1 Understanding fork() and exec()

Creating new processes in the Unix/Linux world is done using fork(). fork() clones an existing process and adds it to the runqueue, rather than really creating a new one. Since fork clones a process, they both execute the line after the fork() call. Now they need to distinguish whether they are parent or child process. This can be done by checking the return value of fork().

- to the parent process, fork() returns the PID of the child process
- to the child process, fork() returns 0

1.1 Playing with fork()

1.1.1 Calling fork() once

Create a program which forks itself once. The parent process should output 100 times “I’m the parent and my child’s PID is <pid>”. The child should output 100 times “I’m the child and my PID is <pid>”.

Do the PIDs match?
Answer: Yes, the PIDs are matching. An example output could look like this
I’m the child and my PID is 4270
I’m the parent and my child’s PID is 4270
I’m the child and my PID is 4270
I’m the parent and my child’s PID is 4270
I’m the child and my PID is 4270
I’m the child and my PID is 4270
...

1.1.2 fork() multiple times

What do you expect to happen here?
Please explain what you think will happen.
int main(int argc, char **argv)
{
  while(1) {
    fork();
  }
}

**Answer:** If you execute this code, your computer will be (almost) dead. Every child forks new children in a loop and this new children fork new children in a loop as well. This consumes too many resources in a very short time. Not only memory, but also CPU cycles, page table entries, descriptors...

### 1.1.3 Executing “ls -l” from your program

*Write a simple program (main function) which executes the ls program. Lookup the manual page for the exec family (man 3 exec). After the exec call in you main function, have a printf which tells that you have called exec now.*

**What do you notice?**

**Answer:** The line after `exec` of your program will not be executed. `exec` replaces the currently executed program by a new one. It does not automatically fork a new process to execute `ls -l`.

**How can you fix that?**

**Answer:** If you want your program to continue to run, first fork and execute `exec` in the child. If you want your program to wait until the child process terminated, you can call one of the `wait` functions (see man 2 `wait`).

### 1.2 Reading the ls output from a pipe

*Create a simple application which opens a pipe, executes “ls” and reads its output via the pipe. Your application should write a sentence and the output of ls to the console (example: “Output from ls: <ls output>”).*

### 2 Implementing cooperative user-level threads-package

**Answer:** You can download an example implementation from the webpage.

#### 2.1 Threads-package

In this section you should implement a cooperative user-level threads-package. Cooperative threads explicitly yield when they no longer require CPU time (for now) and are not interrupted by any other mechanism. When a thread yields, the next thread should be run. To choose the next thread to run, simply keep track of the threads in a circular list and pick the next one (round robin).

You will need to implement at least the following functions:

- `thread_create`
Each thread should be represented by a `struct thread` which contains at least a function pointer to the thread’s function and an argument of type `void *`. The thread’s function should take this `void *` as argument whenever it is executed. This struct should also contain a pointer to the threads stack and two fields which store the current stack pointer and base pointer when it calls `yield`.

To save and restore the context of a thread it should be enough to use `setjmp()` and `longjmp()`. `thread_create` should take a function pointer and a `void*` arg as parameters. It allocates a struct thread, allocates a new stack for this thread and sets default values. It is important that the initial stack pointer (set by this function) is at an address dividable by 8. The function returns the initialized structure.

`thread_add_runqueue` adds an initialized struct thread to the runqueue. To implement the round robin ordering of context switches, it is easiest if you maintain a ring of those structures. You can do that by having a next field which always points to the next to be executed thread.

The static variable `current_thread` always points to the currently executed thread.

`thread_yield` suspends the current thread by saving its context to the struct thread using `setjmp` and calling the `schedule` function to decide what to run and then the `dispatch` function to run the thread. If the thread is resumed later, `thread_yield` returns to the calling place in the function.

`thread_exit` removes the calling thread from the ring, frees its stack and the struct thread, sets the `current_thread` variable to the next to be executed thread and calls dispatch. It is important to dispatch the next thread right here before returning, because we just removed the current thread. Furthermore, think about what happens when the last thread exits.

`schedule` decides which thread to run next. This is actually trivial, because you just run the next thread in the thread struct ring. You can just follow the next field of the current thread. For convenience (for example for the dispatcher), it might be helpful to have another static variable which points to the last executed thread.

`dispatch` actually executes a thread (the thread to run as decided by the `schedule`). It has to save the stack pointer and the base pointer of the last thread to its struct thread and it has to restore the stack pointer and base pointer of the new thread. In case the thread has never run before, it may have to do some initialization using some assembly to set the `rps` and `rbp` registers correctly. If the thread has run before, restoring the context (`longjmp`) is enough. In case the thread’s function just returns, the thread has to be removed from the ring and the next one has to be dispatched. The easiest thing to do here is calling `thread_exit`, since this function does that already.

`thread_start_threading` initializes the threading by calling `schedule` and `dispatch`. This function should be called by your main function (after having added the first thread to the runqueue). It should never return (at least as long as there are threads in your system).

So in summary, to create and run a thread, you should follow the steps below:

```c
static void thread_function(void *arg)
{
    ...
    may create threads here and add to the runqueue;
```
...  
while(some condition, maybe forever) {  
do work;  
thread_yield();  
if (bla) {  
    may call thread_exit();  
}  
}  
}

int main(int argc, char **argv) {  
    struct thread *t1 = thread_create(f1, NULL);  
    thread_add_runqueue(t1);  
    ...  
    may create more threads and add to runqueue here;  
    ...  
    thread_start_threading();  
    // we shouldn’t reach this line  
    printf("exited\n");  
    return 0;  
}

2.2 Test your threads-package

As a second step, implement a main function which creates a couple of threads which perform some operations so that we can see on the console that the threads are really running concurrently. Note: Since this is cooperative threading, your threads have to call thread_yield from time to time.

Two simple functions might have a counter, one counting from 0-9 and another one counting from 1000-1009.

You may download main.c and a skeleton of threads.h from the courses website. The skeleton provides you the prototypes assumed by main.c (you don’t need to follow this, you can have your own prototypes and can have an own main.c). The skeleton does not define the contents of struct thread.

2.3 Extend your threads-package to allow for blocking-I/O

If you use blocking-I/O, your other threads won’t proceed, because you block the whole process. Extend your threads-package in a way that a thread can call block_on_fd(int fd) which causes the function schedule to schedule the thread only if there is incoming data for this file descriptor. The schedule function can figure out which file descriptors have pending data using select. block_on_fd might remove the thread from the runqueue and add it to an io_waiting_queue.

3 Signals

a) Compile and run the following program code. Does it do what you expect? Give an explanation for the behavior.

#include <stdio.h>  
#include <signal.h>  
#include <stdlib.h>
```c
void sighup_handler()
{
    printf("the child has received a SIGHUP\n");
}

int main()
{
    int child_pid = 0;
    child_pid = fork();
    printf("the process id is %d and the child process id is %d\n", getpid(), child_pid);

    if (child_pid == 0) {
        // I am a child
        signal(SIGHUP, sighup_handler);
        printf("the child has registered its handler\n");
        for (;;) { }
    } else {
        // I am the parent
        kill(child_pid, SIGHUP);
        printf("the parent has issued the signal\n");
    }

    return 0;
}
```

**Answer:** The child process and the parent process are running concurrently after the fork. Hence, there is a race condition between the client registering the handler and the parent issuing the signal. The output of the signal handler is only produced when the child wins the race and registers the signal handler first. If not, the signal of the parent is caught by the default signal handler of the child which terminates the process.

b) Write a program that registers a signal handler, forks 10 child processes which send signals to the parent and then terminate. In the parent, count the number of signals that you got and print them to the screen. Is this a valid implementation of a counter? Why not and which behavior of signals is problematic?

**Answer:** The counter will very likely not count the correct number of signals. The reason is that signals are similar to a message queue of size one. The receiver cannot distinguish how many signals have been received when the handler is invoked, only that the signal is set and has to be handled. If the scheduling is in the receiver’s way, many senders can issue their signals before the receiver’s handler function is invoked.