School of Electrical and Computer Engineering
Aristotle University of Thessaloniki
Department of Electronics and Computer Engineering

7th Semester
Winter 2015

OPERATING SYSTEMS

Laboratory Exercise 3

Processes and threads System Calls on C

Introduction

In computing, a process is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently. A computer program is a passive collection of instructions; a process is the actual execution of those instructions. Several processes may be associated with the same program; for example, opening up several instances of the same program often means more than one process is being executed. Multitasking is a method to allow multiple processes to share processors (CPUs) and other system resources. Each CPU executes a single task at a time. However, multitasking allows each processor to switch between tasks that are being executed without having to wait for each task to finish. Depending on the operating system implementation, switches could be performed when tasks perform input/output operations, when a task indicates that it can be switched, or on hardware interrupts. A common form of multitasking is time-sharing. Time-sharing is a method to allow fast response for interactive user applications. In time-sharing systems, context switches are performed rapidly. This makes it seem like multiple processes are being executed simultaneously on the same processor. The execution of multiple processes seemingly simultaneously is called concurrency.

For security and reliability reasons most modern operating systems prevent direct communication between independent processes, providing strictly mediated and controlled inter-process communication functionality.

Representation

In general, a computer system process consists of (or is said to 'own') the following resources:

- An image of the executable machine code associated with a program.
- Memory (typically some region of virtual memory); which includes the executable code, process-specific data (input and output), a call stack (to keep track of active subroutines and/or other events), and a heap to hold intermediate computation data generated during run time.

- Operating system descriptors of resources that are allocated to the process, such as file descriptors (Unix terminology) or handles (Windows), and data sources and sinks.
- Security attributes, such as the process owner and the process' set of permissions (allowable operations).
- Processor state (context), such as the content of registers, physical memory addressing, etc. The state is typically stored in computer registers when the process is executing, and in memory otherwise.

The operating system holds most of this information about active processes in data structures called process control blocks.

Any subset of resource, but typically at least the processor state, may be associated with each of the process' threads in operating systems that support threads or 'daughter' processes.

The operating system keeps its processes separated and allocate the resources they need, so that they are less likely to interfere with each other and cause system failures (e.g., deadlock or thrashing). The operating system may also provide mechanisms for inter-process communication to enable processes to interact in safe and predictable ways.

Primary the executable file has a certain structure (right on Picture *1*) as this is formatted after the compilation process. When this file is loaded in memory, the program has the structure showed on left side of Picture 1.

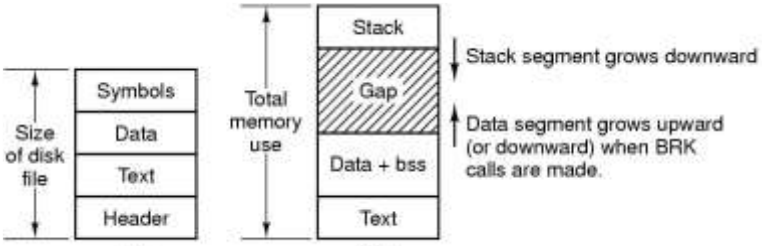

Image 1 : Loading a compiled program into memory.

Process is not only binary code, but a full set of information regarding the program and the state of the code each time is executed.

Process management in multi-tasking operating systems

A multitasking operating system may just switch between processes to give the appearance of many processes executing concurrently or simultaneously, though in fact only one process can be executing at any one

time on a single-core CPU (unless using multithreading or other similar technology).

It is usual to associate a single process with a main program, and daughter (or child) processes with any spin-off, parallel processes, which behave like asynchronous subroutines. A process is said to own resources, of which an image of its program (in memory) is one such resource. (Note, however, that in multiprocessing systems, many processes may run off of, or share, the same reentrant program at the same location in memory— but each process is said to own its own image of the program.)

Processes are often called "tasks" in embedded operating systems. The sense of "process" (or task) is "something that takes up time", as opposed to 'memory', which is "something that takes up space". The above description applies to both processes managed by an operating system, and processes as defined by process calculi.

If a process requests something for which it must wait, it will be blocked. When the process is in the blocked state, it is eligible for swapping to disk, but this is transparent in a virtual memory system, where regions of a process's memory may be really on disk and not in main memory at any time. Note that even unused portions of active processes/tasks (executing programs) are eligible for swapping to disk. All parts of an executing program and its data do not have to be in physical memory for the associated process to be active.

An operating system kernel that allows multi-tasking needs processes to have certain states. Names for these states are not standardised, but they have similar functionality.

- First, the process is "created" - it is loaded from a secondary storage device (hard disk or CD-ROM...) into main memory. After that the process scheduler assigns it the state "waiting".
- While the process is "waiting" it waits for the scheduler to do a so-called context switch and load the process into the processor. The process state then becomes "running", and the processor executes the process instructions.
- If a process needs to wait for a resource (wait for user input or file to open ...), it is assigned the "blocked" state. The process state is changed back to "waiting" when the process no longer needs to wait.

- Once the process finishes execution, or is terminated by the operating system, it is no longer needed. The process is removed instantly or is moved to the "terminated" state. When removed, it just waits to be removed from main memory.

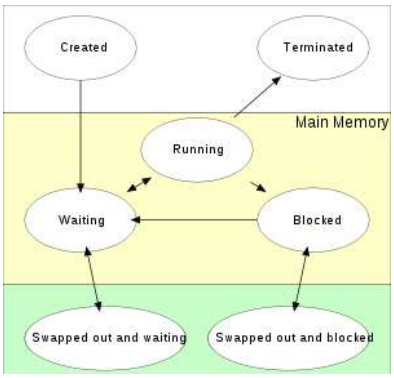

Image 2 : Process States

Syntax and compiling processes

Before we decide how different processes are going to be executed, we need to create the programs of these processes. For the creation of the executable codes of the programs we need to create the source code. These are the files in which certain commands of a high level programming language are typed. Afterwards we need to compile this file into machine language, to the objective program.

Let us guess that we want to create two simultaneous processes. The first one as a program waits for the input for any character from the keyboard and then prints it to screen and the second one waits for two characters as input and then prints them to screen.

testx.c

```
#include <stdio.h>
main(){
        char chin1;
        char chout1;
        chin1 = getchar();
        chout1 = chin1;
        printf( "%c\n", chout1 );
}
```

testy.c

```
#include <stdio.h>
char chin1;
char chin2;
char chout1;
char chout2;
main(){
        chin1 = getchar();
        chin2 =getchar();
        chout1 = chin1;
        chout2 = chin2;
        printf( "%c, %c\n", chout1,
chout2 );
}
```

Compile the first one program with gcc by typing to prompt shell

```
gcc testx.c
```

The execution of this command is the creation of an object program called a.out.

Copy that file into a new executable program with the name prog_a.out by typing to prompt shell the command

```
cp a.out prog_a.out
```

Repeat the previous steps for the second program testy.c and saved the new executable program as `prog_b.out`.

Executing simultaneous processes

Every program can be executed by typing the name of the executable file in the prompt shell. We this commands the operating system creates a new process which its execution is scheduling among the others processes execution, aka shell bash. By typing the command for execution a new process in the shell prompt we say that we run this process on the foreground. It is possible although to postpone the execution of a process or to set the execution in the background and simultaneous to work with another process on the foreground. By using the programs that we wrote previously we will show a way in which we can simultaneous execute processes on foreground and background.

We can set in execution state on background the program prog_a.out by typing its name on prompt shell followed by a space and the character &,

```
prog_a.out &
```

Repeat the previous step for the program prog_b.out.

With the command `jobs` we can observe the catalog of all processes that are executing or have been postponed. Just type on prompt shell the command:

```
jobs
```

On the right of each process of the showed catalog, operating system sets a number. Next to that number is the state in which the process is on.

Processes prog_a.out and prog_b.out that are set on background are on idle state because their programs are executed until the point in which characters must be inserted from the keyboard. To restore them on foreground

in order to insert characters and to continue their execution we must use on prompt shell the command `fg` like

```
fg <process number>
```

By inserting the data, e.g. the character X the execution of the process continues on foreground and the character X is printed on screen..

Each process on a UNIX/Linux when it starts takes, besides other, by the system a unique number – the process identification number (`process ID`), `pid`.

Command ps show the current processes among with their information.

Finding and deleting processes that are executed

Many time it is desirable to delete a process that is been executing on the foreground or is idle or is been executing on the background. For this purpose we can use the command shell `kill`. For example if you want to delete the process prog_a.out after you have brought it back into the foreground you must follow the following procedure

- Execute the command shell jobs to retrieve the process number or the name of the process.
- Type the command shell

```
kill %<process number>
```

By typing the command shell ps you can see the catalog of the current active processes. On this catalog it is marked the identification number of the process that has been given from the system when the process has been created and was recorder to the control board of the process (PCB). On this catalog there is not the process prog_a.out because it has been deleted on previously step.

The deletion of a process can been made by using its identification number, for example after you retrieve the process into foreground and execute it or set it to idle state on foreground, you can find the process identification number by typing the command ps and afterwards you can type

```
Kill <identification number>
```

If the process cannot be deleted, kill command must be used among with a certain signal number. If signal number is omitted, like the above example, kill command uses the default signal 15 (software termination signal). Signal number 9 makes certain termination of the process (sure kill). Therefore for a certain termination of a process you must type

```
Kill -9 <process identification number>
```

Executing shell commands from C

We can execute shell commands (like scenarios or even shell functions) of UNIX/Linux from inside of a C program just like from shell prompt with the help of the C function system().

Note: this can save us from great effort since we can easily embed any command, scenario, service, daemon that is available on shell prompt into our C program.

For the execution of a shell command in C we use the C command

`int system(char *string)` – *where string can be the path of any executable file from the prompt shell of UNIX/Linux. The function system is defined inside the library* `stdlib.h`

Example: Call of `ls` from a C program:

```
#include <stdio.h>
#include <stdlib.h>
main()
{
  printf("Files in Directory are:\n");
  system("ls -l");
}
```

The function returns the exit state of the executed shell command (usually is 0 on success). In case of failure it returns -1.

Another method for executing shell commands is the following

```
#include <unistd.h>
#include <sys/syscall.h>
#include <errno.h>

...

int rc;

rc = syscall(SYS_chmod, "/etc/passwd", 0444);

if (rc == -1)
  fprintf(stderr, "chmod failed, errno = %d\n", errno);
```

The call of function `system` includes several calls to other 3 functions: `execl()`, `wait()` and `fork()` (which are defined on library `unistd`).

The command execl()

The name `execl` comes from abbreviation of the words *execute and leave which means that the called from* `execl()` process will be executed and afterwards terminated.

The function is defined inside the library `unistd.h` as:

```
int execl(char *path, char *arg0,...,char *argn, 0);
```

The lust parameter must be always *NULL* (0, μηδέν). It relies on the *NULL terminator*. Since the number of the origins on the list is variable there must be a NULL termination.

The `path` is the name of the path for the command that is going to be executed, `argo` is the same name like path or at least the same with the last part path name (remember that on shell prompt of UNIX/Linux, zero origin is the name of the command).

The `arg1 ... argn` are the origins that command receives and `(0)` marks the end of the origins.

Therefore the previous example can be written like:

```
#include <stdio.h>
#include <unistd.h>
main()
{
  printf("Files in Directory are:\n");
  execl("/bin/ls","/bin/ls"', "-l",NULL);
}
```

The `execl()` has several repetitions (wrappers). For example the function `execlp()` allows the system to search the for the executable command inside all catalogs that are stored on the environmental shell variable `PATH`. The `execle()` allows to define more values to the environmental shell variables during execution, etc.

The `execv()` functions like `execl()` but is more useful when the number of the origins is not noted from the beginning. The insertion of the origins is

been made with the logic of a origin table. Furthermore `execvp()` is equivalent to `execlp()` and `execve()` to `execle()`.

The `execl()` returns the exit state of the command that has been executed (on success is usually 0). In case of failure it returns -1. Also sets `errno`, with error values that are analyzed on the equivalent manual.

Command fork ()

The `int fork()` converts a process into 2 same processes, noted as parent and child process. On a success execution fork() returns 0 to child process and the pid of the child process to parent. On a failure `fork()` returns -1 to parent – mother process, sets `errno` to an appropriate value, and of course, there is not child process.

Note: The child process inherits the shell environment of the mother process but has its own pid.

The following program demonstrates a simple usage of `fork()`, in which two copies of the same program are "simultaneously" are executed (multitasking)

```
#include <stdio.h>
#include <unistd.h>
main()
{
 int some_value;

 printf("Forking process\n");
 fork();

 /* This part of the program is executed by two different proceses */
 printf("The process id is %d \n", getpid());

 some_value = getpid() + 10;
 printf("Some value is %d", some_value);

 execl("/bin/ls","/bin/ls","-l",NULL);

 /* This line is not executed because of th execl function */
 printf("This line is not printed\n");
}
```

The result will be something like that :

```
Forking process
The process id is 6753
Some value is 6763
The process id is 6754
```

```
Some value is 6764
```

Note: the pid's are changing in each execution. Also the presentation order of the results is changing in each execution. Commands after `execl()` are not executed because we have termination of the program. The variable `some_value` (like all the other issues of the two processes) are private on each process.

When we create 2 processes by `fork()` we can easily identify which process is the child because `fork()` returns zero to the child process. We can trap the errors that `fork()` returns like -1. , for example:

```
int pid; /* process identifier */

pid = fork();
if ( pid < 0 ) { printf("Cannot fork!!'\n");
        exit(1);
        }
if ( pid == 0 )
        { /* Child process */
 ......
 }
else
        { /* Parent process value of pid variable is child's pid */
        ....
 }
```

For example

```
#include <stdio.h>   /* printf, stderr, fprintf */
#include <unistd.h>  /* _exit, fork */
#include <stdlib.h>  /* exit */
#include <errno.h>   /* errno */

int main(void)
{
   pid_t  pid;

   pid = fork();
   if (pid == 0)
   {
      /* child process:
       * when fork() returns 0, then the executable code refers to child
process
       *
       * Count till 10, stepping 1 second
       */
      int j;
      for (j = 0; j < 10; j++)
      {
```

```
         printf("child: %d\n", j);
         sleep(1);
      }
      _exit(0);  /* We are not using exit() but _exit(0) which is not
calling for exiting handlers */
   }
   else if (pid > 0)
   {
      /* Mother process:
       * When fork() return a positive integer then the executable code
refers to mother process and that positive integer is the pid of the
constructed child process
       */
      int i;
      for (i = 0; i < 10; i++)
      {
         printf("parent: %d\n", i);
         sleep(1);
      }
      exit(0);
   }
   else
   {
      /* Error
       * When a call system returns negative an error just happened
       * (e.g.  the system has reached the maximum number of processes).
       */
      fprintf(stderr, "can't fork, error %d\n", errno);
      exit(1);
   }
}
```

It follows an example in C with the usage of execvp:

```
int main(int argc, char *argv[])
{
   int i;
   printf("exec %s via fork",argv[1]);
   printf("with parameters:\n");
   for (i=1; i<argc; i++)
   {
      printf("argv[%d]=%s\n",i-1,argv[i]);
   }
   if (!fork())
   {
      execvp(argv[1],&argv[1]);
      /* The executable code of the process (child process)has just been
       * replaced by the executable file that has been transferred into
mother process as an origin of command shell therefore the next commands
will not be executed unless the call of
       * execv fails/
      perror("execvp");
      return;
   }
```

```
}
```

Command wait ()

The syntax of the command is showed beneath:

```
int wait (int *status_location)
```

This function forces the parent process to wait until the child process is terminated. The function `wait()` returns the `pid` of the child process or `-1` for error. The state of exit for the child process is stored on the `status_location`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
main()
{
  int pid, status;

  printf("Forking process\n");
  pid = fork();

  wait(&status);
  printf("The process id is %d \n", getpid());
  printf("value of pid is %d\n", pid);
  if (pid == 0)
    printf("I am the child, status is %d\n", status);
  else
    printf("I am the parent, status is %d\n", status);
  execl("/bin/ls","/bin/ls","-l", NULL);

  printf("This line is not printed\n");
}
```

Via the value of variable `pid` we can control the processes. Because of called of function `wait()` the mother process will be always executed after child. Also the order that results are showed will be defined since first the result of child process will be showed. Also `status` gets value through `wait()` after termination of child process.

Command exit ()

The syntax of the command is showed beneath:

```
void exit(int status)
```

This function terminated the process that this function called and returns value to `status`. This value can be used by UNIX/Linux and C programs, that have been started with the usage of `fork()`.

The rule that `status` is equal to `0` means a normal termination. Any other value means that there is an error or something unusually happened (usually it is the value -1). Many functions of prototype library of C have error codes that are defined on header file `sys/stat.h`.

Beneath there is an example in which the usage of function `system()` is simulated. Typically it is equivalent to the execution on shell prompt of the command `bash -c command`.

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

/* Execute the command using this shell program. */
#define SHELL "/bin/bash"

int my_system (const char *command)
{
 int status;
 pid_t pid;

 pid = fork ();
 if (pid == 0)
 {
 /* This is the child process. Execute the shell command. */
 execl (SHELL, SHELL, "-c", command, NULL);
 /* If this point is reached execl failed */
 exit (EXIT_FAILURE);
 }
 else
 if (pid < 0)
 /* The fork failed. Report failure. */
 status = -1;
 else
 /* This is the parent process. Wait for the child to complete. */
 if (waitpid (pid, &status, 0) != pid)
 status = -1;
 return status;
```

```
}
```

In this second example user inserts shell prompt commands with their origin. The program splits the series of symbols into tokens that are inserted to an array of series-symbols. The commands are executed into a child process by using `exccvp()`.

```
/* fork.c - example of a fork in a program */
/* The program asks for UNIX commands to be typed and inputted to a
string*/
/* The string is then "parsed" by locating blanks etc. */
/* Each command and sorresponding arguments are put in a args array */
/* execvp is called to execute these commands in child process */
/* spawned by fork() */

/* cc -o fork fork.c */

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

main()
{
 char buf[1024];
 char *args[64];

 for (;;) {
 /*
 * Prompt for and read a command.
 */
 printf("Command: ");

 if (gets(buf) == NULL) {
 printf("\n");
 exit(0);
 }

 /*
 * Split the string into arguments.
 */
 parse(buf, args);

 /*
 * Execute the command.
 */
 execute(args);
 }
}

/*
 * parse--split the command in buf into
 * individual arguments.
 */
```

```
parse(buf, args)
char *buf;
char **args;
{
 while (*buf != NULL) {
 /*
 * Strip whitespace. Use nulls, so
 * that the previous argument is terminated
 * automatically.
 */
 while ((*buf == ' ') || (*buf == '\t'))
 *buf++ = NULL;

 /*
 * Save the argument.
 */
 *args++ = buf;

 /*
 * Skip over the argument.
 */
 while ((*buf != NULL) && (*buf != ' ') && (*buf != '\t'))
 buf++;
 }

 *args = NULL;
}

/*
 * execute--spawn a child process and execute
 * the program.
 */
execute(args)
char **args;
{
 int pid, status;

 /*
 * Get a child process.
 */
 if ((pid = fork()) < 0) {
 perror("fork");
 exit(1);

       /* NOTE: perror() produces a short error message on the standard
 error describing the last error encountered during a call to
 a system or library function.
 */
 }

 /*
 * The child executes the code inside the if.
 */
 if (pid == 0) {
 execvp(*args, args);
 perror(*args);
 exit(1);
```

```
 /* NOTE: The execv() vnd execvp versions of execl() are useful when the
 number of arguments is unknown in advance;
 The arguments to execv() and execvp() are the name
 of the file to be executed and a vector of strings contain-
 ing the arguments. The last argument string must be fol-
 lowed by a 0 pointer.

 execlp() and execvp() are called with the same arguments as
 execl() and execv(), but duplicate the shell's actions in
 searching for an executable file in a list of directories.
 The directory list is obtained from the environment.
 */
 }

 /*
 * The parent executes the wait.
 */
 while (wait(&status) != pid)
 /* empty */ ;
}
```

Zombie Processes

When a process creates several other child processes it must, before it completes its execution, to wait until all its children complete themselves. If it not then the child's, when they are finished for example by making `exit()`, are becoming "zombies". In other words the child – zombie processes are not vanished from the system and they keep wasting some resources. For that reason every mother process must execute the call `wait()` for each children that has created. If it has created N child's , then the mother process can do:

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

  #include <sys/wait.h>

  for (i = 0; i < N; i++)

    wait(0);…
```

The `wait(0)` stops the execution of the parent until it terminates (whenever happens) any of their children. If we are interesting for a termination of a specific child we can use beneath `wait()`, the system call `waitpid()` that takes three origins (for more information take a look on man pages).

Exercises – Activities

Type and compile the following program on C

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

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

int var_glb; /* A global variable*/

int main(void)
{
    pid_t childPID;
    int var_lcl = 0;

    childPID = fork();

    if(childPID >= 0) // fork was successful
    {
        if(childPID == 0) // child process
        {
            var_lcl++;
            var_glb++;
            printf("\n Child Process :: var_lcl = [%d], var_glb[%d]\n",
var_lcl, var_glb);
        }
        else //Parent process
        {
            var_lcl = 10;
            var_glb = 20;
            printf("\n Parent process :: var_lcl = [%d], var_glb[%d]\n",
var_lcl, var_glb);
        }
    }
    else // fork failed
    {
        printf("\n Fork failed, quitting!!!!!!\n");
        return 1;
    }

    return 0;
}
```

What does this program do exactly? What are the values of variables var_lcl and var_glb for each process?