UiT The Arctic University of Norway

Project 2: Non-preemptive Scheduling

INF-2201 Operating System Fundamentals

Spring 2023

Overview

- In P1 you implemented a boot loader
  - But the OS kernel was very minimalistic
- Here you implement a simple OS kernel
- You can (but are not required to) use the code you wrote in P1

Kernel

- Multitasking support
- *Non-preemptive* scheduling
- A running process/thread has to relinquish control explicitly by:
    - Yielding
    - Exiting
    - Blocking on a lock

Processes and Threads

- Processes, threads and the kernel share the same flat address space
  - But how do they differ?
  - And how do they differ from Linux Threads and Processes?
- No real protection
- Separate address spaces later

Protected Mode

- Kernel runs in protected mode
    - Setup done in boot loader
- Everything runs at highest CPU privilege level (ring 0)

- More about protected mode in colloquium groups

Tasks

- Initialization
- Process Control Block (PCB)
- Context switching
- System call mechanism
- Stacks
- Synchronization
- Assembly / inline assembly
- Performance measurements
- Design review

Initialization

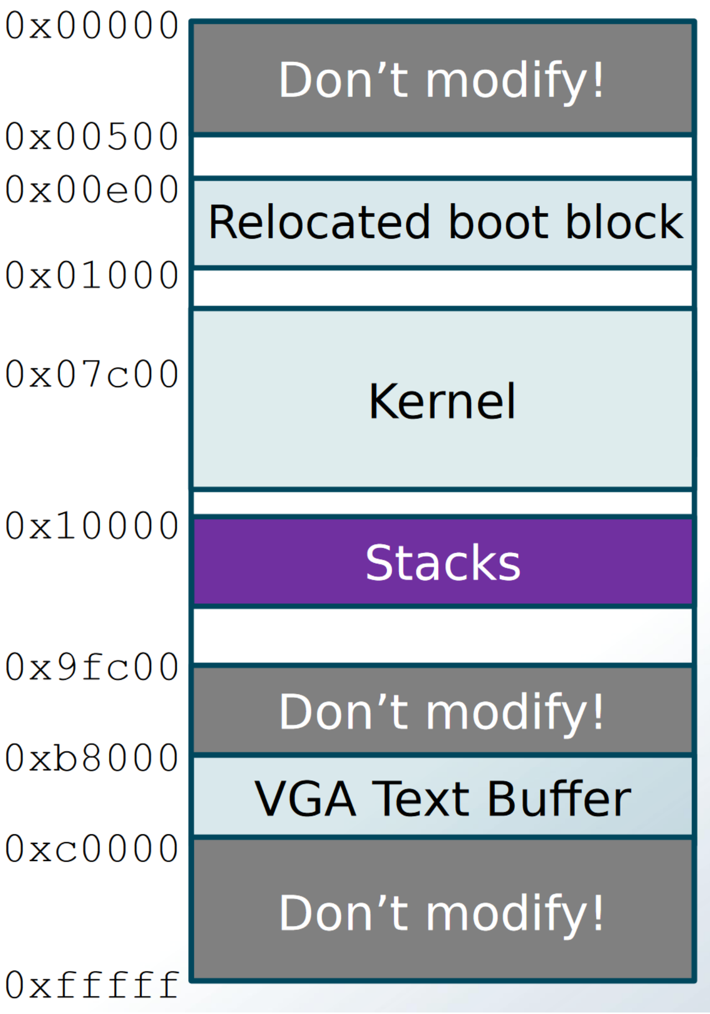
- Initialization
- Setup the required data structures for processes and threads
    - No dynamic loading
    - Everything set at startup

Process Control Block (PCB)

- In kernel.h
- What should be in the PCB?
    - pid?
    - is_thread?
    - stack?
    - next, previous PCB?
- Anything else?
    - What is in the Linux PCB? Or Windows PCB?
    - ⇒ Describe this in the design review!

Stacks

- How many stacks?
  - 2 per process, 1 per thread
    - Why?
- Where to put them in memory?
  - Upper limit: 0x9fc00
    - Suggestion: between 0x10000 and 0x20000
- Stack size?
  - 8KB should be fine.



Memory layout
(not to scale)

Scheduler

- Simple non-preemptive scheduling
  - IS round-robin good enough?
  - How to do blocking and unblocking?
- ⇒ Design review!

Context Switch Procedure

- How to switch between processes and threads?
  - They must call yield() explicitly (non-preemptive)
  - In next project: time slice expires (preemptive)
- What to save?
  - GPRs?
  - More?
- Where to save it?
  - Stack?
  - PCB?
- ⇒ Design review!

System Call Mechanism

- How does a process get services from the kernel?
  - This assignment: a special function call using a single entry “jump table”
  - Later: interrupt/trap mechanism
- Why cannot system calls be implemented as ordinary function calls?
- Is our approach better?

System Call Mechanism

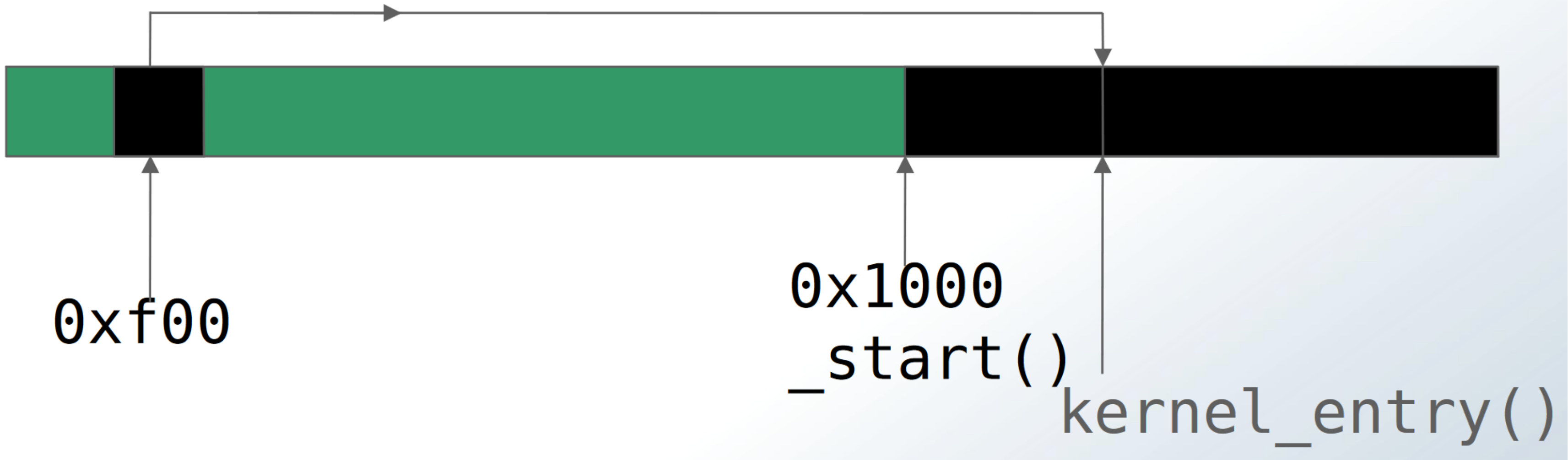
- At runtime, load the address of `kernel_entry()` into memory location `0xf00`
  - Already done in pre-code
- Prototype: `void kernel_entry(int fn);`
- Define this in syslib.h

  ```
  #define ENTRY_POINT ((void (**)()) 0xf00)
  ```
  - Pointer to pointer to function with non-defined argument list returning void
  - ...at address `0xf00`
- Declare the following in syslib.c

  ```
  static void (**entry_point) () = ENTRY_POINT;
  ```
- `entry_point` has address `0xf00`, and `*entry_point` is the address of our kernel entry point function
  - `*entry_point = kernel_entry;` (done in kernel.c)
- Macro for invoking syscall:

  ```
  #define SYSCALL(i) ((*entry_point)(i))
  ```

`kernel_entry()` in memory



The kernel sets the memory at address `0xf00` to the address of the `kernel_entry()` function. The user process can then read the memory at `0xf00` to learn where the `kernel_entry()` function is located in memory.

Synchronization

- Locks are only used by threads
- Many threads can try to acquire a lock
  - Need to maintain queue of threads waiting for a lock
    - Where?
- `lock_acquire()`
  - Check lock
  - Got lock? Great!
  - If not, block itself
- `lock_init(), lock_release()`

Measure Context Switch Time

- What to measure and how (methodology)
  - ⇒ Design review
- Do measurements
- Get results
- Report and discuss results in report

Extra Credit 1 – time

- Implement something similar to the Unix command “time”
  - Measure time spent in user mode and kernel mode
- Get an A+ grade
- ..or a TA job next year?

Extra Credit 2 and 3 – more threads and processes

- Add a new thread to your kernel
- Add a new process to your OS

Design review

- For each assignment you are required to give a design report for you plan to implement the solution
    - Note! describe it at the “design level”
    - Not a formal presentation - just you and your TA
- You should be prepared
    - Have something to show during the design review
    - Only oral presentation is not acceptable
    - You need to convince the TA that you understand the project
- The design review is a mandatory “assignment”:
    - Pass / no-pass grading

Possible topics for design review

- What’s in the PCB? Why is it there?
- How will you implement locks? What about process queues?
- Where should context be saved?
- What is the context (what must be saved?)
- What is the difference between processes and threads?
  - In this assignment?
  - How do “our” processes and threads differ from “normal” processes and threads?
- How many stacks per process? And why?
- How many stacks per thread? And why?
- Non-preemptive vs preemptive scheduling; What is similar? What is different?

Code

- Comment your code
  - Give a high-level overview of what the code does
  - Especially important when dealing with hardware, entry points, and synchronization/ blocking
  - Comments are part of the grading
- Test your code
- Don’t cheat

Report

- Maximum 4 pages
- Give an overview of how you solved each task
  - including extra credits
- Describe how you have tested your code
  - and known bugs/ issues
- Describe the methodology, results, and conclusions for your performance measurements

Handin

- This is an exam, so there is a strict deadline
  - Extensions only for valid and preapproved reasons
- Wiseflow for handin
  - PDF of your report
  - Zip file with source code (remember to make clean before ziping)
- Add your name, GitHub username, and email to the report

Hints - Flat Address Space

- The bootblock code:
  - Switches to protected mode
  - Sets up the CS, DS, SS, and extra segment registers
    - You can use the entire memory by utilizing registers like EAX
    - Do NOT modify the segment registers
  - You have access to the first 1MB of memory
    - Actually, you have access to a 32-bit address space, but we won’t use memory > 1MB limit in this assignment)
    - Including the video-memory area (VGA text buffer @ 0xb8000)
    - Make sure your code, data, and stacks do not overwrite “important” memory areas (see the memory map)

Hints - Threads and Processes

- Your OS will have processes and threads
  - Threads are linked with the kernel
  - Processes are separate executable files
  - Note: they all end up together in the same image file
- Processes need a special way to call functions in the kernel
- Threads can just call kernel functions directly
- In a more advanced OS, there are many other differences between threads in the kernel and processes

Hints – Synchronization

- The synchronization primitives can only be used by threads within the kernel
- The two functions block() and unblock() are to be used by the synchronization code
- Blocking a process is not specific to locks, but is a general purpose service that the kernel should support