Lab 4 Documentation

Bujar Tagani/btagani
Jonathan Lim/jlim2
Norman Wu/luow

Contents

Design:

Timer Driver

The design is an improvement from Lab 3. For Lab 4, we've decided to modify the OS timer configurations. We constantly update the value in OSMR while the OSCR continues to count up in order to reduce drift.

Time system call - unmodified from Lab 3.
Sleep system call - unmodified from Lab 3.
Read system call - unmodified from Lab 3.
Write system call - modified from Lab 3 to check that we can write to certain areas

Task_create system call

In task_create, we will decide if the given set of tasks are schedulable using UB testing. For now, we create an array of task_t pointers to make it easier to sort the tasks by their priority. The tasks are sorted by the rate-monotonic scheduling algorithm. Once we sort the tasks, we call **allocate_tasks()** which will create TCBs for the tasks. Once all the tasks are set to be runnable, we call **dispatch_nosave()** to run the highest priority task without needing to save any previous context.

Implementation

Changes to Lab 2 Code

S_Handler

We needed to change this function so that it would enable interrupts before making SWI calls. We found that we needed to do this because interrupts were disabled on all SWI calls and this would not allow our timer_inc function to run.

exit

Although we do not use exit call in this lab, we changed our exit sys call so that it works properly. We added code to restore the SVC registers back to their original state upon entering kernel. After restoring all registers, including lr, our exit now functioned properly and returns to the GUM prompt.

install_handler

We changed this function so it can take any vector table address as an argument and hijack the corresponding U-Boot handler with our custom handler. Previously, this function was exclusively a SWI installer function.

user_Setup

We updated this code so that it sets up an IRQ stack in kernel before entering the user program. We also added code to clear the cpsr in user mode and disable FIQ interrupts. Clearing the cpsr set the IRQ bit to 0 which will allow IRQ interrupts in user mode.

Kernel Functions

int install_handler(int vec_pos, int my_SWIaddr)

This function is used to access U-Boot vector table and install our own handlers for each vector position. In this lab we use this function to install our own IRQ and SWI handlers.

void timer_init(void)
This is our timer driver function which is called in kernel main before we execute the user program. The purpose of this function is to configure the OS timer registers. We used a resolution of 10ms per tick because we evaluated that this would have the best results.

void timer_inc(void)
This is the function that is called on all IRQ interrupts. When an IRQ is generated, our timer_inc function increments the global variable num_timer_tick which we use to hold the OS time.

void C_IRQ_Handler (void)
In this function, we call timer_inc to increment our OS timer. We chose to write timer_inc as a separate function instead of code contained in C_IRQ_Handler in case we decide to add additional functionality to the IRQ Handler in the future. This was an effort to produce cleaner code.

unsigned long time(void)
This is the time sys call (defined above). It reads the volatile global variable num_timer_tick and returns it in milliseconds. This functionality gives the current OS time.

void sleep(unsigned long ms)
This is the sleep sys call (defined above). It takes one argument, ms which the program will sleep for. The sleep sys call takes this number and adds it to the current OS time to construct the variable target time. It then enters a while loop which polls the OS time and stops when it reaches the target_time.

void sched_init(task_t* main_task)
This function initializes our scheduler with main_task serving as our launcher into the user program.

void dev_init(void)
This function initializes our devices.

void runqueue_init(void)
This function initializes our runqueue which will hold tasks that are ready to run.

Testing

Dagger

This test program creates two tasks, fun1 and fun2 and runs them with our task_create system call. Our scheduler decides when to run these functions and when they should be pre-empted.

Typo

```
while(TRUE){
    print '>'  //prompt
    start the timer
    read a line of characters
    echo characters and display time in seconds
}
```

The program takes a string as input and displays the amount of time it took for a user to type the line. The program tests the time system call.

Splat

The program displays a spinning cursor, using the sleep system call to slow down the animation.

Stopwatch

The program uses the time system call to record time instances. User can start the stopwatch and keep recording times. Once finished, user can print out all recorded times or restart the stop watch. This program terminates upon user's request.

Math Game

The program uses time seeds and our pseudo random number algorithm to generate pseudo random number. Using the pseudo random number, it generates math challenges with different levels (easy, medium, hard). Players will earn either one, two, or three points for each problem depending on the mode. The program also calculates the time players use to solve each problem and will offer one bonus point for correct solutions that were submitted in under five seconds. After each question is answered, the program sleeps for two seconds before moving on to the next question, so that the user can have time to rest and evaluate the result. A total score is then calculated and output for the user to view. The math game then restarts with the menu to choose modes or quit.

System Time (sys_time)

Trivial program that sleeps for a specified amount of time and reports how long the system has been running for.