Gravel2 – Task Scheduling and Concurrency

18-342: Fundamentals of Embedded Systems

Rajeev Gandhi
INI & ECE
Carnegie Mellon University

Carnegie Mellon

Electrical & Computer ENGINEERING

Lab 4 Requirements

- Build a real-time operating system called Gravelv2
- Part I
  - Port your libc, swi and irq installation from lab 3
  - Create, schedule tasks according to rate monotonic policy
  - Create, manage mutexes
  - Schedule tasks according to a fixed pre-assigned priority
- Part II
  - Utilization bound (UB) test – to be covered in "Advanced Real-Time" lecture
  - Highest locker priority protocol – to be covered in "Advanced Real-Time" lecture

Quick Look at Gravelv2

- Preemptive scheduling
- Maximum of 64 tasks
  - 63 “real” tasks, and one idle task
    - If you reserve the highest priority for special conditions (as in part 2), then 62 “real” tasks
  - Idle task executes when all other tasks are waiting on events
  - Idle task has the lowest task priority
  - Tasks are periodic (hence they never finish)

- Graavelv2 priorities
  - Higher numbers ➪ lower priorities
  - Highest-priority task has priority 0
  - Two tasks should not have the same priority – you need to make sure of this

Tasks

- A task in Gravelv2 is typically an infinite loop function

```
void MyTask(void *pdata)
{
      for(;;) {
      /*   some application code */
      /*  call one of Gravelv2’ services: */

              mutex_lock(…);

       /*   some application code */
      /*  call one of Gravelv2’ services: */

           mutex_unlock(…);

      /* more application code and Gravelv2 calls */
      event_wait(dev_num);
      }
}
```

Task Creation

- Application may create tasks as follows

```
   int main(int argc, char** argv)
{
   task_t tasks[2];
   tasks[0].lambda = fun1;
   tasks[0].data = (void*)'@';
   tasks[0].stack_pos = (void*)0xa2000000;
   tasks[0].C = 1;
   tasks[0].T = PERIOD_DEV0;

   tasks[1].lambda = fun2;
   tasks[1].data = (void*)'<';
   tasks[1].stack_pos = (void*)0xa1000000;
   tasks[1].C = 1;
   tasks[1].T = PERIOD_DEV1;

   task_create(tasks, 2);

   puts("Elvis could not leave the bu
   return 0;
}
```

```
/* fun2 is defined as a function in the same file as
   main
*/
void fun2(void* str)
{
          while(1)
          {
                    putchar((int)str);
                    if (event_wait(1) < 0)
                              panic("Dev 1 failed");
          }
}
```

Task Creation

- Since tasks are periodic, they should never end
  - Hence no need for the `exit` syscall
- Once `task_create` function is called, you can assume that all the old task cease to exist (scheduler does not care about them anymore)
  - `task_create` function should never return
- Inside the `task_create` syscall, your code will look at the `task_t` data structure to learn everything needed about the tasks
- **Warning: Don't assume that user code will be sane**
  - **Your kernel should deal with unexpected input not conforming to specifications**

What does `dev` data structure do in `device.c`

- Problem: How do we create periodic tasks? What do we do with a task after it has finished one of its instances?
  - Suspend the task until the next time period
  - Place the suspended task (it's TCB) in a sleep_queue
  - Create events to occur periodically (with periodicity of the task)
  - On each event, check which tasks should be re-instantiated

- `dev` data structure does precisely that

```
struct dev
{
  tcb_t* sleep_queue;
  unsigned long   next_match;
};
```

- `dev_wait` is called by tasks (through `event_wait`) once they have finished execution of their current period and want to suspend themselves until the next period
-

What does `dev` data structure do in `device.c`

- `dev_update` should be called by your timer interrupt handler code
  - `dev update` should check whether the next event for every device has occurred
  - If the next event has occurred
    - Wake up all the tasks on this device's sleep_queue
    - Make these tasks ready to run

SWI Handling

- You will port your SWI Handling code from Lab 3 to this lab
- Need to add support for additional SWIs as documented in Gravelv2 API
  - Need to remove `exit` syscall as well
- Difference from previous lab
  - Interrupts (IRQs) should be enabled when user programs make syscalls
  - Your kernel should be preemptible
- Modify your top level SWI handler to
  - Enable interrupts (at a safe point) before executing the actual syscall code
  - Disable interrupts (if needed, to prevent concurrency issues)

IRQ Handling

- Since IRQs can occur while kernel is running, the IRQ handler is modified to run in SWI mode
  - After an IRQ occurs and context (not necessarily user context) has been saved, we switch to SVC mode
- The top-level interrupt handler has been provided to you (called `irq_wrapper` in file `int_asm.S`)
- You should understand what is going on in `irq_wrapper` and then modify your swi handler (top level) if needed
- Even though interrupts are handled in SWI mode, does not mean we allow nested interrupts

Task Scheduling

- Each task in Gravelv2 is maintained using a Task Control Block (TCB)
  - Since the number of tasks in Gravelv2 is fixed, TCBs for all tasks are statically allocated as an array (`system_tcb`) in `sched.c`
  - The `TCB` structure is declared in `tasks.h`

```
struct tcb
{
  uint8_t          native_prio;
  uint8_t          cur_prio;
  sched_context_t  context;
  int              holds_lock;
  volatile struct tcb* sleep_queue;
  /** Embed the kernel stack here -- AAPCS wants 8 byte alignment
  */
  uint32_t         kstack[OS_KSTACK_SIZE/sizeof(uint32_t)]
                       __attribute__((aligned(8)));
  uint32_t         kstack_high[0];
};
typedef volatile struct tcb tcb_t;
```

Task Scheduling

- When `task_create` is executed, you must execute a function (`allocate_tasks in sched.c`)
  - In this function, you will
    - Set up the TCBs for all the tasks
    - Put the task in `runqueue` (make it available for running)
    - Set up the TCB for idle task
    - Make the idle task schedulable

- After all TCBs have been set up, you must context switch to the highest priority task that has been setup
  - This should occur in a function called `dispatch_nosave` in (`ctx_switch.c`)
  - All the dispatch functions are used for context switching in specific scenarios
    - `dispatch_save`: save the context of current task and context switch to highest priority task
    - `dispatch_nosave`: context switch to the highest priority task without worrying about saving current task's context
    - `dispatch_sleep`: save current task's context, make the current task not runnable and context switch

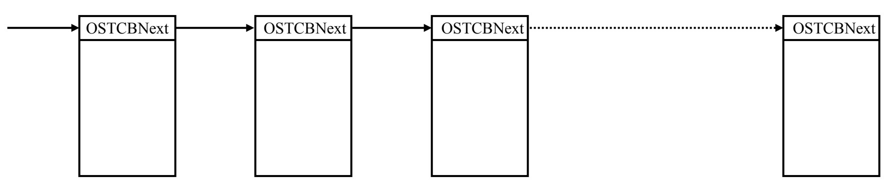
Context Switching

- You will need to write code for context switching
- These functions are to be written in `ctx_switch_asm.S`
  - `ctx_switch_full` will get called from `dispatch_save` and `dispatch_sleep`
  - `ctx_switch_half` will get called from `dispatch_nosave`

- An assembly function `launch_task` has been provided to you
  - Understand this function
  - You will need to call this function the first time a task is launched
    - Don't define global/static variables called first or something similar
    - Once you understand what is going on in the code, it might help you with determining how context of a task will be initialized (before a task is launched for the first time)

Run queues

- Gravelv2 maintains the list of currently runnable tasks on run queues
- Typically there will be a run queue consisting of tasks that are ready to run
  - On a context switch, the scheduler can go through this run queue and look for the highest priority task that is ready to run
- Maintain run queue as a linked list of TCBs of all tasks in your system



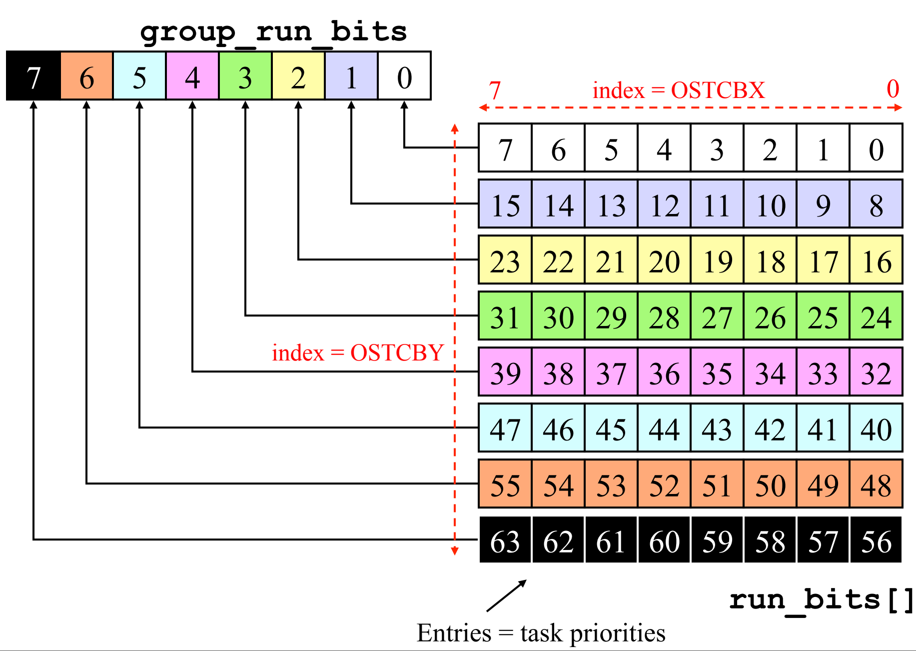
- How do you find the highest priority task in this list?
- Disadvantages?

Runqueue in Gravelv2

Where is the runqueue maintained in Gravelv2?

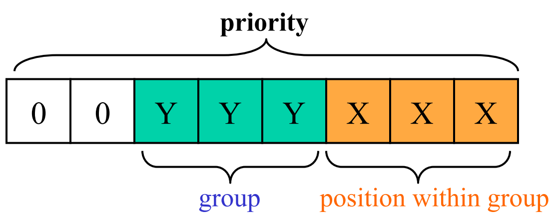
Using the array `run_bits[8]` and variable `group_run_bits` in `run_queue.c`

Ready Queue



Task Groups

- Gravelv2 divides its 64 priority levels into 8 groups
  - How do you determine a task's group?
  - **priority >> 3**
  - This value is stored in the **OSTCBY** variable
    - Task priorities 0, 1, 2, ….., 7 ⇨ task group = 0
    - Task priorities 8, 9, 10, ….., 15 ⇨ task group = 1
  - i.e., **OSTCBY should** contain the 3 MSBs of the task's priority number
- What is the task's position within the group?
  - **priority & 0x07**
  - This value is stored in the **OSTCBX** field of the task's OS_TCB
  - **OSTCBX** contains the 3 LSBs of the task's priority number



priority =
OSTCBY << 3
+ OSTCBX

Ready List

- Gravelv2 maintains two variables `group_run_bits` and `run_list[]` which are used to store the list of tasks ready to run

- When a task is created or is made ready to run, the appropriate bits of `group_run_bits` and `run_list[]` are set to indicate to the scheduler that the task is ready to run

- Example: Assume `group_run_bits` and `run_list[8]={0,0,…0}` and a task of priority `prio` = 17 is created, in order to let the scheduler know the presence of the task at `prio = 17`, `TaskCreate` must
  - Compute priority group (`OSTCBY` field) of the task

    `OSTCBY = (prio >> 3)    = 2`
  - Compute task's position in the priority group (`OSTCBX` field)

    `OSTCBX = (prio & 0x07) = 1`
  - Set bit number `OSTCBY` of `group_run_bits` to 1
  - Set bit number `OSTCBX` of `run_bits[OSTCBY]` equal to `1`

Example

- What will be the values of `group_run_bits` and `run_bits[]` when 3 tasks of priorities 16, 17 and 25 are created?
- Assume that initially group_run_bits = 0, and run_bits = {0, 0, 0, 0, 0, 0, 0, 0}

Finding the Highest-Priority Ready Task

- Suppose that you want to find the highest-priority task (let's denote its priority by `prio`) that is ready to run
- Can be done quickly (without scanning the entire linked list of created tasks) through `group_bits` and `run_bits[]`
  - Algorithm:
    - Find the least significant bit set in `group_bits` (call it `y`)
    - Find the least significant bit set in `run_bits[y]` (say `x`)
    - Highest priority task ready to run is `(y<<3)+x`
- Optimization – Since you will be doing this often, to make this process faster, there is a special table called `prio_unmap_table[]` with 256 entries that helps you do the above

```
y   = prio_unmap_table[group_bits];
x   = prio_unmap_table[run_bits[y]]
prio = (y << 3) + x
```

- You are likely to need this kind of code in `run_queue()`

Finding the Highest-Priority Ready Task (contd.)

- Assume `group_bits=12` and `run_bits={0,0,3,2,0,0,0,0}`
- How do you determine the highest priority task to run

```
y  =  UnMapTbl[group_bits]
x  =
UnMapTbl[run_bits[y]]
prio = (y << 3) + x

Once prio is known, you
can get
to the TCB of this task by
accessing system_tcb[prio]
```

```
UnMapTbl[] = {
    7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0
};
```

- Question: Do you need to do anything to `group_run_bits` and `run_bits[]` when suspending a task?

Summary

- Lab 4 overall summary
- Please follow course mailing list for more details
  - You may have several questions
  - Please use mailing list rather than sending individual requests
  - You may be helping others with similar problem
- Aim to finish task management first – you should try and use what we've covered today to do this as soon as possible
  - Lab is even more challenging than Lab 3
  - Very useful to understand embedded OS concepts if you have never done a course on OS