Operating Systems Practical 2008/2009 – Documentation for assignment 3

Richard van Heuven van Staereling <richard@few.vu.nl>
Marc Hage <mrhage@few.vu.nl>

For this assignment, we had to implement a new server and a library that makes use of its services. The library provides the user access to operations to mutexes, condition variables, and threads. The synchronization server (SS) runs independently from the program requesting its services, allowing it to house synchronization primitives.

Communication between user program and server is done through system calls, like in the first assignment.

Mutex development

Since there is a fixed number of mutexes allowed, storing them in an array makes sense. We created a **mutex_t** type containing the mutex's state and, if applicable, its holder. Processes may wait until a lock on a mutex is released, so they need to be stored in the mutex as well. Since the assignment does not state that there is a fixed number of waiting processes allowed, we implemented it as a simple linked list **wait_queue** (of **wait_t** type, containing the number of the waiting process and a reference to the next one, **NULL** implying the end of the linked list) that we treat like a queue. The **mutex_create()** and **mutex_destroy()** calls are initialization of these (globally defined) data structures and setting the state to unused respectively.

For implementing **mutex_lock()**, we had to figure out a way to block the program in case the critical region is already locked by someone else. In the beginning, we had thought about "moving" the wait to the SS, resulting in the SS blocking until it can send the reply back to the system call. However, since the SS is just one running program, blocking it completely disables its functionality. Through inspection of other servers, we learned that simply skipping the reply is the way to block system calls. But, this requires us to take extra care in making sure replies are being sent eventually, to avoid infinitely blocking programs.

The sending of the reply is taken care of in **mutex_unlock()**. Since the waiting processes are stored in a linked list, we have to make sure not to forget to **free()**. Our solution is to declare another linked list similar to the one stored in the mutex: the **reply_queue**. Sending a reply is as simple as moving an element in the mutex's linked list to this list and the main loop in the SS will take care of the rest (i.e., check if the list contains something and send any outstanding replies). The benefit of doing this is that everything is put in one place; if we put the **free()** here then we will not have to worry about that ever again. Also, all the replies generated by a service call are queued up and then sent at once. Although it should not be a problem to send everything one by one, it is a bit more efficient and easier to debug.

All that is left is making sure all items from the mutex's **wait_queue**s are moved to the **reply_queue** eventually. Therefore we had to think of situations in which this doesn't happen. One example would be a faulty user program or one that is inadvertently killed. The only way to destroy/unlock the mutex afterwards is manually, but an automatic way is preferred. We considered two methods for this. The first is to change the reply mechanism. If the SS finds out a pending reply is destined for a dead process, it unlocks the mutex and removes any items in queues matching the process number. The second method is modifying the **do_exit()**-call, letting it inform the SS whenever a process dies.

We tried the first method first because modifying other servers seemed a bit too drastic. However, the end result was an infinite loop. We realized that when a process dies, a new one may be created, getting the same process number. The SS sees that the process is still alive and believes everything is still OK. Also, cleanup, if needed and if it were possible, would not be immediate. This is not the case if we modify the exit routine. Although we were reluctant to do this, we noticed that what we want is already done by the file system: there is a function **tell_fs()** that informs the FS whenever a process ends. **tell_ss()** was then a piece of cake.

Condition variables development

Implementing the condition variables proved to be a bit more difficult. We started with a three-dimensional array next to the mutex-array, but ended up adding an array inside the mutex structure. Since condition variables belong to the mutex, this is only logical. By giving each condition variable its own **wait_queue**, we could reuse the linked list manipulating functions. Another side effect is that it is very simple to implement **cond_signal()**. All that is required is moving an element from the condition variable's **wait_queue** to the mutex's **wait_queue**; the rest of the code will take care of it.

Threads development

For threads, the final implementation is rather simple: creating a thread is basically just a **fork()** and waiting consists of a call to **waitpid()** (with flag **WNOHANG** in case **blocking_flag** is not set). The assignment is to make sure the functions work according to their interface but a lot is left unanswered. Behavior in special cases is not described in the assignment. For example, a process creating a thread and then forking should copy the thread for correct execution. Or, should a thread be able to create a thread? Although there are no tests for this, we feel that they are still important situations and that their behavior should be specified.