P1: Implement a Multi-Threading Package (in user space)

Robbert van Renesse
Implement the following interface:

void thread_init();
  • initialize the user-level threading module (process becomes a thread)
void thread_create(void (*f)(void *arg), void *arg, unsigned int stack_size);
  • create another thread that executes f(arg)
void thread_yield();
  • yield to another thread (thread scheduling is non-preemptive)
void thread_exit();
  • thread terminates and yields to another thread or terminates entire process
Example usage

```c
static void test_code(void *arg){
    int i;

    for (i = 0; i < 10; i++) {
        printf("%s here: %d
", arg, i);
        thread_yield();
    }
    printf("%s done
", arg);
}

int main(int argc, char **argv){
    thread_init();
    thread_create(test_code, "thread 1", 16 * 1024);
    thread_create(test_code, "thread 2", 16 * 1024);
    test_code("main thread");
    return 0;
}
```
You’ll need to understand stacks *really well*
Review: stack (aka call stack)

```c
int main(argc, argv){
    f(3.14)
    ...
}

int f(x){
    ...
    g();
    ...
}

int g(y){
    ...
}
```

stack frame for main()
Review: stack (aka call stack)

```c
int main(argc, argv){
    ... 
    f(3.14)
    ... 
}

int f(x){
    ... 
    g();
    ... 
}

int g(y){
    ... 
}
```
Review: stack (aka call stack)

```c
int main(argc, argv){
    ...
    f(3.14)
    ...
}
int f(x){
    ...
    g();
    ...
}
int g(y){
    ...
}
```
Review: stack (aka call stack)

```c
int main(argc, argv){
    ...
    f(3.14)
    ...
}

int f(x){
    ...
    g();
    ...
}

int g(y){
    ...
}
```

- stack frame for main()
- stack frame for f()
- stack frame for g()

- arguments (3.14)
- return address
- saved FP (main)
- local variables
- saved registers
- scratch space
Review: stack (aka call stack)

```c
int main(argc, argv){
    ...
    f(3.14)
    ...
}

int f(x){
    ...
    g();
    ...
}

int g(y){
    ...
}
```

- Stack frame for `main()`
- Stack frame for `f()`

- Arguments (3.14)
- Return address
- Saved FP (main)
- Local variables
- Saved registers
- Scratch space

PC/IP → FP → SP
Review: stack (aka call stack)

```c
int main(argc, argv){
    ...
    f(3.14)
    ...
}
int f(x){
    ...
    g();
    ...
}
int g(y){
    ...
}
```
Each thread has its own stack!!
Each thread has its own stack!!
Each thread has its own stack!!

• And its own PC (aka IP), SP, FP, general purpose registers
But we have only one CPU, one core

• And the process has only one stack

We need some magic...

(where’s the thread?)
We run one thread at a time

- and save the state of the other threads in a secret location

- The state of a thread (aka context) consists of
  - its registers (including PC, SP, and FP)
  - its stack
  - possibly more stuff (scheduling state)
Context Switching

- When a thread exists (thread_exit) or yields (thread_yield) another thread, if any, gets to run
- If a thread yields, we need to save its context
- We need to be able to restore another context
Where to store the context of a thread?

• Convenient to push a thread’s registers onto the stack
  • but you can’t save the stack pointer on the stack...
• Keep the stack pointer in a *Thread Control Block*
  • one TCB per thread
Thread Control Block

- **SP**
- **BASE**

Stack frame

saved registers
Thread Control Block (initial state)
Scheduling State of a Thread

• Running
  • currently running

• Runnable (aka Ready)
  • TCB on the run queue (aka ready queue)

• Terminated
  • TCB marked as having terminated
thread_init()

• Initializes thread package
• Maintains run queue and current thread
• Allocates a TCB, but *not* a stack
  • because process already has one in use
• Set TCB->base to NULL to mark no stack has been allocated
• Initial run queue is empty
• Current thread points to allocated TCB
thread_create(f, arg, stack_size)

• Create a new thread
• Allocates a TCB and a stack (of the given size)
  • set TCB->base to “bottom”, and TCB->sp to “top”
• May or may not immediately switch to the new thread
  • I think it’s easier if you switch immediately
thread_yield()

• See if the run queue is empty
  • if so, we’re done
• Get next TCB of the run queue
• Put current TCB on the run queue
• **Switch contexts**
  • Save registers on the stack
  • Save sp in current TCB
  • Restore sp of next TCB
  • Restore registers from the stack
thread_exit()

• See if the run queue is empty
  • if so, exit from the process using exit(0)
• Mark TERMINATED in TCB
• Get next TCB of the run queue
  • **Switch contexts**
    • Save registers on the stack
    • Save sp in current TCB
    • Restore sp of next TCB
    • Restore registers from the stack
• Next thread cleans up last thread
ctx_switch(&old_sp, new_sp)

ctx_switch: // ip already pushed!
    pushq  %rbp
    pushq  %rbx
    pushq  %r15
    pushq  %r14
    pushq  %r13
    pushq  %r12
    pushq  %r11
    pushq  %r10
    pushq  %r9
    pushq  %r8
    movq  %rsp, (%rdi)
    movq  %rsi, %rsp
    popq  %r8
    popq  %r9
    popq  %r10
    popq  %r11
    popq  %r12
    popq  %r13
    popq  %r14
    popq  %r15
    popq  %rbx
    popq  %rbp
    retq

USAGE:

struct tcb *current, *next;

void yield()
{
    assert(current->state == RUNNING);
    current->state = RUNNABLE;
    runQueue.add(current);
    next = scheduler();
    next->state = RUNNING;
    ctx_switch(&current->sp, next->sp)
    current = next;
}
Starting a new process

```c
void thread_create( func ){
    current->state = RUNNABLE;
    runQueue.add(current);
    next = malloc(…);
    next->func = func;
    next->stack = malloc(…)
    next->state = RUNNING;
    ctx_start(&current->sp, top_of_stack)
    current = next;
}

void ctx_entry(){
    current = next;
    (*current->func)();
    current->state = FINISHED;
    finishedQueue.add(current);
    next = scheduler();
    next->state = RUNNING;
    ctx_switch(&current->sp, next->sp)
    // this location cannot be reached
}
```

```assembly
ctx_start:
    pushq  %rbp
    pushq  %rbx
    pushq  %r15
    pushq  %r14
    pushq  %r13
    pushq  %r12
    pushq  %r11
    pushq  %r10
    pushq  %r9
    pushq  %r8
    movq   %rsp, (%rdi)
    movq   %rsi, %rsp
    callq  ctx_entry
```
Synchronization Primitives
Semaphores

• We’re not teaching general semaphores in CS4410 anymore
• A semaphore is a kind of counter:

```c
struct sema;
void sema_init(struct sema *sema, unsigned int count);
void sema_dec(struct sema *sema);
void sema_inc(struct sema *sema);
bool sema_release(struct sema *sema);
```
Semaphore interface

void sema_init(struct sema *sema, unsigned int count)
• Initialize the semaphore to the given counter

void sema_dec(struct sema *sema)
• Wait until sema > 0, then decrement semaphore

void sema_inc(struct sema *sema)
• Increment the semaphore

bool sema_release(struct sema *sema)
• Release the semaphore
Example usage: Producer/Consumer

Producers block when buffer is full

Consumers block when buffer is empty
Example usage: Producer/Consumer

```c
#define NSLOTS 3

static struct sema s_empty, s_full, s_lock;
static unsigned int in, out;
static char *slots[NSLOTS];

int main(int argc, char **argv){
    thread_init();
    sema_init(&s_lock, 1);
    sema_init(&s_full, 0);
    sema_init(&s_empty, NSLOTS);

    thread_create(consumer, "consumer 1", 16 * 1024);
    producer("producer 1");
    return 0;
}
```
Example usage: Producer/Consumer

```c
static void producer(void *arg){
    for (;;) {
        // first make sure there's an empty slot.
        sema_dec(&s_empty);

        // now add an entry to the queue
        sema_dec(&s_lock);
        slots[in++] = arg;
        if (in == NSLOTS) in = 0;
        sema_inc(&s_lock);

        // finally, signal consumers
        sema_inc(&s_full);
    }
}
```
Example usage: Producer/Consumer

```c
static void consumer(void *arg){
    unsigned int i;

    for (i = 0; i < 5; i++) {
        // first make sure there's something in the buffer
        sema_dec(&s_full);

        // now grab an entry to the queue
        sema_dec(&s_lock);
        void *x = slots[out++];
        printf("%s: got '%s'", arg, x);
        if (out == NSLOTS) out = 0;
        sema_inc(&s_lock);

        // finally, signal producers
        sema_inc(&s_empty);
    }
}
```
Semaphore implementation

• Associate a queue with the semaphore
• If thread tries to decrement a zero semaphore, put its TCB on the queue
• If thread increments a semaphore with a non-empty queue, don’t increment the semaphore but move one TCB from the semaphore’s queue to the read queue
EGOS (Earth and Grass O.S.)
Overview

• Runs as a process in user space on Linux, Mac OS X, ...
  • as long as it supports mmap()

• Architecture:
  • Earth: a virtual machine monitor (like VMWare, VirtualBox, KVM, ...)
  • Grass: a microkernel operating system
    • microkernel: file system etc. runs mostly in user space
Earthbox

• Emulates a computer
  • Interrupts
  • TLB
  • Devices (disks, tty, clock, etc.)
• Sets up the address spaces for Grass kernel and EGOS processes
• Then context switches to Grass kernel
Grass Microkernel

- Organized as a collection of processes
  - processes communicate through exchanging messages
  - process can only block by waiting for a message
- Some are purely kernel processes, some support user space
- Device drivers are implemented as kernel processes
  - invoke Earth’s virtual devices
- Main file system implemented in user space
  - a simple file system implemented in kernel space for booting
Address Space Regions

- **real kernel space**
  - 0xFFFFFFFFFFFFF
  - 0x800000000000

- **real user space**
  - 0x002000000000
  - 0x001000000000
  - 0x000100000000
  - 0x000000000000

- **fake kernel space**
  - 0xFFFFFFFFFFFFF

- **fake user space**
  - 0x800000000000

- **Host Kernel (Linux, MacOSX, ...)**
- **Grass Microkernel**
- **EGOS processes**
- **Earthbox virtual machine monitor**
Very very small system call interface

- `sys_getpid()`
- `sys_recv(&message)`
- `sys_send(message)`
- `sys_rpc(request, &response)`
- `sys_exit(status)`
- `sys_gettime()`
- `sys_print(string)`
Other O.S. services

• spawn a process:
  • send request to kernel spawn server

• read/write/create a file:
  • send request to one of the file servers

• print something:
  • send request to kernel tty server

• read from keyboard:
  • send request to kernel tty server

• ...