A4: Synchronisation in KUDOS

Department of Computer Science (DIKU)
Computer Systems 2016

November 28, 2016 (v1)

This is the fifth in a series of seven assignments in the DIKU course Computer Systems 2016. We encourage pair programming, so please form groups of 2 students, but no more.

For this assignment you will receive a mark out of 3 points. You must attain at least half of the possible points to be admitted to the exam. Resubmission is not possible. Furthermore, you must get at least 2 points in each of the topics (ARC, OS, and CN); each topic will have two assignments. This is the *second* assignment in operating systems (OS).

The deadline for the assignment is Tuesday Dec 6 at 10 pm.

If you work alone you have to submit

- `report.pdf` — with your report.
- `kudos.zip` — with the directories `kudos/kudos/` and `kudos/userland`. The `kudos/` directory should be a working KUDOS source tree. Do not select single files, but ZIP the whole folder. We hand out a `Makefile` to make this as quick and painless as possible.

If you work in pairs (only) *one* of you should submit the above, and a third file, `group.txt` — containing your *KU-ids* separated by line breaks.

Before You `_start`

In this assignment you will be required to modify the KUDOS source code. There is a source code handout alongside this assignment. Use this as the basis for your solution. If you haven't already done so, please also take a look at the KUDOS style guide (`STYLE.md`) and KUDOS documentation[1].

Since the last handout, we have:

- Updated the TTY driver, and provided a working implementation of `syscall_read` and `syscall_write` for file descriptors 0, 1, and 2.
- Provided a baseline implementation of `syscall_spawn`, without support for the flags from A3.
- Provided system calls and userland system call wrappers for the semaphore functionality asked for below.

You are welcome to build upon your own solution to A3.

[1] http://kudos.readthedocs.io/en/latest/

1 Shutdown of User Processes (30 %)

KUDOS supports kernel threads as defined in `kudos/kernel/thread.h` with type `thread_table_t` and in A3 you extended it to support start-up of *user processes*. Your task is now to implement the correct shut-down of user processes. This includes both reaping child processes, and joining on threads through user-level synchronization.

1.1. Extend your library of helper functions for process management in the kernel, to also contain the functions described below.

```
/// Stop the current process and the kernel thread in which it runs
/// Argument: return value
void process_exit(int retval);

/// Wait for the given process to terminate, return its return value,
/// and mark the process-table entry as free
int process_join(process_id_t pid);
```

When implementing `process_join()`, the calling process will need to wait until a given event occurs. This is best implemented using the kernel-provided sleep queue; see the section on advanced synchronization[2] in the KUDOS documentation.

In `process_finish()`, use the code shown below before calling `thread_finish()`, where `thr` is the kernel thread executing the process that exits. This cleans up the virtual memory used by the running user process.

```
vm_destroy_pagetable(thr->pagetable);
thr->pagetable = NULL;
```

1.2. Expose the above functions to userland via the following system calls:

```
/// Exit the current process with exit code 'retval'. Note that
/// 'retval' must be non-negative since syscall_join's negative
/// return values are interpreted as errors in the join call
/// itself. This function will never return.
void syscall_exit(int retval)
{
  _syscall(SYSCALL_EXIT, (uintptr_t)retval, 0, 0);
}

/// Wait until the execution of the process identified by 'pid' is
/// finished. Returns the exit code of the joined process, or a
/// negative value on error.
int syscall_join(int pid)
{
  return (int)_syscall(SYSCALL_JOIN, (uintptr_t)pid, 0, 0);
}
```

[2] http://kudos.readthedocs.org/en/latest/advanced-synchronization.html

1.3. Write a number of userland programs and test your implementation.

At the very least, you should implement one program `userland/proc.c`.

You should be able to reuse your test programs from A3.

Userland semaphores for KUDOS (30 %)

The basic KUDOS system supports kernel semaphores as defined in the file `kudos/kernel/semaphore.h` with the type `semaphore_t`, but these cannot be used to synchronize userland processes – yet!

Your task is to use kernel semaphores to implement userland semaphores.

The system call interface in KUDOS userland libc already provides wrappers for these calls, so you should not modify those files.

A kernel semaphore is available to any kernel thread that knows its address; for userland semaphores we instead identify a semaphore by a text string.

In the kernel, incorporate the following files into KUDOS:

- `kudos/proc/usr_sem.h`
- `kudos/proc/usr_sem.c`

Use them to implement the following types and functions, using kernel semaphores for all underlying operations:

```
typedef struct {
  ...
} usr_sem_t;
```

The `usr_sem_t` type is at the center of your solution. Define enough attributes for the functions below to work correctly.

You will also need to define a static array of `usr_sem_t` entries and initialize them, like with `process_table` for user processes, or `thread_table` for kernel threads.

```
usr_sem_t* usr_sem_open(const char* name, int value);
```

Return a handle to a userland semaphore identified by the string `name`, which can then be used in the future to refer to this semaphore.

If the argument `value` is zero or positive, a fresh semaphore of the given name will be created with `value`, unless a semaphore of that name already exists, in which case `NULL` should be returned.

If `value` is negative, the call to `usr_sem_open` returns an existing semaphore with the given name, unless such a semaphore does not exist, in which case `NULL` is returned.

```
int usr_sem_close(usr_sem_t* sem);
```

Try to remove the userland semaphore `sem`, making its space available for future semaphores. Fail if there are processes currently blocked on the semaphore. *Be aware* that there is a potential race condition if a process blocks on a semaphore while it is being closed. You should not handle this race condition, but you should write about it. Return 0 if nothing went wrong, or a negative number in case of an error.

```
int usr_sem_p(usr_sem_t* sem);
```

Procure (execute the P operation on) the userland semaphore `sem`. Return 0 if nothing went wrong, or a negative number in case of an error.

```
int usr_sem_v(usr_sem_t* sem);
```

Vacate (execute the V operation on) the userland semaphore `sem`. Return 0 if nothing went wrong, or a negative number in case of an error.

Then call these user semaphore functions from `kudos/proc/syscall.c`. Use the matching system call numbers defined in `kudos/proc/syscall.h`:

```
...
#define SYSCALL_SEM_OPEN  (0x301)
#define SYSCALL_SEM_CLOSE (0x302)
#define SYSCALL_SEM_P     (0x303)
#define SYSCALL_SEM_V     (0x304)
...
```

Finally, add a userland program to test that your userland semaphores work correctly. Use `syscall_spawn` to spawn several processes, and make them do work with the same semaphore(s).

Note that, while the KUDOS handout code does contain a basic spawn-capable `syscall_spawn`, there is no proper support for `syscall_join` or `syscall_exit` (since they are not critical to being able to test userland semaphores).

The userland semaphore `syscall` functions have the same names and type names as in your `kudos/proc/usr_sem.h`, except that function names have a `syscall_` prefix.

Please consult the chapter on advanced synchronization in the KUDOS documentation for background knowledge[3].

Hints:

- Specify a maximum length for the semaphore name in order to have equally-sized struct elements in your semaphore table.
- To tell KUDOS' `make` system that there is a new file in town, modify `module.mk` – each subdirectory has one of these.
- Even though the `usr_sem_t` type is called the same in both the kernel and in userland, it does not actally have to be the same type. The only important part is that every `usr_sem_t*` value in userland means something in the kernel, as userland programs should never need to dereference it.

Optional: IPC Namespaces

Much like PIDs and FDTs, semaphores can be treated as a resource shared only among some processes, but isolated from others. Named semaphores, are one of a range of classical mechanisms for *interprocess communication* (IPC).

Implement support for the spawn flag `SPAWN_NEWIPC` (`0x4`), which when specified starts the process in a new IPC namespace. A process that starts in a new IPC namespace has no semaphores allocated, and cannot access already allocated, or new allocate semaphores in a parent namespace.

[3] https://kudos.readthedocs.org/en/latest/advanced-synchronization.html

2 Report (40%)

Alongside your solution, you should submit a short report. The report should:

2.1. Discuss how your implementation fits within the overall design of KUDOS.

2.2. Discuss in-how-far your implementation is "bullet-proof". (See the KUDOS system calls documentation [4] for a definition of bullets.)

2.3. Discuss other non-trivial parts of your implementation and your design decisions, if any.

2.4. Discuss how you tested your solution and assess the quality of your implementation. Make your test results easily reproducible (e.g., using a phony `Makefile` target), and tell us how to reproduce your test results.

2.5. Disambiguate any ambiguities you might have found in the assignment.

2.6. Answer the following question: Could you use semaphores (rather than sleep queues) to implement process join? Argue why, or why not.

[4] http://kudos.readthedocs.org/en/latest/system-calls.html.