

Programing with Multiple Processes in C

fork, wait, execlp, file operations, and make

Assignment Information

- Four executables will be needed
 - **Initiator** – Main program, that opens, reads the characters and closes the file, forks child processes.
 - **Pell, Composite, Total**

Outline

- Learn how to use the following
 - `fork()`
 - `wait()`
 - `execlp()`
 - file operations
 - make

fork()

-
- Generates an exact copy of parent process except for the value it returns
 - In a child process, fork() returns zero
 - In the parent process it will return the child's process ID
 - If return value is -1, then fork() failed.
 - Any process can retrieve its process ID with getpid(), and its parent process ID with getppid()
 - Syntax:
 - `pid_t fork();`


```

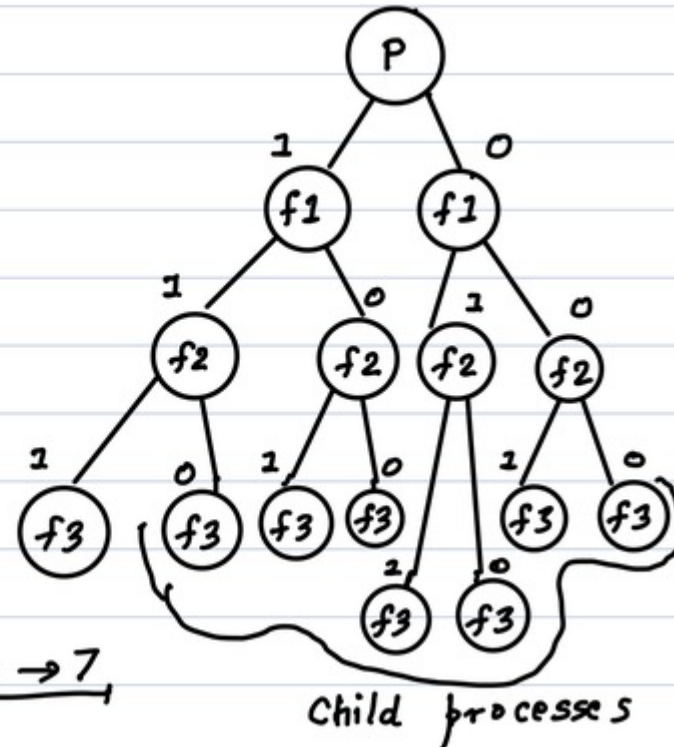
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
fork();
fork();
fork();
printf("hello\n");
return 0;
}

```

fork(); $\rightarrow f_1$
fork(); $\rightarrow f_2$
fork(); $\rightarrow f_3$

Parent $\rightarrow 1$
child $\rightarrow 0$

Total child process $\rightarrow 7$



wait()

- Makes parent process wait until the child has been entirely executed
- Use `WIFEXITED()` to check whether child process has terminated normally, as opposed to dying with a signal
- Use `WEXITSTATUS()` to retrieve return value of child process
- Syntax: `pid_t wait(int *stat_loc);`

execvp()

- Executes a new program within a child process
- Arguments passed - the name of the executable and filename like `"./Starter"`, `"Starter"`
- Also pass any needed command line arguments as parameters
- Terminate list of arguments with `NULL`
- Syntax
 - `int execvp("./executable_path", "program_name", const char *arg, ..., NULL);`
- Regard the arguments as program followed by `argv[]`.

File Operations

- We need several functions for this assignment.
- They are:
 - `fopen()`
 - `fclose()`
 - `fgets()` or `fgetc()`

fopen()

- Used to open a file, whose name is given as the argument.
- It returns a pointer to the opened file.
- Syntax:
 - `FILE * fp = fopen(const char *filename, const char *mode)`

fclose()

- Closes the stream to the file.
- Buffers are flushed.
- Syntax
 - `int fclose(FILE *stream)`

fgets()

- Reads a line from a file
- Puts the line into the provided array/string
- Syntax:

```
int fgets(char *s, int size, FILE *stream)
```

- Use:

```
char buf[256];  
while (fgets(buf, sizeof(buf), in)  
    // deal with the string in buf
```


Why use make?

- Enables developers to easily compile large and complex programs with many components.
- Situation: There are thousands of lines of code, distributed in multiple source files, written by many developers and arranged in several sub-directories. This project also contains several component divisions and these components may have complex inter-dependencies.

Demo Makefile

Simple version

CC = gcc

.c.o:
\$(CC) -o P1.c P2.c P3.c p4.c

default: all
all: p1 p2 p3 p4

package:
zip -r Mohit.zip p1.c p2.c p3.c p4.c Makefile
input.txt

clean:
rm -f *.o *~ p1 p2 p3 p4

Variable assignments in make

- By convention, predefined variable names used in a Makefile are in upper case, and user-defined variables are lower case.

Example: `CC = gcc`

- We can use the value assigned later on as `$()`

Example: `$(CC)`

Makefile Structure

- Makefile contains definitions and rules.
- A definition has the form:

VAR = value

- A rule has the form:

Output files: input files

<tab>Commands to turn inputs to outputs

- All commands must be tab-indented. Spaces don't work!
- The make <target> command executes the rule with the <target>. If target not is specified, it defaults to the first rule defined in the Makefile.

Patterns and Special variables

- % : Wildcard pattern-matching, for generic targets.
- \$@ : Full target name of the current target.
- \$? : Returns the dependencies that are newer than the current target.
- \$* : Returns the text that corresponds to % in the target.
- \$< : Name of the first dependency.
- \$^ : Name of the first dependency with space as the delimiter.

Demo Makefile

What we're trying to create:

C_SRCS = p1.c p2.c p3.c p4.c

Compiler command & flags

-g : generate debug info.

-Wall: enable many compiler warnings

CC = gcc

CFLAGS = -std=c11 -g -Wall -c -l

Default (first) target is the executables
build: \$(programs)

Create Fibb from LowAlpha.c:

Fibb: Fibb.c

\$(CC) \$(CFLAGS) \$(LDFLAGS) -o \$@ \$?

Actually, make already knows how to create
a program from a .c file, so the rule above
was unnecessary.

Clean up the directory

clean:

rm -f *.o *~ \$(programs)

Demo program output

\$./Initiator input.txt

Initiator[735610]: Forked process with ID 735614.

Initiator[735610]: Waiting for Process [735614].

Pell[735614] : Number of terms in Pell series is 15

Pell[735614] : The first 15 numbers of the Pell sequence are:

0, 1, 2, 5, 12, 29, 70, 169, 408, 985, 2378, 5741, 13860, 33461, 80782,

Initiator: Child process 735614 returned 142.

Initiator[735610]: Forked process with ID 735615.

Initiator[735610]: Waiting for Process [735615].

Composite[735615]: First 15 composite numbers are:

4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 22, 24, 25,

Initiator: Child process 735615 returned 25.

Initiator[735610]: Forked process with ID 735616.

Initiator[735610]: Waiting for Process [735616].

Total[735616] : Sum = 120

Initiator : Child process 735616 returned 120.

Pell: 142

Composite: 25

total Count: 120

Thank You

Acknowledgements

- These slides are based on contributions of current and past CS370 instructors and TAs, including J. Applin, A. Yeluri, Y. K. Malaiya, Phil sharp and S. Pallickara.