CS 202: Advanced Operating Systems
University of California, Riverside

Lab #3: xv6 Threads

Due: 06/03/2022, Friday, 11:59 p.m. (Pacific time)

Overview

In this project, you will be adding kernel-level thread support to xv6. First, you will implement a new system call to create a kernel-level thread, called `clone()`. Then, using the `clone()` system call, you will build a simple user-level library consisting of `thread_create()`, `lock_acquire()` and `lock_release()` for thread management. Finally, you will show these things work by using a user-level multi-threaded test program. That's it!

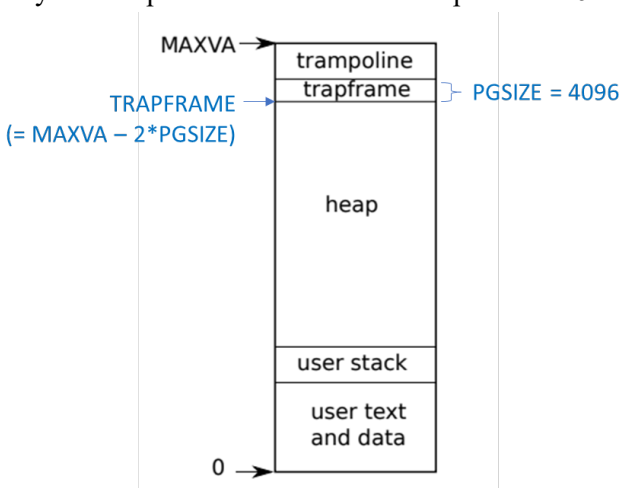
Background: xv6 virtual address space memory layout

In xv6, every process has its own page table that defines a virtual address space used in the user mode. When a process enters the kernel mode, the address space is switched to the kernel's virtual address space. Because of this, each process has separate stacks for the kernel and user spaces (aka. user stack and kernel stack). Also, in xv6, each PCB maintains separate objects to store process's register values:

```
struct proc {
  …
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
```

`trapframe` stores registers used in the user space when entering the kernel mode. `context` is for registers in the kernel space when context-switched to another process.

Below figure illustrates the layout of a process's virtual address space in xv6-riscv.



In the virtual address space, user text, data, and user stack are mapped at the bottom. At top, you can see two special pages are mapped: `trampoline` and `trapframe`, each has the size of `PGSIZE` (= 4096 bytes). The `trampoline` page maps the code to transition in and out of the kernel. The `trapframe` page maps the PCB's `trapframe` object so that it is accessible by a trap handler while in the user space (see Chapter 4 of the xv6 book for more details).

The mapping of those pages to process's address space is done when a process is created. In `fork()`, it calls `proc_pagetable()` which allocates a new address space and then performs mappings of `trampoline` and `trapframe` pages. For example, in `proc_pagetable()`

```
  if(mappages(pagetable, TRAPFRAME, PGSIZE,
              (uint64)(p->trapframe), PTE_R | PTE_W) < 0){ ...
```

This means mapping the kernel object `p->trapframe` to the user address space defined by `pagetable` at the memory location of `TRAPFRAME`.

Part 1: Clone() system call

In this part, the goal is to add a new system call to create a child thread. It should look like:

```
int clone(void *stack);
```

`clone()` does more or less what `fork()` does, except for the following major differences:

- **Address space:** Instead of creating a new address space, it should use the parent's address space. This means a single address space (and thus the corresponding page table) is shared between the parent and all of its children. Do not create a separate page table for a child.
- **File descriptors:** should not be duplicates of the parent's – the parent and the child should share the same file descriptors.
- **`stack` argument:** This pointer argument specifies the **starting address** of the user-level stack used by the child. The stack area must have been allocated by the caller (parent) before the call to clone is made. Thus, inside `clone()`, you should make sure that, when this syscall is returned, a child thread runs on this stack, instead of the stack of the parent. Some basic sanity check is required for input parameters of `clone()`, e.g., `stack` is not null.

Similar to `fork()`, the `clone()` call returns the PID of the child to the parent, and 0 to the newly-created child thread. And of course, the child thread created by `clone()` must have its own PCB.

There are also some modifications required for the `exit()` and `wait()` syscalls.

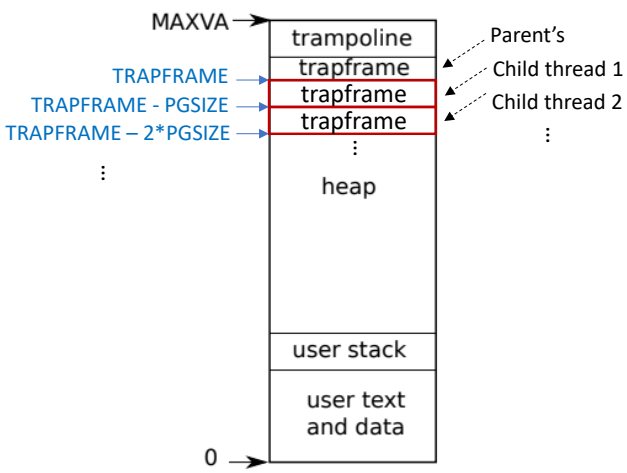
- **`exit()`:** By default, `exit()` closes all the file descriptors that the process uses. However, since these are now shared across all threads, `exit()` must close the file descriptors only when the parent is exiting. In this project, it is fine to assume that the parent is the last one to exit (i.e., after all of its child threads have exited).
- **`wait()`:** The parent process uses `wait()` to wait for a child process to exit and returns the child's PID. Also, `wait()` frees up the child's resources such as PCB, memory space, page table, etc. This

becomes tricky for child threads created by `clone()` because some resources are now shared among all the threads of the same process. Therefore, if the child is a thread, `wait()` must deallocate only the thread local resources, e.g., clearing PCB and freeing & unmapping its trapframe, and must not deallocate the shared page table.

For simplicity, we will assume that only parent process calls `clone()`. A thread created by `clone()` does not call `clone()` to create another child thread. Also, assume that a process does not call `clone()` more than 20 times (i.e., up to 20 child threads). For `exit()` and `wait()`, it is fine to assume that the parent is the last one to exit (i.e., after all of its child threads have exited). These assumptions will make the implementation a lot easier.

Tips:

- The best way to start would be creating `clone()` by duplicating `fork()`. `fork()` uses `allocproc()` to allocate PCB, trapframe, pagetable, etc. However, `clone()` must not allocate a separate page table because the parent and child threads should share the same page table. But each thread still needs a separate trapframe. So, modify `allocproc()` or create a new function (e.g., `allocproc_thread`) for `clone()`.
- In `clone()`, you need to specify the child's user stack's starting address (hint: `trapframe->sp`).
- `wait()` uses `freeproc()` to deallocate child's resources, so you will need to make appropriate changes to `freeproc()`.
- In `clone()`, you should map each thread's `trapframe` page to a certain user space with no overlap. One simple way would be to map it below the parent's trapframe location. For example, see the figure on the right. If your child thread has a thread ID (> 0), map it to `TRAPFRAME - PGSIZE * (thread ID)`. So your first child thread's trapframe is mapped at `TRAPFRAME - PGSIZE`, second one at `TRAPFRAME - PGSIZE * 2`, and so on. This can easily avoid overlap.



- You also need to tell the kernel the new trapframe locations for your child threads. At the end of `usertrapret()` in kernel/trap.c, change

  ```
  ((void (*)(uint64,uint64))fn)(TRAPFRAME, satp);
  ```

  to

  ```
  ((void (*)(uint64,uint64))fn)(TRAPFRAME - PGSIZE * (thread ID), satp);
  ```

  for child threads. Normal processes (or thread ID == 0) should continue to use the default `TRAPFRAME` address.
- Trampoline is already mapped by the parent and it can be shared with childs. So child threads must **not** remap it to the page table (doing so will crash). Only map the trapframe of each child.

Part 2: User-level thread library

The first thread library routine to create is `thread_create()`:

```
int thread_create(void *(start_routine)(void*), void *arg);
```

You can think of it as a wrapper function of `clone()`. Specifically, this routine must allocate a user stack of `PGSIZE` bytes, and call `clone()` to create a child thread. Then, for the parent, this routine returns 0 on success and -1 on failure. For the child, it calls `start_routine()` to start thread execution with the input argument `arg`. When `start_routine()` returns, it should terminate the child thread by `exit()`.

Your thread library should also implement **simple user-level spin lock** routines. There should be a type `struct lock_t` that one uses to declare a lock, and two routines `lock_acquire()` and `lock_release()`, which acquire and release the lock. The spin lock should use **the atomic test-and-set operation** to build the spin lock (see the xv6 kernel to find an example; you can use GCC's built-in atomic operations like __sync_lock_test_and_set). One last routine, `lock_init()`, is used to initialize the lock as need be. In summary, you need to implement:

```
struct lock_t {
     uint locked;
};
int thread_create(void *(start_routine)(void*), void *arg);
void lock_init(struct lock_t* lock);
void lock_acquire(struct lock_t* lock);
void lock_release(struct lock_t* lock);
```

These library routines need be declared in `user/thread.h` and implemented in `user/thread.c`. Other user programs should be able to use this library by including the header `"user/thread.h"`.

Tips:

- In RISC-V, **the stack grows downwards**, as in most other architectures. So you need to give the correct stack starting address to `clone()` for the allocated stack space.
- How to create a library? Once you write `user/thread.c`, find the line starting with `ULIB` in Makefile and modify as follows:

  ```
  ULIB = $U/ulib.o $U/usys.o $U/printf.o $U/umalloc.o $U/thread.o
  ```

  This will compile it as a library and make it usable by other user-level programs.

How to test:

We will be using a simple program that uses `thread_create()` to create some number of threads. The threads will simulate a game of frisbee where each thread passes the frisbee (token) to the next thread. The location of the frisbee is updated in a critical section protected by a lock. Each thread spins to check the value of the lock. If it is its turn, then it prints a message, and releases the lock. Below shows the program code. This program should run as-is. Do not modify. Add this program as `user/frisbee.c`

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
#include "user/thread.h"

lock_t lock;
int n_threads, n_passes, cur_turn, cur_pass;

void* thread_fn(void *arg)
{
  int thread_id = (uint64)arg;
  int done = 0;
  while (!done) {
    lock_acquire(&lock);
    if (cur_pass >= n_passes) done = 1;
    else if (cur_turn == thread_id) {
      cur_turn = (cur_turn + 1) % n_threads;
      printf("Round %d: thread %d is passing the token to thread %d\n",
        ++cur_pass, thread_id, cur_turn);
    }
    lock_release(&lock);
    sleep(0);
  }
  return 0;
}

int main(int argc, char *argv[])
{
  if (argc < 3) {
    printf("Usage: %s [N_PASSES] [N_THREADS]\n", argv[0]);
    exit(-1);
  }
  n_passes = atoi(argv[1]);
  n_threads = atoi(argv[2]);
  cur_turn = 0;
  cur_pass = 0;
  lock_init(&lock);
  for (int i = 0; i < n_threads; i++) {
    thread_create(thread_fn, (void*)(uint64)i);
  }
  for (int i = 0; i < n_threads; i++) {
    wait(0);
  }
  printf("Frisbee simulation has finished, %d rounds played in total\n", n_passes);

  exit(0);
}
```

It takes two arguments, the first is the number of rounds (passes) and the second is the number of threads to create. For example, for 6 rounds with 4 threads:

```
$ frisbee 6 4
Round 1: thread 0 is passing the token to thread 1
Round 2: thread 1 is passing the token to thread 2
Round 3: thread 2 is passing the token to thread 3
```

```
Round 4: thread 3 is passing the token to thread 0
Round 5: thread 0 is passing the token to thread 1
Round 6: thread 1 is passing the token to thread 2
Frisbee simulation has finished, 6 rounds played in total!
$
```

Test your implementation with up to 20 threads on one emulated CPU. If it works, also test on multiple CPUs. You can change the number of emulated CPUs in Makefile (CPUS parameter).

The Code

Download a fresh copy of xv6 and add the above-mentioned functionalities.

Reference Material

This Lab may take additional readings and through understanding of the concepts discussed in the handout.

1. Chapters 2 and 4 of the xv6 book discusses the essential background for this Lab.
2. You may also find this book useful: Programming from the Ground Up. Attention should be paid to the first few chapters, including the calling convention.

What to submit:

Your submission should include:

(1) **XV6 source code** with your modifications ('make clean' to reduce the size before submission)

(2) **Writeup (in PDF).** Give a detailed explanation on the changes you have made (Part 1 & 2). Add the screenshots of the frisbee program results for "frisbee 20 7" and "frisbee 20 20". Also, a brief summary of the contributions of each member.

(3) **Demo video** showing that all the functionalities you implemented can work as expected, as if you were demonstrating your work in person. Demonstrate the results of "frisbee 20 7" and "frisbee 20 20" on three CPUs. Your video should show that xv6 is running with three CPUs (e.g., 'hart 1 starting' and 'hart 2 starting' messages when booting up).

Grades breakdown:

- Part I: clone() system call: 50 pts
  - clone() implementation
  - modifications to exit() and wait()
- Part II: user-level thread library: 30 pts
  - thread_create() routine
  - spinlock routines
- Writeup and demo: 20 pts

Total: 100 pts