Project 2: Cooperative Multitasking

Managing processes

INF-2201 Staff

Spring 2024

Contents

Project 2 **2**
Project Overview 2
Project 2: Administrative Info 2
Project 2: Cooperative multitasking 2
Project 2: Simple Environment 2
Project Tasks 3
Tasks Overview 3
. . . . 3
Processes and Threads / Process Control Block (PCB) 3
Task: Design Process Control Block (PCB) 3
Task: Initialize Kernel 5
Memory Map 5
Task: Implement Scheduler 6
Task: Implement Context Switching 6
Incorporating Assembly with C 7
Task: Implement Locks: Blocking and Unblocking 7
Task: Implement System Call Mechanism 8
Syscall vs Scheduler 8
Task: Measure Context Switch Time 9
Extra Challenges 9
Administrative Details 9
Design Reviews 9
Possible Topics for Design Review 10
Code 10
Report 10
Hand In via Canvas 10

Project 2

Project Overview

Project 2: Administrative Info

- Mandatory assignment
  - This is a mandatory assignment
  - It will be graded as pass/fail by TAs
  - You must pass to gain admission to the exam
- Groups of two
  - You must work in groups of two
- Design review
  - Meet with TA and present your design
  - Informal, but must have some kind of slides / presentation
- Hand-in
  - Hand-in via Canvas
  - Code: whole repository (working tree + `.git/` dir)
  - Report: place in a `report/` directory in your repo
  - Zip up and upload to Canvas

Project 2: Cooperative multitasking

- Switch between multiple threads of execution
- Threads can give up control via function:
  - `yield`: give up control
  - `block`: wait on a lock
  - `exit`: stop execution, allow kernel to clean up
- Kernel takes control and then gives it to another thread
  - Save context for yielding/blocking thread
  - Choose next thread
  - Set up / restore context for next thread

Project 2: Simple Environment

- Protected mode, but no active protection
  - CPU in 32-bit Protected Mode
  - No protection active: all runs at kernel level (Ring 0)
  - Flat 32-bit address space
  - Kernel and processes share one address space
- No `malloc` / `free`
  - Statically allocate any globals / arrays you need
  - Allocate single structs from arrays

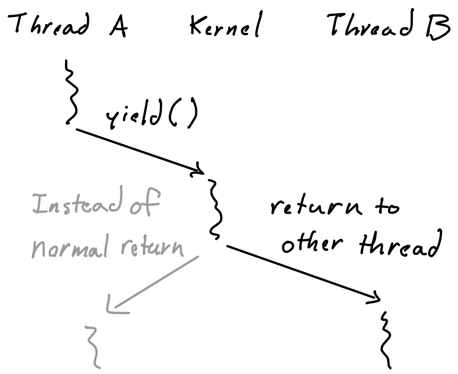

Figure 1: Returning to another thread

Project Tasks

Tasks Overview

- Design Process Control Block (PCB)
- Initialize kernel
- Implement scheduler
- Implement context switching
- Implement locks, blocking, and unblocking
- Implement system call mechanism
- Measure context switch time

Processes and Threads / Process Control Block (PCB)

Process	Thread	File
Kernel	clock_thread	th1.c
Kernel	lock_thread[0–1]	th2.c
Kernel	mcpi_thread[0–3]	th3.c
Plane	main	process1.c
Math	main	process2.c

- `struct pcb` (Process Control Block)
  - One struct for process info + thread info
  - Don't let this blur your understanding of processes vs threads

Task: Design Process Control Block (PCB)

We give you

- `struct pcb` in pcb.h
- `next` / `previous` pointers + fns to put PCBs into a linked list

```
struct pcb {
    /* PCBs as a doubly-linked list */
    struct pcb      *next;
    struct pcb      *previous;

    /* TODO */
};
```

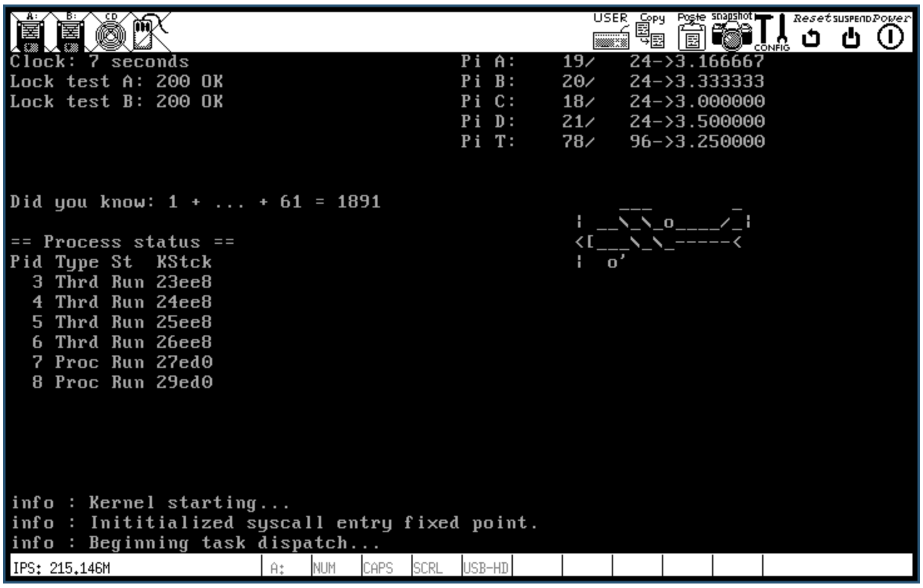

Figure 2: Project 2 example screenshot

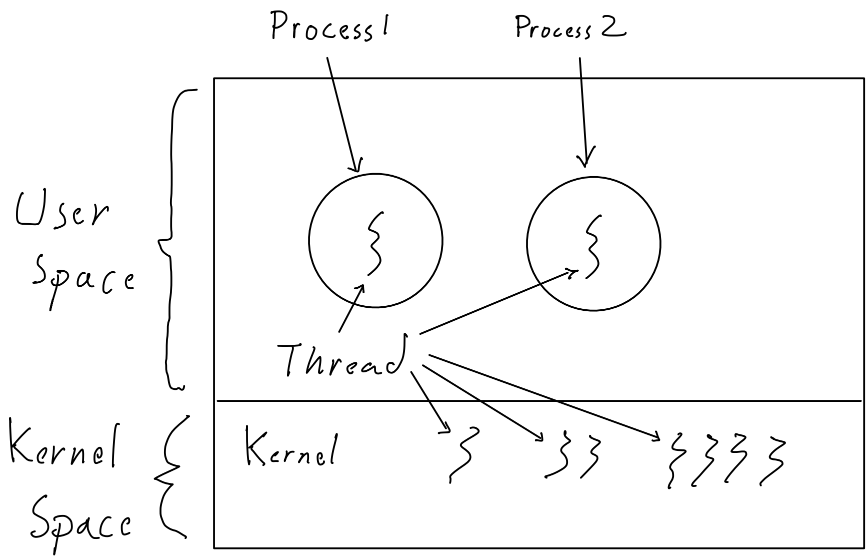

Figure 3: Project 2 processes and threads

- stack pointer? stack pointers?

Look at other OSes

- What does Linux do? Windows?
- Read case studies in the textbook
- Search online
- Comparing OSes is a great thing to bring up in your design reviews

Task: Initialize Kernel

- We give you:
  - `init_cpu()`: sets up Global Descriptor Table
  - `init_syscalls()`: sets up function pointer for system calls
- You must:
  - Initialize PCB table
  - Initialize PCBs for in-kernel threads and in-process threads
    * No need to do dynamic loading
    * Just set up an array at startup

Memory Map

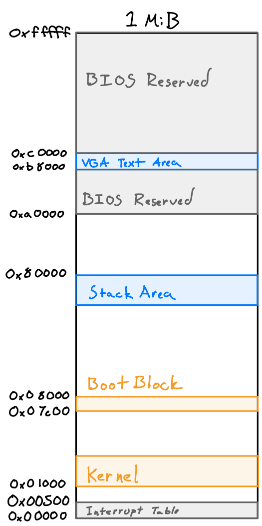

Figure 4: Project 1 memory map

- Kernel + Process 1 + Process 2 will overwrite boot block
- Bootblock must relocate itself
- This used to be a gotcha in the assignment

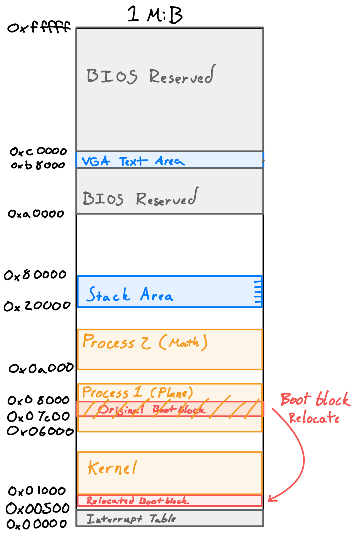

Figure 5: Project 2 memory map

- Stack area
  - Used for boot
  - Divide to allocate stacks to threads
  - How many stacks per thread?
- File: `addrs.h`

Task: Implement Scheduler

- How do you want to pick the next process?
- Simple round-robin is fine
- But you are welcome to try more advanced options

Task: Implement Context Switching

- How do you switch between threads?
- Cooperative: thread gives up control
  - `yield`: give up control
  - `block`: wait on a lock
  - `exit`: stop execution, allow kernel to clean up
- What state to save?
  - Registers
  - Stack pointer
  - Anything else?
- Where and how to save it?

- Store in PCB fields?
- Push everything onto the stack?
- Hybrid?

Incorporating Assembly with C

- Some things must be done in assembly
- Anything where you manipulate the stack

Infrastructure in precode

- Extra `_asm.S` file for code units that need assembly

header	scheduler.h	syscall.h
C code	scheduler.c	syscall.c
assembly code	scheduler_asm.S	syscall_asm.S

- Include file: `asm_common.h.S`
  - Place to define asm macros
  - Recommend: asm `.macro` over C `#define`
- Export file: `asm-offsets.c`
  - Export C constants to asm
  - Especially struct field offsets
    e.g. `offsetof(struct pcb, next)`

Task: Implement Locks: Blocking and Unblocking

- Locks in `sync.[ch]`
- We don't really *need* synchronization at this point
- Because threads can't be interrupted
- But these locks are here to test your scheduler
  - The lock-test and Monte-Carlo Pi threads use the lock API
  - If you mishandle the locks, they will not behave properly
- Can you block and unblock threads?
  - Not specific to locks
  - General service that the kernel should support

```
struct _lock {
    /* TODO: Design your lock struct.
     * You'll need at the very least a flag
     * and a wait queue. */
};

typedef struct _lock lock_t;
```

```c
#define LOCK_INIT \
    { \
    }

void lock_acquire(lock_t *);
void lock_release(lock_t *);
```

Task: Implement System Call Mechanism

- How does a user process *outside* of the kernel make a call *into* the kernel?
  - Kernel and process are separate executables
  - Not linked together
  - Do not know exact memory addresses of kernel functions
- Need some kind of agreed-upon protocol
  - Project 2 approach: Indirect function call via fixed address (`0xf00`)

```
USER SIDE
process     syslib      syslib
main()  ->  yield() ->  invoke_syscall_entry_fixedpoint(YIELD) ->

            -> pointer (*0xf00)(YIELD) ->

KERNEL SIDE
    kernel asm                  kernel C                    kernel C
->  syscall_entry(YIELD)    ->  syscall_dispatch(YIELD)  -> yield()
```

Syscall vs Scheduler

- Both require low-level context switch
- Syscall:
  - same thread
  - switch from *user* context to *kernel* context
- Scheduler:
  - already in kernel context
  - switch from *one thread* to *another*
- Keeping this stuff straight can be a little mind-bending
  - Need to look behind abstractions like flow of execution and function calls
  - Think about raw program state: register values, stack pointer, instruction pointer
  - Low-level kernel code has to get very meta: manipulate state, shuffle stack pointers

Task: Measure Context Switch Time

- You decide what to measure and how (methodology)
  - There is code in `util.[ch]` for measuring CPU time
- Do your measurements
- Get results
- Write it up in your report

Extra Challenges

- We can't actually give extra credit on an assignment that doesn't count towards your grade
  - But trying these challenges will enhance your understanding
  - And make you stand out
  - Maybe get a TA job next year?

1. Implement something similar to the unix `time` command

   ```
   $ time make clean all
   real    0m3,454s
   user    0m2,517s
   sys     0m0,923s
   ```

   - Measure time in user space vs time in kernel space ("sys")
   - Don't bother trying to calculate real-time
2. More threads
   - Add a new in-kernel thread
   - Add a new user process

Administrative Details

Design Reviews

- Meet with your TA and present your design for the project
  - In colloquium period ~1 week after hand-out
  - TAs will schedule design review times
- This is mandatory. The design review is part of the assignment.
- This is not a formal presentation
  - But you should have some slides/visuals to show
  - Your task is to convince them that you understand the project
- Keep it at the design level
  - You don't need to go deep into implementation details
  - But we want to see that you have a ideas for implementation

Possible Topics for Design Review

- What will go into your PCB? And why?
- How will you implement locks and process queues?
- What do you need to save on context switch?
  - Where will you save it?
- What is the difference between processes and threads?
  - How are the concepts muddied in our precode?
- How many stacks will you use per thread? And why?

Code

- Code should be as readable as possible
  - Put thought into names and order
- Be sure to comment your code
  - Comments should explain the reasoning behind the code
  - This is especially important when dealing with entry points, synchronization, and blocking
  - Comments are part of the grading
- Be sure to test your code
  - Run your OS in the emulator
  - If writing code in `lib/`, you can try using the unit test framework

Report

- Should be around 4 pages
- Give an overview of how you solved each task (or extra challenge)
- Describe how you tested your code
  - Point out any known bugs/issues
  - Describe how you would try to fix the bugs if you had more time
- Describe the methodology, results, and conclusions of your performance measurements

Hand In via Canvas

- Put your report in your repository, under a `report/` dir
  - Report shoud be a PDF format
  - If you write in Word or other WYSIWYG word processor, export to PDF
- If you write in a document prep system like Markdown or LaTeX, you can include the report source if you like, but it's not required.

- Zip up your entire repository (code tree + report + `.git/` dir)
- Submit via Canvas