

*Forfattere:*

13.07-89 Rane Eriksen BVL193  
05.01-90 Christian Nielsen BNF287  
14.06-79 Sebastian O. Jensen GJX653  
Hold 1

DATALOGISK INSTITUT  
KØBENHAVNS UNIVERSITET

OPERATIVSYSTEMER OG MULTIPROGRAMMERING

---

# G-Assignment 1

---

23. FEBRUAR 2015



UNIVERSITY OF  
COPENHAGEN

# Indhold

1	Task 1. Types and functions for userland processes in Buenos .	2
	Defining the data structure . . . . .	2
	System calls . . . . .	3
2	Task 2. System calls for user-process control in Buenos . . . .	4
	System calls . . . . .	4
3	Tests . . . . .	7
4	Worth mentioning . . . . .	7

# 1 Task 1. Types and functions for userland processes in Buenos

## Defining the data structure

In our data structure of our processes we wanted to make sure that the parent process could administer all of its children in case of early termination. We therefor implemented an array with the same size as the maximum process to ensure that any process had capacity for all possible processes.

We also implemented a count which counts how many the current parent has - This is only to shorten the algorithm when terminating the processes of a parent.

We initially wanted to create a linked list that could increase in size as needed, but `kmalloc` is only available while the kernel is initializing, hence the static size of our array.

The name of the executable file also has limitations due to `kmalloc` not being available and therefor we set the maximum size of the executable file name to be 64 characters.

It is also essential to a process to hold the return given by the process. This is used in two places, when exiting a and when a process is waiting for another process.

A process state is also important when a process needs to wait for another process, we therefor implemented three different states: `PROCESS_FREE` which states that the process is available to use, `PROCESS_RUNNING` which states that the process is busy, however it does not state if it is the actual processes running, one will have to look at the state of a thread to determine this. The last state is `PROCESS_ZOMBIE` which is set when a process terminates, this is used for processes waiting for another process to terminate.

lastly we implemented a container for the `thread_id` belonging to the process.

The reason for this container is to actually terminate the thread of a process if its parent die - We do not want to have any running threads in the background because of an eternal loop.

## System calls

### **SYSCALL\_EXEC**

To execute another program a program must call SYSCALL\_EXEC with a string as argument, the argument is a string that describes the disk it resides on and the name on the disk an example is "[disk]initprog".

The call to the system call returns the process\_id of the newly created process created by process\_spawn which is called by SYSCALL\_EXEC.

The process id can be used to ask the program to wait until the newly executed program has terminated.

### **SYSCALL\_EXIT**

There are two ways a program can exit itself, one way is by returning an integer and the other way is to call SYSCALL\_EXIT with an integer.

return actually calls SYSCALL\_EXIT so these are the same.

SYSCALL\_EXIT never returns a value and it calls process\_finish which is explained in detail later.

### **SYSCALL\_JOIN**

To have a program wait for another program, the program must call SYSCALL\_JOIN, the program calling this system call will wait until the system call returns with a response, there is no telling of how long this system call will take to return a value and have the program continue its execution.

SYSCALL\_JOIN takes one argument which is the process id of the process it wants to wait for, this id is granted when using SYSCALL\_EXEC.

SYSCALL\_JOIN calls process\_join which is explained in detail later.

## 2 Task 2. System calls for user-process control in Buenos

### System calls

We have implemented:

- `process_init`  
Initializes the `process_table`.  
When initializing the `process_table` we want to make sure that every process is available this is why we set the process state of every process to `STATE_FREE` which indicates that the process is not in use by any thread. We also want to initialize the `retval` to 0 and set the `parent_id` to -1.  
We set the `parent_id` to -1 to indicate that the process has no parent since a `process_id` must be at least 0.  
Lastly we set the `child_count` to zero and every entry in our `children` array to 0.
- `process_add_child` and `process_remove_child` We have implemented these two functions to keep track of a parents child processes.  
The two functions assumes that a lock is already acquired.  
The `child` array works in way that makes insertion and deletion run in  $\Theta(1)$  because the index is actually the id of the process and the value of the entry is either true or false.  
It is worth mentioning that when a process terminates it will run through every entry of its `child` array, looking for an entry with the value 1, the index is then used to terminate the child, and the index is also used to terminate every child of the child, the runtime of terminating a process can therefor be expensive which is why we implemented a count for the `children` array to keep track of how many children a process actually has. This is a substitute to a dynamic linked list.
- `process_spawn`  
This function is not a replacement of the `process_init`, but an initializer to our processes that we want to create.  
When `process_spawn` is called it will look for a process with the state `PROCESS_FREE`, if it fails to find a process with this state, the function will return -1 otherwise it will return the id of the newly created process.  
Once a process has been found and the state of it is `PROCESS_FREE`

we initialize the process in the same way that we initialize every entry of the `process_table` in `process_init` but we set the state to `PROCESS_RUNNING` instead of `PROCESS_FREE` to state that the process is not available.

We use two kinds of locks in this function, the first lock is acquired in the beginning of the function which holds a lock for the `process_table` that we wish to access and manipulate, we release this lock when the function has started the new process, before the function ends we wish to store the `thread_id` of the newly created thread which holds the newly created process.

We do this by acquiring the lock for threads which, we want to store thread `thread_id` because we want to be able to terminate the child threads and their process when a parent terminates.

We will explain this in the next.

- `process_finish` and `process_kill_children` When a process finishes we wish to keep the return value in the process in case of another process waiting for the current to terminate. We do this by storing the return value in the process.

The return value is given by the `syscall_exit`(some integer) or by the program return an integer as its last program command.

Because we are storing the value in the process we also set the state of the process as `PROCESS_ZOMBIE`. This is done for two reasons, the first reason is that we know that the function is not running anymore, but is not completely finished and therefor it should not be marked as `PROCESS_FREE`, we use `PROCESS_ZOMBIE` to indicate that the process is in a state where some other process needs to evaluate the return value of this process, we explain this in the next section.

Once the the return value has been set, we signal any process waiting for this process to terminate that it has by using `sleepq_wake` with the address of the process which is about to terminate.

Before finishing the thread completely we also want to terminate any child that the current process possess, we do so by calling the recursive function `process_kill_children` with the id of the currently running process.

what `process_kill_children` does is, that it runs through every entry of the `children` array belonging to the process associated with the `process_id` given as parameter when the function is called.

Every index in the children array is the actual process\_id of a process and the value of every entry in the children array is either 1 or 0.

When the value of the entry is 1 then this index is a child and this child must be terminated, not only the process but the thread that holds the process too.

We do this by modifying the program counter of the thread to point to process\_finish, because of this we also remove the index in the children array since we do not want this process to enter an eternal loop.

Once the thread we want to terminate gets CPU time it will terminate itself thus making it available in the thread\_table again.

As mentioned, this function is called recursively and every entry in a child array that has a value of 1 will call the function until every child has been set to self terminate.

- process\_join This function has only one job to do, it is designed to make a process wait for another process.

The function starts by checking that the pid(process\_id) of the process that the currently running process wants to wait for is within the range of valid process id's, it also checks that the process associated with the pid does not have the state PROCESS\_FREE. If either the pid is not within range or the state of the process is PROCESS\_FREE process\_join returns -1, we do this to make sure, that the currently running process do not risk waiting forever, because the process associated with the pid is either not running or simply does not exist.

If it is the case that the pid is valid the function acquires a lock for the process\_table, it then enters a while loop which terminates when the process associated with the pid changes its state to PROCESS\_ZOMBIE, however the lock is released inside the loop and the thread is switched - This ensures that the the process waiting for another process is not obtaining the CPU time, before releasing the lock, the address of the process associated with the pid is given to the function sleepq\_add which sets the sleeps\_on in the thread to the address of the process associated with the pid.

When process\_finish signals the cpu that a process is about to terminate, this process is reactivated, but only if it's the process associated with pid. Once reactivated the loop runs again and if the state is PROCESS\_ZOMBIE it leaves the loop and the the process state is set to PROCESS\_FREE, the lock is released and the return value is return

to the process that was waiting.

### 3 Tests

### 4 Worth mentioning

While our `process_join`, `process_finish` and `process_spawn` works as intended we have discovered an error when shutting down our children while they are in the process of being initialized, although this error only occurs when a program runs, spawns a new program and immediately shuts down.

Our tests have found it to happen within `process_start` when trying to use kernel functions that have already been shut down by `halt_kernel()`.

The problem may run deeper in the system, but a solution to the current problem could be to create a global lock for `buenos/halt.c` and `proc/process.c` to use.

When `halt_kernel(void)` in `kernel/halt.c` is called it should acquire a lock for the global lock, once obtained it should shut down the system without releasing the lock.

`process_start` in `proc/process.c` should also acquire the global lock before running its function, once obtained it should finish what is started and before exiting the function it should release the lock.

This way the kernel will only shut down while some of its is not in use.