CG2271 Real Time Operating Systems

Tutorial 5

Question 1

In a co-operative multitasking operating system, tasks surrender control to other tasks on their own accord. I.e. they are never pre-empted. How is such an OS similar to function queue scheduling, and how is different? List down and discuss as many points as possible.

Similar:

- **If several tasks are ready at the same time, the highest priority one is picked for execution.**
- **Control is passed to the next task (even if a "higher priority" task) only when the current task surrenders control.**

Different:

- **An RTOS would provide many more services than just task management, like coordination services (semaphores, barriers, etc), communication services (queues, mailboxes), memory management and interrupt management.**
- **In an FQS system the programmer would have to implement all of these services himself.**

Question 2

C supports a special data type called "void *", and this is commonly used in RTOS. For example in uC/OS-II and FreeRTOS, tasks are usually declared as:

```
void task1(void *param)
{
     !! task1 body
}

void task2(void *param)
{
     !! task2 body
}
```

a. A variable that is declared of type int * (e.g. int *ptr) is essentially a pointer to an integer variable. By the same logic, a variable that is of type void * (e.g. void *ptr) is a pointer to void variable. However there is no such thing as a void variable in C. So what is a void * variable?

   A void * (read as void star or void pointer) is used to create parameters with a "neutral" data-type. I.e. to create function parameters whose types you don't know in advance.

b. Discuss why void * variables are useful.

As stated in part a, this is very useful when you need to pass in an argument whose type is not known ahead of time. In an RTOS context "param" might be an initial parameter we want to pass to a task (e.g. the task's own task number). We just need to cast the argument into a void * when passing to the function, and then re-casting to the correct type within the function. For example:

```
void task1(void *param)
{
        printf("I am task number %d\n", (int) param);
}

void task2(void *param)
{
        printf("The character passed in was %c\n", (char) param);
}

int main()
{       …
        // OSCreateTask takes in the following parameters:
        // Pointer to task function, argument to task function, priority level

        int i;

        // Creates 5 tasks using the task1 function, passing in i as argument.
        for(i=0; i<5; i++)
                OSCreateTask(task1, (void *) i, i);

        // Creates 1 task using task2, passing in a character 'c'
        OSCreateTask(task2, (void *) 'c', 6);
}
```

c. What are the pitfalls of using void * variables?

You must ensure that "both ends" interpret the variable correctly. For example if we had a queue that queued void * variables, you can easily make this mistake.

```
int r=255;

…
// Enqueue r, which is a pointer to integer variable r. enq takes two arguments: the first
// is the queue to write to, and the second is the item to queue, cast into a (void *) first.
enq(queue1, (void *) r);

…

// Dequeue an item from queue1

int *s = (int *) deq(queue1);

*s=123;
```

The code above is likely to fail because on one end we have queued an integer variable, and on the other end we have de-queued the same integer variable but mis-interpreted it as a pointer.

Question 3

In C, a pointer to a function with prototype *void fun(int *)* is defined as:

```
int (*fpointer)(int);
```

So you can for example do this.

```
#include <stdio.h>

int f(int x)
{
	return 2*x;
}

int main()
{
	int (*fptr)(int);

	fptr=f; // Assign function f to pointer fptr

	printf("%d\n", fptr(3)); // Exactly the same as calling
	                         // function f  itself.
	return 0;
}
```

a. Discuss, within the context of operating systems, the different ways that function pointers are useful.

- **In task creation function pointers allow us to pass code over to the operating system to turn the function into a task.**
- **Function pointers are useful as "call-backs", which are functions that are written by a user but called by the operating system when an event occurs, e.g. when an interrupt is triggered:**

  void (*isr1_callback)(void);

  ISR(ISR1_vect)
  {
  // Just call the call-back routine.
  isr1_callback();
  }

  This allows a programmer to specify the behaviour of the ISR without actually having to write the ISR himself.

b. Using what you've just learnt about function pointers, implement a priority queue. Use the following structure:

```
typedef struct pq
{
      void (*fp)(void *);
      int priority;
      struct pq *next, *prev;
} TFuncQ;
```

Your priority queue has the following functions to queue and dequeue functions.

```
void enq(void (*fp)(void *), int priority);
TFuncQ  *deq(); // Returns the structure at the head of the queue.
```

You can assume that priority=0 is for highest priority functions and priority=255 is for lowest.

```
TFuncQ *q=NULL;
…

// Assume smaller priority number = higher priority task
void enq(void (*fp)(void *), int priority)
{
      TFuncQ *node = malloc(sizeof(TFuncQ));
      node->fp=fp;
      node->priority=priority;
      node->prev=node->next=NULL;

      if(!q)
            q=node;

      TFuncQ *trav=q;

      // Traverse to correct insertion point
      while(trav->next != NULL && trav->priority < priority)
            trav = trav->next;
```

```
// Check if trav->priority < priority. Will happen if we
      // are currently inserting a task with the lowest priority.

      if(trav->priority < priority)
      {
            trav->next=node;
            node->prev=trav;
      }
      else
      {
            // if trav->priority >= priority, we insert ourselves
            // before trav
            node->next=trav;
            trav->prev=node;

            // We might be at the head of the queue
            if(q==trav)
                  q=node;
      }
}

TFuncQ *deq()
{
      // Head item has highest priority. We return that.
      TFuncQ *tmp=q;

      // Delete the head item
      if(q)
            q=q->next;

      return tmp;
}
```

c. Suppose that there are three handler functions adc_func, timer0_func and timer1_func defined as:

```
void adc_func(void *ptr)
{
      !! Handle ADC stuff
}

void timer0_func(void *ptr)
{
      !! Handle timer 0 stuff
}
void timer1_func(void *ptr)
{
      !! Handle timer 1 stuff
}
```

Write the ISRs for the ADC, Timer 0 and Timer1 compare interrupts that insert the respective handler functions into the priority queue. You only need to show the ISR code, you do not need to show any initialization/startup code for the ADC or the timers. ADC has highest priority, Timer 0 next highest, and Timer 1 the lowest priority.

```
ISR(ADC_vect)
{
      enq(adc_func, 0);
}

ISR(TIMER0_COMPA_vect)
{
      enq(timer0_func, 1);
}

ISR(TIMER1_COMPA_vect)
{
      enq(timer1, 2);
}
```

d. Continuing on with c., implement a full function-queue scheduling system, except for the code to initialize/startup the timers and ADC.

… codes from part c…

```
int main()
{
      // In this question the parameters to the handler functions
      // is not used so we just pass in a NULL.
      // The handlers in an FQS system are examples of call-back
      // functions.
      while(1)
      {
            q=deq();

            // Check to make sure q is not NULL, which happens if
            // the queue is empty.
            if(q)
                  q->fp(NULL);
      }
}
```

…