CMPT 300 — Operating Systems I

Summer 1999

Segment 2:

Unix Basics

Melissa O'Neill

Unix Basics — Compiling and Using

You had to learn how to do the basics on a Unix system, including:

- Look up a manual page on a Unix system
  - man fork
  - man -s 2 intro
- Compile a C file with gcc
  - *unix%* cat > hello.c
    ```
    #include <stdio.h>

    int main(int argc, char **argv) {
        printf("Hello World\n");
        return 0;
    }
    ```
    unix% gcc -Wall -o hello hello.c

    unix% hello
    Hello World

  - *unix%* cat > hello.c
    ```
    #include <stdio.h>

    int main(int argc, char **argv) {
        puts("Hello World");
        return 0;
    }
    ^D
    ```
    unix% gcc -Wall -o hello hello.c

    unix% hello
    Hello World

    unix% truss hello
    ```
    ioctl(1, TCGETA, 0xEFFFEA3C)            = 0
    write(1, "H e l l o  W o r l d\n", 12)    = 12
    Hello World
    llseek(0, 0, SEEK_CUR)                  = 693
    _exit(0)
    ```


- Compile a C file with gcc (*continued*)
  - *unix%* cat > hello.c
    ```
    #include <unistd.h>

    int main(int argc, char **argv) {
        write(1, "Hello World\n", 12);
        return 0;
    }
    ^D
    ```
    unix% gcc -Wall -o hello hello.c

    unix% hello
    Hello World

    unix% truss hello
    ```
    write(1, "H e l l o  W o r l d\n", 12)    = 12
    Hello World
    _exit(0)
    ```

- Compile a C++ file with g++
  - *unix%*␣cat > bye.cc

```
#include <iostream>

int main(int argc, char **argv) {
    cout << "Goodbye, cruel world..." << endl;
    return 0;
}
^D
```

```
unix%␣g++ -Wall -o bye bye.cc
```

```
unix%␣bye
Goodbye, cruel world...
```


Unix Basics — The Shell

A *shell* is a *front end* to a system. Usually Unix shells are text-based programs.

Neat things to *do* with the shell:

```
unix%␣echo Hello World
Hello World
```

```
unix%␣who
csilop    pts/0  May 11 08:34  (maia.cs)
meoneill  pts/1  May 13 21:26  (geranium.css)
```

```
unix%␣who am i
meoneill  pts/1  May 13 21:26  (geranium.css)
```

```
unix%␣pwd
/gfs1/meoneill
```

```
unix%␣mkdir cmpt300
```

```
unix%␣cd cmpt300
```



```
unix%␣pwd
/gfs1/meoneill/cmpt300
```

```
unix%␣cd
```

```
unix%␣pwd
/gfs1/meoneill
```

```
unix%␣grep SETUP_GNU .CSIL.rc
setenv SETUP_GNU          YES  # FSF Software
```

```
unix% wc < .CSIL.rc
      56     353    2565
```

```
unix%␣grep SETUP .CSIL.rc                              \
               | perl -pe 's/#.*//'                     \
               | perl -pe 's/.*SETUP_//'                \
               | perl -pe 's/\s+/ /'                    \
               | sort                                   \
               > setup.mine
```

```
unix% echo Hello `whoami`
Hello meoneill
```


Unix Basics — Processes and Files

The two most important concepts:

- Process — a program that is being executed
- File — a named repository for data

Unix Basics — Processes

Two fundamental operations

- fork — create a clone of the current process
- execv/execve — replace the program part of the process with another program

```
#include <iostream>          // declaration of cerr & <<
#include <unistd.h>          // declaration of fork

int main(int argc, char **argv) {
    const char *const args[] = { "echo", "Hello World", 0 };
    execv("/bin/echo", args);
    std::cerr << "execv failed!" << endl;
    return -1;
}
```


Unix Basics — forktest1.cc

```
#include <iostream>          // declaration of cout, cerr & <<
#include <unistd.h>          // declaration of fork, getpid & getppid

int main(int argc, char **argv) {
    const int childpid = fork();
    if (childpid > 0) {                                  // in parent
        std::cout << "Process " << getpid()
            << ": created child with id " << childpid << endl;
    } else if (childpid == 0) {                          // in child
        std::cout << "Process " << getpid()
            << ": parent is " << getppid() << endl;
    } else {
        std::cerr << "fork failed" << endl;
    }
    std::cout << "Process " << getpid() << ": exiting." << endl;
    return 0;
}
```



```
unix% forktest1
Process 24994: created child with id 24995
Process 24995: parent is 24994
Process 24995: exiting.
Process 24994: exiting.

unix% forktest1
Process 25013: parent is 25012
Process 25013: exiting.
Process 25012: created child with id 25013
Process 25012: exiting.

unix% forktest1
Process 25024: parent is 25023
Process 25023: created child with id 25024
Process 25024: exiting.
Process 25023: exiting.

unix% forktest1
Process 25031: parent is 25030
Process 25030: created child with id 25031
Process 25030: exiting.
unix% Process 25031: exiting.
```


fork — I lied!

Here's what actually happened every time I ran the program on my workstation at home:

```
unix% forktest1
Process 25039: created child with id 25040
Process 25039: exiting.
Process 25040: parent is 1
Process 25040: exiting.

unix% forktest1
Process 25041: created child with id 25042
Process 25041: exiting.
unix% Process 25042: parent is 1
Process 25042: exiting.

unix% forktest1
Process 25043: created child with id 25044
Process 25043: exiting.
unix% Process 25044: parent is 1
Process 25044: exiting.
```

fork — I lied! (contd.)

But here's what happened when I ran the program on a four processor SPARCStation:

```
unix% forktest1
Process 23827: created child with id 23828
• Process 23828: parent is 23827
Process 23827: exiting.
• Process 23828: exiting.

unix% forktest1
Process 23829: created child with id 23830
• Process 23830: parent is 23829
Process 23829: exiting.
• Process 23830: exiting.
```

Class Exercise: Why do we see different results on different kinds of machine.

Why does my computer say the parent is process 1?

Unix Basics — forktest2.cc

```
#include <iostream>          // declaration of cout, cerr & <<
#include <unistd.h>        // declaration of fork, getpid & getppid

int main(int argc, char **argv) {
    const int childpid = fork();

    if (childpid > 0) {                                    // in parent

        std::cout << "Process " << getpid()
            << ": created child with id " << childpid << endl;
        wait(0);        // <————— this added line is the only change

    } else if (childpid == 0) {                            // in child

        std::cout << "Process " << getpid()
            << ": parent is " << getppid() << endl;

    } else {
        std::cerr << "fork failed" << endl;
    }

    std::cout << "Process " << getpid() << ": exiting." << endl;
    return 0;
}
```


fork with wait — running forktest2

With wait, the program behaves more consistently…

```
unix% forktest2
Process 25059: created child with id 25060
• Process 25060: parent is 25059
• Process 25060: exiting.
Process 25059: exiting.

unix% forktest2
Process 25061: created child with id 25062
• Process 25062: parent is 25061
• Process 25062: exiting.
Process 25061: exiting.

unix% forktest2
Process 25064: parent is 25063
• Process 25064: exiting.
• Process 25063: created child with id 25064
Process 25063: exiting.
```


Unix Basics — filecopy.cc (main)

```
#include <unistd.h>
#include <fcntl.h>

void copyout(const int in_fd, const int out_fd) {
    ;                        // we'll write code for this function shortly
}

int main(int argc, char **argv) {
    const char *const in_name = argv[1];
    const char *const out_name = argv[2];

    const int in_fd = open(in_name, O_RDONLY);
    const int out_fd =
        open(out_name, O_WRONLY | O_CREAT | O_TRUNC, 0666);

    copyout(in_fd, out_fd);                                // copy the file

    close(in_fd);
    close(out_fd);
}
```

Unix Basics — filecopy.cc (err_sys)

```
#include <iostream>
#include <string.h>
#include <errno.h>

void err_sys(const char *const error_message)  {
    std::cerr << error_message << " (" << strerror(errno) << ")"
        << endl;
    exit(1);
}
```


Unix Basics — filecopy.cc (main)

```
int main(int argc, char **argv) {
    if (argc != 3)
        err_sys("Need exactly two arguments");
    const char *const in_name = argv[1];
    const char *const out_name = argv[2];

    const int in_fd = open(in_name, O_RDONLY);
    if (in_fd < 0)                                  // open failed, die
        err_sys("Can't open input file");

    const int out_fd =
        open(out_name, O_WRONLY | O_CREAT | O_TRUNC, 0666);
    if (out_fd < 0)                                 // open failed, die
        err_sys("Can't open output file");

    copyout(in_fd, out_fd);                         // copy the file

    close(in_fd);
    if (close(out_fd) < 0)
        err_sys("Error closing output file");

    return 0;
}
```


Unix Basics — filecopy.cc (main, alternative)

```
int main(int argc, char **argv) {
    if (argc != 3)
        err_sys("Need exactly two arguments");
    const char *const in_name = argv[1];
    const char *const out_name = argv[2];

    const int in_fd = Open(in_name, O_RDONLY);
    const int out_fd =
        Open(out_name, O_WRONLY | O_CREAT | O_TRUNC, 0666);

    copyout(in_fd, out_fd);                         /* copy the file */

    Close(in_fd);
    Close(out_fd);
    return 0;
}

void Close(int fd) {
    const int retval = close(fd);
    if (retval < 0)
        err_sys("Close failed!");
}
```


Unix Basics — filecopy.cc (copyout)

```
const int BUFFER_SIZE = 4096;

//  copyout assumes the file descriptors are in blocking mode
//  (i.e., weren't opened with O_NONBLOCK)

void copyout(int in_fd, int out_fd) {
    char buffer[BUFFER_SIZE];

    for( ; ; ) {
        const int read_count = read(in_fd, buffer, BUFFER_SIZE);
        if (read_count < 0)                         // Read failure
            err_sys("Read failed");
        else if (read_count == 0)
            return;                                 // End of file — return

        const int write_count = write(out_fd, buffer, read_count);
        if (write_count != read_count)              // Didn't write it all
            err_sys("Write failed (or partial write only)");
    }
}
```

Segment Review

You should be able to write programs using:

- fork and exec
- open and close
- read and write