OS LAB2: NON-PREEMPTIVE KERNEL

Miro Mannino

25 September 2022

LAB2 OVERVIEW

- Done with the Bootloader!
- Now we want to work on the actual kernel
- Add multiprogramming to the kernel to be able to run multiple programs at the same time
  - Non-preemptive scheduler
  - User space processes and kernel threads
  - Process Control Blocks
- Context switching Timing
- Mutual exclusion Lock

KERNEL THREADS AND USER SPACE PROCESSES

Implement context switch to switch between the process listed in `task[]` in `tasks.c`

- Kernel threads
  - `task1(clock_thread())` is defined in `th1.c`
  - `task2(thread2())` is defined `th2.c`
  - `task3(thread3())` is defined in `th2.c`
- User space processes
  - Task4-5 were defined `process[1-2].c`
  - Look at `process1.c` and try to understand it

Implement synchronization primitives

ASSUMPTIONS

- Protected Mode
  - No more segment registers: 32 bit memory
  - No more BIOS
- Non-Preemptive Tasks:
  - Run code until yield, block, or exit
- Fixed Number of Tasks:
  - Allocate per-task state (PCB) statically in your program at compile time in `kernel.c:_start()`
- Fixed Task Stack Size

NON-PREEMPTIVE

- `do_yield()` & `do_exit()` within the kernel (kernel threads can call these directly)
- `yield()` & `exit()` for processes (dispatches a desire to call `do_yield()` or `do_exit()` to the kernel)
    - User processes use library `syslib.h` for these
- Threads need to explicitly call `yield()` or `exit()`, in order to invoke the scheduler, otherwise a thread can run forever.
    .

YIELD

- When `yield` is called, the "context" of a task (thread or process) must be saved
- Process Control Block (PCB)
  - Process ID (PID)
  - Stack Info
  - Registers
  - CPU Time
  - Etc.
- Once the context is saved, the scheduler is run to pick a new task

PICKING A NEW TASK

- All tasks are waiting in a queue to be run
- Pick the next one from the front
- Restore its state from the PCB
- Return to where the task was executed before

MUTUAL EXCLUSION

Spinlock implementation is provided, you must implement a blocking lock

- `lock_init(lock_t * l)`
- `lock_acquire(lock_t * l)`
- `lock_release(lock_t * l)`

Questions?