OS192: Assignment 2

Kernel Threads and Synchronization

03/04/2019

Responsible TAs: Or Dinari, Simion Novikov
Due Date: 29/4/2019 23:59

1 Introduction

The assignment main goal is to teach you about synchronization mechanisms and XV6 process management. First, you will implement a Kernel-level threads framework, which will enable the user to use multi-threading. Next, you will implement in the kernel a simple synchronization primitive, the mutex, and using the mutex you have added, you will implement a tournament based mutex for N processes in the user space. Lastly, you will create a sanity test for the above implementations.
The assignment is composed of the following parts:

1. Implement the Kernel-level threads framework
2. Implement the mutex synchronization primitive in the kernel space.
3. Implement a tournament based mutex for N threads in the user space.
4. Create a sanity test for testing the above 3 implementations.

Use the following git repository:
http://www.cs.bgu.ac.il/~os192/git/Assignment2

 Before writing any code, make sure you read the **whole** assignment.

2 Kernel Level Threads

Kernel level threads (KLT) are implemented and managed by the kernel. Our implementation of KLT will be based on a new struct you **must** define called *"thread"*. One key characteristic of threads is that all threads of the same process share the same virtual memory space. You will need to decide (for each field in the proc struct) if it will stay in proc or will move to the new struct thread. Be prepared to explain your choices in the frontal check. Thread creation is very similar to process creation in the fork system call but requires less initialization, for example, there is no need to create a new memory space.

1. We added a file named *kthread.h* which contains some constants, as well as the prototypes of the KLT package API functions. Review this file before you start.
2. Read this whole part thoroughly before you start implementing.
3. Understand what each of the fields in proc struct means to the process.

To implement kernel threads as stated above you will have to initially transfer the entire system to work with the new thread struct. After doing so every process will have 1 initial thread running in it. Once this part is done you will need to create the system calls that will allow the user to create and manage new threads.

2.1 Moving to threads

This task includes passing over a lot of kernel code and understanding any access to the proc field. Once this is done, figure out how to replace those accesses appropriately in order to replace the proc field with a new thread field. Another important issue is understanding how to synchronize changes in process state in a multi-CPU environment. Add a constant to *proc.h* called **NTHREAD = 16**. This will be the maximum number of threads each process can hold.

Every CPU holds specific data, this data can be accessed by using the function *mycpu()*, which is defined in *proc.c*. In *proc.h* you will find both the structs for *cpu* and *proc*, you must update the *cpu* struct to support the *thread* struct you have defined earlier. In addition, you should define a new function *mythread*() to access the current running thread (and fix *myproc*() if needed), consider the places where *myproc*() is used, and decide if it should be changed to *mythread*() instead.

As some pointers, you can ask yourself the following questions:

- What is the role of *allocproc*()? which parts of it should be moved to the thread creation, and which part should remain as it is?
- What is happening during the initialization of the system?
- What is happening when a process terminates? what happens when just a thread terminates?
- Where is the process kernel stack allocated? where is the process user stack allocated?
- And more...

The scheduler needs to be updated as well, instead of checking processes for $RUNNABLE$ status, it must go over threads. The scheduler will pass through all the process threads before moving to the next process.

Note that now multiple cores may run different threads of the same process simultaneously. Specifically, they will share the process resources, and must be synchronized to allow normal flow. Consider all the places where multiple CPU's may access the same resource, and either synchronize it or do not, be prepared to explain your decision. One example of such places is the function *growproc*(*int n*), defined in *proc.c*, which is responsible for expanding a process memory when needed.

Existing system calls:

fork Should only duplicate the calling thread, if other threads exist in the process they will not exist in the new process

exec Should start running on a single thread of the new process. Notice that in a multi-CPU environment a thread might be still running on one CPU while another thread of the same process, running on another CPU,is attempting to perform exec. The thread performing exec should "tell" other threads of the same process to destroy themselves upon exit from user mode and only then complete the exec task. Hint: You can review the code that is performed when the $proc \rightarrow killed$ field is set and write your implementation similarly.

exit Should kill the process and all of its threads, note that while a thread running on one CPU is executing exit, other threads of the same process might still be running.

Note that after completing the part above, you should be able to pass the **usertests** successfully. **This will be checked during the frontal exam!**

2.2 Thread system calls

You will implement several system calls to support the kernel threads framework, which will enable the user to create new kernel threads.

```
int kthread_create(void (*start_func)(), void* stack);
```

Calling *kthread_create* will create a new thread within the context of the calling process. The newly created thread state will be *RUNNABLE*. The caller of *kthread_create* must allocate a user stack for the new thread to use, whose size is defined by the MAX_STACK_SIZE macro in *kthread.h*. This is in addition to the kernel stack, which you will create during the function. $start\_func$ is a pointer to the entry function, which the thread will start executing. Upon success, the identifier of the newly created thread is returned. In case of an error, a non-positive value is returned.

> The kernel thread creation system call on real Linux does not receive user stack pointer. In Linux the kernel allocates the memory for the new thread stack. Since we did not teach memory management yet, you will need to create the stack in user mode and send a pointer to its end in the system call. (Allocating the stack in the kernel, which we don't do in this assignment, means you will later need to manage virtual memory addresses when a stack is removed or created.)

```
int kthread_id();
```

Upon success, this function returns the caller thread's id. In case of error, a negative error code is returned. Note that the ID of a thread and the ID of the process to which it belongs are, in general, different.

```
void kthread_exit();
```

This function terminates the execution of the calling thread. If called by a thread (even the main thread) while other threads exist within the same process, it shouldnt terminate the whole process. If it is the last running thread, the process should terminate. Each thread must explicitly *call kthread_exit()* in order to terminate normally.

```
int kthread_join(int thread_id);
```

This function suspends the execution of the calling thread until the target thread, indicated by the argument *thread_id*, terminates. If the thread has already exited, execution should not be suspended. If successful, the function returns zero. Otherwise, -1 should be returned to indicate an error.

3 Synchronization

In this task you will implement two types of mutexes, the first one, a simple *mutex*, will be implemented in the kernel space. The second *mutex* will be a tournamed-tree bases *mutex* for N processes, which you will implement as a user library, using the *mutex* you have implemented.

3.1 Mutex

Before starting the implemntation you should do the following:

- Read about mutexes and condition variables in POSIX. Our implementation will emulate the behavior and API of *pthread_mutex*. A nice tutorial is provided by Lawrence Livermore National Laboratory.
- Examine the implementation of spinlocks in xv6's kernel (for example,the scheduler uses a spinlock). Spinlocks are used in xv6 in order to synchronize kernel code. Your task is to implement the required synchronization primitives as a kernel service to applications, via system calls. Locking and releasing

can be based on the implementation of spinlocks to synchronize access to the data structures which you will create to support mutexes.

Examine *pthreads'* implementation, and define the type *kthread_mutex_t* inside the xv6 kernel to represent a mutex object. You can use a global static array to hold the mutex objects in the system. The size of the array should be defined by the macro $MAX\_MUTEXES$.
The API for mutex functions is as follows:

```
int kthread_mutex_alloc();
```

Allocates a mutex object and initializes it; the initial state should be unlocked. The function should return the ID of the initialized mutex, or -1 upon failure.

```
int kthread_mutex_dealloc(int mutex_id);
```

De-allocates a mutex object which is no longer needed.The function should return 0 upon success and -1 upon failure(for example, if the given mutex is currently locked).

```
int kthread_mutex_lock(int mutex_id);
```

This function is used by a thread to lock the mutex specified by the argument *mutex_id*. If the mutex is already locked by another thread, this call will block the calling thread (change the thread state to $BLOCKED$) until the mutex is unlocked.

```
int kthread_mutex_unlock(int mutex_id);
```

This function unlocks the mutex specified by the argument *mutex_id* if called by the owning thread, upon unlocking a mutex one of the threads which are waiting for it will wake up and acquire it, we leave the waking up order to you, can be in any way you prefer.

3.2 Tournament Trees

In this task you will implement a tournament-tree synchronization through a user library. Use the mutex you have implemented in the previous section.

Given a mutex, it is possible to implement a tournament-tree library in user space. You are required to implement such a library in *tournament_tree.h*.

Note that the tree should work as long as up to 2^{depth} threads are using it, irrespective of which threads exactly they are: For example, a tree of depth 2 should be able to deal with being used simultaneously by threads #1,3,9,15. This is done by the user mapping the threads to the numbers 0 to $2^{depth} - 1$. The mapping is managed by the caller.

Follow the API exactly in order to pass the automatic tests. First define the type *trnmnt_tree* in *tournament_tree.h* .
The API for Tournament-tree functions is as follows:

```
trnmnt_tree* trnmnt_tree_alloc(int depth);
```

Allocates a new tournament tree lock and initializes it. The function returns a pointer to the initialized tree, or 0 upon failure. Depth indicates the depth of the tree, at least 1. As such the tree can accommodate 2^{depth} threads.

```
int trnmnt_tree_dealloc(trnmnt_tree* tree);
```

De-allocates the tournament tree lock, which is no longer needed. Returns 0 upon success and -1 upon failure (e.g., if there are threads waiting).

```
int trnmnt_tree_acquire(trnmnt_tree* tree,int ID);
```

This function is used by a thread to lock the tree specified by the argument trnmnt_tree. ID indicates the ID of the lock acquirer thread, a number between 0 and $2^{depth} - 1$. It is not necessarily the Thread ID. There should be no 2 threads in the tree with the same ID.

```
int trnmnt_tree_release(trnmnt_tree* tree,int ID);
```

This function is used by a thread to release the tree specified by the argument trnmnt_tree. ID indicates the ID of the lock acquirer thread, a number between 0 and $2^{depth} - 1$. It should be the same as the one used by the thread in acquiring the lock.

4 Sanity Test

In this part you are required to test all the above implementations. There are no strict guidelines, just keep in mind the following:

- You **MUST** test all the tasks in the assignments, and explain to the grader how you are testing each task.
- While the makefile default $CPUS$ is set to 4, the grader might increase/decrease it.
- The grader will use a different sanity test. You must restrict yourself to the functions supplied by our API, we will not make modifications to our test to accommodate new functions which you defined.

5 Submission Guidelines

You should download the XV6 code that belongs to this assignment here:

http://www.cs.bgu.ac.il/~os192/git/Assignment2

Make sure that your Makefile is properly updated and that your code compiles with no warnings whatsoever. We strongly recommend documenting your code changes with comments - these are often handy when discussing your code with the graders. Due to our constrained resources, assignments are only allowed in pairs. Please note this important point and try to match up with a partner as soon as possible. Submissions are only allowed through the submission system. To avoid submitting a large number of xv6 builds you are required to submit a patch (i.e. a file which patches the original xv6 and applies all your changes). You may use the following instructions to guide you through the process:

1. Backup your work before proceeding!
2. Before creating the patch review the change list and make sure it contains all the changes that you applied and nothing more. Modified files are automatically detected by git but new files must be added explicitly with the "git add" command:

   ```
   > git add . –Av
   > git commit –m "commit message"
   ```

3. At this point you may examine the differences (the patch)

   ```
   > git diff origin
   ```

4. Once you are ready to create a patch simply make sure the output is redirected to the patch file:

   ```
   > git diff origin > ID1_ID2.patch
   ```

5. Tip: Although grades will only apply your latest patch, the submission system supports multiple uploads. Use this feature often and make sure you upload patches of your current work even if you havent completed the assignment

6. Finally, you should note that the graders are instructed to examine your code on lab computers only! We advise you to test your code on lab computers prior to submission, and in addition after submission to download your assignment, create a clean xv6 folder (by using the git clone command), apply the patch, compile it, and make sure everything runs and works. The following command will be used by the testers to apply the patch:

```
> patch -p1 < ID1_ID2.patch
```