Lab3 Report by Enzuo Zhu and Shixun WU

This report describes our work on Lab 3. Section 1, 2 describes our implementation of Part 1, and 2. Section 3 demonstrates our experimental results. Here is the link to our video https://drive.google.com/file/d/19TDBcl8n51UYBU05qlp_Fv2H-bxFdYXz/view?usp=sharing.

Enzuo Zhu and Shixun Wu contribute to this project equally.

Part 1:

PCB modification

We first add two int32 variables, namely `thread_id`, `num_thread` into the PCB struct as follows:

```
// Per-process state
struct proc {
  // Lab3 addition
  int thread_id; // thread id of a process
  int num_thread; // number of thread of a process
};
```

allocproc() modification

The newly added variables are initialized in function `allocproc` as follows:

```
found:
  p->thread_id = 0;
  p->num_thread = 0;
```

New kernel function allocproc_thread()

We add a function `allocproc_thread(int pid, int tid, pagetable_t pagetable)`. In this function, we first find an unused proc struct from process array. We initialize the `thread_id`, `pagetable` with the input argument. The `thread_id` is obtained from the variable `num_thread` of parent process. The `pagetable` is same with the parent process. After that, we allocate one page of `trapframe` for this thread. Then we map the physical address of `trapframe` below other `trapframe` at the top of the process's shared virtual address using the following code `mappages(p->pagetable, TRAPFRAME - PGSIZE * (p->thread_id), PGSIZE, (uint64)(p->trapframe), PTE_R | PTE_W)`. Finally, we empty the `context` of the thread, initialize `context.ra` with `forkret`, and `context.sp` with `p->kstack + PGSIZE;`.

```
struct proc*
allocproc_thread(int pid, int tid, pagetable_t pagetable)
{
  struct proc *p;
```

```
  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state == UNUSED) {
      goto found;
    } else {
      release(&p->lock);
    }
  }
  return 0;

found:
  p->pid = pid;
  p->state = USED;
  p->num_syscall = 0;
  p->tick = 0;
  p->thread_id = tid;
  p->tickets = 10000;
  p->pagetable = pagetable;
  #if defined (STRIDE)
  p->stride = 1;
  p->pass = 0;
  #endif

  // Allocate a trapframe page.
  if((p->trapframe = (struct trapframe *)kalloc()) == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
  }

  if(mappages(p->pagetable, TRAPFRAME - PGSIZE * (p->thread_id), PGSIZE,
              (uint64)(p->trapframe), PTE_R | PTE_W) < 0){
    uvmunmap(pagetable, TRAMPOLINE, 1, 0);
    freeproc(p);
    release(&p->lock);
    return 0;
  }
  memset(&p->context, 0, sizeof(p->context));
  p->context.ra = (uint64)forkret;
  p->context.sp = p->kstack + PGSIZE;

  return p;
}
```

New syscall clone()

The system call `clone()` is initialized with syscall number `26`. In `clone()`, we first fetch the argument `stack` with function `argaddr()`. After that, we increment the `num_thread` of caller process by one. The `thread_id` of the cloned thread is initialized with current value of `num_thread`. The `pid` and `pagetable` of the cloned thread is the smae with parent process. After that, the `allocproc_thread()` is called to initialized the PCB, trapframe and context of the cloned thread. When the `allocproc_thread()` returns, the `clone()` syscal initialized the content of `trapframe`, `sz`, file descriptor `opfile`, runnable variable `state`, and `parent` variables. The implementations of `clone()` is as follow

```
uint64 sys_clone(void)
{
  uint64 stack;
  argaddr(0, &stack);
  if(stack == 0)return -1;
  int i;
  struct proc *np;
  struct proc *p = myproc();

  acquire(&p->lock);
  p->num_thread++;

  int tid = p->num_thread;
  int npid = p->pid;
  pagetable_t npagetable = p->pagetable;
  release(&p->lock);
  // Allocate process.
  if((np = allocproc_thread(npid, tid, npagetable)) == 0){
    return -1;
  }


  np->sz = p->sz;

  // copy saved user registers.
  *(np->trapframe) = *(p->trapframe);

  // Cause fork to return 0 in the child.
  np->trapframe->a0 = 0;
  np->trapframe->sp = (uint64)stack;

  // increment reference counts on open file descriptors.
  for(i = 0; i < NOFILE; i++)
    if(p->ofile[i])
      np->ofile[i] = filedup(p->ofile[i]);
  np->cwd = idup(p->cwd);

  safestrcpy(np->name, p->name, sizeof(p->name));

  release(&np->lock);

  acquire(&wait_lock);
  np->parent = p;
  release(&wait_lock);

  acquire(&np->lock);
  np->state = RUNNABLE;
  release(&np->lock);
```

```
    return tid;
  }
```

Modification on wait() and freeproc()

Instead of modifying `wait()`, we rewrite the kernel function `freeproc()`. We add a branch for thread and process in `freeproc()`. If the `proc *p` is a thread, namely `p->thread_id > 0`, we remove the `trapframe` from the `pagetable` using the following code `uvmunmap(p->pagetable, TRAPFRAME - (PGSIZE * p->thread_id), 1, 0);`. Unlike the process, free a thread does not free the `pagetable`.

```
static void
freeproc(struct proc *p)
{
  if(p->thread_id != 0){
    if(p->trapframe)
    uvmunmap(p->pagetable, TRAPFRAME - (PGSIZE * p->thread_id), 1, 0);
    p->trapframe = 0;
    p->pagetable = 0;
    p->thread_id = 0;
    p->sz = 0;
    p->pid = 0;
    p->parent = 0;
    p->name[0] = 0;
    p->chan = 0;
    p->killed = 0;
    p->xstate = 0;
    p->state = UNUSED;
  }
  else{
    if(p->trapframe)
      kfree((void*)p->trapframe);
    p->trapframe = 0;
    if(p->pagetable)
      proc_freepagetable(p->pagetable, p->sz);
    p->pagetable = 0;
    p->sz = 0;
    p->pid = 0;
    p->parent = 0;
    p->name[0] = 0;
    p->chan = 0;
    p->killed = 0;
    p->xstate = 0;
    p->state = UNUSED;
  }
}
```

Part 2:

thread_create() routine

The `thread_create()` function first allocate one page of user stack, and then call the `clone()` syscall. The newly cloned thread obtains a zero return value, then the `start_routine` is called. The parent process return directly.

```
int thread_create(void *(start_routine)(void*), void *arg){
    uint64* user_stack = malloc(PGSIZE) + PGSIZE;
    int ret = clone((void*) user_stack);
    if(ret < 0) return -1;
    if(ret != 0) return 0;
    start_routine(arg);
    exit(0);
}
```

spinlock routine

When a lock is initialized, the `lock-locked` is set to zero. In `lock_acquire()`, `while(__sync_lock_test_and_set(&lock->locked, 1) != 0)` is used to acquire the lock. After that `__sync_synchronize();` is called for synchronization. In `lock_release()`, `__sync_lock_release(&lock->locked);` is called after the synchronization.

```
struct lock_t {
uint locked;
};
void
lock_init(struct lock_t* lock)
{
  lock->locked = 0;
}

// Acquire the lock.
// Loops (spins) until the lock is acquired.
void
lock_acquire(struct lock_t *lock)
{
  while(__sync_lock_test_and_set(&lock->locked, 1) != 0)
    ;
  __sync_synchronize();
}

// Release the lock.
void
lock_release(struct lock_t *lock)
{
  __sync_synchronize();
  __sync_lock_release(&lock->locked);
}
```

Part 3: Experimental Results

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ lab3_test 10 3
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 0
Round 4: thread 0 is passing the token to thread 1
Round 5: thread 1 is passing the token to thread 2
Round 6: thread 2 is passing the token to thread 0
Round 7: thread 0 is passing the token to thread 1
Round 8: thread 1 is passing the token to thread 2
Round 9: thread 2 is passing the token to thread 0
Round 10: thread 0 is passing the token to thread 1
Frisbee simulation has finished, 10 rounds played in total
$ lab3_test 21 20
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 4
Round 5: thread 4 is passing the token to thread 5
Round 6: thread 5 is passing the token to thread 6
Round 7: thread 6 is passing the token to thread 7
Round 8: thread 7 is passing the token to thread 8
Round 9: thread 8 is passing the token to thread 9
Round 10: thread 9 is passing the token to thread 10
Round 11: thread 10 is passing the token to thread 11
Round 12: thread 11 is passing the token to thread 12
Round 13: thread 12 is passing the token to thread 13
Round 14: thread 13 is passing the token to thread 14
Round 15: thread 14 is passing the token to thread 15
Round 16: thread 15 is passing the token to thread 16
Round 17: thread 16 is passing the token to thread 17
Round 18: thread 17 is passing the token to thread 18
Round 19: thread 18 is passing the token to thread 19
Round 20: thread 19 is passing the token to thread 0
Round 21: thread 0 is passing the token to thread 1
Frisbee simulation has finished, 21 rounds played in total
$
```