OPERATING SYSTEMS – ASSIGNMENT 2

SIGNALS, USER LEVEL THREADS AND SYNCHRONIZATION

Responsible TAs: Vadim Levit & Benny Lutati

Introduction

In this assignment we will extend xv6 to support a simple signal framework, a user level threads library and a set of synchronization primitives.

You can download xv6 sources for the current work by executing the following command:

```
git clone http://www.cs.bgu.ac.il/~os172/git/Assignment2
```

Task 1: The Signals framework

As you have seen in class, signals are a simple form of inter process communication (currently not implemented in xv6). In this part of the assignment, you will add a framework that will enable the passing of signals from one process to another. This implementation will cover the basic features needed for a signals framework, and despite its resemblance to the Linux framework it is far from being complete.

1.1. Updating the process data structure:

Our basic implementation will be composed of a single bit array per process that will hold the pending signals, and an array of signal handlers (pointers to handler functions) together with several system calls that will allow the process to register signal handlers, send signals, etc.

The first step towards meeting this goal is to extend the '*proc*' *struct* located at *proc.h*. The struct should contain a field called "*pending*" that represents all currently unhandled (pending) signals. For example, when this field's value is 0x3 (the hex representation of the binary word 00…0011) then the process will eventually receive two signals whose identifiers are 1 and 0 (pending is a bit array). For simplicity, you may assume that your implementation will never have to support more than *32* signals (you can add an appropriate definition: *#define NUMSIG 32*). Each signal must also be associated with some action. To support this, add an array of *NUMSIG* entries where every entry is a pointer to a function.

- Note that this representation means that multiple instances of a signal with the same type (having the same identifier) do not stack (are treated as a single signal).

- Every signal is assumed to have a default handler. This handler must only print the following message: "A signal %num% was accepted by process %pid%", where %num% is the current signal identifier and %pid% is the pid of the process.

1.2. 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:

```
sighandler_t signal(int signum, sighandler_t handler)
```

This system call will register a new handler for a given signal number (signum). If failed, -1 is returned. Otherwise, the previous value of the signal handler is returned. The type sighandler_t should be defined as:

```
typedef void (*sighandler_t)(int);
```

1.3. Sending a signal:

So far, we described how a process should register to new signals. Next we will add the ability to send a signal to a process. Add the following system call:

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

Although "kill" is the standard name for the system call in charge of sending a signal, it is already used in xv6 for terminating processes. Given a process id (pid) and a signal number (signum), the sigsend 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 sigsend can fail).

1.4. Getting the process to handle the signal:

Finally, you are required to implement a mechanism that will ensure that a process receiving a signal actually executes the relevant signal handler. In your extension of xv6, **a signal should be handled (if at all) whenever control for the receiving process is passed from the kernel space to the user space**. That is, before restoring the process context (see `trapret` at `trapasm.S`), the kernel first checks for the pending signals of that process. If signal handling is required, the kernel must create a signal-handling stack frame which is a (user) stack frame for the appropriate signal handler execution. The signal-handling stack frame should contain the values of the CPU registers, stored before the execution of the signal handler in the trapframe of the current process, the signal number and a return address that points to an invocation of the system call `int sigreturn(void)`. The values in the signal-handling stack frame should be arranged so that they will be used as follows:

1. The signal handler should receive a single argument – the signal number
2. After the signal handler termination an invocation of the `sigreturn` system call should be performed (If you like, you can do this in a manner similar to the implicit invocation of the exit

system call you implemented in the first assignment)

3. As a response to the sigreturn system call the kernel should restore the trapframe which is stored in the signal-handling stack frame of the calling process

Finally, the kernel should change the trapframe in the process' control block so that it will execute the signal handler. This way, when the process returns from kernel space to user space, the code for the signal handler is executed. Note that in case several signals are pending, the kernel should prepare signal-handling stack frame and execute the signal handler of a ***single*** signal.

1.5. Implement the alarm system call:

The alarm (`int alarm(int);`) system call shall cause the kernel to generate a SIGALRM=14 signal for the process after a given number of ticks, specified by the system call argument. If the argument ticks is 0, a pending alarm request, if any, is canceled. Alarm requests are not stacked; only one SIGALRM generation per process can be scheduled in this manner. If the SIGALRM signal has not yet been generated, the call shall result in rescheduling the time at which the SIGALRM signal is generated.

- Note: the scheduler may prevent the process from handling the signal as soon as it is generated

Task 2: User Level Threads

User level threads (ULT) avoid changing the kernel (i.e., are transparent to it), and manage their own structures and their own scheduling algorithm (thus allowing more efficient, problem-specific scheduling). Context switching between threads within the same process is achieved by manipulating the CPU state (registers). ULTs are usually cooperative i.e., threads have to voluntarily give up the CPU by calling the thread_yield function (non-preemptive schduling). In this task you will implement preemptive (non-cooperative) user space threads.

In order to manage threads within a process, a struct is required to represent a thread (much like a process control block PCB for a process). We will refer to this as the thread control block (TCB). The threads framework must maintain a table of the threads that exist within the process. You can define a static array for this purpose, whose size will be defined by the *MAX_UTHREADS=64* macro. The application programming interface (API) of your threads package has to include the following functions:

2.1. int uthread_init():

The purpose of this function is to prepare the user space process to deal with ULTs. The function must be called by the process (that wants to use the uthread library). This function performs the following steps:

- Initializes the process' threads table
- Creates the main thread

- Registers the SIGALRM handler to the uthread_schedule function
- Executes the alarm system call with parameter UTHREAD_QUANTA=5

After performing the initialization steps you are free to use the uthread library.

2.2. int uthread_create(void (*start_func)(void *), void*arg);

This function receives as arguments a pointer to the thread's entry function and an argument for it. The function should allocate space for the thread, initialize it but not run it just yet. Once the thread's fields are all initialized, the thread is inserted into the threads table. The function returns the identifier of the newly created thread or -1 in case of a failure.

- Don't forget to allocate space for the thread's stack. A size of 4096 bytes should suffice.
- Consider what happens when the thread's entry function finishes: does it have an address to return to? Hint: you can
    - Wrap the entry function, so that a thread_exit call will be carried out implicitly after the entry function finishes.
    - On thread creation, push the thread_exit as the return address.

2.3. void uthread_schedule():

This function should choose (schedule) the next thread from the threads table, according to the round robin scheduling policy. Note that uthread_schedule should be called only from a signal handler, therefore it can assume that the trapframe of the current process, which actually represents the context of the current thread, is stored at the stack. Therefore, uthread_schedule should not backup the current process trapframe but only perform the thread "context switch" similarly to the ideas behind the context switch of processes. Note that context switching requires inlining assembly code into your C code.

2.4. void uthread_exit():

Terminates the calling thread and transfers control to some other thread (similar to uthread_schedule). Don't forget to clear the terminated thread from the threads table, and free any resources used by it (the stack, in our case). When the last thread in the process calls uthread_exit the process should terminate (i.e. exit should be called).

2.5. int uthred_self():

Returns the thread id associated with the calling thread.

2.6. int uthred_join(int tid):

The uthread_join function waits for the thread specified by tid to terminate. If that thread has already terminated/not exists, then uthread_join returns immediately.

2.7. int uthred_sleep(int ticks);

The uthread_sleep function should suspend the execution of the current thread for at least *ticks* (here ticks represent the value of ticks variable defined at trap.c). Namely, uthread_sleep resembles sleep system call but is applied to the user threads.

Task 3: Synchronization primitives

In this task we you are required to implement two types of synchronization primitives: binary and counting semaphores. Both synchronization primitives will be implemented as a user library (implemented in user space). 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. Think how you can update the current implementation of the uthread library to achieve this goal. In your implementation you can assume that the maximum number of binary semaphores is MAX_BSEM=128.

3.1. Binary semaphore

Your implementation of the binary semaphore should support the following functions:

- `int bsem_alloc();` – the bsem_alloc function should allocate a new binary semaphore and return its descriptor. 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);` – the bsem_free function should free 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 build must satisfy the mutual exclusion and deadlock freedom conditions but is not required to be starvation free.

3.2. Counting Semaphore

In order to implement the counting semaphore you should use the above binary semaphore implementation (you should not add direct support for counting semaphores in the uthreads library).

The counting semaphore you are required to implement should adhere to the following interface:

- `void 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 up(struct counting_semaphore *sem)`: Increments the value of semaphore variable by 1.

As before, you are free to build the `struct counting_semaphore` as you see fit. Additionally you should create a methods to allocate and free the counting semaphores.

Task 4: Sanity tests

In this task you should create a user space program which will use the above implementation of threads and synchronization primitives. The program should create four threads, one of which will be referred to as the producer and the others as the consumers. These threads will use a shared array of 100 integers which will be referred to as the *queue*. The behavior of these threads is defined as follows:

- Producer – sequentially adds the numbers from 1, 2, 3, …, 1000 to the queue (note that the producer should be blocked if the queue is full).
- Consumers – take a number *i* from the queue and sleep for *i* ticks. Afterwards, the thread should print a message: “Thread %tid% slept for %i% ticks.” Note that the consumers should be blocked if the queue is empty. Additionally, you must ensure there is no two consumers taking the same value from the queue.

The process should terminate when one of the consumers finish handling the value *i=1000*.

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 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:

Back-up your work before proceeding!

Before creating the patch review the change list and make sure it contains all the changes that you applied and noting 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"
```

At this point you may examine the differences (the patch):

```
> git diff origin
```

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
```

- 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 haven't completed the assignment.

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, apply the patch, compile it, and make sure everything runs and works.

Enjoy !!!