Lab 06 – Martino Mensio

Exercise 1

The debugging of the system calls with an emulated multiprocessor produces every time a different sequence of system calls, and it seems that gdb (via ddd) is not able to capture every time one of them is called (only sometimes the `sys_exec`, `sys_pipe` and `sys_sbrk` are captured). To run qemu with only a single emulated processor, I modified the `Makefile` to use a single processor.

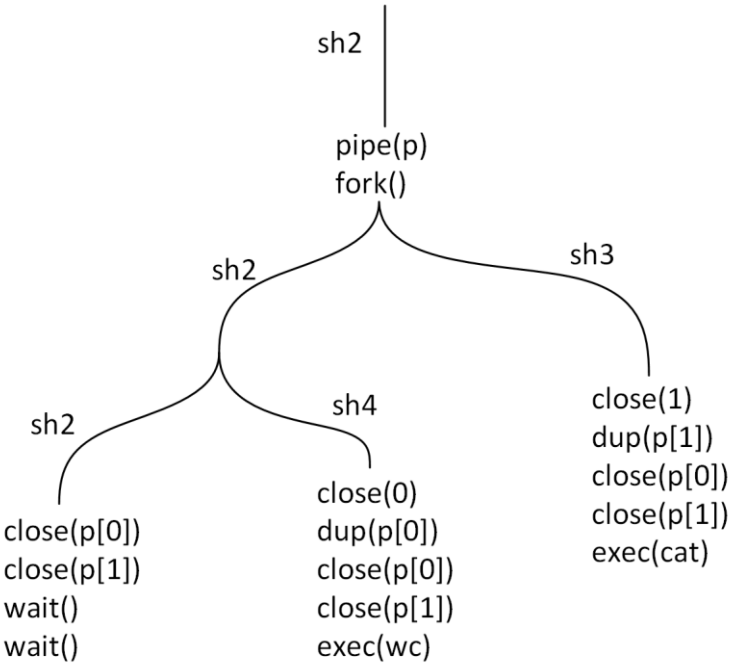
The execution of the command cat myname.txt | wc produces the following sequence of system calls, with small variations depending on each execution:

```
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 1 3 13 4 1 1 26 11 26 26 8 26 26 3 26 11
26 26 8 20 6 21 6 26 2 3 6 6 21 21 21 21 21 21 21 2 2 21 21 6
```

The analysis is the following:

- 6 (x20) (`SYS_read`): called 20 times by `sh` to read from the input the command, one character per time ("`cat myname.txt | wc\n`")
- 1 (`SYS_fork`): called by `sh` to fork. The child process (named "`sh2`" below) will execute the command
- 3 (`SYS_wait`): `sh` waits for child `sh2` to end. When child will terminate, `sh` will be able to read next command and fork again
- 13 (`SYS_sbrk`): this system call is for memory management, is similar to `malloc` and probably is called to allocate some space for the newly created process `sh2`
- 4 (`SYS_pipe`): `sh2` creates a pipe, that will be used by `cat` and `wc` to communicate. fd[0] = 3, fd[1] = 4
- 1 (`SYS_fork`): `sh2` forks for becoming `cat` (named as "`sh3`" below)
- 1 (`SYS_fork`): `sh2` forks for becoming `wc` (named as "`sh4`" below)
- 26 (`SYS_close`): `sh3` closes fd = 1 because `cat` will not use stdout
- 11 (`SYS_dup`): `sh3` calls the dup on the writing end of the pipe, so that when `cat` will use fd = 1, it will write on the pipe
- 26 (`SYS_close`): `sh3` closes fd = 3 because it already has fd = 1 to write on the pipe
- 26 (`SYS_close`): `sh3` closes fd = 4 because `cat` will not need to read from pipe
- 8 (`SYS_exec`): `sh3` becomes `cat`
- 26 (`SYS_close`): `sh2` closes fd = 3 because this process doesn't need the pipe
- 26 (`SYS_close`): `sh2` closes fd = 4 because this process doesn't need the pipe
- 3 (`SYS_wait`): `sh2` waits for the first child termination (could be both `cat` or `wc`)
- 26 (`SYS_close`): `sh4` closes fd = 0 because `wc` will not use stdin
- 11 (`SYS_dup`): `sh4` calls dup on the reading end of the pipe, so that when `cat` will use fd = 0, it will read from the pipe
- 26 (`SYS_close`): `sh4` closes fd = 3 because `wc` will not need to write to pipe
- 26 (`SYS_close`): `sh4` closes fd = 4 because it already has fd = 0 to read from the pipe
- 8 (`SYS_exec`): `sh4` becomes `wc`
- 20 (`SYS_open`): `cat` opens the file "`myname.txt`" receiving fd = 3

- 6 (`SYS_read`): `cat` asks for 512 bytes from "`myname.txt`", and receives 7 bytes containing "`Martino`"
- 21 (`SYS_write`): `cat` writes on the pipe the 7 bytes
- 6 (`SYS_read`): `cat` asks to read from the file, but receives 0 as return value, understanding the file was terminated
- 26 (`SYS_close`): `cat` closes fd = 3 associated to "`myname.txt`"
- 2 (`SYS_exit`): `cat` terminates
- 3 (`SYS_wait`): `sh2`, after being woken up by the child termination, waits for the other child to die
- 6 (`SYS_read`): `wc` reads from the pipe and receives 7 bytes
- 6 (`SYS_read`): `wc` reads from the pipe and receives 0 bytes, meaning that the pipe has no more data
- 21 (x7) (`SYS_write`): `wc` writes one after the other the resulting string to stdout. Since the string contains "`0 1 7 \n`", the write is called 7 times
- 2 (`SYS_exit`): `wc` terminates
- 2 (`SYS_exit`): `sh2` terminates
- 21 (x2) (`SYS_write`): `sh` writes again the prompt "`$ `", one character per time (two characters)
- 6 (`SYS_read`): `sh` waits for a new command



Exercise 2

In the second exercise we are required to write the system calls that manage the handling of condition variables. The specifications are given by a set of methods that we need to implement. Those are:

- `int cond_alloc()`
- `void cond_set(int cond, int val)`
- `int cond_get(int cond)`
- `void cond_destroy(int cond)`
- `void cond_wait(int cond)`

- `void cond_signal(int cond)`
- `void cond_broadcast(int cond)`

Since different processes, created using the system call `fork()`, don't have shared variables, the value to be tested is embedded inside the condition variable, and can be read using `cond_get`, and can be set using `cond_set`. The functions `cond_alloc` and `cond_destroy` allow the user programmer to manage the lifecycle of a condition variable. The other functions allow to wait on a condition variable (`cond_wait`), wakeup a single process waiting (`cond_signal`) and wakeup every process waiting (`cond_broadcast`).

Since the condition variables are used together with a mutex, the mutex can be both external (passed as a parameter to the `cond_wait` that needs to perform some actions with it), or can be embedded inside the condition variable itself. I chose the second approach, to have a centralized management of the lock.

The lock is not visible to the user program, but some actions need to be done by the user programmer on it (lock and unlock). So the condition variable must expose on its interface two more methods: `cond_lock` and `cond_unlock`, that respectively acquire the mutex and release it.

Writing system calls in xv6

The first part of the work has been a preliminary one, making modifications to some files in order to create the system calls and be able to call them from a user program. The files that have been modified are:

- `param.h`
- `user.h`
- `usys.S`
- `syscall.h`
- `syscall.c`
- `main.c`
- `defs.h`
- `Makefile`
- `sysfile.c`
- `file.c`

Also the files `st.c` and `cond_test.c` have been modified in order to test the behavior of the condition variables and of the semaphores.

The implementation

The implementation is contained in the file `file.c`. The internal structure of a condition variable contains:

- a spinlock: this is used to be able to protect the execution of functions in an exclusive way. Once it is acquired by a process, other processes that need it have to wait until it is released.
- locked: this variable can have values 0 or 1, and represents the state of the internal lock. The lock can be required by using `cond_lock` and released by using `cond_unlock`.
- count: represents the number of processes that are currently waiting on the condition variable.

- value: this is an internal variable that is stored inside the condition structure in order to be able to share its value between different processes. A process can read this value using `cond_get` and can write into this variable using `cond_set`.
- ref: this variable can have values 0 or 1, and if set to 1 means that the condition variable is currently used. Its value is set to 1 when calling the `cond_alloc`, to 0 when the `cond_destroy` is executed.
- resurrection_token: the content of this variable represents how many processes have the rights to continue their execution. Its value can be increased by the `cond_signal` and `cond_broadcast`, and is decremented by 1 by processes that are restarting their execution (see below the tokens system).
- q: this variable represents the channel where processes go to sleep during the `cond_wait`.
- m: this is a second channel, where processes asking for the lock (both inside `cond_lock` and `cond_wait`) can be sent to sleep.

Sleep and wakeup (proc.c)

The implementation is based on the use of the functions sleep and wakeup (both belong to the `proc.c` file). The `sleep` function receives two arguments: an address and a spinlock. The address is used as a channel: a process can be sent to a sleep state on different channels, and different channels can be used to partition the processes into groups. For this reason, when calling the `sleep` function this argument is set to an address of a variable inside the condition structure (q or m), because in this way we are sure that the channels do not overlap. The second argument of the `sleep` function is a spinlock, that is released atomically together with the suspension of the process.

The `wakeup` function receives a single parameter (the channel), that is used to wake up every process that is sleeping on this channel.

Those two functions are very similar to the condition variables primitives. The `sleep` can be considered a basic version of the wait primitive, while the `wakeup` is analogue to the broadcast primitive.

Extending functionalities of sleep and wakeup

The `sleep` and `wakeup` have the following limitations:

- they don't manage in a strong way the number of processes waiting
- they don't handle the signal scenario (we want to be able to wake up a single process)
- for a user program it can be dangerous to manipulate directly the spinlocks (the schedule will complain easily about spinlocks acquired and not released, calling the `panic` function)

For this reasons the condition functions need to be written as wrappers extending the functionalities provided by `sleep` and `wakeup`.

The condition variable contains a counter that represents the number of processes that are waiting. This value is modified only in `cond_wait`: is increased before going to sleep, and is decreased when the process is woken up. In this way, inside `cond_signal` and `cond_broadcast` we can read this value and decide some other things.

Tokens

Instead for the problem of being able to choose how many processes can wake up effectively, my idea was to implement a solution based on tokens, that are generated from one side and are consumed on

the other side. Each token can be used only by a process to continue its execution: if there are no tokens available the process is sent to sleep. So on the side that generates tokens we are able to decide how many processes can wake up.

The function `cond_signal` generates a token (after having checked the number of processes sleeping) and adds it to the tokens that can be used. Instead the function `cond_broadcast` sets the number of tokens available to the number of processes that are sleeping, so that there is one for each of them. Both functions then perform a wakeup on the channel. The processes sleeping are woken up and if a token is available they consume it and continue the execution, instead if there is no token they go to sleep again.

For this reason the `sleep` is enclosed inside a while loop, because a process always has to check again if there are tokens before continuing. The function `cond_wait` contains two while loops of this type: the first one is for the tokens generated by `cond_signal` and `cond_broadcast`, while the second one is for acquiring the lock (as in the `cond_lock`) because the process must resume execution with the lock acquired.

The `cond_lock` and `cond_unlock` use the same principle of tokens, but in a simplified way because there can be only 0 or 1 tokens (it is a mutex) respectively when the lock is already acquired by another process or when the lock is available.

Spinlock management

The spinlock inside the condition variable is necessary for guaranteeing the correct behavior of this system: in the sections of code protected by the spinlock, the functions are executed in mutual exclusion, not interleaved. When there is a call to the `sleep` function, the spinlock is passed as a parameter: in this way another process can acquire it. When the `sleep` function ends its execution the spinlock is acquired again. So there starts another section of code inside which the execution is exclusive.

If we consider in detail the function `cond_wait`, the critical regions (that will be executed in mutual exclusion with respect to all the other functions on the same condition variable) are the following:

- from line 227 to 236: update of the counter and unlocking of the lock
- from line 235 (the condition of the while evaluating to false) to 245: consumption of a token and update of the count of waiting processes. The protection of this set of instructions guarantees that a token is used by a single process
- from line 244 (the condition of the while evaluating to false) to 250: the acquisition of the lock. As for the previous region, the atomicity of those instructions on the condition variable guarantees that the lock is acquired only by one process, the fastest one that was able to acquire the spinlock

```
222  void cond_wait(int cn)
223  {
224    struct condition *c;
225
226    c = &condition_table.condition[cn];
227    acquire(&c->lock);
228    c->count++; // this process must be counted
229    // unlock the lock (same code as above in cond_unlock)
230    if(c->locked) {
231      c->locked = 0;
232      wakeup(&c->m);
233    }
234    // the WHILE puts back to sleep all the processes that have been woken up but weren't able to get a resurrection token
235    while(c->resurrection_token == 0) {
236      sleep(&c->q, &c->lock);
237    }
238    // take a resurrection token, those are so limited!
239    c->resurrection_token--;
240    c->count--; // update the count of waiting processes
241
242    // same code as in cond_lock
243    // WHILE instead of IF, makes sleep again processes that weren't able to acquire the lock
244    while(c->locked) {
245      sleep(&c->m, &c->lock);
246    }
247    // set c->locked to 1 so no other process can exit the preceding while loop
248    c->locked = 1;
249
250    release(&c->lock);
251  }
```

It is very important to notice that the spinlock is always released: directly by calling release or indirectly when a process goes to sleep. In this way the scheduler does not complain and the user program don't have to manage spinlocks.

Bug in the semaphores implementation

Examining the code of the semaphores and doing some tests, I found that the primitives don't work as expected in some situations: if for example there are some processes sleeping, and another one signals on the semaphore, all the processes sleeping are waked up (because they are all on the same channel) and continue their execution. There is no check after the sleep, and things go wrong. I added the idea of tokens also to the semaphore: the tokens are generated both at the beginning when `sem_init` is called with a positive argument, and also when the `sem_post` is called. They are consumed when available inside `sem_wait` and in this way I can control how many processes continue the execution.

Using condition variables

The pattern for using the `cond_wait` is the following:

- acquire the lock (using `cond_lock`)
- inside a while loop get the value stored inside the condition variable (using `cond_get`) and decide if suspend the execution (using `cond_wait`) or exit the loop
- perform some actions and release the lock when not more needed (using `cond_unlock`)

On another process we can set the value using `cond_set` and decide if wake up a single process waiting or all of them. Inside the file `cond_test.c` there are some tests that are written using this pattern and have been used to check the correct behavior of the condition variable.