Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Lab: Threads and Synchronization in xv6

The goal of this lab is to understand the concepts of concurrency, synchronization and threads in xv6.

Before you begin

- Download, install, and run the original xv6 OS code. You can use your regular desktop/laptop to run xv6; it runs on an x86 emulator called QEMU that emulates x86 hardware on your local machine. In the xv6 folder, run `make`, followed by `make qemu` or `make qemu-nox`, to boot xv6 and open a shell.
- We have modified some xv6 files for this lab, and these patched files are provided as part of this lab's code. Before you begin the lab, copy the patched files into the main xv6 code directory. The modified files are as follows.
  - The files `syscall.c`, `syscall.h`, `sysproc.c`, `user.h`, `ulib.c` and `usys.S` have been modified to add the new system calls of this lab.
  - Some of the new synchronization primitives you need to implement are declared in `barrier.h`. These functions must be implemented in `barrier.c` in the placeholders provided.
  - Testcases have been provided for each part.
  - An updated Makefile has been provided, to include all the changes for this lab. You need not modify this file.
  - You may modify any of the header or code files to solve the questions in this lab.

Part A: Waitpid system call

Implement the `waitpid` system call in xv6. This system call takes one integer argument: the PID of the child to reap. This system call must return the PID of the said child if it has successfully reaped the child, and must return -1 if the said child does not exist or if the process is not a child process. Once a child is reaped with `waitpid`, the same child should not be returned by a subsequent `wait` system call. You must write your code in the placeholder provided in `proc.c`.

Hint: Since the waitpid implementation needs to access `ptable`, it must be written in `proc.c`, and you must include suitable function definitions in `defs.h` to call your new code of `proc.c` from `sysproc.c`.

You have been provided a test program `t_waitpid.c` to test your code. The program forks a child and tries to reap it via `waitpid` and `wait`. If waitpid functionality is implemented correctly, the child will be reaped by the `waitpid` call and not by the subsequent `wait`.

The expected correct output is as shown below.

```
$ t_waitpid
return value of wrong waitpid -1
return value of correct waitpid 4
return value of wait -1
child reaped
```

Part B: Userspace barrier in xv6

In this part, you will implement a barrier in xv6. A barrier works as follows. The barrier is initialized with a count N, using the system call `barrier_init(N)`. Next, processes that wish to wait at the barrier invoke the system call `barrier_check()`. The first $N-1$ calls to `barrier_check` must block, and the N-th call to this function must unblock all the processes that were waiting at the barrier. That is, all the N processes that wish to synchronize at the barrier must cross the barrier only after all N of them have arrived. You must implement the core logic for these system calls in `barrier.c`.

You may assume that the barrier is used only once, so you need not worry about reinitializing it multiple times. You may also assume that all N processes will eventually arrive at the barrier.

You must implement the barrier synchronization logic using the `sleep` and `wakeup` primitives of the xv6 kernel, along with kernel spinlocks. Note that it does not make sense to use the userspace sleep/wakeup functionality developed by you in this part of the assignment, because the code for these system calls must be written within the kernel using kernel synchronization primitives.

We have provided a test program `t_barrier.c` to test your barrier implementation. In this program, a parent and its two children arrive at the barrier at different times. With the skeleton code provided to you, all of them clear the barrier as soon as they enter. However, once you implement the barrier, you will find that the order of the print statements will change to reflect the fact that the processes block until all of them check into the barrier.

Part C: Threads in xv6

xv6 in its original form does not provide any primitives to create multiple threads within a process. In order to provide concurrency to users, we will implement three system calls in this part which essentially are all you need to use basic multithreading. In what follows, both `tid` and `pid` refer to the same quantity, the unique identity of a unit of execution.

- `thread_create(uint*,void* (*func)(void*),void* arg)`: This is an analogue of the `fork` syscall which will be used to spawn threads. It takes as arguments
  - a pointer to an integer in which the `thread_id` would be stored. For convenience, this value can be the same as the `tid` of the spawned thread
  - a function that takes a `void*` argument and has return type `void*`. This would be the entry point of the spawned thread
  - the `void*` argument passed to that function

  It returns the `pid` (or `tid`) allocated to the newly created thread.

- `thread_exit()`: This is an analogue of the `exit` syscall and is to be used exclusively by spawned threads. If this syscall is called from a main thread, the behaviour should be similar to a no-op.
- `thread_join(uint)`: This is an analogue of the `waitpid` syscall and takes the `thread_id` of a thread as argument. Similar to `thread_exit`, this syscall is to be used exclusively to reap spawned threads.

Note that for this lab, we will not be testing usage of certain syscalls such as `fork` or `sbrk` alongside the thread-based syscalls. Moreover, discussion regarding the "parent process" of a thread is not of interest in this lab. Some hints to get you started on the lab are given below.

Threads are fundamentally different from processes because in a way, they are linked to their main parent process. As a result, we need to keep track of the main parent thread inside each of the spawned threads. `struct proc` can be modified accordingly to contain new fields in this regard. Now, we will also have to modify the behaviour of `fork`, `exit` and `wait` syscalls to acknowledge the presence of threads.

- `fork`: You should modify this syscall to ensure that the mainthread of the newly forked process is the process itself (that is, `p->mainthread = p;`)
- `wait`: Let process A fork a process B which in turn has spawned a thread T. When process A calls `wait()`, we want to ensure that A waits until the mainthread of B finishes execution i.e. even if T finishes execution before the mainthread, `wait()` must block until the mainthread finishes executing. Secondly, once a process is reaped, all threads spawned by it must also get reaped at the same time i.e., they should not get scheduled anytime post the completion of the `wait()` syscall. This includes freeing any memory related to the threads and cleaning up the `ptable` state (see `wait()` function in `proc.c` for more details).
- `exit`: We want to ensure that this function is only called by the mainthread of a process, hence if it is called by a non-main thread, we simply call the `thread_exit` function instead. Note that we are not going to reap the spawned threads at this point, they will be reaped forcefully once the mainthread is reaped.

Some helpful tips when writing the `thread_create` function. This is similar to `fork` but:

- Threads share memory and hence page tables, so there is no need to create a new set of page tables everytime a thread is created.
- Forked processes are mainthreads themselves whereas spawned threads are not.
- Though threads share memory, they cannot share stacks (imagine the chaos if stacks were shared). So, each thread would need a new writeable user page as its user stack. This would also mean that each time a new thread is spawned, the size of the process grows. You might run into trouble if you do not track this.
- The spawned thread must start executing at the entrypoint function and not the next instruction. You must modify the EIP in the trapframe to achieve this.
- The `void*` argument being passed into the entrypoint function needs to be placed on the user stack so that the compiled code reads the correct data. You can look at how arguments can be put on the stack in the implementation of `exec()` syscall.

- A number `thread_id` is to be assigned to the spawned thread - it is convenient to just use it as an alias for `tid`. This number then has to be copied into memory at the address passed as the first argument to this syscall

When writing the `thread_exit` function, note that this is similar to `exit`. But the `exit` syscall wakes up the parent of the exiting process so as to reap it using `wait`. An analogue must be done so as to reap the exiting thread using `thread_join`.

Some tips for writing the `thread_join` function. This is similar to `waitpid` but:

- Mainthreads cannot be joined to themselves.
- Mainthreads can only join their spawned threads, not their children created using `fork`.
- Threads share memory, so one thread joining doesn't necessarily mean that the entire memory address space needs to be freed up (as is done in `waitpid`). You might run into problems if you free memory prematurely. As a rule of thumb, memory does get freed when the mainthread gets reaped because we are ensuring that all the threads are also reaped along with it.
- There is still a portion of memory intended for a thread which can safely be deallocated regardless of memory address space being shared, this region of memory must be freed while joining.

A test case `t_threads.c` has been provided to you, where new threads update some shared variables. If thread creation works properly, the expected output will be as follows.

```
$ t_threads
Hello World
Thread 1 created 10
Thread 2 created 11
Value of x = 11
```

Part D: Userspace spinlocks in xv6

In this part of the lab, we will be implementing spinlocks in xv6, which will be used by the threads of a process to protect access to shared memory. Please solve this part only after completing part C above.

. Spinlocks can be implemented entirely in userspace using atomic instructions. Hence the implementation of spinlocks do not require you to add any new syscalls. All of your changes for implementing locks are to be done in `user.h` and `ulib.c`. You are given template code for the lock in `user.h`, and are expected to implement the following functionality in `ulib.c`:

1. `void initiateLock(struct lock* l)`: Initializes the lock by setting the value of `l->lockvar` to 0
2. `void acquireLock(struct lock* l)`: Check if the lock has been initialized, if so, use the atomic `xchg` instruction to set `l->lockvar` to 1, thus acquiring the lock
3. `void releaseLock(struct lock* l)`: Check that the lock has been initialized and acquired, then set the value of `l->lockvar` back to 0

You are provided testcase `t_lock.c`. The expected output after a correct implementation is shown below.

```
$ t_lock
Final value of x: 2000000
```

Part E: Userspace conditional variables in xv6

We will now implement conditional variables in xv6, to enable synchronization between the threads created in part C. Please solve this part only after you complete parts C and D above.

Conditional variables, unlike spinlocks, require you to use syscalls to sleep and wake up processes. On top of some syscalls, you will build userspace APIs to initialize, wait, and signal condition variables.

Remember that waiting on a conditional variable and a lock involves releasing the lock, going to sleep on a particular channel, and reacquiring the lock on waking up. While we have made methods to acquire and release locks, we still lack a userspace API to sleep on a particular channel, because the current sleep and wakeup functions inside xv6 are only used by kernel functions and not exposed to user code. Therefore, to correctly implement condition variables, you must first implement the following system calls, to expose sleep and wakeup APIs to user code.

1. `int getChannel(void)`: Should return the value of a channel that is unused and will remain unused. You must generate unique channels inside xv6 somehow, and the channel chosen should not clash with a channel that is being used, otherwise this will lead to spurious wakeups.
2. `void sleepChan(int)`: Should check that the provided argument is a valid channel value and then sleep on that channel value.
3. `void sigChan(int)`: Should check that the provided argument is a valid channel value and then wake up all processes sleeping on that channel value.
4. `void sigOneChan(int)`: Should check that the provided argument is a valid channel value and then wake up one specific process sleeping on that channel value.

Once the above syscalls are available to user code, you must implement condition variable functionality, which is already defined in `user.h`, by completing the following functions in `ulib.c`:

1. `void initiateCondVar(struct condvar* cv)`: Allocate a channel through `getChannel` and set `cv->var` to the channel's value. Also track that the variable has been initialized.
2. `void condWait(struct condvar* cv, struct lock* l)`: Check that both the conditional variable and lock are initialized, then release the lock, sleep on the channel and re-acquire the lock after waking up.
3. `void broadcast(struct condvar* cv)`: Check that the conditional variable has been initialized and then call `sigChan` with appropriate arguments to wakeup all threads sleeping on the CV's channel.
4. `void signal(struct condvar* cv)`: Check that the conditional variable has been initialized and then call `sigOneChan` with appropriate arguments to wake up one particular thread sleeping on the CV's channel.

We have provided the following test cases for you to test the system calls as well as the CV implementation: `t_sleepwake.c, t_l_cv1.c t_l_cv2.c`. The expected output from the test cases is as follows.

```
$ t_sleepwake
going to sleep on channel 4
Woken up by child
$ t_l_cv1
Hello World
Gonna acquire lock
Acquired lock successfully
Value of x = 11
$ t_l_cv2
I am thread 1
I am thread 2
I am thread 1
I am thread 2
I am thread 1
I am thread 2
I am thread 1
I am thread 2
I am thread 1
I am thread 2
I am thread 1
I am thread 2
I am thread 1
I am thread 2
I am thread 1
I am thread 2
I am thread 1
I am thread 2
I am thread 1
I am thread 2
```

Part F: Semaphores in xv6

In this part you will use the conditional variables and locks you implemented earlier to implement semaphores in xv6. Please solve this part only after you finish parts C, D, and E above.

The structure for the semaphore is given in `user.h`, it consists of a counter variable, a lock and a conditional variable, the implementation for the semaphore consists of three functions:

1. `void semInit(struct semaphore*, int initval)`: This function initializes the lock and conditional variable in the semaphore and sets the counter to `initval`.
2. `void semUp(struct semaphore* s)`: This function should acquire the lock, increment the counter, wake up *exactly one* process that is sleeping on the channel of the conditional variable.
3. `void semDown(struct semaphore* s)`: This function should acquire the lock, decrement the counter, if after decrementing, the counter becomes negative, it should call `condWait` on the conditional variable and the lock, then it should release the lock.

You are given testcases `t_sem1.c, t_sem2.c`. The expected output is as follows.

```
$ t_sem1
Hello World
Gonna acquire semaphore
Acquired semaphore successfully
Value of x = 11
$ t_sem2
I am thread3 and I should print first
I am thread 1
Only one other thread has printed by now
I am thread 2
Both the other threads have printed by now
```

Submission instructions

- For this lab, submit all the files you changed.
- Place all the files you modified in a directory, with the directory name being your roll number (say, 12345678).
- Tar and gzip the directory using the command tar -zcvf 12345678.tar.gz 12345678 to produce a single compressed file of your submission directory. Submit this tar gzipped file on Moodle.