Karlsruhe Institute of Technology

Operating Systems 2018/19
Assignment 4

Prof. Dr. Michael Beigl
Mathias Gottschlag, M.Sc.
Jonas Kiemel, M.Sc.

Submission Deadline: Sunday, November 25th, 2018 – 23:59

A new assignment will be published every week, right after the last one was due. It must be completed before its submission deadline.

The assignments must be filled out online in ILIAS. Handwritten solutions are no longer accepted. You will find the online version for each assignment in your tutorial's directory. **P-Questions** are programming assignments. Download the provided template from ILIAS. Do not fiddle with the compiler flags. Submission instructions can be found on the first assignment.

In this assignment you will get familiar with the pthread library.

T-Question 4.1: Processes and Threads

a. A process creates a new kernel level thread. Name at least two structures in memory and one operation that the kernel needs to allocate or perform until the CPU executes the new thread. For each structure, denote if they are placed in kernel or user space. **2 T-pt**

b. What command could you use in Linux to list the threads of a certain process? **1 T-pt**

c. A process creates two threads T1 and T2. When T1 executes a recursive function, the process crashes with a stack overflow error. When instead T2 executes the same function, the exception does not occur. What might be the reason? **1 T-pt**

d. Describe how dynamic shared libraries can be loaded at any virtual address. **2 T-pt**

P-Question 4.1: Worker Pool

Download the template **p1** for this assignment from ILIAS. You may only modify and upload the file `workerpool.c`.

Threads allow an application to perform tasks asynchronously and in parallel to other jobs (e.g., process a web request in a web server). However, spawning a new, dedicated thread for every task introduces considerable time and memory overhead. This is especially the case if tasks are short. Hence, in practice, it is better to create a (fixed) pool of worker threads that receive work from a work queue, execute it, and then return to the queue for more work. If the queue is empty, the threads block and wait until new jobs appear.

In this question you will build a simple worker pool with the help of pthreads. Jobs are expressed by providing a function pointer (`WorkFunc`) and an integer argument. **Use `assert()` or explicit error handling where appropriate!**

a. The work queue in this question will be build with a ring buffer of a fixed size (`MAX_JOBS`). The template already contains the necessary structures to represent the buffer (`_workItems`) and tasks (`WorkItem`). The variable `_nextJob` holds the index of the next valid task (if any) within the buffer. The variable `_numJobs` provides the number of jobs in the buffer. It may not exceed `MAX_JOBS`. Implement the functions to add and remove jobs. Use the following guideline: **3 P-pt**

`_enqueue` Adds a new task to the queue's *tail*.

- Returns -1 if the queue if full.
- Otherwise, adds the new work item to the tail of the queue by copying it at the right location into the buffer and updating the variables `_nextJob` and `_numJobs` if necessary.
- Returns 0 on success.

`_dequeue` Removes an item from the queue's *head*.

- Returns -1 if the queue is empty.
- Otherwise, copies the next work item from the queue into the supplied `item` argument and updates the variables `_nextJob` and `_numJobs` if necessary.
- Returns 0 on success.

Hints: If the queue is empty `_numJobs` should be `0`. `_nextJob` should always point at the head of the queue (i.e., the next open job), if any.

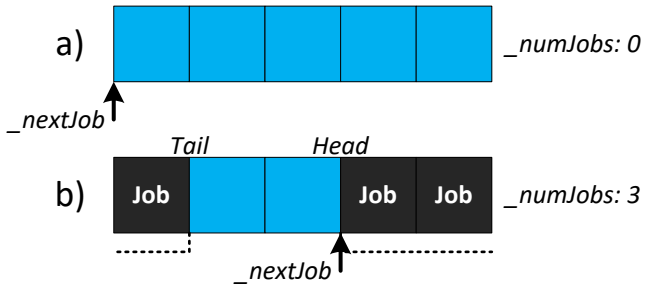

Abbildung 1: Example queue of size 5. a) Empty queue b) Queue with 3 open jobs after 3 jobs have already been pulled.

```
int _enqueue(WorkFunc func, int arg);
int _dequeue(WorkItem *item);
```

b. A worker pool should be tuned to the resources provided by the hardware. Modify the initialization of `n` in `initializeWorkerPool()` so that the variable holds the *number of available processors*. Use the `sysconf` system call. `n` should be at least 4, even if less processors are available. **1 P-pt**

```
int initializeWorkerPool(void);
```

c. Implement the `_startWorkers()` function that creates `num` worker threads. Your implementation should satisfy the following requirements: **2 P-pt**

- Creates new worker threads, which start execution in `_workerMain()` and receive their thread number (i.e., 0, 1, 2, 3, etc.) as argument.
- Returns 0 on success, otherwise does not terminate already created threads, but instead just returns -1. Expect a call to `finalizeWorkerPool()` in that case.

Hints: You may define further global variables and initialize them at the marked position in `initializeWorkerPool()`. Do not forget to free any additional resources in `finalizeWorkerPool()`.

```
int _startWorkers(uint32_t num);
```

d. Implement the `_waitForWorkers()` function, which waits until all worker threads have exited. Your implementation should be callable from `initializeWorkerPool()` (indirectly) and thus should not expect all data structures and the final number of worker threads to be created and initialized. **1 P-pt**
Hints: The function does not need to wait until all pending work has been processed. However, it should not forcibly terminate the workers!

```
void _waitForWorkers(void);
```

e. Complete the `_workerMain()` function, which is executed by the worker threads. Your implementation should satisfy the following requirements: **1 P-pt**

- Repeatedly calls `_waitForWork()` until the function returns 0.
- Executes each received job, by calling the work items function with the appropriate argument.

```
void* _workerMain(void *arg);
```

Total:
6 T-pt
8 P-pt