L:EIC / SO2122:
Process Management
(using the Kernel API)

Q1. 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 a child process */
  fork();

  /* fork another child process */
  fork();

  /* and fork another */
  fork();

  return EXIT_SUCCESS;
}
```

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

Q2. Consider still 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();
  return EXIT_SUCCESS;
}
```

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

Q3. 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? Hint: 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[]) {
  pid_t pid;
  int value = 0;

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

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

Q4. Considere the following program that, when executed, creates a child process that then executes a comand 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[]) {
  pid_t pid;

  /* fork a child process */
  if ((pid = fork()) == -1) {
    perror("fork");
    return EXIT_FAILURE;
  } else if (pid == 0) {
    /* child process */
    if (execlp(argv[1],argv[1],NULL) == -1) {
      perror("execlp");
      return EXIT_FAILURE;
    }
  } else {
    /* parent process */
    if (waitpid(pid, NULL, 0) == -1) {
      perror("waitpid");
      return EXIT_FAILURE;
    }
    printf("child exited\n");
  }
  return EXIT_SUCCESS;
}
```

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

Q5. 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[]) {
  char  buf[1024];
  char* command;
  pid_t pid;

  /* do this until you get a ^C or a ^D */
  for( ; ; ) {

    /* give prompt, read command and null terminate it */
    fprintf(stdout, "$ ");
    if((command = fgets(buf, sizeof(buf), stdin)) == NULL)
      break;
    command[strlen(buf) - 1] = '\0';

    /* call fork and check return value */
    if((pid = fork()) == -1) {
      fprintf(stderr, "%s: can't fork command: %s\n",
              argv[0], strerror(errno));
      continue;
    } else if(pid == 0) {
      /* child */
      execlp(command, command, (char *)0);
      /* if I get here "execlp" failed */
      fprintf(stderr, "%s: couldn't exec %s: %s\n",
              argv[0], buf, strerror(errno));
      /* terminate with error to be caught by parent */
      exit(EXIT_FAILURE);
    }

    /* shell waits for command to finish before giving prompt again */
    if ((pid = waitpid(pid, NULL, 0)) < 0)
      fprintf(stderr, "%s: waitpid error: %s\n",
              argv[0], strerror(errno));
  }
  exit(EXIT_SUCCESS);
}
```

Note the alternative use (with respect to function `perror()`) of function `strerror()` to understand why the system call failed. The later function returns a string with the error description associated with the number in variable `errno`. The value of variable `errno` is set by the kernel before returning from the failed system call with a value of `-1`. The string returned by `strerror()` can be included in richer error messages using, for example, buffered and formated I/O functions such as `fprintf()`.

Why can't you execute commands with arguments, `ls -l` ou `uname -n` with this code?

Q6. Change the previous program so that the commands can be executed with arguments. Hint: check the manual pages for `exec` and its numerous variants. Some of these receive, besides the command name, these functions receive a variable number of arguments read from the shell. You can gather these arguments using, for example, function `strtok` from the Standard C Library.

Q7. 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. Hint: you may use the usual `fork()-exec()` sequence, as in all the other shell commands; check the Bash shell command `tail` (you have seen it in the first set of exercises - "ficha 0").

Q8. Finally, change the program again to implement a command `exit` that terminates the shell.