CS 202: Advanced Operating Systems
University of California, Riverside

Lab #3: xv6 Threads

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.

Before your start:

1. In `Makefile`, set the number of CPUs to 3 (`CPUS := 3`). You may debug your code using one CPU, your demo and submission should have `CPUS := 3`.
2. Replace `kernel/trampoline.S` with the one provided at the end of this document. This new `trampoline.S` is also available to download from eLearn.

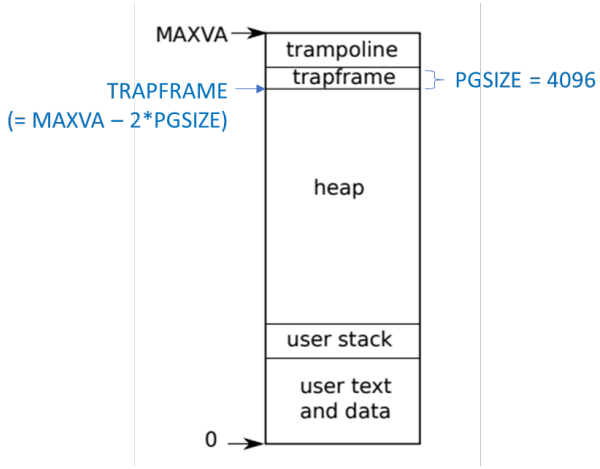
Background: xv6 virtual address space memory layout

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.
- **`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. The number of child threads per process is assumed to be at most 20.

To manage threads, add an integer type `thread_id` variable to PCB. The value of `thread_id` is 0 for the parent process and greater than 0 (e.g., 1, 2, …) for its child threads created using `clone()`.

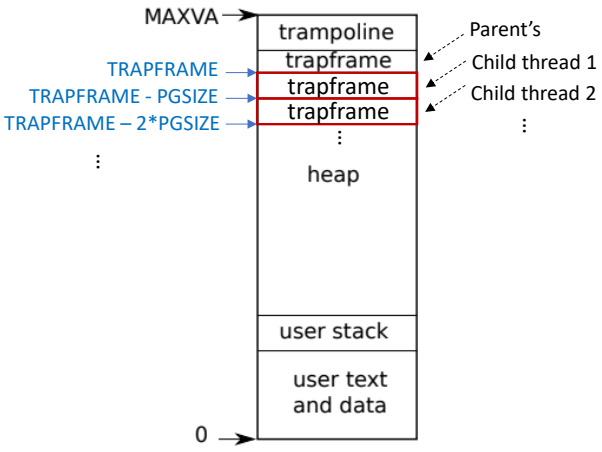
There are also some modifications required for the `wait()` syscall.

- **`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 own 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). It is fine to assume that only the parent uses `wait()` and the parent is the last one to exit (i.e., after all of its child threads have exited). In addition, parent and child do not need to share file descriptors. 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`).
- 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 explicitly the new trapframe locations for your child threads. Update kernel/trampoline.S as explained earlier. Then, at the end of `usertrapret()` in kernel/trap.c, change

  ```
  ((void (*)(uint64))trampoline_userret)(satp);
  ```

  to

  ```
  ((void (*)(uint64,uint64))trampoline_userret)(TRAPFRAME - PGSIZE * p->thread_id, satp);
  ```

  for child threads. Normal processes (or thread ID == 0) should continue to use the default `TRAPFRAME` address as follows:

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

- Trampoline (not trapframe) is already mapped by the parent and it can be shared with childs. So you must **not** map it again to the page table when creating child threads (doing so will crash). Only map the trapframe of each child (see `mappages()` function in the background).
- `wait()` uses `freeproc()` to deallocate child's resources, so you will need to make appropriate changes to `freeproc()`.

Part 2: User-level thread library

You need to implement a user-level thread library in `user/thread.c` and `user/thread.h`. **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 `user/thread.c` as a library and make it usable by other user-level programs that include `user/thread.h`.

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 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/lab3_test.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:

```
$ lab3_test 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 3 emulated CPUs.

The Code and Reference Materials

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

This Lab may take additional readings and through understanding of the concepts discussed in the handout. Especially, Chapters 2 and 4 of the xv6 book discusses the essential background for this Lab.

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 "**lab3_test 10 3**" and "**lab3_test 21 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 "**lab3_test 10 3**" and "**lab3_test 21 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: 45 pts
  - clone() implementation
  - modifications to wait()
  - other related kernel changes
- Part II: user-level thread library: 25 pts
  - thread_create() routine
  - spinlock routines
- Writeup and demo: 30 pts

Total: 100 pts

Appendix: kernel/trampoline.S

```
   #
        # code to switch between user and kernel space.
        #
        # this code is mapped at the same virtual address
        # (TRAMPOLINE) in user and kernel space so that
        # it continues to work when it switches page tables.
   #
   # kernel.ld causes this to be aligned
        # to a page boundary.
        #
   .section trampsec
.globl trampoline
trampoline:
.align 4
.globl uservec
uservec:
   #
        # trap.c sets stvec to point here, so
        # traps from user space start here,
        # in supervisor mode, but with a
        # user page table.
        #
        # sscratch points to where the process's p->trapframe is
        # mapped into user space, at TRAPFRAME.
        #

   # swap a0 and sscratch
        # so that a0 is TRAPFRAME
        csrrw a0, sscratch, a0

        # save the user registers in TRAPFRAME
        sd ra, 40(a0)
        sd sp, 48(a0)
        sd gp, 56(a0)
        sd tp, 64(a0)
        sd t0, 72(a0)
        sd t1, 80(a0)
        sd t2, 88(a0)
        sd s0, 96(a0)
        sd s1, 104(a0)
        sd a1, 120(a0)
        sd a2, 128(a0)
        sd a3, 136(a0)
        sd a4, 144(a0)
        sd a5, 152(a0)
        sd a6, 160(a0)
        sd a7, 168(a0)
        sd s2, 176(a0)
        sd s3, 184(a0)
        sd s4, 192(a0)
        sd s5, 200(a0)
        sd s6, 208(a0)
        sd s7, 216(a0)
        sd s8, 224(a0)
        sd s9, 232(a0)
        sd s10, 240(a0)
        sd s11, 248(a0)
        sd t3, 256(a0)
        sd t4, 264(a0)
        sd t5, 272(a0)
        sd t6, 280(a0)

   # save the user a0 in p->trapframe->a0
        csrr t0, sscratch
        sd t0, 112(a0)

        # restore kernel stack pointer from p->trapframe->kernel_sp
        ld sp, 8(a0)

        # make tp hold the current hartid, from p->trapframe->kernel_hartid
        ld tp, 32(a0)

        # load the address of usertrap(), p->trapframe->kernel_trap
```

```
        ld t0, 16(a0)

        # restore kernel page table from p->trapframe->kernel_satp
        ld t1, 0(a0)
        csrw satp, t1
        sfence.vma zero, zero

        # a0 is no longer valid, since the kernel page
        # table does not specially map p->tf.

        # jump to usertrap(), which does not return
        jr t0

.globl userret
userret:
        # userret(TRAPFRAME, pagetable)
        # switch from kernel to user.
        # usertrapret() calls here.
        # a0: TRAPFRAME, in user page table.
        # a1: user page table, for satp.

        # switch to the user page table.
        csrw satp, a1
        sfence.vma zero, zero

        # put the saved user a0 in sscratch, so we
        # can swap it with our a0 (TRAPFRAME) in the last step.
        ld t0, 112(a0)
        csrw sscratch, t0

        # restore all but a0 from TRAPFRAME
        ld ra, 40(a0)
        ld sp, 48(a0)
        ld gp, 56(a0)
        ld tp, 64(a0)
        ld t0, 72(a0)
        ld t1, 80(a0)
        ld t2, 88(a0)
        ld s0, 96(a0)
        ld s1, 104(a0)
        ld a1, 120(a0)
        ld a2, 128(a0)
        ld a3, 136(a0)
        ld a4, 144(a0)
        ld a5, 152(a0)
        ld a6, 160(a0)
        ld a7, 168(a0)
        ld s2, 176(a0)
        ld s3, 184(a0)
        ld s4, 192(a0)
        ld s5, 200(a0)
        ld s6, 208(a0)
        ld s7, 216(a0)
        ld s8, 224(a0)
        ld s9, 232(a0)
        ld s10, 240(a0)
        ld s11, 248(a0)
        ld t3, 256(a0)
        ld t4, 264(a0)
        ld t5, 272(a0)
        ld t6, 280(a0)

   # restore user a0, and save TRAPFRAME in sscratch
        csrrw a0, sscratch, a0

        # return to user mode and user pc.
        # usertrapret() set up sstatus and sepc.
        Sret
```