SdP 2022 – OS161 Laboratory – 4

To address this laboratory you need:

- ***having carried out (and <u>understood</u>) laboratories 2 and 3,***
    - ***laboratory 2 is needed to understand system calls and (above all) the end/exit of a process,***
    - ***laboratory 3 to be able to use synchronization primitives.***

Implement the waitpid system call (wait for the end of the process)

We want to create support for the `waitpid` system call (in Unix/Linux there is also the `wait`, which awaits any child process), which allows a process to wait for the change of state of another process, whose identifier (`pid`) is known.

For example, see the `waitpid` documentation on https://linux.die.net/man/2/waitpid or https://www.freebsd.org/cgi/man.cgi?query=waitpid

For simplicity, we ask you to manage only the change of state when the process is "*finished*" (it would be necessary to manage also other states, such as *wait/resume* connected to a signal). In summary, after `thread_exit` (of the last/only thread of a given process) a process remains in a "*zombie*" state until another process does a `wait` or `waitpid` (in OS161 only `waitpid`), and therefore obtains the exit status.

The laboratory can be divided into parts, which are recommended to be made one piece at a time, moving on to the next after having finalized (including execution/debugging) the previous one:

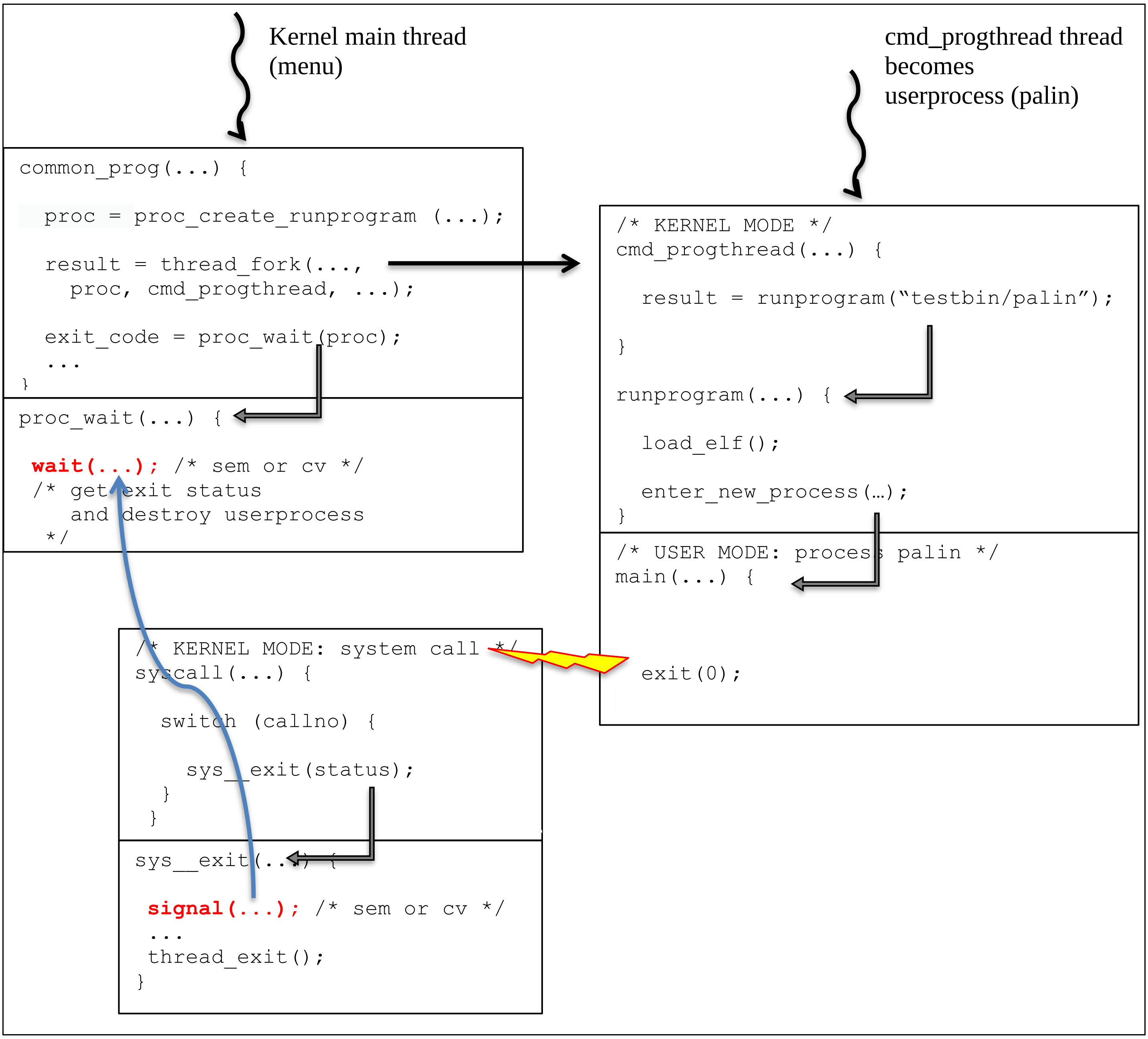
- waiting for the termination of a user process, returning its exit status;
- destruction of the data structure of a process (`struct proc);`
- pid assignment to process, process table management and `waitpid`;
- (optional) create system calls `getpid` and `fork`.

Waiting for process termination

It is recommended to first create an `int proc_wait(struct proc *p)` function. The function waits (*without the need to manage the pid and to support the system call waitpid*) for the end (with a call to the `syscall _exit`()) of the process (whose related `struct proc` is) pointed to by `p`. Waiting can be implemented by means of a semaphore or condition variable (added as a field to the `struct proc`).
It is therefore a kernel function, usable only within it (because it uses the pointer to `struct proc`). This function could be implemented in `kern/proc/proc.c` and called by `common_prog`, after this has successfully called `thread_fork`, in order to wait for the end of the activated process (and not immediately return to the menu that would require another command). The `common_prog` function could then wait for the child process to finish by

```
exit_code = proc_wait(proc);
```

and print the return code (the one received from `_exit - sys__exit` and saved in the `struct proc`) on the console before returning to the calling program. The figure represents the call and synchronization scheme. Note that the calls to `wait()` and `signal()` must be replaced by appropriate functions, which depend on the implementation choices made (semaphore, condition variable, wrapper function or direct call).



Kernel main thread (menu)

cmd_progthread thread becomes userprocess (palin)

```
common_prog(...) {

  proc = proc_create_runprogram (...);

  result = thread_fork(...,
    proc, cmd_progthread, ...);

  exit_code = proc_wait(proc);
  ...
}
```

```
proc_wait(...) {

 wait(...); /* sem or cv */
 /* get exit status
    and destroy userprocess
  */
```

```
/* KERNEL MODE */
cmd_progthread(...) {

  result = runprogram(“testbin/palin”);

}

runprogram(...) {

  load_elf();

  enter_new_process(…);
}
```

```
/* USER MODE: process palin */
main(...) {



  exit(0);
```

```
/* KERNEL MODE: system call */
syscall(...) {

  switch (callno) {

    sys__exit(status);
  }
 }
```

```
sys__exit(...) {

 signal(...); /* sem or cv */
 ...
 thread_exit();
}
```

Destruction of `struct proc`

ATTENTION: follow the advice to go forward one step at a time. Implement the destruction of the `struct proc` only after having done and verified, possibly with debugging, the `proc_wait()` function.

The `struct proc` of a process cannot be destroyed (during `_exit`) until another process that calls `wait/waitpid` receives the signal (and reads the exit status).

Among the various possible solutions to free a `struct proc` through `proc_destroy`, it is suggested to call the latter inside `proc_wait`, after waiting on a semaphore or condition variable (i.e. the struct is not destroyed when the process ends, but later on, inside the `proc_wait` function, called by another process, possibly the kernel).

This probably implies a modification to the previous implementation of the `sys__exit`, which now would no longer need to release the address space, but only to signal (to the semaphore or condition variable) the end of the process, before calling `thread_exit`. In other words, `sys__exit` terminates the thread, does not destroy the data structure of the process, but simply signals its termination.

The complete destruction of a process is performed by `proc_wait` after it has received the end signal. The `proc_wait` function also handles the return of the process exit status.

ATTENTION: the `proc_destroy()` function requires that the process being destroyed (data structure) no longer has active threads (see the `proc destroy` code, which contains the assertion `KASSERT(proc->p_numthreads == 0)`;). Since `sys__exit()` signals the end of the process before calling `thread_exit()`, it is possible that the kernel on hold (in `common_prog()`) is woken up and calls `proc_destroy()` before `thread_exit()` "detaches" the thread from the process. (see the `thread_exit()` code, especially the call to `proc_remthread()`).
The solution to avoid this race is not univocal. We suggest, as a possible option, to call `proc_remthread()` explicitly in the `sys__exit()`, "before" signalling the end of the process, and modify the `thread_exit()` function so that it accepts a thread already detached from the process (be careful, not to REQUIRE `thread_exit()` to ALWAYS see a "detached" thread: the `thread_exit()` is called also in other contexts, outside the `sys__exit()`).

Assigning `pid`

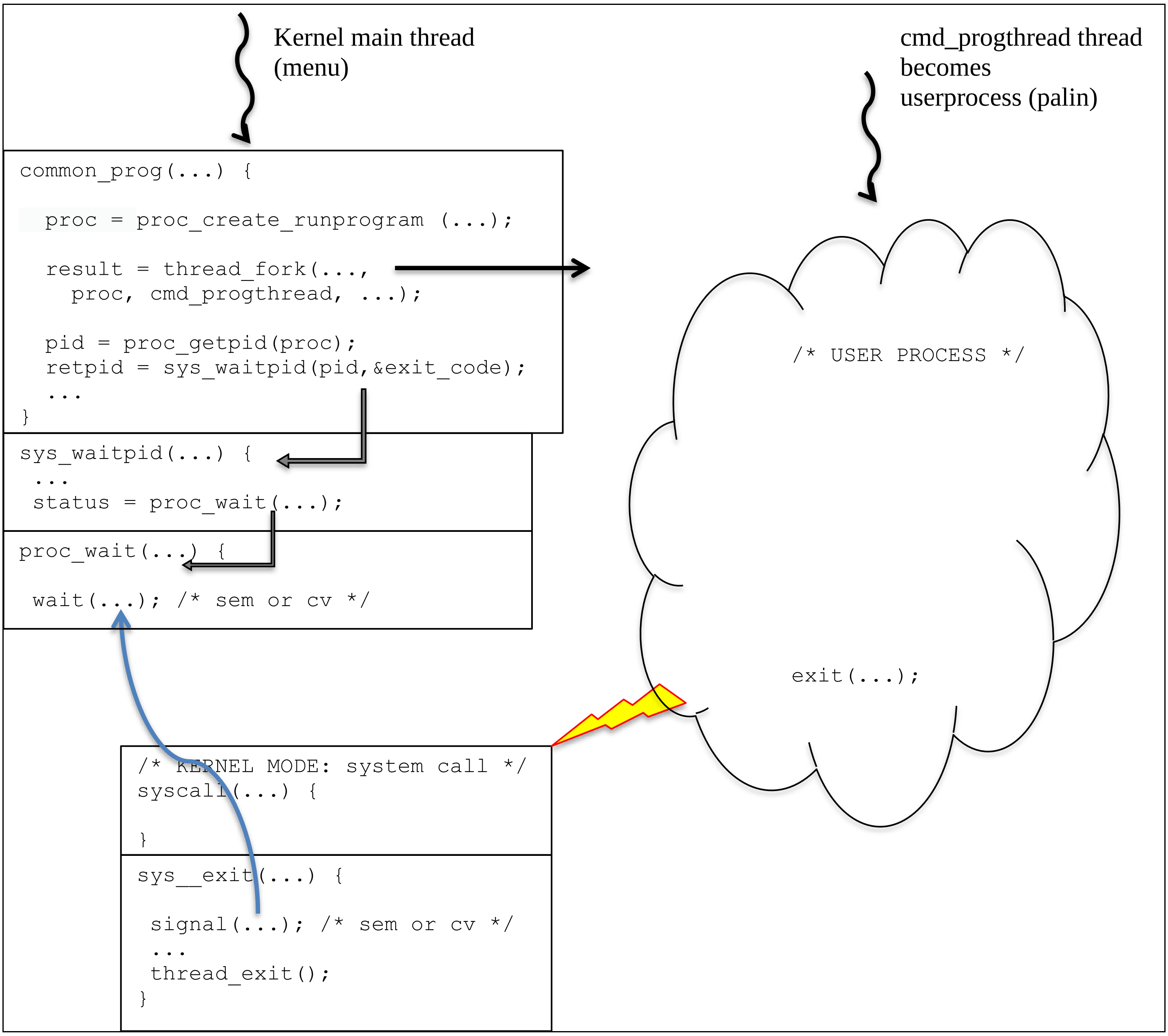
The `proc_wait` function does not completely accomplish the work required by the `waitpid`, as it starts from a process pointer (instead of a `pid`, an integer value).
For the attribution of a `pid` (process id) to a process, it must be taken into account that it is a single integer (type `pid_t`), with a value between `PID_MIN` and `PID_MAX` (`kern/include/limits.h`), defined according to `__PID_MIN` and `__PID_MAX` (`kern/include/kern/limits.h`). In order to assign a `pid` and to go from process (pointer to `struct proc`) to `pid` and vice versa, it is necessary to create a table. For simplicity, it is recommended to create an array of `struct proc` pointers, in which the index corresponds to the `pid` (use an acceptable number as the maximum `pid`, e.g. 100), or an array of pairs (`pid`, process pointer). A suitable `pid` field in the `struct proc` can instead allow to find the `pid` starting from the pointer. Each newly created process must be added to the table, generating its `pid`, each destroyed process must be removed from the table (once a `wait/waitpid` is completed). For example, the table can be created as a global variable in `kern/proc/proc.c`. It is then necessary to create a possible `sys_waitpid` function to call in `syscall`() (it is recommended to use the same file, e.g. `proc_syscalls.c`, already used for the `sys__exit`).

Process table and waitpid

The `common_prog`, internal function of the kernel, does not need to manage the `pid` of a created process (it already has the pointer), therefore it does not need, to wait for the end of the created process, support for the `waitpid` (which requires to manage the process table). Basically, the end of a process with `_exit` (and system call `sys__exit`) does not need, if it is the kernel that has the pointer waiting for it, `waitpid` (with process identified by `pid`), but `proc_wait` (process identified by pointer).
The `waitpid`, on the other hand, is necessary for user programs to manage processes. For example, `testbin/forktest` would allow you to check the `waitpid` operation. However, for this test program to work, it is necessary to implement the `getpid`, which obtains the `pid` of the current process (pointed to by `curproc`), and the `fork`, which allows you to generate a user-level child process.
This part (making `getpid` and `fork`) can be considered optional (it is recommended to try to make it only once the rest of the laboratory has been successfully completed): implementing the `getpid` is simple, the `fork` less, as it is cloning (duplicating, the entire address space of a process (the child is a copy of the father) and to start it correctly).

An easy way to test (without the `fork`), not the `waitpid` directly, but the possible `sys_waitpid` called in `syscall` to support it, is to obtain the child process `pid` in `common_prog` (through `proc_getpid()` or other strategy), and subsequently wait with `sys_waitpid` instead of `proc_wait`.

The following figure shows a possible scenario to verify the correctness of the `getpid/waitpid` chain and process table management, not directly, but indirectly through the `proc_getpid()` and `sys_waitpid()` functions.



```
common_prog(...) {

  proc = proc_create_runprogram (...);

  result = thread_fork(...,
    proc, cmd_progthread, ...);

  pid = proc_getpid(proc);
  retpid = sys_waitpid(pid,&exit_code);
  ...
}
```

```
sys_waitpid(...) {
 ...
 status = proc_wait(...);
```

```
proc_wait(...) {

 wait(...); /* sem or cv */
```

```
/* USER PROCESS */

exit(...);
```

```
/* KERNEL MODE: system call */
syscall(...) {

}
```

```
sys__exit(...) {

 signal(...); /* sem or cv */
 ...
 thread_exit();
}
```