Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών
Τμήμα Πληροφορικής και Τηλεπικοινωνιών

Operating Systems

Assignment 2

Name: Helen Fili **sdi:** 1115202100203

---

1 System Calls

To implement the new system calls, the following modifications are required:

- **usys.pl**: Two entries, one for each new system call, were added to generate the corresponding assembly code.
- **syscall.h**: System call numbers are defined here.
- **syscall.c**: Prototypes for the functions handling system calls were added, and syscall numbers from `syscall.h` were mapped to the corresponding functions.
- **user.h**: Prototypes for the two system calls were added.
- **sysproc.c**: The implementation of the two system calls is located here.

To implement the priority based scheduler, it's necessary to introduce an additional field (`priority`) in the definition of `struct proc` within `proc.h`.

The default priority is 10, and during initialization in the `allocproc()` function within the file, all processes are set to this value. In the `freeproc()` function, when a process is freed, its priority is reset to 0.

1.1 `int setpriority(int num)`

In the `sysproc.c` file, the `setpriority` system call is implemented using the function `sys_setpriority()`. The function starts by retrieving the argument using `argint()`, followed by a call to the auxiliary function `int setpriority(int n)`.

The code for the `setpriority` function is in the `proc.c` file, and its prototype has been added to `defs.h`. The function ensures that the provided priority is within the acceptable range. Subsequently, it retrieves the active process and sets its priority to the specified value, acquiring and releasing the corresponding lock.

1.2 `int getpinfo(struct pstat* pstat)`

We created `pstat.h`, which contains the definition of `struct pstat`, responsible for storing information about each process. This struct encompasses details such as the total number of processes, arrays for the process IDs, parent IDs, priorities, states, names, and sizes of all processes. For the proper inclusion of the `getpinfo` definition in the `user.h` file, it is essential to add the incomplete definition of `struct pstat` to it.

In the `sysproc.c` file, the `getpinfo` system call is implemented using the function `sys_getpinfo()`. The function starts by retrieving the argument using `argaddr()` -which is a user space address-, followed by a call to the auxiliary function `int getpinfo(struct pstat* pstat)`.

The code for the `getpinfo` function is in the `proc.c` file, and its prototype has been added to `defs.h`. The function iterates through the processes, collecting information for each process whose state is not `UNUSED`, and stores the relevant details in a kernel space `struct pstat` variable, acquiring and releasing the appropriate locks when necessary. The final step involves using `copyout` to securely transfer the contents of the kernel space variable to the designated user space address.

Note: In the case of the `init` process, the parent ID is specifically set to 0. This choice reflects the nature of the `init` process as the initial user-space process initiated by the kernel during system startup and therefore has no parent process.

2 User Program `ps`

We added the `user/ps.c` file, which includes the implementation of the `ps` command. To enable program execution, we updated the Makefile by incorporating `$U/_ps` into `UPROGS`.

The function invokes `getpinfo` syscall to obtain a `struct pstat` containing the relevant information for each active process. It then iterates through this structure to print the details of each one of them.

3 Priority Based Scheduler

3.1 Implementation Details:

To implement the **priority-based scheduler**, we made modifications to the `scheduler()` function in `proc.c`. The updated scheduler selects the `RUNNABLE` process with the highest priority. In the event that multiple processes share the same priority, the scheduling mechanism employs the round-robin algorithm.

As in the previous implementation, the core logic of the scheduler resides within an infinite loop. Specifically, it begins with identifying the runnable process with the highest priority, retaining the lock of that process until a process with a higher priority is encountered. Once all processes have been considered, the scheduler checks whether the current highest priority matches the priority of the last executed process. If they are the same, the scheduler continues the execution using the round-robin algorithm. Otherwise, it initiates a new round-robin execution, starting from the process with the highest priority.

3.2 Scheduler Soundness

To assess the reliability of the scheduler, we compiled the `xv6` kernel using the following flags and executed the provided `priotest` file:

- `CPUS=1`: This setting ensures that priorities are consistently respected, and the results remain deterministic. In a multi-CPU scenario, processes with lower priorities might complete their execution earlier than anticipated, introducing uncertainty into the evaluation of our implementation's correctness. As observed in Figure 1, our implementation is sound.
- `CPUS=3`: This setting ensures that priorities are consistently respected in a multi-CPU scenario. The potential for lower-priority processes to finish earlier is expected due to the availability of multiple CPUs for concurrent execution.
- `CPUS=8`: This configuration is essential to test for the occurrence of deadlocks. Running `priotest` did not reveal any deadlocks.

```
$ priotest
Child pid 19, small, with priority 2 finished. Useless sum: 448743748
Child pid 38, small, with priority 2 finished. Useless sum: 448743748
Child pid 39, small, with priority 3 finished. Useless sum: 448743748
Child pid 20, small, with priority 3 finished. Useless sum: 448743748
Child pid 40, small, with priority 4 finished. Useless sum: 448743748
Child pid 21, small, with priority 4 finished. Useless sum: 448743748
Child pid 41, small, with priority 5 finished. Useless sum: 448743748
Child pid 22, small, with priority 5 finished. Useless sum: 448743748
Child pid 42, small, with priority 6 finished. Useless sum: 448743748
Child pid 23, small, with priority 6 finished. Useless sum: 448743748
Child pid 43, small, with priority 7 finished. Useless sum: 448743748
Child pid 24, small, with priority 7 finished. Useless sum: 448743748
Child pid 44, small, with priority 8 finished. Useless sum: 448743748
Child pid 25, small, with priority 8 finished. Useless sum: 448743748
Child pid 45, small, with priority 9 finished. Useless sum: 448743748
Child pid 26, small, with priority 9 finished. Useless sum: 448743748
Child pid 8, small, with priority 10 finished. Useless sum: 448743748
Child pid 27, small, with priority 10 finished. Useless sum: 448743748
Child pid 46, small, with priority 10 finished. Useless sum: 448743748
Child pid 28, small, with priority 11 finished. Useless sum: 448743748
Child pid 47, small, with priority 11 finished. Useless sum: 448743748
Child pid 9, small, with priority 11 finished. Useless sum: 448743748
Child pid 29, small, with priority 12 finished. Useless sum: 448743748
Child pid 10, small, with priority 12 finished. Useless sum: 448743748
Child pid 30, small, with priority 13 finished. Useless sum: 448743748
Child pid 11, small, with priority 13 finished. Useless sum: 448743748
Child pid 31, small, with priority 14 finished. Useless sum: 448743748
Child pid 12, small, with priority 14 finished. Useless sum: 448743748
Child pid 32, small, with priority 15 finished. Useless sum: 448743748
Child pid 13, small, with priority 15 finished. Useless sum: 448743748
Child pid 14, small, with priority 16 finished. Useless sum: 448743748
Child pid 33, small, with priority 16 finished. Useless sum: 448743748
Child pid 5, large, with priority 16 finished. Useless sum: 1818368003
Child pid 15, small, with priority 17 finished. Useless sum: 448743748
Child pid 34, small, with priority 17 finished. Useless sum: 448743748
Child pid 6, large, with priority 17 finished. Useless sum: 1818368003
Child pid 16, small, with priority 18 finished. Useless sum: 448743748
Child pid 35, small, with priority 18 finished. Useless sum: 448743748
Child pid 7, large, with priority 18 finished. Useless sum: 1818368003
Child pid 36, small, with priority 19 finished. Useless sum: 448743748
Child pid 17, small, with priority 19 finished. Useless sum: 448743748
Child pid 18, small, with priority 20 finished. Useless sum: 448743748
Child pid 37, small, with priority 20 finished. Useless sum: 448743748
Child pid 4, large, with priority 20 finished. Useless sum: 1818368003
```

Figure 1: Priotest Results for `CPUS=1`