1/9/2018 – Exam of Real-Time Operating Systems

ACME "boredparents" is producing a new smart toy for 2-year-old children, named "Xplore." The toy is basically a colored plastic cube embedded with different kinds of sensors:

a) One accelerometer
b) Touch sensors on the six faces of the cube
c) One light sensor

Additionally, the system is equipped with

d) a speaker that can produce different 256 different sounds when the child plays with the toy.

The a) and b) collect data at a frequency of 20Hz, whereas c) collects data at 50Hz. Sounds are produced with a maximum frequency of 1 sound per second.

All data are collected by an embedded PC within the toy, running Linux. The PC includes a wi-fi network for monitoring that allows the PC to send messages to parents in a nearby room.

Figure 1. ACME Xplore^{TM} system

Technical specifications

The PC has a control board connected to sensors (a), (b), (c), and to the speaker (d).

To read the most recent values from sensors a), b), and c), it is necessary to read from 8-bit registers at the address (a) 0x260, (b) 0x261, and (c) 0x262, respectively.

To play a sound with (d), it is necessary to write into an 8-bit register at the address 0x360 (there are 256 sounds possible, each corresponding to a different byte).

Also, notice that, by opportunely programming an 8-bit control register at the address 0x361 (it is necessary to write the value 0x10), the control board is enabled to launch an interrupt on IRQ9. The interrupt is triggered after the speaker has received a byte, played the sound, and is ready to receive a new byte (remember that it is not allowed to play more than one sound per second). Indeed, if a byte is written on the register before the speaker has played the sound and the interrupt has been triggered, the second byte

(and sound) is lost: for this reason, an interrupt-based mechanism can be crucial for writing bytes/playing sounds.

The onboard PC is required to perform the following tasks:

- J1 acquires and pre-processes data from the accelerometer;
- J2 acquires and pre-processes data from the touch sensors;
- J3 acquires and pre-processes data from the light sensor;
- J4 merges all data received from sensors and selects the next sound to be produced according to rules that are not reported here; also, J4 detects anomalous sensor patterns that can be caused by an emergency situation;
- J5 sends the most appropriate sound to be produced to the speaker;
- J6, if and only if an emergency situation is detected, calls the parents by sending an alarm through the wi-fi network.

As a consequence of statistical analysis, the computational times required for the execution of tasks J1…J6 turns out to be, in the worst case, **C1 = 3ms; C2 = 2ms; C3 = 3ms; C4 = 12ms; C5 = 3 ms; C6=6ms.**

Exercise 1 (max score 9): **a - max score 1)** tell which tasks are to be considered periodic, which sporadic, and which aperiodic. **b - max score 3)** propose a period of periodic tasks and manage opportunely sporadic tasks in order to meet i) the conditions of the general schedulability theorem and ii) sufficient conditions for RM. iii) Assign the appropriate priorities to tasks. **c - max score 5)** After an analysis of requirements, it turns out that we must add one more aperiodic task J7, with a maximum duration of C7 = 2 ms. Assume to schedule the aperiodic tasks using a Polling Server with a FIFO policy (whose characteristics must be defined) and compute the maximum delay in execution (termination time – arrival time) of C7 in the worst case by using necessary and sufficient conditions.

Exercise 2 (max score 6): After the first set of tests, it turns out that more complex algorithms are required to pre-process data and merge them to produce sounds. Specifically, it turns out that – as a consequence of the new rules – a set of four periodic tasks is required with the following characteristics: J1: C1=4; T1=50; J2: C2=4; T2=50; J3: C3=4; T3=20; J4: C4=12; T4=20. Check if the tasks can be scheduled using the Audsley theorem (or algorithm).

Exercise 3 (max score 4): describe, in a bullet list, the basic ideas to implement the Rate Monotonic algorithm for exchanging messages on a Can Bus, by outlining i) what C and T mean in the case of RM scheduling on a Can Bus and ii) similarities and differences with the RM used for scheduling tasks on a processor.

Exercise 4 (max score 7): for the tasks J1 to J7, we use Posix threads, and therefore we decide to design a char driver. **a (max score 3)** Write a schematic diagram of the code contained in a "minimal" char driver, implemented as a module loadable in runtime, assuming the existence of a device file "/ dev / xplore_sens" for reading sensor data and a device file "/ dev / xplore_speak" for controlling the speaker. The driver should provide system calls for reading and writing and the transition from blocking to non-blocking state (and vice versa). Report only the prototypes of each function and comment on their use, clarifying, for each function, which event (shell commands or system calls) determines its execution. **b (max score 4)** Clarify WHY it is necessary to associate the special device files with two different sets of file operations and show WHERE and HOW this should be done. Show how the driver can read data from sensors and transfer them from kernel space to user space.

Exercise 5 (10 points): To perform write operations on the control board required by J5 to play the sound, we decided to implement an interrupt-based mechanism. The idea is that once programmed, the control board triggers an interrupt whenever the user has sent a command for playing the sound, and the sound has been successfully played. **a - 2 points)** write the code in the proper place to "connect" and "disconnect" the IRQ handler and program the board to run it. Declare, if necessary, the variables used. **c - b points)** Write, in the most appropriate system call, i) the code to transfer sound commands from the task J5 to the driver; ii) the code to send sound commands to the control board, iii) the code to make the system call above blocking. Specifically, once the sound command has been sent to the control board, the system call must block until the sound has been completely played. The system call can return only once the operations to play the sound have been successfully achieved (approximately 1 second after the command has been sent) **c – 3 points)** Suppose that the operations performed by the interrupt are computationally very demanding. Show how it is possible to divide the interrupt handler into a top half and a bottom half.

$$\forall i = 1,..,n \quad deve \quad valere \quad \sum_{j=1}^{i} c_j(t) \le \Omega(t, d_i)$$

$$\forall i, 1 \le i \le n, \quad \exists k : \left(C_i + I_i^k \le t_k\right) and \left(t_k \le D_k\right)$$

$$dove \quad I_0 = \sum_{j=1}^{i-1} C_j \qquad t_k = I_i^{k-1} + C_i \qquad I_i^k = \sum_{j=1}^{i-1} ceiling\left(\frac{t_k}{T_j}\right) C_j$$

$$\forall i = 1,..,n \quad deve \quad valere \quad t + ceiling\left(\frac{S_i}{\Phi}\right) H \le d_i$$

$$dove \quad \Phi = (1-U)H \quad Si = \sum_{j=1}^{i} c_j(t)$$

$$\forall i = 1,..,n \quad deve \quad valere \quad \sum_{j=1}^{i} c_j(t) \le \Omega(t, d_i)$$

r_a istante di arrivo, s_a istante di inizio

f_a istante di completamento

$$F_a = \begin{matrix} (C_a / C_s) - 1 & se \quad C_a \quad multiplo\_di \quad C_s \\ floor(C_a / C_s) & altrimenti \end{matrix},$$

$$G_a = floor(r_a / T_s)$$

$$R_a = (C_a - F_a C_s), \quad s_a = (G_a + 1) T_s$$

$$f_a = (G_a + 1) T_s + F_a T_s + R_a$$

$$U_{lub} = n\left[\left(\frac{U_s + 2}{2U_s + 1}\right)^{1/n} - 1\right]$$

$$\lim_{n \to \infty} U_{lub} = \ln 2 - \ln\left(\frac{4U_s + 2}{U_s + 2}\right)$$

$$2\left(2^{1/2} - 1\right) = 0.828$$

$$3\left(2^{1/3} - 1\right) = 0.779$$

$$4\left(2^{1/4} - 1\right) = 0.756$$

$$5\left(2^{1/5} - 1\right) = 0.743$$

$$6\left(2^{1/6} - 1\right) = 0.734$$

$$U_{tot} = \sum_i \frac{C_i + \delta_i}{T_i}$$

$$\forall i, 1 \le i \le n, \quad \sum_{k=1}^{i} \frac{C_k}{T_k} + \frac{B_i}{T_i} \le i(2^{1/i} - 1)$$

$$\forall h = 1,..,m \quad f_h \le d_h$$

$$f_h = \begin{cases} t + S_h & se \quad S_h \le c_s(t) \\ (F_h + G + 1) T_s + R_h & altrimenti \end{cases}$$

$$G = floor(t / T_s) \quad F_h = floor[(S_h - \Delta_h) / C_s] \quad \Delta_h = c_s(t)$$

$$Sh = \sum_{i=1}^{h} c_i(t) \qquad R_h = (S_h - \Delta_h - F_h C_s) \qquad T_s \le \min_i(T_i)$$

cycles_t get_cycles(void);
int rt_free_global_irq (unsigned int irq);
int rt_request_global_irq (unsigned int irq, void(*handler)(void));
int rt_sem_wait_timed (SEM *sem, RTIME delay)int rt_sem_wait_until (SEM *sem, RTIME time)
int (*fsync) (struct file *file, struct dentry *dentry, int datasync);
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
int (*ioctl) (struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);
int (*mmap) (struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*release) (struct inode *, struct file *);
int access_ok(int type, const void *addr, unsigned long size);
int check_mem_region(unsigned long start, unsigned long len);
int close(int fd);
int gettimeofday(struct timeval *tv, struct timezone *tz);
int ioperm(unsigned long *from*, unsigned long *num*, int *turn_on*);
int iopl(int *level*);
int nanosleep(const struct timespec **req*, struct timespec **rem*);
int open(const char *pathname, int flags);
int printk(const char*fmt, ...)
int probe_irq_off(unsigned long);
int pthread_attr_init(pthread_attr_t *attr)
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
int pthread_attr_setinheritsched(pthread_attr_t *attr, int inherit);
int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_cancel_(pthread t THREAD) ;
int pthread_cond wait (pthread cond t *COND, pthread mutex t *MUTEX) ;
int pthread_cond_broadcast (pthread cond t *COND) ;
int pthread_cond_destroy (pthread cond t *COND) ;
int pthread_cond_init (pthread cond t *COND, pthread condattr t *cond ATTR) ;
int pthread_cond_signal (pthread cond t *COND) ;
int pthread_cond_timedwait (pthread cond t *COND, pthread mutex t *MUTEX, const struct timespec *ABSTIME) ;
int pthread_create(pthread_t * thread, const pthread_attr_t * attr, void * (*start_routine)(void *), void *arg);
int pthread_detach(pthread_t th);
int pthread_join (pthread t TH, void **thread RETURN) ;
int pthread_mutex_destroy (pthread mutex t *MUTEX);
int pthread_mutex_init (pthread mutex t *MUTEX, const pthread mutexattr t *MUTEXATTR) ;
int pthread_mutex_lock (pthread mutex t *mutex)) ;
int pthread_mutex_timedlock (pthread mutex t *MUTEX, const struct timespec *ABSTIME) ;
int pthread_mutex_trylock (pthread mutex t *MUTEX) ;
int pthread_mutex_unlock (pthread mutex t *MUTEX);
int pthread_setschedparam(pthread_t target_thread, int policy, const struct sched_param *param);
int register_chrdev(unsigned int major, const char *name, struct file_operations *fops);
int request_irq(unsigned int irq, void (*gest)(int, void *, struct pt_regs *), unsigned long flags, const char *devname, void *dev_id);
int rt_get_task_state (RT_TASK **task*);
int rt_sem_delete (SEM *sem)
int rt_sem_signal (SEM *sem)
int rt_sem_wait (SEM *sem)
int rt_task_delete (RT_TASK **task*);
int rt_task_make_periodic(RT_TASK *task, RTIME start_time, RTIME period);
int rt_task_make_periodic_relative_ns (RT_TASK **task*, RTIME *start_delay*, RTIME *period*);
int rt_task_resume (RT_TASK **task*);
int rt_task_suspend (RT_TASK **task*);
int rt_task_use_fpu (RT_TASK* *task*, int *use_fpu_flag*);
int rtf_create (unsigned int *fifo*, int *size*);
int rtf_create_handler (unsigned int *fifo*, int (**handler*)(unsigned int *fifo*));

int rtf_destroy (unsigned int *fifo*);
int rtf_get (unsigned int *fifo*, void **buf*, int *count*);
int rtf_put (unsigned int *fifo*, void **buf*, int *count*);
int rtf_reset (unsigned int *fifo*);
int settimeofday(const struct timeval *tv , const struct timezone *tz);
int setuid(uid_t uid);
int unregister_chrdev(unsigned int major, const char *name);
int wait_event_interruptible(wait_queue_head_t queue, int condition);
loff_t (*llseek) (struct file *, loff_t, int);
MAJOR(kdev_t dev);
MINOR(kdev_t dev);
MOD_DEC_USE_COUNT (decrementa)
MOD_IN_USE restituisce true se il contatore non è uguale a 0
MOD_INC_USE_COUNT (incrementa)
PTHREAD ADAPTIVE MUTEX INITIALIZER NP,
PTHREAD ERRORCHECK MUTEX INITIALIZER NP
PTHREAD MUTEX INITIALIZER,
PTHREAD RECURSIVE MUTEX INITIALIZER NP,
rdtsc(low,high);
rdtscl(low);
RT_TASK *rt_whoami (void);
rt_task_init(RT_TASK *task, void *rt_thread, int data, int stack_size, int priority, int uses_fp, void *sig_handler);
RTIME nano2counts(int nanoseconds);
RTIME rt_get_cpu_time_ns (void);
RTIME rt_get_time (void);
RTIME rt_get_time_ns (void);
RTIME start_rt_timer(RTIME period);
SEM sem;
ssize_t (*read) (struct file *, char *, size_t, loff_t *);
ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
ssize_t read(int fd, void *buf, size_t count);
ssize_t read(int fd, void *buf, size_t count);
unsigned rt_startup_irq (unsigned irq);
unsigned inb(unsigned port);
unsigned inl(unsigned port);
unsigned int (*poll) (struct file *, poll_table *);
unsigned inw(unsigned port);
unsigned long copy_from_user(void *to, const void *from, unsigned long count);
unsigned long copy_to_user(void *to, const void *from, unsigned long count);
unsigned long probe_irq_on(void);
unsigned readb(address);
unsigned readl(address);
unsigned readw(address);
void rt_disable_irq (unsigned irq);
void rt_enable_irq (unsigned irq);
void rt_shutdown_irq (unsigned irq);
void, rt_spv_RMS (int cpuid)
void rt_task_set_resume_end_times (RTIME resume_time, RTIME end_time)
void (*handler)(int, void *, struct pt_regs *)
void *ioremap(unsigned long phys_addr, unsigned long size);
void *iounmap (void *addr)
void *kmalloc (size_t size, int prio);
void *rtai_kmalloc(unsigned long name, int size); void *rtai_malloc (unsigned long name, int size);
void barrier(void)
void disable_irq(int irq);
void disable_irq_nosync(int irq);
void do_gettimeofday(struct timeval *tv);
void enable_irq(int irq);
void init_waitqueue_head (wait_queue_head_t *queue);

void interruptible_sleep_on(wait_queue_head_t *queue);
void interruptible_sleep_on_timeout(wait_queue_head_t *queue, long timeout);
void kfree (const void *ptr);
void mb(void);
void mdelay(unsigned long msecs);
void memcpy_fromio(unsigned long, unsigned long, unsigned int count);
void memcpy_toio(unsigned long, unsigned long, unsigned int count);
void memset_io(unsigned long, char fill, int count)
void outb(unsigned char byte, unsigned port);
void outl(unsigned longword, unsigned port);
void outw(unsigned short word, unsigned port);
void pthread cancel (pthread t THREAD) ;
void pthread exit (void *RETVAL) ;
void pthread_exit(void *retval);
void release_mem_region (unsigned long start, unsigned long len);
void request_mem_region(unsigned long start, unsigned long len, char * name);
void rmb(void);
void rt_busy_sleep (int *nanosecs*);
void rt_linux_use_fpu (int *use_fpu_flag*);
void rt_sem_init (SEM *sem, int value)
void rt_set_periodic_mode(void);
void rt_sleep (RTIME *delay*);
void rt_sleep_until (RTIME *time*);
void rt_task_signal_handler (RT_TASK **task*, void (**handler*)(void));
void rt_task_wait_period(void);
void rt_task_yield (void);
void rtai_free (int name, void *adr)
void rtai_kfree (int name)
void sleep_on(wait_queue_head_t *queue);
void sleep_on_timeout(wait_queue_head_t *queue, long timeout);
void spin_lock(spinlock_t *lock);
void spin_lock_init(spinlock_t *lock);
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
void spin_unlock(spinlock_t *lock);
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
void udelay(unsigned long usecs);
void wait_event(wait_queue_head_t queue, int condition);
void wake_up(wait_queue_head_t *queue);
void wake_up_interruptible(wait_queue_head_t *queue);
void wake_up_interruptible_sync(wait_queue_head_t *queue);
void wake_up_sync(wait_queue_head_t *queue);
void wmb(void);
void writeb(unsigned value, address);
void writel(unsigned value, address);
void writew(unsigned value, address);
void free_irq(unsigned int irq, void *dev_id);