Week 4: Processes

Processes and Threads

In this exercise, we will look at two mechanisms for parallel and simultaneous execution of different execution paths. In general, this may be the code of one or more programs.

Introduction to C

Here you can find basics on C programming languages.

Processes

The first mechanism is the process. The process is an instance of the execution of an individual program. In modern systems, the process allows isolation and virtualization of the basic resources of the system - processor and memory. Each process gets its time on the processor and its virtual memory. This allows simultaneous execution of different programs on the same processor and in the same physical memory.

Translated code in binary format, put on a medium is called a program. When you run the program, its code (commands) and data are loaded into memory, a bit of initial memory is reserved (the stack) and the commands are executed. At this point the program becomes a process. Typically, different processes execute a variety of different programs. Often several processes are also driven by the same program. Here we'll look at how to create new processes within the same program and how to manage the new process.

Process management

Different OSes supplies us with a variety of tools to organize and manage our processes. Some use a graphical interface (system monitor, task manager) while other are command line tools. Here we will look at some command line tools available in linux.

ps - print information of currently running processes

At any given moment your OS is running multiple processes, that are necesary for a working environment. In Linux each process is identified using a process identification number - **PID**. This is a number that uniquely identifies a process - more so than the programs name, as one program could run as multiple processes. Since the PID is usualy 16 bit, there is a finite number of PIDs available.

pstree - print a process tree

Processes are created by other processes. In linux the process that creates a new process is called the parent, while the created processes are called its children. The parent of a process is identified using a PPID - the parents PID number.

proc - file system with process information, mounted at /proc

In Unix like operating sistems, most resources are represented in the file system as files. This is also true for all processes, which are listed in the /proc directory. This directory contains information on any process - its memory, its files, ...

top - command line tool, similar to a GUI task manager.

Creating new processes

New processes are created using a system call *fork*:

```
pid_t fork()
```

A simple example:

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

int main() {
    int pid = fork();
    printf("pid=%dn", pid);
    sleep(1);
    return 0;
}
```

When the *fork* function is called, a new process is created, a copy of the current process. The original process is called parent and the new process is called child.

Execution of both processes resumes immediately after the fork function is called. We can distinguish between parent and child by the return value of the fork function. It returns the following values:

- 0 - In a child, the *fork* function returns 0.
- > 0 - In the parent, the *fork* function returns a value greater than 0. The returned value is the process id of the child.
- -1 - When an error occurs, the *fork* function returns -1, **without creating** a new process. The error code is written to **errno**.

The previous example can be modified so that the parent and the child print a different strings to the output:

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

int main() {
    int pid = fork();
    switch(pid) {
        case -1:
            perror("Error calling fork.");
            return -1;
        case 0:
            printf("I am the child.n");
            break;
        default:
            printf("I am the parent. %d is my child.n", pid);
            break;
    }
    sleep(1);
    return 0;
}
```

The process id is obtained by calling the **getpid** function. The process id of the parent is obtained by calling the **getppid** function. Each process that we will be operating with has in addition to its **pid**, also the **ppid** of its current parent (we will see later that the parent of the process can change). The only process without the parent is the first process - the so-called **init** process.

Because the child is a copy of the parent, **it has access to all values set before the fork function is called**. However, it does not have access to the values set after the function call.

The following example creates a child that outputs the value of variable *a* and then waits 20,000 microseconds with a call to the *usleep* function. The parent changes the value of *a* during the sleep of the child and displays it. The child wakes up and display the value of *a* one more time.

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

int main() {
    int a = 0;
    int pid = fork();
    switch(pid) {
        case -1:
            perror("Error calling fork.");
            return -1;
        case 0:
            printf("Child heren");
            printf("a is %dn", a);
            usleep(20000);
            printf("Child againn");
            printf("a is %dn", a);
            break;
        default:
            usleep(10000);
```

```
            printf("Parent heren");
            a = 4;
            printf("a is %dn", a);
            usleep(20000);
            break;
    }
    sleep(1);
    return 0;
}
```

Because variable *a* is a copy of the parent variable, the child does not see the changes to the variable made by the parent after calling the fork function, and vice versa.

Ending the process

There are several ways to complete the program. The process (running program) is terminated when:

- a value is returned from the **main** function
- **exit** function is called
- the **main** function runs out (pass its end)

The last example is generally unwanted, but still normal way of completing the process. Before the process is actually completed, it can run additional functions and clean after itself accordingly. What is left in the memory is then cleared and removed by the operating system.

The process can also be completed when some signals are received. That is when the process ends in an unusual way. The running process is terminated immediately. Anything left in the memory is cleared by the operating system. While some cases of such interruptions can be avoided by catching signals, the SIGKILL signal is inevitable and stops any active process.

Normal proces termination

After normal termination additional functions and routines will be executed:

- additional registered function will be called
- memory buffers used by *stdio* are flushed (written to files) and freed
- the function **_exit** is called

After this the OS takes care of any remaining resources, discarding any changes:

- all file descriptors are closed
- any memory allocated by *mmap* is released
- any temporary sistem structures are removed and closed (semaphors, memory locks,...)
- **all remaining children of the process get a new PPID, usualy the init process (PPID becomes 1)**

We can register additional function to be called on normal termination. This is done using the function:

```
int atexit(void (*func)(void));
```

As mentioned earlier, calling the fork function creates a new process, which is a copy of the existing process. The new process has the same file descriptors (closing a child file descriptor does not close the file descriptor in the parent), registered the same functions with atexit, and has copies of all memory - heap and stack. Therefore, we must pay attention when the child terminate.

The following example registers two functions with atexit, one for the parent and one for the child. For the parent, it registers the method before and for the child after calling the fork function.

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

void parent_exit() {
    char msg[] = "The parent ended.n";
    write(1, msg, strlen(msg));
}

void child_exit() {
    char msg[] = "The child ended.n";
    write(1, msg, strlen(msg));
}

int main() {
    int a = 0;
    atexit(parent_exit);
```

```
    int pid = fork();
    switch(pid) {
        case -1:
            perror("Error calling fork.");
            return -1;
        case 0:
            atexit(child_exit);
            sleep(1);
            printf("I am the child.n");
            exit(0);
        default:
            sleep(2);
            printf("I am the parent.n");
            exit(0);
    }
    sleep(1);
    return 0;
}
```

We see that the child called both functions at the exit. Depending on the functionality, this may be a desirable or unwanted feature, and we need to pay attention to it.

The easiest way is to avoid difficult cases and always end the child with a call to the **_exit** function.

If we also want to register methods with *atexit* in a child, we can make all *atexit* calls after calling the *fork* function. Thus, the parent and child each have their own list of functions to call when the process is completed.

An example where not using **exit** method has negative consequences:

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

int main() {
    printf("I am the parent.n");
    write(1, "hellon", strlen("hellon"));
    int pid = fork();
    switch(pid) {
        case -1:
            perror("Error calling fork.");
            return -1;
        case 0:
            printf("I am the child.n");
            break;
    }
    sleep(1);
    return 0;
}
```

Runing the process in the terminal gives us the expected output:

```
I am the parent.
hello
I am the child.
```

But if we redirect the output to a file, we get the following output:

```
hello
I am the parent.
I am the parent.
I am the child.
```

Waiting for a child

In the previous examples the parent and child processes ran independently. We can achieve some basic synchronization by calling **wait** function:

```
pid_t wait(int *status)
```

The *wait* function call in the parent process waits for the child to finish first. It then returns the child's pid and sets the child's return value to the **status** variable.

For this call to work correctly the child must remain partially in memory after the completion and wait for the *wait* function to be called. This leads to two situations:

- the child has terminated before the parent called *wait*, the child becomes a **zombie** process
- the parent terminates without calling *wait*, the child becomes an **orphan** process

The first case is practically unavoidable in normal operation - the child must close before the parent can receive confirmation with the *wait* function. However, the situation can become a problem when the parent creates new processes and does not call *wait* for completed ones. Zombie processes do not consume memory and are not consuming CPU, however, they occupy valid pid values. Until the zombie process is completed, another process cannot get its pid number. If **we run out of valid pid numbers**, we cannot start a new process. Therefore, the parent must call *wait* as soon as possible to remove the children who have completed or will be. The Zombie process cannot be killed by the SIGKILL signal either - the process is not running, so it cannot be stopped meaningfully.

The second case only occurs when the process is completed before waiting for all of its children with the *wait* function. At that time, children are labeled as orphans. All orphans are transferred to the init process (init becomes their new parent). Init calls the *wait* function at the right time for each such process. Orphan running is, of course, not interrupted. If the process is still running while its parent is completing, the process can still be completed normally. **In your programs, the parent must always wait for their children by calling the wait function. Orphaned processes are inadmissible.**

The following example creates a zombie process:

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

int main() {
    switch(fork()) {
        case -1:
            perror("Error calling fork.");
            return -1;
        case 0:
            printf("I am the child.n");
            _exit(0);
        default:
            sleep(60); // The child will be a zombie for 60 seconds
            wait(0);
    }
    return 0;
}
```

We compile the program into an executable module named 'zombie', run it and run the following command in another terminal:

```
$ ps axo fname,pid,ppid,stat | grep zombi
zombi    16382 14649 S+
zombi    16383 16382 Z+
```

The parent process in our case has PID 16382 and status S+ (is in sleep), and the child process has PID 16383 and has status Z+ (zombie).

The following example creates an orphan process:

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

int main() {
    switch(fork()) {
    case -1:
        perror("Error calling fork.");
        return -1;
    case 0:
        printf("I am the child.n");
        sleep(60);
        _exit(0);
    default:
        sleep(10); // The child will become an orphan after 10 seconds
    }
    return 0;
}
```

We compile the program into an executable module named 'orphan', run it, and run the following command in another terminal:

```
$ ps axo fname,pid,ppid,stat | grep orphan
orphan   16476 14649 S+
orphan   16477 16476 S+
```

The parent process in our case has PID 16476 and the child process has PID 16477. After 10 seconds, the parent closes. We run the following command:

```
$ ps axo fname,pid,ppid,stat | grep orphan
orphan   16477 1 S
```

We see that the PPID has changed from 14676 to 1. As the father closed, the process has been adopted by another process - in this case the 'init' process.

Modern Linux OSes can have multiple init like processes, that will catch orphan processes.

The child return value

After calling the *wait* function, the **status** variable contains information about the child's termination. From this we can see the value that the child returns at successful completion or the signal number that interrupted the child's execution. We can find individual values from the status variable with macros in the header <sys / wait.h>:

- WIFEXITED(status) - returns ***true***, if the child terminated normaly
- WEXITSTATUS(status) - returns the value, returned by the child using *_exit*
- WIFSIGNALED(status) - returns ***true***, if the child was terminated using a signal
- WTERMSIG(status) - returns the number of the signal terminating the child

Informations are packed as 2 bytes and are set by *wait* function. One byte is used to save the information returned from the process. Even though the **main** and **_exit** function return an **int** value, only the **char** size can be read from the wait.

An example reading the return value of the child:

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

int main() {
    int pid = fork();
    switch(pid) {
        case -1:
            perror("Error calling fork.");
            return -1;
        case 0:
            sleep(10);
            _exit(0x04);
        default:
            printf("The child is process: %dn", pid);
    }
    int status = 0;
    pid = wait(&status);
    if(WIFEXITED(status)) {
        printf("The child returned value: %dn", WEXITSTATUS(status));
    }
    else if(WIFSIGNALED(status)) {
        int sig_id = WTERMSIG(status);
        printf("The child was interrupted by signal: %sn", strsignal(sig_id));
    }
    return 0;
}
```

The example above creates 1 child. The child waits 10 seconds and returns a value of 0x04 via the *_exit* function. The parent prints out the child's PID and waits for its completion by calling the *wait* function. It then reads the status and displays the returned value or the name of the signal that interrupted the child.

If we run this example in parallel with the & operator, we can try to interrupt the process whose PID is written to us by the parent or we can wait for the child to call _exit.

Sequential and parallel execution

In practice, there are two ways to use additional processes. When we have tasks that we need to perform one after the other, we use what we call sequential execution. We only create the next process when the current process is completed. We use this mode of operation when the calculations of one process represent the input of the next.

When we have tasks that can be performed simultaneously, we use so-called parallel execution. All processes are created simultaneously or closely one by one. This type of operation can be much more efficient and take advantage of modern multi-core processors. However, such implementation must be carefully planned. It is also very difficult to eliminate errors that become non-deterministic in such environment (the same input does not cause an error every time it starts).

Because language C is not adapted to execute multiple processes at the same time, we do not have explicit commands for sequential or parallel execution. Alternatively, you can achieve the desired method of execution with the appropriate calls of the *wait* function.

We are saying that processes run in parallel when at some point in time several processes are running simultaneously. In C, it is not possible to run multiple processes at the same time.

Example of sequential execution:

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

int main() {
    for(int i=0; i < 10; i++) {
        switch(fork()) {
            case -1:
                perror("Error calling fork.");
                return -1;
            case 0:
                usleep(1000);
                printf("I am the child number: %dn", i);
                _exit(0);
            default:
                wait(0);
        }
    }
    return 0;
}
```

Example of parallel execution:

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

int main() {
    for(int i=0; i < 10; i++) {
        switch(fork()) {
            case -1:
                perror("Error calling fork.");
                return -1;
            case 0:
                usleep(1000);
                printf("I am the child number: %dn", i);
                _exit(0);
        }
    }
    for(int i=0; i < 10; i++) {
        wait(0);
    }
    return 0;
}
```

1.

Guided Exercise

You will prepare a set of programs for operation with phonebook. Thus, you will have to implement the following programs: *phonebook*, *phonebook_add*, *phonebook_delete*, *phonebook_printout*, *phonebook_search*, *phonebook_close*.

We suggest that you write programs in source codes named phonebook.c, phonebook_add.c, phonebook_delete.c, phonebook_printout.c, phonebook_search.c in phonebook_close.c.

phonebook

Program *phonebook* is a service that responds to queries of other processes through Unix sockets. Its functionality is to manage the content of the phonebook data in the background (names, surnames and telephone numbers). The names should have max. 100 characters, also the surnames should have max. 100 characters and the telephone numbers should have max. 30 characters.

On each request, standard output should print out a type of a query and result (successful, unsuccessful, number of results found). Queries are the following: add, delete, printout, search and close.

Adding is unsuccessful when a telephone number already exists in a phonebook.

Deleting is unsuccessful when a telephone number does not exist in a phonebook.

Printout is unsuccessful when a telephone number does not exist in a phonebook.

The result of a search is a number of found and returned entries.

A hint: first implement a program *phonebook*, that will read commands and data from the standard input (keyboard). For now, individual actions mentioned above implement as separate functions. In such manner you will be able to modify and upgrade your program more efficiently in the following two weeks.

phonebook_add <name> <surname> <telephone_number>

A program sends a query for adding a new entry in a phonebook. It gets the following arguments through the command line: name, surname and telephone number. After query is performed, it prints out whether the adding was successful or not.

phonebook_delete <telephone_number>

A program sends a quers for deleting an entry in a phonebook. It gets a single argument through the command line: a telephone number which we want to delete. After query is performed, it prints out whether the deleting procedure was successful or not.

phonebook_printout <telephone_number>

A program sends a query for print out the information about the telephone number which is provided to a program through the argument in a command line. If telephone number does not exist, you should also print that out.

phonebook_search

A program reads a search string from a standard input (max. 100 characters). Then, it sends this search string to a main program *phonebook* that returns all entries in a phonebook, that contain search string in a name OR surname. Then, it prints out all entries, each in own line in a form *<name> <surname> : <telephone number>*. Then, it reads the next search query. Program terminates, when user closes the standard input (i.e. by pressing Ctrl+D on keyboard).

phonebook_close

A program sends a query for terminating the main program *phonebook*. Before doing that, it should wait to complete all open connections with all other processes.

Upgrades

Use of threads

In some cases, the main process *phonebook* must communicate with more than one client in the same time. In order to keep a program responsive also by using more complex queries, we can communicate with each of the program in its own thread. By doing so, we must synchronize the access to the phonebook. Than means, that adding and deleting can be performed by only one thread at the time. Printout and search can be perfomed in parallel, but during that time none of the treads is allowed to add or delete the entries.

Use of a shared memory for transferring the results of a query

Results of a query could be quite long. In order to transfer the data faster, we can use POSIX shared memory. Thus, main program *phonebook* should for the results of a search create a new shared memory with a unique name which is than used for storing the found results. Then, main program sends only this unique name to a program *phonebook_search* to access to the result stored in this shared memory.

Scoring

- basic functionalities **20 points**
  - phonebook_add
  - phonebook_delete
  - phonebook_printout
  - phonebook_close
- search **10 points**
- use of threads **10 points**

- use of threads **10 points**
- use of a shared memory **10 points**

Makefile

```
all: phonebook phonebook_search phonebook_delete phonebook_printout phonebook_add phonebook_close

phonebook: phonebook.c
    gcc phonebook.c -g -o phonebook -lpthread -lrt

phonebook_add: phonebook_add.c
    gcc phonebook_add.c -g -o phonebook_add

phonebook_printout: phonebook_printout.c
    gcc phonebook_printout.c -g -o phonebook_printout

phonebook_delete: phonebook_delete.c
    gcc phonebook_delete.c -g -o phonebook_delete

phonebook_search: phonebook_search.c
    gcc phonebook_search.c -g -o phonebook_search -lrt

phonebook_close: phonebook_close.c
    gcc phonebook_close.c -g -o phonebook_close

clean:
    rm -f phonebook phonebook_search phonebook_delete phonebook_printout phonebook_add phonebook_close
```

100 pts

Submission deadline **18. 12. 2022 at 23:59:59**

Number of submissions: 0/2

Number of tests: 15

Choose File No file chosen

Submit Check