Lab 4 Documentation

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

Design:

In this lab, we are no longer setting up the user stack and launching the user program this way. Instead we are creating a main task in kernel to launch the user program. We call the function sched_init which will set up the main task and then call dispatch_nosave to dispatch the the main task(user program).

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.

Scheduler

In this lab, we implemented a scheduler which uses rate monotonic policy. The scheduler determines which task to run next by looking through the run queue and seeing which is next to run. In the run queue, tasks are sorted by priority based on their Period, with the highest priority being tasks with the lowest Period.

Dispatcher

After all tasks are scheduled, the dispatcher finds the highest priority in the run queue and "dispatches" that task so that it will run. Our dispatcher consists of three functions dispatch_nosave, dispatch_save and dispatch_sleep which are all used for different purposes. This is described in full detail below. When we initialize the TCBs, we set the link register to point to launch_task(). We reached this design choice because we felt that this would allow us to do less comparisons, and not need to explicitly compare the context of a newly run task and an already-run task.

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.

In regards to error checking, we sort the tasks by stack pointers in order to detect duplicates. It is difficult to figure out how much stack space is needed per task function, so we just kernel panic when at least two stack pointers are the same. We also make sure that the user does not submit more tasks for scheduling than the OS can support.

Run queue

We've followed the designs of the lab, where we've used the run queue as a way to keep track of the runnable tasks. However, the current running task would not be found in the run queue, but instead would get put back in the run queue if it is pre-empted. If the task waits on a lock or a device, then it wouldn't be placed back into the run queue until it acquires the desired resources.

Mutex

A mutex must be created before it can be acquired by a user task. It is important to make the mutex lock and unlock functions as atomic, to avoid any race conditions.

Implementation

Changes to Lab 2 Code

S_Handler

From the previous version of S_Handler, we added code to store and load the user link register and stack pointers before and after executing a SWI. We also make sure to disable interrupts when loading the values so that context switching wouldn't occur during that time.

install_handler

We didn't change this function from lab 3.
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 got rid of user setup for this lab. We do not need to do this since we don't need to set up the user stack. We assume argc and argv will not be used by the user program. We also schedule the initial user program as a "main task" and launch it as a task.

Write system call
Modified from Lab 3 to check that we can write to certain areas

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. This function also calls *dev_update(),* so tasks waiting on devices can be made runnable again when the device is ready.

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.

Scheduler Functions

void allocate_tasks(void)

This function loops through each user task and allocates user stacks and initializes kernel contexts of each task.

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 the sleep queues and match values for all devices.

void dev_wait(unsigned int dev)

This function puts a task to sleep on the sleep queue until the next event is signalled for the device. The argument dev contains the device number. This function dispatches to the next runnable task with the highest priority with dispatch_sleep().

void dev_update(unsigned long millis)

This signals the occurrence of an event on all applicable devices. This function is called on timer interrupts to determine that the interrupt corresponds to the event frequency of a device. If the interrupt corresponded to the interrupt frequency of a device, this function should ensure that the task is made ready to run. It calls dispatch_save() to run the next task with the highest priority after updating.

void sleepqueue_wake(unsigned int dev)

This function removes all tasks waiting on dev’s sleep queue and adds them to the runqueue, ensuring that the tasks are made runnable.

void runqueue_init(void)

This function initializes our runqueue which will hold tasks that are ready to run.

Dispatcher Functions

For the assembly context switch functions, we use LDMIA and STMIA because we use C structs to save the context. C structs grow from smallest addresses to largest addresses, while the program stack grows from higher to lower addresses.

void dispatch_save(void)

This function context switches the current task to the highest priority task. It calls ctx_switch_full which saves the previous context before switching to the new context. If a task is running for the first time, it will call the function launch_task().

void dispatch_nosave(void)

This function context switches the current task to the next highest priority task. It calls ctx_switch_half which does not save the previous context before switching to the new task. If a task is running for the first time, it will call the function launch_task().

void dispatch_sleep(void)

This function context switches to the highest priority and saves the current task but does not mark it as runnable; it does not add it back to the runqueue. If a task is running for the first time, it will call the function launch_task().

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.

We also ran the given lab4grading tests, and make sure they all checked out.

To make sure that the tasks were properly sorted, we modified the order of the tasks given in simple_mutex.c and other user programs. We verified that the same output was given regardless of the order of the tasks in the task array.