Process Management
(using the kernel API)

---

1. Consider the following program that calls function `fork()` multiple times. Compile it and run it. How many processes, including the parent process, are created? Why?

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

int main(int argc, char* argv[]) {
  fork();
  fork();
  fork();
  exit(EXIT_SUCCESS);
}
```

Check your guess by changing the program in such a way that all processes print their process ids (`PID`). Suggestion: check function `getpid()`.

2. Consider also this other program that also calls `fork()` repeatedly. Compile it and run it. How many processes, including the parent process, are created? Why?

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

int main(int argc, char* argv[]) {
  for (int i = 0; i < 4; i++)
    fork();
  exit(EXIT_SUCCESS);
}
```

Again, check your guess by changing the program in such a way that all processes print their process ids (`PID`).

3. Consider now the following program that, when executed, creates a child process. How do you explain the value of variable `value` in the parent and child processes? Suggestion: make a drawing of their respective address spaces as the parent runs and the `fork()` is executed. What happens to the variable then?

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

int main(int argc, char* argv[]) {
  int value = 0;

  pid_t pid = fork();
  if (pid == -1) { perror("fork"); exit(EXIT_FAILURE); }

  if (pid == 0) {
    /* child process */
    value = 1;
    printf("CHILD: value = %d, addr = %p\n", value, &value);
    exit(EXIT_SUCCESS);
  }
  else {
    /* parent process */
    int retv = waitpid(pid, NULL, 0);
    if (retv == -1) { perror("waitpid"); exit(EXIT_FAILURE); }
    printf("PARENT: value = %d, addr = %p\n", value, &value);
    exit(EXIT_SUCCESS);
  }
}
```

Observe the values and addresses of the variable `value` printed by the parent and child processes. Can you explain the results?

Note: the function `perror()` prints the string provided as its argument followed by a string with the cause of the error as determined by an internal variable `errno` whose value is set by the kernel before returning from the system call with a value of `-1`.

4. Consider the following program that, when executed, creates a child process that then executes a command provided in its command line arguments. Compile it and run it. Pay close attention to the code and understand how it works.

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

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main(int argc, char* argv[]) {
  /* fork a child process */
  pid_t pid = fork();
  if (pid == -1) {
    perror("fork");
    exit(EXIT_FAILURE);
  }
  if (pid == 0) {
    /* child process */
    int retv = execlp(argv[1],argv[1],NULL);
    if (retv == -1) { perror("execlp"); exit(EXIT_FAILURE); }
  }
  else {
    /* parent process */
    int retv = waitpid(pid, NULL, 0);
    if (retv == -1) { perror("waitpid"); exit(EXIT_FAILURE); }
  }
  exit(EXIT_SUCCESS);
}
```

If the function `execlp` executes successfully, how does the child process signal its end to the parent process?

5. The following program implements a very simple command line shell. Compile it and run it. Pay close attention to the code and understand how it works.

```
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>

int main(int argc, char* argv[]) {
  for( ; ; ) {
    /* give prompt, read command and null terminate it */
    fprintf(stdout, "$ ");
    char  buf[1024];
    char* command = fgets(buf, sizeof(buf), stdin);
```

```
    if(command == NULL)
      break;
    command[strlen(buf) - 1] = '\0';
    /* call fork and check return value */
    pid_t pid = fork();
    if(pid == -1) {
      perror("fork");
      exit(EXIT_FAILURE);
    }
    if(pid == 0) {
      /* child */
      int retv = execlp(command, command, (char *)0);
      if(retv == -1) {
        perror("execlp");
        exit(EXIT_FAILURE);
      }
    }
    /* shell waits for command to finish before giving prompt again */
    int retv = waitpid(pid, NULL, 0);
    if (retv == -1) {
      perror("waitpid");
      exit(EXIT_FAILURE);
    }
  }
  exit(EXIT_SUCCESS);
}
```

Why can't you execute commands with arguments, e.g., `ls -l` or `uname -n`, with this code? Change the program to implement a command `exit` that terminates the shell.

6. Change the previous program so that the commands can be executed with arguments. Suggestion: use function `execvp` (instead of `execlp`) and use `strtok` to break the `command` string into the name of the command and the arguments.

7. Change the previous program so that it keeps a history of all commands it executed. Implement a command `myhistory` that gets an integer `n` and prints the last `n` commands executed by the shell. Suggestion: take advantage of the `tail` Bash shell command.

8. The following program - `par` - provides a basic template that will help you solve the subsequent exercises.

```
/* complete ... */
```

```
int main(int argc, char* argv[]) {
    for(int i = 1 ; i < argc ; i++) {
       pid_t pid = fork();
       if( pid == 0 ) {
          printf("%d: %s\n", getpid(), argv[i]);
          exit(EXIT_SUCCESS);
       }
    }
    for(int i = 1 ; i < argc ; i++)
       wait(NULL);
    exit(EXIT_SUCCESS);
}
```

Compile it and execute it as follows:

```
$ gcc -Wall par.c -o par
$ ./par mercury venus earth mars jupiter saturn uranus neptune
```

Study the code and try to understand what is going on.

9. Write a program `parcc` that compiles a C program composed of (possibly) multiple source files into a binary executable file in parallel. Note that each source file `prog.c` should be compiled with the command `gcc -Wall -c prog.c`. There should be one of these commands for each source file given in the command line of `parcc`. Each of these commands should be executed in an independent process created by the main process. The latter waits until all the commands are complete and then runs the command: `gcc prog1.o ... progn.o -o prog` to generate the executable. The command should look like:

```
$ parcc -o prog prog1.c prog2.c ... progn.c
[pid:2751] compiling prog1.c ...
[pid:2749] compiling prog2.c ...
...
[pid:2767] compiling progn.c ...
$ ls
prog1.c  prog2.c  ...  progn.c
prog1.o  prog2.o  ...  progn.o
prog
$ ./prog
```

10. Write a program `txt2epub` that takes as arguments multiple books in plain text format (`book1.txt`, `book2.txt`, ..., `bookn.txt`) in the current directory and converts them to EPUB format files using the `pandoc` command (see its man page) in parallel. In the Bash shell this would be achieved by running the commands:

```
$ pandoc book1.txt -o book1.epub
...
$ pandoc bookn.txt -o bookn.epub
```

Each `.txt` file must be converted into its `.epub` version by a process specifically created by the main program for this purpose. Each process is assigned one file name from the command line and runs `pandoc` on that file as shown above and in parallel with the others. The main program waits for these processes to finish and, finally, generates a `.zip` file with all the `epub` converted files as in:

```
$ zip ebooks.zip book1.epub ... bookn.epub
```

The command should look like:

```
$ txt2epub iliad.txt odyssey.txt aeneid.txt metamorphoses.txt
[pid:2751] converting iliad.txt ...
[pid:2749] converting metamorphoses.txt ...
[pid:2752] converting odyssey.txt ...
[pid:2766] converting aeneid.txt ...
$ ls *.epub *.zip
ebooks.zip  aeneid.epub  iliad.epub  metamorphoses.epub  odyssey.epub
```