THE UNIVERSITY OF SYDNEY

COMP2017 / COMP9017 Week 8 Tutorial

Introduction to processes

Introduction to processes

Memory layout of a process

A process memory layout is typically made of (from lowest memory address to highest):

- Program text (Program instructions!) (.text)
- Initialised Global and Static Variables (Data Segment) (.data)
- Uninitialised global and static variables (.bss)
- Heap Memory (from low to high)
- Memory mapped .data and .text regions (Shared)
- Stack Memory (from high to low)
- Environment and args

To examine the memory layout of your program you can use the `readelf` command that will allow you to examine the section headers of your executable.

```
readelf --section <your compiled program>
```

You can examine your executable by using the program objdump to grab the disassembly of `.text`

```
objdump -D -M intel <your compiled program>
```

Output from `hello_world`

```
...
40050d:         c7 45 fc 04 00 00 00 mov   DWORD PTR [rbp-0x4],0x4
400514:         c7 45 ec 00 00 00 00 mov   DWORD PTR [rbp-0x14],0x0
40051b:         eb 13                jmp   400530 <main+0x43>
...
```

Using `fork()`

The fork function is used when creating a new child process. This function will duplicate the current calling process and it's current execution context and allocate separate memory. Pointers and variables that are written to will eventually be written to a new memory space via CoW (Copy-on-Write). A new pid is generated for the sub process that can be isolated from the parent process.

Quote from man pages:

```
fork()  creates  a new process by duplicating the calling process.  The
       new process is referred to as the child process.
       The calling  process is referred to as the parent process.
```

Fork usage example:

```c
#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>
#include<sys/types.h>

int main(void) {

  int n = 4;
  puts("about to fork");
  pid_t pid = fork();

  if (pid < 0) {
    perror("unable to fork");
    return 1;
  }

  if (pid == 0) {
    puts("child");
    n++;
    // sleep(1);
  } else {
    puts("parent");
    n *= 2;
    // wait(NULL);
  }

  printf("fork returned %d, n is %d\n", pid, n);
  return 0;
}
```

`wait` and `waitpid`

The `wait` or `waitpid` system call will halt a process until another process has finished executing. The wait function will return the `pid` value of the child process.

```
pid_t pid = fork();
...

wait(NULL);
//alternatively waitpid(pid, NULL, 0);
```

The `wait` will implicitly wait for one of its children finishes execution while `waitpid` will explicitly wait for a process to finish with a specified pid.

Question 1: Forking processes

UNIX systems use the `fork()` system call to allow existing processes to create new ones. The new process created by fork is called the child process, its creator is the *parent process*. fork is called once, but returns different values depending on which process you are in. The parent receives the *process ID* of the child, and the child receives zero. The return value of fork is the simplest way to distinguish between parent and child.

Both the child and parent process continue executing the instructions following the call to fork. The child is a copy of the parent, meaning the child gets a copy of the parent's stack, heap, and data space. In this exercise, we will investigate how to use `fork()`. The headers in `unistd.h` and `sys/*.h` are included as part of the POSIX standard and rather than the C standard library.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int main(void) {

  int n = 4;

  puts("about to fork");
  pid_t pid = fork();

  if (pid < 0) {
    perror("unable to fork");
    return 1;
  }
  if (pid == 0) {
    puts("child");
    n++;
```

```
    // sleep(1);
  } else {
    puts("parent");
    n *= 2;
    // wait(NULL);
  }

  printf("fork returned %d, n is %d\n", pid, n);
  return 0;
}
```

- Compile and run the program a few times. Do the lines always get outputted in the same order?
- Uncomment the sleep function so the child sleeps for a second, and try running the program.
- Uncomment the wait function so the parent waits for the child to exit, and try running the program.

Question 2: Signal Talk between Parent and Child

Using the scaffold from the previous exericise, extend your the program to allow for communication between the child process and the parent process. In this instance, the processes will communicate using signals, with the following signal handler behaviour.

The parent program should handle the following signals:

- SIGINT -> The child wants to know the time

The child program should handle the following signals:

- SIGINT -> Child program should output: `"No!"`

Allow the child, once it has been forked to ask the parent for the time. The parent should output the time, followed by "Have you finished your homework".

The parent will know the pid of the child when forked and you will need to also ensure that the child knows its parents pid.

Once a fork has been executed, the two processes will communicate via signals.

Example with output and actions annotated.

```
$ ./proc_talk

What's the time? <-- Child outputs and sends signal to parent
The time is 5:37am <-- Parent output the time
Have you finished your homework? <--- Out puts the next line and sends a si
No!
<program ends>
```

Please look at the `man` pages for `waitpid`, `getpid`, `kill` and `pause`

`exec()` family calls

The exec family of functions allow a process to start an executable file. When a process calls exec, it is replaced in memory by the new process. If we want to start a new process and keep the current one running, we need to call fork before calling exec. The following code uses exec to invoke the sort program located in /usr/bin/sort to sort the lines in the previous program, forkdemo.c. exec is given the path to a program to execute, then the contents of argv[] terminated by a null pointer. There are many variations of exec documented in the man pages.

```c
#include <stdio.h>
#include <unistd.h>

int main(void) {
  puts("about to launch sort");

  if (execl("/usr/bin/sort", "sort", "forkdemo.c", NULL) == -1) {
    perror("exec failed");
    return 1;
  }

  puts("finished sort");
  return 0;
}
```

Question 3: Miniterminal

Shells! You have interacted with your computer's shell since beginning this course, however we will be exploring how your shell is able to execute commands and manage them.

Although we are not expecting you to completely recreate `bash`, however using your knowledge of fork and exec, create a simple shell that will ask for a command to be inputted and then, if the program exists, attempt to execute the specified program that exists in /usr/bin.

Usage:

```
<command/program> <arguments>
```

Your miniterminal application will need to pass in command line arguments when initialised. Your terminal program should parse the command line and split them by the spaces as defined and be fault tolerant.

Example:

```
cat file.txt
```

For this task you will need to use `fork` and `exec`. Once a command has been inputted by the user, your terminal must fork the current process and then proceed to call exec with the command and arguments provided.

Note: Start with getting a simple command launching before adding arguments. Once you have a separate process launched, you can focus on launching a command with arguments.

Question 4: Controlling Processes

After you have written a simple miniterminal that can execute a processes, we will extend this to maintain a list or (some data structure of your choosing) of running processes and using the `waitpid` function, after a command is executed you can check to see if all processes that have been executed by miniterminal are still running.

If a process is no longer running, it is the job of your terminal to remove the process from your table.

This kind of functionality is aimed at maintaining background processes when using `&`.

For example: Example:

```
cat file.txt &
```

If the input provided by the user contains an &, your program should not provide any focus for this process.

You can observe this behaviour in your own termainl window by executing a process with &.

It is advised you read `waitpid` man page for information on how to retrieve the status without blocking.

popen and pclose

In a similar manner to open and close we are able to run a command and in turn we are provided a `FILE*` object that we can read from or write to (not both however as the object is a connection to stdin or stdout pipes). We can retrieve output using `stdout` from the specified process and pipe it into our own program.

```
FILE* f = popen("<program>", )
...
fread(...)
...

//Close it in similar fashion
pclose(f);
```

Question 5: Log output from a program

Using `popen` and `pclose` create a process logger. A process logger will launch a program and extract the output from that program to a file specified.

```
./proc_log --cmd="python3 say_hello_100.py" --out=test.out
```

With the provided command and out flags, your program should execute the program with the specified command and pipe its output to a file.

Something fun: Use the socat command and create a channel between from your process logger and use the cat command to read the other end.

Example use for `socat`

```
socat pty,raw,echo=0,link=./program_input pty,raw,echo=0,link=./program_out
```

Question 6: Reading a linux executable (Extension)

You are tasked with reading the data of program executable, this program can be the program you are currently writing to read a program executable.

Specification of a linux executable

```
typedef struct elf64_hdr {
        unsigned char          e_ident[EI_NIDENT];
        Elf64_Half e_type;
        Elf64_Half e_machine;
        Elf64_Word e_version;
        Elf64_Addr e_entry;
        Elf64_Off e_phoff;
        Elf64_Off e_shoff;
        Elf64_Word e_flags;
        Elf64_Half e_ehsize;
        Elf64_Half e_phentsize;
        Elf64_Half e_phnum;
        Elf64_Half e_shentsize;
        Elf64_Half e_shnum;
        Elf64_Half e_shstrndx;
} Elf64_Ehdr;
```

Question 7: Break your compiled program (Extension)

Using `nm` or `readelf` and `objdump`, you will change the regularly printed out value of your code to be something entirely different.

Constructing a `hello_number` application you can change the register or static variable's data so it will print out the number `7`.

References

Chapter 9, Page 839-843