Info4 – Cours Système – TP.1

Pr. Olivier Gruber

olivier.gruber@univ-grenoble-alpes.fr

Laboratoire d'Informatique de Grenoble

Université de Grenoble-Alpes

Outline

- About processes
  - Create child processes (fork,execv)
  - Waiting for a child process (wait)
- About files and pipes
  - Files are persistent seekable streams of bytes
  - Pipes are for inter-process communication, a stream of bytes or packets
- About shells
  - Command line interpretor
  - Internal versus external commands
  - Piping and redirecting commands

Processes

- Tree of processes
  - Each process is uniquely identified (pid)
  - Linux manages its processes as a tree
  - Each process has a parent and may have children
- Processes can be programmatically
  - Created (fork,execv) and killed (kill)
  - Use the command “man” to have the details
  - Do not kill the process with pid==1

Duplicate a Process

- Fork function
  - Duplicate the process calling the function “fork”
  - Look at the man for more infos

$ man -man 2 fork

```
#include <unistd.h>

int main(int argc, char** argv) {

  int pid = fork();

  switch(pid) {
    case -1: /* error */
      perror("fork failed: ");
      break;
    case 0:  /* child code */
      …
      break;
    default: /* parent code */
      …
      break;
  }
  return;
}
```

Debugging: we remind you that it is possible to **attach gdb to a running process**, if you know the pid of that process, via the gdb command:

(gdb) attach ***pid-of-the-process-to-attach-to***

Do not forget to **load the symbol table** for the executable file running in that process

(gdb) symbol ***path-to-executable-file***

Note: *Eclipse does support GNU toolchain via the CDT plugin*

Fork – Wait

- The function "fork"
  - Duplicate the process calling the function “fork”
- Wait function
  - Waits for a state change(1) in a child process
  - Wait for the identified child process

$ man -man 2 waitpid

```
#include <unistd.h>

int main(int argc, char** argv) {
  int pid = fork();
  int status;
  switch(pid) {
    case -1: /* error */
      perror("fork: ");
      exit(-1);
    case 0:  /* child code */
      printf("Child exiting...");
      break;
    default: /* parent code */
      if (-1==waitpid(pid,&status,0)) {
          perror("waitpid: ");
          exit(-1);
      }
      break;
  }
  exit(0);
}
```

(1): A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal.

Fork – Wait

- The function "fork"
  - Duplicate the process calling the function “fork”
- Wait function
  - Waits for a state change(1) in a child process
  - Wait for the identified child process to exit
  - Wait for any child process

$ man -man 2 wait

Note: the wait call is equivalent to:

waitpid(-1,&status, flags);

which can be **non-blocking** with the flag WNOHANG that forces the call to return immediately if no child has exited.

```
#include <unistd.h>

int main(int argc, char** argv) {
  int pid = fork();
  int status;
  switch(pid) {
    case -1: /* error */
      perror("fork: ");
      exit(-1);
    case 0:  /* child code */
      printf("Child exiting...");
      break;
    default: /* parent code */
      pid = wait(&status);
      if (-1==pid) {
          perror("wait: ");
          exit(-1);
      }
      break;
  }
  exit(0);
}
```

(1): A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal.

Loading an executable file

- The function "execve"
  - Loads and executes, in the process calling the function "execve", the program pointed to by filename

$ man -man 2 execve

Note the real signature for main:

int main(int argc, char *argv[], char *envp[])

where both argv and envp are null-terminated arrays

Thus (argv[argc]==NULL) is true.

What does this call do, with those arguments?

```
#include <unistd.h>

int main(int argc, char** argv,
         char *envp[]) {

  int pid = fork();
  int status;
  switch(pid) {
    case -1: /* error */
      perror("fork: ");
      exit(-1);
    case 0: { /* child code */
      char * args[3];
      args[0] = "ls";
      args[1] = "-al";
      args[2] = argv[1];
      execve("/bin/ls", args, envp);
      break;
    }
    default: /* parent code */
      if (-1==waitpid(pid,&status,0))
         perror("waitpid: ");
      break;
  }
  exit(0);
}
```

About Files

- Reminder: file concept
  - A file is a persistent sequence of bytes
  - Used a stream of bytes
- Reminder: stream-related functions
  - For more information, use "man"

$ man -man 2 open

$ man -man 2 close

$ man -man 2 read

$ man -man 2 write

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);

  Flags: O_RDONLY, O_WRONLY, or O_RDWR.
  These flags respectively mean read-only,
  write-only, or read/write.

  Use the flag O_CREAT to create a file
  Use the flag O_TRUNC to truncate it

int close(int desc);

ssize_t read(int desc, void *ptr,size_t nb_octet);

ssize_t write(int desc, void *ptr, size_t nb_octet);
```

About pipes

- A pipe is a inter-process communication mechanism
  - A pipe is a unidirectional stream of bytes between two processes
  - One process writes bytes into the pipe
  - The other process reads bytes from the pipe
  - For more information, use "man"

$ man -man 2 pipe

```
#include <unistd.h>

int pipe(int pipefd[2]);

/*
 * The array pipefd is used to return two file
 * descriptors referring to the ends of the pipe.
 * pipefd[0] refers to the read end of the pipe.
 * pipefd[1] refers to the write end of the pipe.
 *
 * On  success,  zero  is  returned.
 * On error, -1 is returned, and errno is set
 * appropriately.
 */
```

About pipes and the shell

- Well, the shell provide a way to pipe commands
  - Via the character '|', like in these examples

$ ls | grep toto

$ cat toto | wc

- Requires a pipe in order to:
  - Connect the standard output stream of the command on the left
  - To the standard input stream of the command on the right
- Using the function "dup2"
  - The dup2() system call creates a copy of the file descriptor oldfd to the file descriptor newfd

```
#include <unistd.h>

int dup2(int oldfd, int newfd);
```

About pipes and the shell

- Example using the function "dup2"

So, what does this program do?

```
#include <unistd.h>

int pipe[2];
#define READ_END 0
#define WRITE_END 1

int main(int argc, char** argv) {

  pipe(pipe);

  int pid = fork();

  switch(pid) {
    case -1: /* error */
      perror("panic: ");
      break;
    case 0:  /* child code */
      dup2(pipe[READ_END],STDIN_FILENO);
      ...
      break;
    default: /* parent code */
      dup2(pipe[WRITE_END],STDOUT_FILENO);
      ...
      break;
  }
  return;
}
```

About Terminal and Shells

- Terminal window
  - The graphical user interface for a shell... running as a child process...
- A shell is just another program
  - It can be written in C, the needed functions are in the standard C library
  - It may print stuff through its standard output
  - It reads its standard input for the command lines typed by the end user
  - It interprets the command lines and executes their semantics
- Internal versus external commands
  - Some commands are internal commands: known to the shell implementation
    - Example: the command "cd" ou "pwd" are internal commands
  - Other commands are executable files on a "PATH"
    - Example: the command "ls" is in fact the program /bin/ls
  - The PATH is an environment variable, the environment variables are maintained by the shell
    - Example: PATH="/bin:/sbin:/home/droopy"