ITT440 – NETWORK PROGRAMMING

Introduction To Unix Multi-Process Programming

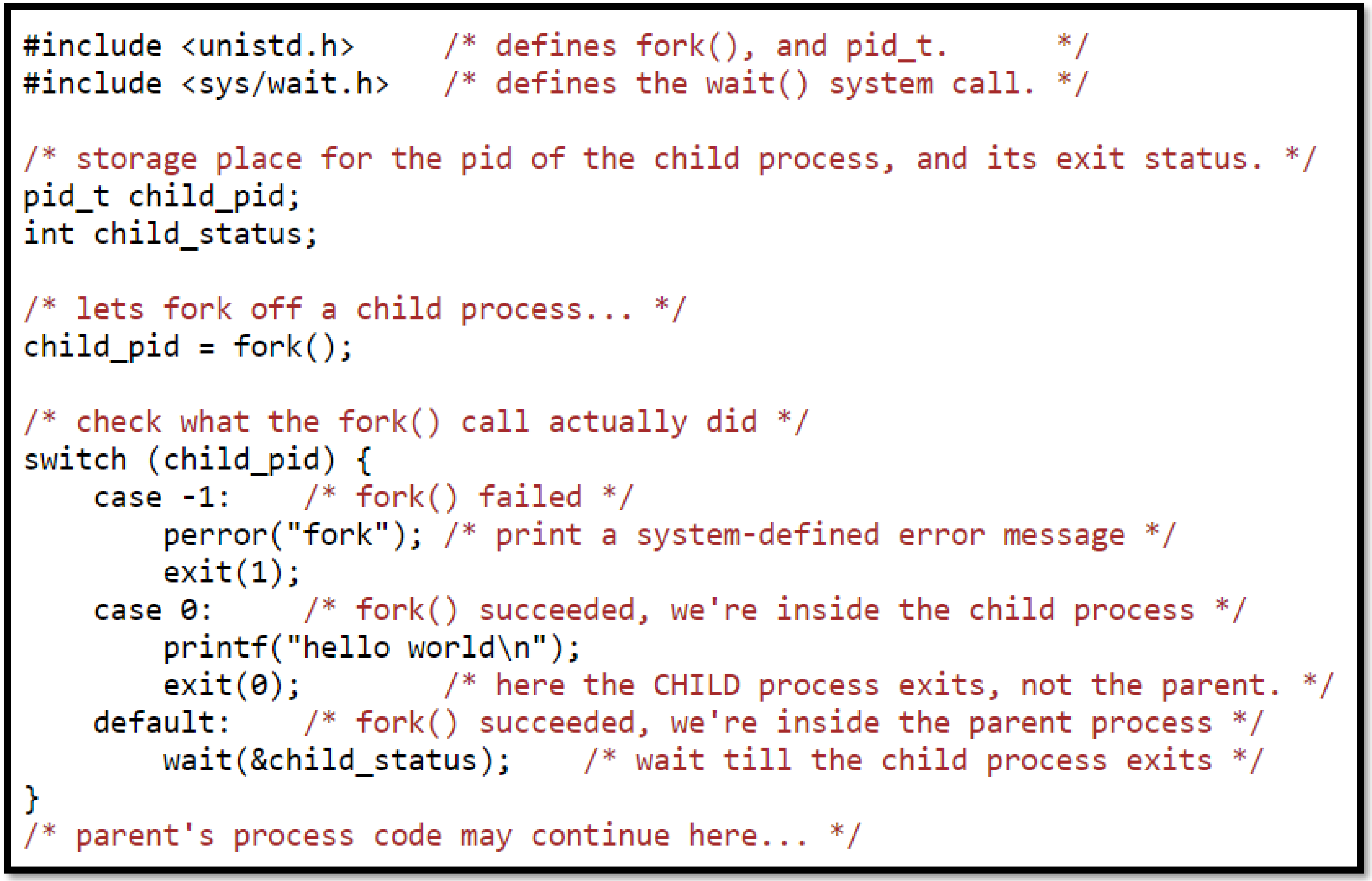
Basic Knowledge

- **Unix_Process**
  - An entity that executes a given piece of code
  - has its own execution stack
  - has its own set of memory pages
  - has its own file descriptors table
  - A unique process ID
- **The fork() System Call**
  - basic way to create a new process.
  - It is also a very unique system call, since it returns twice to the caller.

fork()

- This system call causes the current process to be split into two processes
  - a parent process
  - a child process
- All of the memory pages used by the original process get duplicated during the fork()call
  - so both parent and child process see the exact same image.
- The only distinction is when the call returns.
  - When it returns in the parent process, its return value is the process ID (PID) of the child process. When it returns inside the child process, its return value is '0'. If for some reason this call failed (not enough memory, too many processes, etc.), no new process is created, and the return value of the call is '-1'. In case the process was created successfully, both child process and parent process continue from the same place in the code where the fork() call was used.

Unix-fork.c



```c
#include <unistd.h>     /* defines fork(), and pid_t.      */
#include <sys/wait.h>   /* defines the wait() system call. */

/* storage place for the pid of the child process, and its exit status. */
pid_t child_pid;
int child_status;

/* lets fork off a child process... */
child_pid = fork();

/* check what the fork() call actually did */
switch (child_pid) {
    case -1:    /* fork() failed */
        perror("fork"); /* print a system-defined error message */
        exit(1);
    case 0:     /* fork() succeeded, we're inside the child process */
        printf("hello world\n");
        exit(0);        /* here the CHILD process exits, not the parent. */
    default:    /* fork() succeeded, we're inside the parent process */
        wait(&child_status);    /* wait till the child process exits */
}
/* parent's process code may continue here... */
```

Compile the program above and note the output.

Notes

- The

  perror()

  function prints an error message based on the value of the errno variable, to stderr.

- The

  wait()

  system call waits until any child process exits, and stores its exit status in the variable supplied.
  - There are a set of macros to check this status, that will be explained in the next section.

Source

http://neuron-ai.tuke.sk/hudecm/Tutorials/C/special/multi-process/multi-process.html