OS212: Assignment 2

Signals, Threads, and Semaphores

14/04/2021

Responsible TAs: Semion Novikov, Nitsan Soffair
Due Date: 02/05/2021 23:59

1 Introduction

The main goal of the assignment is to teach you about XV6 synchronization mechanisms and process management. Initially, you will implement a signal framework in XV6, implementing the sigaction, sigprocmask, and sigret system calls, and all the required infrastructure, which is specified at section 2.

Then you will implement Kernel-Level Threads and Semaphore.
You will write a sanity test for testing both parts of the assignment, in addition you will need to make sure that all of the usertests pass.

The assignment is composed of four main parts:
1. Implement a simple signals framework.
2. Using the signals framework, create a multi process application.
3. Add threads support for XV6.
4. Add semaphore support for XV6.

Follow the instructions available on the course website to run XV6 on a virtual machine:
https://moodle2.bgu.ac.il/moodle/mod/page/view.php?id=1733250

or alternatively via VPN:
https://moodle2.bgu.ac.il/moodle/mod/page/view.php?id=1734080

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

2 Implementing Signals In XV6

In practical session 3 you were introduced to the idea of signals. Using signals is a method of Inter-Process Communication (IPC). In this part of the assignment, you will add a framework that will enable the passing of signals from one process to another.

2.1 Creating the signal framework

2.1.1 Updating the process data structure

In order to implement signals, you will initially have to enhance the process data structure (located at proc.h) to accommodate the required features:

- **Pending Signals:** 32bit array, stored as type uint.
- **Signal Mask:** 32bit array, stored as type uint.
- **Signal Handlers:** Array of size 32, of type void*.
- **User Trap Frame Backup:** Pointer to a trapframe struct stored as struct trapframe*.

In addition to modifying the proc data structure, you will add the following macros:

```
#define SIG_DFL 0 /* default signal handling */
#define SIG_IGN 1 /* ignore signal */
#define SIGKILL 9
#define SIGSTOP 17
#define SIGCONT 19
```

2.1.2 Updating process creation behavior

Now you will need to modify the existing behavior in order to use signals:
1. **Default Handlers:** The default handler, for all signals, should be SIG_DFL.
2. **Process creation:** When a process is being created, using fork(), it will inherit the parent's signal mask and signal handlers, but will not inherit the pending signals.
3. **Executing a new process:** When using exec, we will return all custom signal handlers to the default, note that SIG_IGN and SIG_DFL should be kept.

2.1.3 Updating the process signal mask

You will implement a new system call:

```
uint sigprocmask (uint sigmask);
```

This will update the process signal mask, the return value should be the old mask.

The option you should implement is:
SIG_SETMASK: The set of blocked signals is set to the argument set.
You can read here for more details.

2.1.4 Registering Signal Handlers

A process wishing to register a custom handler for a specific signal will use the following system call which you should add to XV6:

```
int sigaction (int signum, const struct sigaction *act, struct sigaction *oldact)
```

This system call will register a new handler for a given signal number (signum).
sigaction returns 0 on success, on error, -1 is returned.
If act is non-NULL, the new action for signal signum is installed from act. If oldact is non-NULL, the previous action is saved in oldact.
You will define the new struct required for sigaction.

```
struct sigaction {
        void (*sa_handler) (int);
        uint sigmask;
}
```

sa_handler is the signal handler, and sigmask specifies a mask of signals which should be blocked during the execution of the signal handler.

sa_handler can also be a reference to kernel space actions, that is to be one of SIG_IGN, SIGSTOP, SIGKILL, SIGCONT, SIG_DFL.

Any positive value is valid for sigmask, see PS3.
Make sure that SIGKILL and SIGSTOP cannot be modified, blocked, or ignored!
Attempting to modify them will result in an error.

For additional info regarding the sigaction check the man.

2.1.5 The sigret system call

You will implement a new system call:

```
void sigret (void);
```

This system call will be called implicitly when returning from user space after handling a signal.

This system call is responsible for restoring the process to its original workflow, its purpose will be clear after section 2.4.

2.2 Sending a Signal

2.2.1 Updating the kill system call

So far, we described how a process should register new handlers.
Next we will add the ability to send a signal to a process.
You will modify the existing kill system call, and change it to the standard linux kill system call:

```
int kill (int pid, int signum);
```

Given a process id (pid) and a signal number (signum), the kill system call will send the desired signal to the process pid.
Upon successful completion, 0 will be returned.
A -1 value will be returned in case of a failure (think about the cases where kill can fail).

2.2.2 Fixing current kill occurrences

Now you will make sure kill is always used properly.

There are not many calls to it in XV6 code.

When finishing section 2.4 usertests should pass, and in addition the program 'kill' will work, thus enabling you to write in shell 'kill <pid> <signal>' in order to send signals to processes.

2.3 Implement kernel space signals

Not all signals are being handled in the user space, specifically, SIGSTOP, SIGKILL, SIGCONT, are all handled in the kernel space.

You will now implement those signals:

- **SIGKILL:** Will cause the process to be killed, similar to the original XV6 kill.
- **SIGSTOP:** Will cause the process to freeze until SIGCONT is received.
- **SIGCONT:** Will cause a process to resume, if freezed by a previous SIGSTOP.

After a process handles SIGSTOP it will become suspended (frozen).

Only when in this mode SIGCONT should be handled, and in all other cases it will be ignored.

2.3.1 SIGSTOP and SIGCONT

When handling SIGSTOP the process will be freezed, until a SIGCONT is received.

This cannot be done by simply blocking the process, as then it will not receive any CPU time, and will not check for pending signals.
In order to overcome this, whenever a process is being stopped, its state will remain ready, but whenever it receives CPU time, the system will check for a pending SIGCONT signal.
if none are received, it will yield the CPU back to the scheduler.

2.4 Handling Signals

When a process is about to return from kernel space to user space (using the function usertrapret which can be found at trap.c) its pending signals must be checked by the kernel. If a pending signal exists and it is not blocked by the process signal mask then the process must handle the signal.

The signal handling is done by either discarding the signal (if the signal handler is SIG_IGN) or by executing a signal handler.
Executing a kernel space signal handler is straightforward, and should be done before returning to user space.

Notice that you should call the signal handler function after setting the Exception PC in the saved user PC.

If the signal handler is defined to be SIG_DFL, it will attempt to execute the proper default action. For SIGSTOP, SIGCONT and SIGKILL it will attempt to execute the behavior you implemented at 2.3. For all other signals the default should be the SIGKILL behavior.

User space handlers (as can be defined by the signal function) are more difficult to execute: Initially, you must create a backup of the user current trap frame, backing up all relevant registers. You should use the field you added earlier to struct proc for holding the trapframe backup.

After backing up the user trap frame, you will have to modify the user space stack and the instruction pointer of the process, in order to execute the user-space handler. This requires knowledge of conventions of function call.

Notice that in RISC-V, the calling conventions are a bit different, read here for more information, or here (Pages 25-37) for more information.

Function call steps:
1. Store arguments values at dedicated registers
2. Move SP down
3. Store return PC
4. Jump to the called function address

In order to execute the body of the signal handler upon return to user space, one must update the instruction pointer's value to be the address of the desired function. When the signal handler returns, the user space program should continue from the point it was stopped before.

Since the signal handler can change the CPU register values this can cause unpredictable behavior of the user space program once jumping back to the original code.

In order to solve this problem, you must save the CPU registers values before the execution of the signal handler and restore them after the execution of the signal handler finishes.

You will use the field you created (in Assignment 1) in struct proc for holding the trapframe backup.

When the signal handler finishes, your code must return to kernel space in order to restore the original trapframe.

This is the responsibility of the sigret system call, which will only restore the original trapframe.

The main problem here is that the signal handler can accidentally not call the sigret system call on exit and this may cause unpredictable behavior in the user code.

To solve this problem, you need to "inject" an implicit call to the sigret system call after the call to the signal handler.

This can be done by modifying the user space stack directly, creating an explicit call to sigret in the user stack, and pointing the return address of the signal handler to the sigret call.

You can do so by using the memmove function which is located at string.c.
You learned how to create a function and copy the compiled code to another location in the memory in the architecture and splab courses you already took.

Note that you should not support nested user-level signal handling.
That is, once a user-level signal handler starts executing, it will execute until it terminates.

Just before a signal handler starts executing, the process sigmask should be replaced by the sigaction mask corresponding to the signal.

Once the signal handler returns, the value of the process sigmask should be restored to its previous value.

2.5 Testing your signal framework

You should write a user program to test all relevant parts of the framework.

The test should include spawning of multiple processes, modifying the handlers using sigaction, restoring previous handlers using the sigaction oldact, blocking signals.

All possible actions should be tested, as well as user space signals.

You should update the existing kill user program to support signal sending, as in Linux kill shell command.

3 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. Read this whole part thoroughly before you start implementing.
2. 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 define and implement the system calls that will allow the user to create and manage new threads.

Notice that for the signal handling, you should use a shared signal table for all the threads of a particular process.

3.1 Moving to threads

This task includes passing over a lot of kernel code and understanding every 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 = 8.
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.

Hints:
• 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?
• When switching from userspace to kernel space, a pointer to the trapframe is “saved” so that the correct trapframe is filled up upon an interrupt. Now each thread needs to do this.

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 CPUs 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 → 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.

3.2 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 = 4000.

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.
Obviously in real OS’s, the kernel does the memory allocation and management for the additional threads’ stacks, but we haven’t covered memory management yet.

```
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(int status);
```

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 shouldn't 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, int* status);`

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 0 and fills status with the threads exit status. Otherwise, -1 should be returned to indicate an error.

4 Semaphore

In this task you are required to implement two types of synchronization primitives: binary and counting semaphores.

Binary semaphores will be implemented in kernel space, and the counting semaphore will be implemented as a library in user space.

The header of the user library should be called Csemaphore.h.

Both counting and binary semaphores should adhere to the well-known interfaces (as learnt in class). Note that none of them should use busy-waiting. In your implementation you can assume that the maximum number of binary semaphores is MAX_BSEM=128.

4.1 Binary semaphore

Your implementation of the binary semaphore should be through the following system calls:

`int bsem_alloc();`

Allocates a new binary semaphore and returns its descriptor(-1 if failure). You are not restricted on the binary semaphore internal structure, but the newly allocated binary semaphore should be in unlocked state.

`void bsem_free(int);`

Frees the binary semaphore with the given descriptor. Note that the behavior of freeing a semaphore while other threads “blocked” because of it is undefined and should not be supported.

`void bsem_down(int);`

Attempt to acquire (lock) the semaphore, in case that it is already acquired (locked), block the current thread until it is unlocked and then acquire it.

void bsem_up(int);

Releases (unlock) the semaphore. Note that the binary semaphore you are required to implement must satisfy the mutual exclusion and deadlock freedom conditions but is not required to be starvation free.

4.2. Counting Semaphore

In order to implement the counting semaphore you should use the above binary semaphore implementation, using code shown in class for the implementation of a counting semaphore using binary semaphores. The counting semaphore you are required to implement should adhere to the following interface:

void csem_down(struct counting_semaphore *sem):

If the value representing the count of the semaphore variable is not negative, decrement it by 1. If the semaphore variable is now negative, the thread executing acquire is blocked until the value is greater or equal to 1. Otherwise, the thread continues execution.

void csem_up(struct counting_semaphore *sem):

Increments the value of the semaphore variable by 1. As before, you are free to build the struct counting_semaphore as you wish.

int csem_alloc(struct counting_semaphore *sem, int initial_value):

Allocates a new counting semaphore, and sets its initial value. Return value is 0 upon success, another number otherwise.

void csem_free(struct counting_semaphore *sem):

Frees semaphore.
Note that the behavior of freeing a semaphore while other threads “blocked” because of it is undefined and should not be supported.

5 Submission Guidelines

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 **Moodle**.

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 **Moodle** system supports multiple uploads.

Use this feature often and make sure you upload patches of your current work even if you haven't completed the assignment.

6. Finally, you should note that the graders are instructed to examine your code on lab computers only (or on the VM image provided)!

We advise you to test your code on lab computers prior to submission (or on the VM image provided), 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
```