



## MODULAR PROGRAMME

### ASSESSMENT SPECIFICATION

#### ***Module Details***

<b>Module Code</b> UFCETY-20-2	<b>Run</b> 10SEP/1 AY	<b>Module Title</b> Operating Systems and Networks Programming
<b>Module Leader</b> Ian Johnson	<b>Module Tutors</b>	
<b>Component and Element Number</b> B1		<b>Weighting: (% of the Module's assessment)</b> 25%
<b>Element Description</b> Coursework 1		<b><u>Total Assignment time</u></b> 12 hours + lab time

#### ***Dates***

<b>Date Issued to Students</b> w/b 01/11/2010	<b>Date to be Returned to Students</b> 20th January 2011
<b>Submission Place</b> <b>PROJECT ROOM - 2Q30</b> (Help Desk open 9.00 - 6.00pm)	<b>Submission Date</b> 9th December 2010
	<b>Submission Time</b> <b>2.00 pm</b>

#### ***Deliverables***

As per attached specification
-------------------------------

#### ***Module Leader Signature***

<i>Ian Johnson</i>
--------------------

## Overview

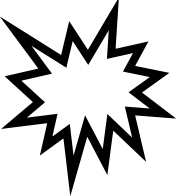
Even in these days of Graphical User Interfaces (GUIs) most modern operating systems still offer a command line interpreter, or shell. Many systems administrators are frequent users of command line interfaces, even on Windows XP!

From an educational point of view, command-line interfaces provide an opportunity to study the underlying operating system calls and to this end you will be developing a simple, restricted shell.

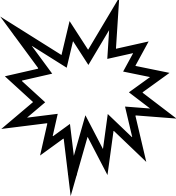
**This assignment will be developed in stages and signed off by your lab tutor as you proceed.**

**You will be expected to work on this during your lab sessions AND in your own time.**

### IMPORTANT!!



For fairness, and to encourage you to work consistently lab tutors have been advised to sign off one category of work at a time for each student. This is to ensure that everyone can be seen in a lab session. No sign-offs will be performed outside of scheduled classes.



**IT IS YOUR RESPONSIBILITY TO ENSURE YOU ALLOW ADEQUATE TIME TO DEMONSTRATE YOUR WORK AND TO DO SO ON A WEEK BY WEEK BASIS**

## Requirements

To develop the following elements, initially as stand-alone programs, and **then to combine them into a simple shell**. It will pay to think ahead and to consider functions that will be common to all/many of the stand-alone versions. Your shell should provide a prompt, error/usage messages for the built-ins and pass other commands to the underlying system for execution.

### Note:

- Credit will be awarded for the use of version control. A worksheet on using version control software is available on Nigel Gunton's web page.
- Credit will be awarded for robust error checking and the use of *perror()*

The elements required for this assignment together with the available marks are listed overleaf:

**a. pwd**

This should print, on **stdout**, the path to the current directory.

( 2 marks )

**b. cd**

This should take an optional path as an argument. If no argument is provided then the default behaviour is to change directory to the users home directory.

( 3 marks )

**c. ps**

Default behaviour is to list all processes owned by the user. It should accept the flag **-A** as an argument and list all current processes and their process Id's .

( 25 marks )

**d. kill**

This command should respond as follows :

- kill **pid** send **SIGTERM** to process **pid**
- kill -l list the signals sent by this command. Your version of kill should recognise SIGTERM, SIGKILL and SIGHUP. It should provide a list of both the names of the signals and their numbers. **RTFM section 7 signal.**
- kill **signal pid(s)** send the specified signal to pid(s). It should recognise both the numeric value and the name of the signals.

( 5 marks )

**e. A basic shell**

<http://www.cs.ucsd.edu/classes/wi97/cse80/dumbshell.html> provides a very simple shell. This is for guidance only. Your shell should at a minimum provide the capability to support executing commands. You **MUST** develop your own shell, for which credit will be given.

( 25 marks )

**f. Integration of [a-e] within f**

Your shell should have the commands you have implemented as built-ins. These should execute as functions of the shell, not as new processes

( 10 marks )

**g. Piping and redirection.**

The Hawklord shell walks you through the process of generating pipelines. Think carefully about your design particularly with respect to pipelining built-in commands.

( 15 marks )

## h. Extras!

A further 15% is available, at the discretion of the markers, for outstanding work. These marks will be awarded for high quality original code, robust code and extensions to your shell.

Version control will also gain credit under this heading.

Examples of work which could be worth extra credit would be adding functionality such as simple control structures ( if, while, for ), filename completion or history to your shell. If you wish to extend your work consult your lab tutor.

(up to 15 marks)

## Constraints

1. The marks shown above are the maximum marks available for each category. Working code alone is not adequate; your work is expected to demonstrate professionalism through adherence to appropriate coding standards, robust error checking, the use of static analysis, the use of version control etc.
2. All code **MUST** be demonstrated and explained to your lab tutor before it will be signed off. All code must be your own or the source acknowledged. Only your own code will attract credit.

Remember, **this is an individual assignment** and that assessment offences are taken seriously. This does not prevent you from discussing problems and ideas with your peers and you are encouraged to do so as long as the final result is your own work. If you use sections of code from other sources then they must be clearly identified and you will be expected to demonstrate your understanding of the code to your lab tutor.

All work **MUST** be demonstrated before the hand-in date, in lab session time. **Do not expect to turn up at the last lab and demonstrate everything.**

3. Support for system calls

Possibly the best advice is Read The Friendly Manual ☺ All system calls are documented in section 2 of the manual. All C library functions are documented in section 3. ***man 2 syscalls*** will give you a list of system calls.

System calls will be covered in Nigel Guntons lectures. There is also a very good web-site that covers much of the assignment material. (see the link from Nigel's home page).

## Other Resources

### WWW

Nigel Gunton's homepage (<http://www.cems.uwe.ac.uk/~ngunton>) has unix/linux system programming related links.

The linux documentation project, has too many resources to list!

### Dead Trees

Stevens, Richard W;                    ***“Advanced Programming in the Unix® Environment”***,  
Addison-Wesley, 1993.

Rochkind, Marc J;                    ***“Advanced Unix Programming”***,  
Prentice Hall, 1985.

These are both excellent system level programming reference guides. Both of these books are not cheap, but provide a professional level reference that will last you through many years of your career.

Alternatively, the library is an excellent place to discover books!

### Your lab tutor

These are often worth talking to ☺ and can provide support for C syntax etc.

## Deliverables

Your sign-off sheet, signed and dated for all completed work.

Signed off code **DOES NOT INDICATE THAT MAXIMUM MARKS HAVE BEEN ACHIEVED**

Copies of all code that has been demonstrated/explained to your tutor. This code **MUST** be signed off by your tutor.

Student Number:

<b>UFCETY-20-2</b>	<b>Assignment 1 Checklist</b>		
	<b>Comments</b>	<b>Signed</b>	<b>Date</b>
pwd			
cd			
ps			
kill			
shell			
integration			
pipng & redirection			
Extras!			

Other comments:

# dumbshell.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define DEBUG 1
#define MAXLINELEN 4096
#define MAXARGS 128
#define END_OF_LINE 0
#define SEQ_OP ';'
#define SEQUENCE 1

struct cmd {
    struct cmd *next;
    int terminator;
    char *exe_path;
    int nargs;
    char *arg[MAXARGS];
};

void *ck_malloc(size_t size)
{
    void *ret = malloc(size);
    if (!ret) {
        perror("dumbshell:ck_malloc");
        exit(1);
    }
    return ret;
}

char *skip_to_non_ws(char *p)
{
    int ch;
    while (ch = *p) {
        if (ch != ' ' && ch != '\t' && ch != '\n') return p;
        ++p;
    }
    return 0;
}

char *skip_to_ws_or_sep(char *p)
{
    int ch;
    while (ch = *p) {
        if (ch == ' ' || ch == '\t' || ch == '\n') return p;
        if (ch == SEQ_OP) return p;
        ++p;
    }
    return 0;
}

struct cmd *parse_commands(char *line)
{
    char *ptr;
    int ix;
    struct cmd *cur;

    ptr = skip_to_non_ws(line);
    if (!ptr) return 0;
    cur = ck_malloc(sizeof *cur);
    cur->next = 0;
    cur->exe_path = ptr;
    cur->arg[0] = ptr;
    cur->terminator = END_OF_LINE;
    ix = 1;
    for (;;) {
        ptr = skip_to_ws_or_sep(ptr);
        if (!ptr) {
            break;
        }
        if (*ptr == SEQ_OP) {
            *ptr = 0;
            cur->next = parse_commands(ptr+1);
            if (cur->next) {
                cur->terminator = SEQUENCE;
            }
            break;
        }
        *ptr = 0;
        ptr = skip_to_non_ws(ptr+1);
    }
}
```

```

        if (!ptr) {
            break;
        }
        if (*ptr == SEQ_OP) {
            /* found a sequence operator */
            cur->next = parse_commands(ptr+1);
            if (cur->next) {
                cur->terminator = SEQUENCE;
            }
            break;
        }
        cur->arg[ix] = ptr;
        ++ix;
    }
    cur->arg[ix] = 0;
    cur->nargs = ix;
    return cur;
}

void execute(struct cmd *clist)
{
    int    pid, npid, stat;

    pid = fork();
    if (pid == -1) {
        perror("dumbshell:fork");
        exit(1);
    }
    if (!pid) {
        /* child */
        execvp(clist->exe_path,clist->arg);
        fprintf(stderr,"No such command: %s\n",clist->exe_path);
        exit(1);
    }
    do {
        npid = wait(&stat);
        printf("Process %d exited with status %d\n",npid,stat);
    } while (npid != pid);
    switch (clist->terminator) {
    case SEQUENCE:
        execute(clist->next);
    }
}

void free_commands(struct cmd *clist)
{
    struct cmd *nxt;

    do {
        nxt = clist->next;
        free(clist);
        clist = nxt;
    } while (clist);
}

char *get_command(char *buf,
                  int size,
                  FILE *in)
{
    if (in == stdin) {
        fputs("@ ",stdout); /* prompt */
    }
    return fgets(buf,size,in);
}

void main(void)
{
    char    linebuf[MAXLINELEN];
    struct cmd *commands;

    while (get_command(linebuf,MAXLINELEN,stdin) != NULL) {
        commands = parse_commands(linebuf);
        if (commands) {
            execute(commands);
            free_commands(commands);
        }
    }
}

```