CSCI 350 Project 2
Spring 2022

Kernel Threads

Kivilcim-Kylie Cumbul

Objectives

1. Synchronization using locks
   - Part 1 - lock *shared resources* & add some *functionality*
2. Implement kernel level threads
   - `kthread_create, kthread_id, kthread_exit, kthread_join`
3. Implement mutex API
   - `kthread_mutex_alloc, kthread_mutex_dealloc, kthread_mutex_lock, kthread_mutex_unlock`

Part 1

Change the implementation of some existing system calls.

- Growproc(), Fork(), Exec(), Exit()

Growproc(), Fork(), and Exit() methods are in **proc.c**

Exec() function is in **exec.c**

Note: Some functions might not need changes you have to pick which ones to change.

Growproc() - example

- Growproc() is responsible for retrieving more memory when the process asks for it
- sz variable — alloc()/delloc() checking that
- We have to synchronize sz variable

```
int
growproc(int n)
{
  uint sz;
  // added for part 1.1
  acquire(&ptable.lock);
  sz = proc->sz;
  if(n > 0){
    if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0){
      // added for part 1.1
      release(&ptable.lock);
      return -1;
    }
  } else if(n < 0){
    if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0){
      // added for part 1.1
      release(&ptable.lock);
      return -1;
    }
  }
  proc->sz = sz;
  switchuvm(proc);

  // added for part 1.1
  release(&ptable.lock);
  return 0;
}
```

in **proc.c**

Fork()

Fork should duplicate only the calling thread, if other threads exist in the process they will not exist in the new process.

Questions to ask:

Are there any conflicts between shared variables?

Do we need to kill any threads after calling fork?

Is the acquired the lock enough for synchronization or should we put more locks?

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct thread *nt;

  acquire(&ptable.lock);

  // Allocate process.
  if((np = allocproc()) == 0){
    release(&ptable.lock);
    return -1;
  }
  nt = np->threads;
```

in **proc.c**

Exit()

Should kill the process and all of its threads, remember while a single threads executing *exit*, others threads of the same process might still be running.

We have to create killall() method:

It kills all alive threads:

```
kill_all():
 Create thread pointer *t
 For each thread t:
 Begin
  If ( thread t is not current thread and not running and not
  unused)
   Make it zombie
 End
 Make current thread zombie
 Kill process
```

```
kill_all();

// Jump into the scheduler, never to return.
thread->state = TINVALID;
proc->state = ZOMBIE;
sched();
panic("zombie exit");
```

in **proc.c**

Exec()

The thread performing exec should “tell” other threads of the same process to destroy themselves and only then complete the exec task.

modify kill_all() method and create kill_others():

It kills all alive threads but itself:

```
kill_others():
 Create thread pointer *t
 For each thread t:
 Begin
  If ( thread t is not current thread and not running and not
  unused)
   Make it zombie
 End
```

in **exec.c**

Other Hints for Part 1

- Find shared variables and put locks before them
- Always release locks before return statement if it is not released previously
- Use kill_all and kill_others in respective functions.
- **Important**: I did not explain all the steps, you might need to think more about synchronization and find where to put methods, locks etc.
- I will explain how to loop through threads and where to find current thread in later slides.

```
int
growproc(int n)
{
  uint sz;
  // added for part 1.1
  acquire(&ptable.lock);
  sz = proc->sz;
  if(n > 0){
    if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0){
      // added for part 1.1
      release(&ptable.lock);
      return -1;
    }
  } else if(n < 0){
    if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0){
      // added for part 1.1
      release(&ptable.lock);
      return -1;
    }
  }
  proc->sz = sz;
  switchuvm(proc);

  // added for part 1.1
  release(&ptable.lock);
  return 0;
}
```

in **proc.c**

Part 2

Implement thread API for kernel.

- `kthread_create, kthread_id, kthread_exit, kthread_join`

You will implement these methods in **proc.c** and create header file **kthread.h**

Header file should include:

```
#define NTHREAD                 16
int kthread_create(void*(*start_func)(), void* stack, int stack_size);
int kthread_id();
void kthread_exit();
int kthread_join(int thread_id);
```

Changing thread states / Finding current thread

```
t->state = TZOMBIE;
```

```
t->tid != thread->tid
```

```
enum threadstate { TUNUSED, TEMBRYO, TSLEEPING,
TRUNNABLE, TRUNNING, TZOMBIE, TINVALID, TBLOCKED };
```

```
struct thread {
  int tid;                    // Thread ID
  enum threadstate state;     // thread state
  char *kstack;               // Bottom of kernel stack for this thread
  struct proc *parent;        // Parent process
  struct trapframe *tf;       // Trap frame for current syscall
  struct context *context;    // swtch() here to run process
  void *chan;                 // If non-zero, sleeping on chan
  int killed;                 // If non-zero, have been killed
};
```

In proc.h

Changing thread states / Finding current thread

```
t->state = TZOMBIE;
```

```
t->tid != thread->tid
```

```
enum threadstate { TUNUSED, TEMBRYO, TSLEEPING,
TRUNNABLE, TRUNNING, TZOMBIE, TINVALID, TBLOCKED };
```

Trap Frame of thread

```
*t->tf = *thread->tf;
```

```
struct thread {
  int tid;                      // Thread ID
  enum threadstate state;       // thread state
  char *kstack;                 // Bottom of kernel stack for this thread
  struct proc *parent;          // Parent process
  struct trapframe *tf;         // Trap frame for current syscall
  struct context *context;      // swtch() here to run process
  void *chan;                   // If non-zero, sleeping on chan
  int killed;                   // If non-zero, have been killed
};
```

In proc.h

How to loop through threads

Allocthread method in proc.c

```
43  struct thread*
44  allocthread(struct proc * p)
45  {
46    struct thread *t;
47    char *sp;
48    int found = 0;
49
50    for(t = p->threads; found != 1 && t < &p->threads[NTHREAD]; t++)
51    {
52      if(t->state == TUNUSED)
53      {
54        found = 1;
55        t--;
56      }
57      else if(t->state == TZOMBIE)
58      {
59        clearThread(t);
60        t->state = TUNUSED;
61        found = 1;
62        t--;
63      }
64    }
```

Create a thread pointer

Loop threads

Allocproc method in proc.c

```
107 allocproc(void)
108 {
109   struct proc *p;
110   struct thread *t;
111
112   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
113     if(p->state == UNUSED)
114       goto found;
115   return 0;
116
117 found:
118   p->state = USED;
119   p->pid = nextpid++;
120
121   t = allocthread(p);
122
123   if(t == 0)
124   {
125     p->state = UNUSED;
126     return 0;
127   }
128   p->threads[0] = *t;
129
130   for(t = p->threads; t < &p->threads[NTHREAD]; t++)
131     t->state = TUNUSED;
132
133   return p;
134 }
```

Loop processes

Loop threads

In proc.c

kill_all()

```
kill_all():
 Create thread pointer *t
 For each thread t: -> loop through threads
 Begin
  If ( thread t is not current thread and not running and not unused)->
  check its state
   Make it zombie-> change its state
 End
 Make current thread zombie -> find current thread and change its state
 Kill process -> proc->killed = 1
```

in **proc.c**

kthread_create()

- Calling kthread_create will create a new thread within the context of the calling process. The newly created thread state will be TRUNNABLE. The caller of kthread_create must allocate a user stack for the new thread to use (it should be enough to allocate a single page i.e., 4K for the thread stack). This does not replace the kernel stack for the thread.

- start_func is a pointer to the entry function, which the thread will start executing. Upon success, the identifier of the newly created thread is returned. In case of an error, a non-positive value is returned.

- The kernel thread creation system call on real Linux does not receive a user stack pointer. In Linux the kernel allocates the memory for the new thread stack. You will need to create the stack in user mode and send its pointer to the system call in order to be consistent with current memory allocator of xv6.

Implement in proc.c

kthread_create

```
kthread_create(void*(*start_func)(), void* stack, int stack_size)
```

Create a thread pointer

Allocate the thread using allocthread method

Check if t is 0 –> allocated correctly?

    If not return -1

Else

    Copy current thread's trap frame

    Find **stack address** of the thread using stack pointer given **parameter**

    Make **stack pointer** inside trap frame stack **address + stack size**

    Update **base pointer** inside trap frame as **stack pointer**

    Find **address of the start function** which is given in **parameter**

    Make **instruction pointer** inside trap frame **start address**

    return t_id

t->tf->esp

t->tf->ebp

t->tf->eip

This is not the only way to create a thread

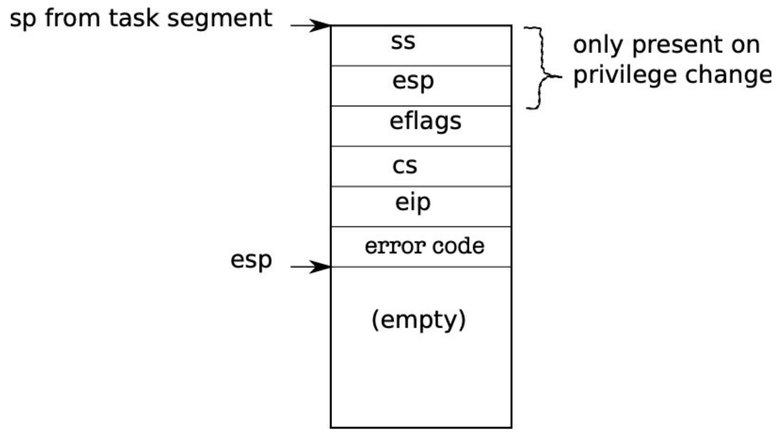
Implement in proc.c

esp, eip, ebp?

For more information you can refer to xv6 manual chapter 3

t->tf->esp

We have to change the stack address of new thread
So we use given parameter to make stack address different than current thread
Add given **stack's address** with **stack size** to find where to put stack pointer



Kernel stack after an int instruction.

t->tf->ebp

Initially base pointer and stack pointer point the same place

t->tf->eip

Instruction pointer points the instructions which will be implemented by thread
This pointer has to point stack function at the beginning

xv6 manual: https://pdos.csail.mit.edu/6.828/2014/xv6/book-rev8.pdf

kthread_id

Easiest function to implement in project 2 :)

- Upon success, this function returns the caller thread's id. In case of error, a non-positive error identifier is returned.
- Remember, thread id and process id are not identical.

Implement in proc.c

kthread_id

Easiest function to implement in project 2 :)

```
kthread_id()

    If process and thread exists
       return t->t_id
    Else
      return -1
```

This is not the only way to return a thread id

Implement in proc.c

kthread_exit

- This function terminates the execution of the calling thread.
- If called by a thread (even the main thread) while other threads exist within the same process, it shouldn't terminate the whole process.
- If it is the last running thread, process should terminate. Each thread must explicitly call *kthread_exit()* in order to terminate normally.

Implement in proc.c

kthread_exit

```
kthread_exit()
Create a thread pointer
Create a found flag
Loop through all threads to find another thread running
    If t is not current thread (because calling thread is current)
        If t is not Unused, not Zombied and not Invalid
            Make flag true
            Break –> only one running t is enough
If (found)
    Wakeup all waiting using wakeup1()
Else –> not found
    exit()
    wakeup()
Make this thread zombie
Call shed to schedule another thread
```

This is not the only way to exit a thread
This is not a complete pseudocode
You have to add locks if necessary

Implement in proc.c

kthread_join

- This function suspends the execution of the calling thread until the target thread (of the same process), indicated by the argument thread_id, terminates.
- If the thread has already exited, execution should not be suspended.
- If successful, the function returns zero.
- Otherwise, -1 should be returned to indicate an error.

Implement in proc.c

kthread_join

```
kthread_join(int thread_id)
Check if thread_id is valid
Create a thread pointer t
Loop through all threads to find target thread id(parameter)
    Make t point target thread with thread_id
If not found
    return -1
While (t->t_id = thread_id and valid)
    Make t sleep using sleep method with a lock
If state of t is zombie
    clearThread(t);
return 0
```

This is not the only way to join threads
This is not a complete pseudocode
You have to add locks if necessary

Implement in proc.c

Part 3

Implement thread API for mutex.

- `kthread_mutex_alloc, kthread_mutex_dealloc, kthread_mutex_lock, kthread_mutex_unlock`

- You will implement these methods in **proc.c** and add methods into header file **kthread.h**

Header file should include:

```
#define MAX_MUTEXES     64
int kthread_mutex_alloc();
int kthread_mutex_dealloc(int mutex_id);
int kthread_mutex_lock(int mutex_id);
int kthread_mutex_unlock(int mutex_id);
```

Changes in proc.h

```
enum threadstate { TUNUSED, TEMBRYO, TSLEEPING, TRUNNABLE, TRUNNING, TZOMBIE, TINVALID, TBLOCKED };
```

Create mutexstate struct similar to this

You need:
Unused, locked, unlocked states

```
struct thread {
  int tid;                    // Thread ID
  enum threadstate state;     // thread state
  char *kstack;               // Bottom of kernel stack for this thread
  struct proc *parent;        // Parent process
  struct trapframe *tf;       // Trap frame for current syscall
  struct context *context;    // swtch() here to run process
  void *chan;                 // If non-zero, sleeping on chan
  int killed;                 // If non-zero, have been killed
};
```

Create kthread mutex struct similar to this

You need:
mutex_id
Mutex state
[other variables depending on implementation]
Ex: blocked threads array

Implement in proc.h

Changes in proc.c

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

```
29   int nextpid = 1;
30   int nexttid = 1;
31   int nextmid = 1;
```

You need to have a spin lock on mutex table similar to this

We will discuss other mutex methods in next slides…

Implement in proc.c

kthread_mutex_alloc()

- Allocates a mutex object and initializes it; the initial state should be unlocked.
- The function should return the ID of the initialized mutex, or -1 upon failure

Implement in proc.c

Looping through mutex table

```
for (i = 0; i < MAX_MUTEXES; i++)
{
  m = &mtable.mutex[i];
  // if found break
  if(m->mid == mutex_id)
    break;
}
```

```
for(m = mtable.mutex; m < &mtable.mutex[NPROC]; m++);
```

You can use other ways to do this, i.e. while loop etc.

kthread_mutex_alloc()

```
Create a mutex pointer m (using struct in proc.h)
Loop through all mutex table
    If m is unused
        m->mid = nextmid++;
        m->state = M_UNLOCKED;
        Initiate all other values
If (&mtable.mutex[NPROC]) -> unused mutex not found
    return -1 –> cannot allocate because there is no unused mutex in mutex table
Else
    Return m->mid
```

This is not a complete pseudocode
You have to add locks if necessary

This is not the only way to implement mutex allocation

Implement in proc.c

kthread_mutex_dealloc()

- De-allocates a mutex object which is no longer needed.
- The function should return 0 upon success and -1 upon failure
- For example, if the given mutex is currently locked

Implement in proc.c

kthread_mutex_dealloc()

```
Create a mutex pointer m (using struct in proc.h)
Loop through all mutex table to find given mutex_id
    If m is locked -> we can't dealloc
        Return -1
If (&mtable.mutex[NPROC]) -> unused mutex not found
    return -1 —> cannot allocate because there is no unused mutex in mutex table
Else -> deallocate all properties of mutex
    m->mid = 0
    m->state = MUNUSED;
    Zero all other values
    Return 0
Return -1 —> if no mutex_id is found
```

This is not a complete pseudocode
You have to add locks if necessary

This is not the only way to implement mutex deallocation

Implement in proc.c

kthread_mutex_lock()

- This function is used by a thread to lock the mutex specified by the argument *mutex_id*.
- If the mutex is already locked by another thread, this call will block the calling thread (change the thread state to *TBLOCKED*) until the mutex is unlocked.

in proc.h

```
enum threadstate { TUNUSED, TEMBRYO, TSLEEPING,
TRUNNABLE, TRUNNING, TZOMBIE, TINVALID, TBLOCKED };
```

Add TBLOCKED state if you did not add earlier

Implement in proc.c

kthread_mutex_lock()

```
Create a mutex pointer m (using struct in proc.h)
Loop through all mutex table to find given mutex_id
    If m->mid == mutex_id
        break;
If (i is MAX_MUTEXES) -> given mutex_id not found
    return -1
while(m->state == M_LOCKED)
    sleep
If (m->state != M_UNLOCKED) -> failed
    Return -1

m->state = M_LOCKED;
Return 0
```

Spin lock

This is not a complete pseudocode
You have to add locks if necessary

YOU HAVE TO IMPLEMENT TBLOCKED!

This is not the only way to implement mutex lock

Implement in proc.c

kthread_mutex_unlock()

- This function unlocks the mutex specified by the argument mutex_id if called by the owning thread, and if there are any blocked threads, one of the threads will acquire the mutex.
- An error will be returned if the mutex was already unlocked.
- The mutex may be owned by one thread and unlocked by another!

Implement in proc.c

kthread_mutex_unlock()

```
Create a mutex pointer m (using struct in proc.h)
Loop through all mutex table to find given mutex_id
    If m->mid == mutex_id
        break;
If (m == &mtable.mutex[NPROC]) -> given mutex_id not found
    return -1
If (m->state is not LOCKED)
    return -1
m->state = M_UNLOCKED;
Call wakeup1 on m to wakeup thread
Return 0
```

This is not a complete pseudocode
You have to add locks if necessary

This implementation does not include last part of description as well:

- The mutex may be owned by one thread and unlocked by another!

This is not the only way to implement mutex unlock

Implement in proc.c

Common Errors

```
./sign.pl bootblock
make: execvp: ./sign.pl: Permission denied
Makefile:95: recipe for target 'bootblock' failed
make: *** [bootblock] Error 127
```

```
sudo chmod +x *.pl
```

Common Errors

```
$ threadtest1
3 threadtest1: unknown sys call 23
thread in main -1,process 3
3 threadtest1: unknown sys call 22
3 threadtest1: unknown sys call 22
3 threadtest1: unknown sys call 25
Got id : -1
3 threadtest1: unknown sys call 25
Got id : -1
Finished.
3 threadtest1: unknown sys call 24
```

Make sure to implement system calls for all kthread and mutex methods.

Common Errors

Most important error in Project 2

```
thread in main 3,process 3
Thread id is: 3
cpu with apicid 0: panic: acquire
 80103e81 80103aa8 80104f3c 80104263 80105117 80104ff3 80111e8c 0 0 0
```

Also, panic: release

panic: acquire and panic: release errors mean that program fails to acquire lock because it is already acquired earlier or it cannot released lock because it is already released.

Common Errors

panic: sched locks

This means process holding multiple locks.
Before calling sched() make sure to release all other locks.

ac(ptable)
ac(ttabke)
ac(mtable)

rel(mtable)
rel(ttable)
rel(ptable)

Questions?