
#include <sys/types.h>
#include <dirent.h>
#include <sys/stat.h>
#include <stdio.h>
#include <errno.h>

int main(int argc, char *argv[])
{
  char          *dirName ;
  DIR           *dir     ; /* 1 mark for declaration */
  struct dirent *dirData ; /* 1 mark for declaration */

  if (argc > 2){
    fprintf(stderr, "Usage: %s <directory>", argv[0]);
    exit(1);
  }

  if ( argc == 2) dirName = argv[1] ; /* 1 mark for arg processing */
  else            dirName = "."     ;

  if (dir = opendir(dirName)){ /* 2 marks for opendir() */
    long        count = 0 ;
    struct stat buf       ;
    while (dirData = readdir(dir)) /* 2 marks for readdir() */
    {
      char filePath[256]; /* let's assume 256 is enough */
      strcpy(filePath, dirName);
      strcat(filePath, "/");
      strcat(filePath, dirData->d_name);
      if (stat(filePath, &buf) != -1) /* 1 mark for stat()/lstat() */
        /* 1 mark for S_ISDIR(), plus calculation */
        { if (S_ISDIR(buf.st_mode)){ count ++ ; }
      else
        fprintf(stderr,"Unable to stat() %s!\n", filePath);
    }
    closedir(dir); /* 1 mark for closedir() */
    printf("Total size = %ld files.\n", count);
  } else {
    fprintf(stderr,"Unable to open %s!", dirName);
    exit(1);
  }
  return 0 ;
}
```

Subtract 2 marks for failing to check return values. I don't think there are many different ways to do this. This solution ignores symbolic links for simplicity's sake.

Q4: [12 marks]

Write the following C program:

`runSafe <progSafe> <prog> [<arg1> [<arg2> … [<argN>] …]]`

`runSafe` creates a child process and uses it to execute `<prog>` with any arguments following `<prog>` on the command line. If `<prog>` terminates normally with status code = 0, then `runSafe` is finished. If `<prog>` terminates normally with status code ≠ 0, then `runSafe` executes `<progSafe>` and reports on its termination status. If `<prog>` terminates abnormally or fails to run at all, `runSafe` reports this but does not run `<progSafe>`. In your solution, you must check the return status of all system calls for errors. It is assumed `<prog>` may reside anywhere in the search path.

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

int main(int argc, char *argv[])
{
  int result, status ;
  if (argc < 3){
    fprintf(stderr,"Usage: %s <progSafe> <prog> {<args>}", argv[0]);
    exit(1);
  }

  switch (fork()) /* 2 marks - create child */
  { case -1: /* error */ /* 1 marks - deal with error */
      fprintf(stderr,"Error: %s: unable to create new process!\n", argv[0]);
      exit(1);
    case  0: /* child */
      execvp(argv[2], argv + 2); /* 1 marks - correct call to execvp() */
      exit(1); /* must have this, in case exec() fails *//* 1 mark - handle exec error */
    default:
      result = wait(&status); /* 1 marks - wait for child */
      if (result == -1){ /* 1 mark - check for error in wait() */
        perror("Error while waiting for child! ");
        exit(1);
      }
      if (WIFEXITED(status)){ /* 1 marks - check status returned by child */
        printf("%s exited with code %d\n", argv[2], WEXITSTATUS(status));
        if (WEXITSTATUS(status)) { /* error */
          switch (fork())/* 2 marks - create child */
          { case -1:
              perror("Unable to create process for <progSafe>\n");
              exit(1);
            case  0: /* child */
              execlp(argv[1], argv[1], (char *)0); /* 1 mark - call to execlp() */
              exit(1); /* 1 mark - handle execlp() failure */
            default:
              result = wait(&status); /* 1 mark - wait for child */
              if (result == -1){ /* 1 mark - check for error in wait() */
                perror("Error waiting for <progSafe>\n");
                exit(1);
              } else {
                if (WIFEXITED(status)) /* 1 mark - check return status */
                  printf("%s exited with status = %d\n", argv[1], WEXITSTATUS(status));
                else
                  printf("%s exited abnormally\n", argv[1]);
              }
          }
        } else printf("%s exited with status = 0\n", argv[2]);
      } else {
```

```
        fprintf(stderr,"%s exited abnormally!\n", argv[2]);
        exit(0);
      }
  }
}
```

This could be done in different ways, for example you might use `if-else` instead of `switch` when invoking `fork()`. Given the way in which command line parameters are set up, it is pretty much necessary to use `execvp()` to handle the execution of `<prog>` (allows for easy use of existing command line parameters), and `execlp()` for the execution of `<progSafe>` since we must pass at least one parameter, `arg[0]`. The "p" versions of `execl/v` are used since we want the system to search the path for us automatically.

It should be noted that the example shown here is complete and tested: you did not need something this complete to get full marks. As stated in class, you did not need to list any of the include files.

Macros & Function Prototypes

I/O

```
char   *fgets(char *s, int n, FILE *stream)
FILE   *fopen(const char *file, const char *mode)
int     close(int fd)
int     fclose(FILE *stream)
int     fflush(FILE *stream)
int     fileno(FILE *stream)
int     fprintf(FILE *stream, const char *format, …)
int     fscanf(FILE *stream, const char *format, …)
int     listen(int soc, int n)
int     open(const char *path, int oflag)
int     sprintf(char *s, const char *format, …)
int     write(int fd, void *buf, int nbyte)
ssize_t read(int fd, void *buf, size_t nbyte)
```

Process Management

```
int   execl(const char *path, char *argv0, …, (char *)0)
int   execle(const char *path, char *argv0, …, (char *)0, const char *envp[])
int   execlp(const char *file, char *argv0, …, (char *)0)
int   execv(const char *path, char *argv[])
int   execve(const char *path, char *argv[], const char *envp[])
int   execvp(const char *file, char *argv[])
int   wait(int &status)
int   waitpid(int pid, int *stat, int options)
int   WIFEXITED(int status)
int   WIFSTOPPED(int status)
int   WIFSIGNALLED(int status)
int   WEXITSTATUS(status)
int   WTERMSIG(int status)
int   WSTOPSIG(int status)
pid_t fork(void)
```

String Handling

```
char *strtok(char *s, const char *delim)
char *strcpy(char *dest, const char *srce)
char *strncpy(char *dest, const char *srce, int count)
int   strlen(const char *s)
int   strcmp(const char *s1, const char *s2)
int   strncmp(const char *s1, const char *s2, int count)
```

Directory Structure

```
DIR           *opendir(const char *filename);
int            closedir(DIR *dirp);
int            lstat(const char *path, struct stat *buf);
int            S_ISDIR(mode);
int            stat(const char *path, struct stat *buf);
long           telldir(DIR *dirp);
struct dirent *readdir(DIR *dirp);
void           rewinddir(DIR *dirp);
void           seekdir(DIR *dirp, long loc);
```

```
struct stat {
 mode_t   st_mode;  /* File mode (see mknod(2)) */
 time_t   st_atime; /* Time of last access */
 time_t   st_mtime; /* Time of last data modification */
 time_t   st_ctime; /* Time of last file status change */
 off_t    st_size;  /* File size in bytes */
 nlink_t  st_nlink; /* Number of links */
 uid_t    st_uid;   /* User ID of the file's owner */
 gid_t    st_gid;   /* Group ID of the file's group */
};
```

```
Struct dirent {
  ino_t            d_ino;
  off_t            d_off;
  unsigned short   d_reclen;
  char             d_name[1];
}
```