Lab4, Part1: Real-Time Operating Systems

Group 2: Shi Su & Mengjin Yan

December 2, 2015

Content

1 Kernel Initiation

- Store the value of r8 (uboot function table) to global variable *global_data*
- Wire in IRQ handler
  - IRQ handler will handle interrupts in supervisor mode
- Wire in SWI handler
  - In SWI handler, addition to the implementation in lab3, we saved the sprs because IRQ is enabled in svc mode and is also handle in svc mode, sprs may get changed during SWI handling.
  - Also we store user mode sp and lr of each task on task's svc stack, for they will get changed by each running task.
- Initiate interrupt controller
- initiate Mutex
  Update global variable *gtMutex*, set all mutex as available
- Initiate Timer

2 Task Creation

After kernel initiated, Gravelv2 enters user mode main function, which may schedule a set of periodic user tasks through task_create syscall. Each task will be maintained in a kernel TCB, which contains it's priority, kernel context, kernel stack, etc.

3 Context Switch

There's only one task in running states on Gravelv2, so it needs to perform context switch when device event (timer interrupt or event_wait syscall) or mutex event (mutex_lock, mutex_unlock syscall) occurs.

When context switch occurs we need to save all callee saved registers of current task, and load the registers of next task, and because of we have a preemptive kernel, the kernel stack is put in TCB of each task.

Even if context switch may be initiated by interrupt or syscall, it only happens in svc mode, so for the 2 kind of event, the context switch implementation is the same and IRQ and SWI use same kernel stack.

When the first time a task launches, the context initiated during task creation make sure that Gravelv2 switches to user mode with sp set to task's user stack, and branches to task's lambda function with argument provided.

When a task is brought back to running state, its context is load from TCB so it can continue running from the point it was switch away in svc mode. And the user mode lr, sp stored in kernel stack make sure it can return to the proper place in user mode.

3.1 Queues

3 sets of queues(Run queue, Device Sleeping Queues, Mutex Sleeping Queues) are handled in the kernel during the whole program running process. Each task changes their status by moving from one queue to another.

- Run queue: Maintained in runqueue.c. Stores the address of all the TCBs in ready state. In order to fetch the highest priority task fast, a bit array (group_run_bits) and a bitmap (run_bits) is maintained to store the runnability of threads in a particular group and in particular priority respectively. When adding and removing TCBs in a run queue, the bit array and the bitmap are also needed to be updated.

- Device Sleep Queues: Maintained in device.c. Stores the address of all TCBs in waiting for event state, specifically, waiting for device update state.

- Mutex Sleep Queues: Maintained in mutex.c. Stores the address of all TCBs in waiting for event state, specifically, waiting for mutex unlock state.

3.2 Devices

Several virtual devices are maintained, each of which has its own frequency. Each of the devices also handled a sleeping queue, storing the address of the TCBs in waiting for device update state which is assign to the certain device. Functions device_wait and device_update handle moving TCBs from or to the sleeping queues when the tasks changing states.

- Device wait
  Called by function event_wait. Change the state of the task from running state to waiting for event state by moving the address of the task from running queue to the sleeping queue of the corresponding device, after a task complete processing for this cycle and waiting to be executed in the next cycle. Then call dispatch_sleep to do context switch to the highest priority task among all tasks in ready state(in running queue).

- Device update
  Called by the time driver. When IRQ interrupt generated every 10 ms, it is called to first check if the tasks in the corresponding device can be woken up. If it is true, change the task's state from waiting for event state to ready state by moving the address of the task from device sleeping queue to run queue. And if the woken up task has the highest priority among all tasks in both running and read state, dispatch_save is called to context switch from the current task to the newly woken up task.

3.3 Mutexes

Several mutexes are initialized during the kernel initialization process. Several mutexes are created when process different tasks. Also each mutex maintains a mutex sleeping queue which stores the address of the TCB which is waiting for on the corresponding mutex.

- Lock
  When mutex_lock is called and the mutex is not available, change the task from running state to waiting for event state by moving the address of the TCB from current running position to the sleeping queue of the corresponding mutex and call dispatch_sleep to context switch to the highest priority task among all tasks in ready state(running queue).

- Unlock
  When mutex_unlock is called and there are tasks waiting on the mutex sleeping queue, change the task from waiting for event state to ready state by moving the address of TCB in sleeping queue to running queue. And if the newly woken up task's priority is the highest in all tasks in both running and ready state, call dispatch_save to context switch to the newly woken up task.

4 New System Calls

4.1 task_create

int task_create(task_t* tasks, size_t num_tasks)

All currently present tasks are cancelled, and new set of tasks got scheduled.
In proc.c:

- Check the tasks are valid
- Call assign_schedule to assign priority to according to their period
- Call allocate_tasks to initiate TCB and scheduler

In sched.c:

- Clear the run queue
- Initiate all devices
- Set all mutex as unacquired, and clear the sleep queue of mutex
- Initiate the TCB for idle task, and add idle task to run queue
- Initiate the TCB according to the task lists, and add them to run queue
- Call dispatch_nosave to switch to highest priority task

4.2 mutex_create

int mutex_create(void);

scan the array of mutex, find the first available mutex and set the availability to false. Return the number of the mutex.

4.3 mutex_lock

int mutex_lock(int mutex);

- Assert that the mutex is available and not already hold by the task
- If the mutex is not been blocked, set the mutex to be blocked and update the task holds the mutex, as well as the holds_lock attribute in TCB
- If the mutex is been blocked, move the address of TCB from current running position to the mutex sleeping queue and then called dispatch_sleep to context switch to another ready task

4.4 mutex_unlock

int mutex_unlock(int mutex);

- Assert that the mutex is available and not already hold by the task
- If there is no element in sleep queue, set the mutex to be not blocked and update the task holds the mutex.
- If the sleep queue have tasks waiting for the mutex, move the head of the sleeping queue to run queue and if the newly woken task's priority is higher among tasks in both running and ready state tasks, call dispatch_save to context with to the newly woken task.

4.5 event_wait

int event_wait(unsigned int dev);

Call dev_wait to put current task to the sleep queue of specified devices.