CS 333 Operating Systems Lab

Autumn 2023

Lab8: xv6 threads for world peace

The goal of this lab is to understand the kernel primitives and implementation details of library functions related to threading. In the xv6-public directory accompanying this lab, look for the comment **WPTHREAD** (World Peace Threads) for a starting point to add implementation details for the tasks ahead.
Use git commands to navigate your way through the changes already implemented in the codebase. Use: `git status` to check which files have been changed, since the last commit (the default xv6 repository). Use `git diff master:<filename> <filename>` to check differences between the specific file from latest commit and current changes to it. For example, to see changes to proc.c, use `git diff master:proc.c proc.c.` Or just use the git extension in VSCode. 🙂

Task 1: clone and cloning

In this question, we will implement the **clone()** system call. This system call creates a new kernel thread for a process. Kernel threads are independently schedulable entities like processes, all the threads of a process share the same virtual address space and file descriptors.
In order to allow for multi-threading in xv6, we need to come up with a design for the Thread Control Blocks (TCBs). For this lab, we will use **struct proc** itself as the PCB-cum-TCB for all the threads. Study the different thread-relevant fields in the PCB:

- `pid`: Same as process id and is synonymous to thread_id
- `tgid`: A thread-group-ID, which is the main (first calling processes) pid). All threads that belong to a process have the same **tgid**
- `thread_count`: number of threads of a process
- `user_stack`: pointer to the user stack of the thread — each thread has its own user stack and kernel stack (similar to a process).

The thread-group-id and pid for the main thread would be the same values, and for the other spawned threads, the thread-group-id would be the same as the pid of the main thread. Threads other than the main thread do not have to bother with the thread_count field. `thread_count` is assigned the value of 1 for each process (refer `allocproc` in `proc.c`) and the count increments when a clone call is made from the main thread process.

Note that our current implementation of clone is not supposed to support multiple level of threading (i.e., a thread spawning threads), because of how we maintain the difference between a main and children threads.

The declaration of the clone system call is as follows:

```
int clone(void(*fn)(int*), int *arg, void *stack);
```

- **fn** : indicates the start point of execution of the thread.
- **arg** : the argument passed to thread function **fn**, of type **int ***.
- **stack** : base address of the user stack allocated for the new thread.

Note that we are designing the system call to work only with functions that take one argument.

On success, the clone call returns the thread-id of the new thread or -1 on error.

A wrapper function is to be used in user space for invoking the clone systems —

```
int create_thread(void(*fn)(int*), int *arg);
```

- **fn** : indicates the start point of execution of the thread
- **arg** : argument of function **fn** (hence, int*)

Notice the distinction between arg in create_thread and clone.

In this wrapper function, **malloc** (defined in `umalloc.c`) one page in user space, and then call the **clone** system call with the malloc'ed page as the stack argument to start the thread. On success, this call returns the thread-id of the new thread and if unsuccessful, it returns -1.

The above step is required, since each thread has its own user space stack (or user stack). All the threads have a common address space and share the entire memory region, but operate using separate user stacks, and per thread state (maintained within the kernel).
As an example, refer to implementation of **exec()** to understand how a user stack is set up.

Following are the implementation details of the **clone** system call:

1. The stack pointer is passed from the thread wrapper function (which allocates memory on the virtual space of the calling process). This stack pointer (virtual address in main process, pointing to a single page) will be used as the user stack of the cloned process. Implement the wrapper function **create_thread** in the file **ulib.c**.
2. Following are the expectations around the system call:
    a. All the threads of a process must have an identical virtual address space and share the same physical pages (and hence share a single page table).
    b. Every thread has its own user stack for user functions.
    c. To initialize a thread, the instruction pointer and the stack pointer of the new thread need to be initialized. The base pointer also needs to be initialized to the bottom of the stack frame, since stack pointer grows relative to it.
    d. A new thread once created and initialized, when scheduled, should start executing at the user specified function.

e. Each thread function **MUST** end execution with an explicit call to **exit()**.

Study the modification to the different system calls, and functions in the `proc.c` source file. The following parts of the implementation have been provided:

1. `exit`: If a thread exits, decrement thread count of parent. If a process exits, pass children processes to init. For children threads, exit all of them (by setting state to ZOMBIE) and reap them, before setting state of parent process to ZOMBIE.
2. `wait`: If thread calls wait, do nothing and return -1. If process calls wait, look for exited child processes and reap them. Ignore child threads.
3. `kill`: If the parent thread is killed, all its threads must be killed.
4. Before the main process exits, it should kill and reap all its threads before exiting.
5. For a process to be reaped, all of its threads must have been reaped before (refer to the **join** system call described in the next subsection).

Further details of implementation are provided in the code, but if you find it overwhelming, we have provided the implementation of `exit`, `wait` and `kill` functions in help.txt, given in the code.

THE OPTIONAL BOX

In our current implementation, we have ignored some details to keep things simple. Following are some of the considerations which we need to pay attention to, in order to make a more robust clone system call.

1. All the threads of a process must share the same set of file descriptors. For this lab, we are implementing it like the way **fork** does it (by copying over the FDs to the new proc table entry) as opposed to sharing them. Irrespective, sharing of file descriptors requires changes to the open and close system calls (or more) to keep fd state consistent across all threads.
2. The children threads (non-main threads) in our implementation have to explicitly call exit(). We can also allow the threads to call return. Think, how one'd handle the correct termination of a thread?

Task 2: join the queue

As part of this task, implement the **join** system call that reaps an exited thread spawned by the process calling **join**. The declaration of the join system call is as follows: `int join();`

Like **wait**, **join** is a blocking system call that returns the thread-ID of a reaped thread, when called from the main thread. If none of the threads have exited, then the call waits for one of the threads to exit. If there are no threads left to exit then join will return -1.

Reaping a thread involves cleaning up its TCB(-cum-PCB), and updating the PCB of the process, which consists of steps like:

- Freeing up the kernel-stack of the thread (we are not freeing the user-stack as explained later on thread exit).
- Updating the TCB entries of the kernel thread to their respective default values, like in the implementation of **wait** (for example, set the state field to UNUSED).
- Decrementing the thread-count of the process.
- Remember that threads share the same page table so we must not deallocate the page table when a thread exits since other threads might be accessing the same. Deallocation of the page table has to be done in **wait** when the process exits.
- A process should only be able to reap threads that it spawned, not the threads created by other processes, this should be checked when iterating over the proc table.

Based on our design, the join may lead to a page-sized memory leak due to the stack memory allocated earlier and not freed on thread termination. Our current design will keep things simple and let this be. As an extension, think how to overcome these leaks cleanly.

Note: The **create_thread** wrapper function should be available as part of every program that runs in xv6. Thus, we should add prototypes to user/user.h and the actual code to implement the library routines in user/ulib.c.

We also have to make changes to the **wait** system call to implement the following:

- **the wait** system call must only consider the exit of **child processes** (and **NOT the threads of the child process**, if any). So, **wait** should not unblock the parent when a thread of a child process exits.
- All else (freeing the memory, setting the PCB to UNUSED,etc) proceeds as normal.

Use the test cases **tc-var.c** and **tc-array.c** to test the implementation.

tc-var.c

Spawns 20 threads each of which increments a global counter by 1. If done right, the final value of VAR should be 20, but since we are trying to simulate race condition here, we may not see the value 20 as output.

tc-array.c

Declares 2 arrays and then spawns 2 threads. The first thread sums the elements of the first half of the first array and second half of the second array. The second thread sums up the remaining elements (second half of the first array, first half of the second array). The program then adds up these two values in the main function and prints the final sum.

THE OPTIONAL BOX

We are not freeing the user stack when a thread terminates, an easy way to do this is to free the malloc'ed user page when join is called. The problem is that join() is a system call implemented in proc.c whereas free() is a user function that cannot be called in kernel code. Figure out a way to get around this, possibly by using wrapper functions as done for clone.

Task3: semaphores cometh!

Next, we will work on a **semaphore** implementation. This would include implementing four system calls, with the following declarations —

```
int semaphore_init(int value);
int semaphore_destroy(int sem);
int semaphore_down(int sem);
int semaphore_up(int sem);
```

For this implementation, we'll maintain a global array of 16 semaphores in the kernel space. Study the fields of `struct semaphore` (in `proc.c`). The `used` field indicates if the semaphore is currently in use or not. This field must be set while initializing the semaphore, and reset when destroying. The locks used in each of the 16 semaphores in the global array are initialized by calling the `sinit` function (defined in `proc.c`) when the kernel boots up.

- The `semaphore_init` call should find an unused semaphore in the array, initialize its value, and return its index.
- `semaphore_down` and `semaphore_up` should implement the typical functionality of a semaphore down and up call. We must make sure that the **down** operation is blocking and not busy-looping. Use `sleep` and `wakeup2` (provided in `proc.c`) functions for this implementation.
- `semaphore_down` should check the current value of semaphore. If it is less than or equal to zero, it should put the calling process to sleep. Otherwise, it should reduce the value of semaphore and return successfully with 0. In any other case, it should return -1 to indicate error.

- `semaphore_up` should increment the value of semaphore, and check if there are any blocked processes, and wake one of them up. If no thread is blocked on the semaphore, it should just increment and return 0. In any other case, it should return -1 to indicate error.
- `semaphore_destroy` should clear the state of semaphore at the index passed, i.e., set the used field to zero.

Find the empty definitions of the above four functions in `proc.c`. and implement the four functions.

The calls above will be used in conjunction with the above threading functions (create_thread and join) to manage concurrent access to shared resources. Study the **tc-semaphore.c** source file to understand a simple usage of semaphore.

tc-semaphore.c
Extends **tc-var.c**, and makes use of the semaphore implementation, to provide mutual exclusion for the critical section.

Task 4: STOP RIGHT THERE!!!

Using the above defined semaphores, we can work on the implementation of a ***barrier*** synchronization primitive in the user space.

A **barrier** is a synchronization primitive that blocks threads from progressing until a required number of threads reach the barrier condition/point. When this happens, the last thread opens the barrier allowing all blocked threads to proceed (we want all blocked threads to be released simultaneously).

Synchronization Barriers

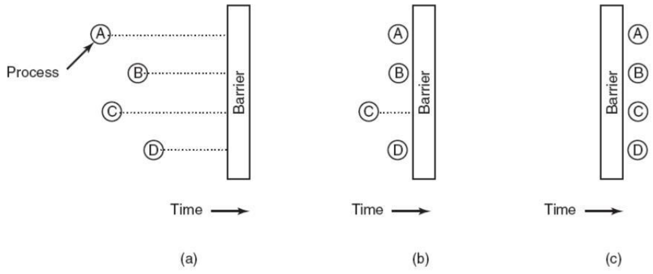

Figure 2-37. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639

Here, we are using the barriers in a loop, and want it to be reset after it allows threads to be passed so that it can be reused. So, when the required number of threads reach the barrier, the barrier should let all threads through and then go back to its original state where it blocks all threads till the required number of threads have reached the barrier.

Modify file **`tc-barrier.c`** using the semaphore functionality to implement barriers at the specified sections in the code. Functionality details.

- All threads should execute Section 1 of the code together and no thread should be allowed to proceed to Section 2 till all threads have completed execution of Section 1.
- When all threads have executed Section 1, the last thread which reaches the barrier should open it and allow all threads to start Section 2 execution (by looping or otherwise).
- No thread should be allowed to loop back to Section 1 till all threads have completed execution of Section 2.
- This process should repeat as long the loop is running, which would require barriers to be reset once they are opened.

The code has 2 barrier placements, one for Section 1 and another for Section 2. The only synchronization primitive that should be used are semaphores implemented in Task 3 (no locks/mutexes allowed). Notice that it is the same barrier struct being used in both the places.

Add variables to

```
struct barrier{}
```

and implement the following functions in tc-barrier.c :-

```
int barrier_init(struct barrier *bar, int num_threads);
int barrier_place(struct barrier *bar);
```

The exact details of the implementation are left to you as long as the required functionality is achieved.

Submission instructions

- Submissions are on the assignment link via Moodle.
  Name your submission as `{rollnumber_lab8}.tar.gz`
  `tar -czvf {rollnumber_lab8}.tar.gz {rollnumber_lab8}`
  (e.g., 200050183_lab8.tar.gz)

- The tar submission should contain all xv6 source files along with the test case files and the makefile

- Run `make clean` before making a tar file for submission

Deadline: 13 October, Friday, 17:00