CS 452 Final Exam

Colby Blair

Due April May 11th, 2012

Grade: _____

1

a)

The C language was not designed for the Harvard architecture, with its seperate program (flash) and data (RAM) memory. It is challenging to get some things like constant data in program space, so it can be used efficiently, using C. Asking for more data (malloc) and accessing address space requires the compiler to do a lot more work, and if the compiler doesn't make the efficient decisions, then the programmer may want to step in (i.e. using the PROGMEM macro).

b

If I wrote a language for embedded systems, I would write a language that was strictly data and resource driven. It seems like we had to write a lot of code in C to figure out what our resources were, where they were located in memory space, and where the data lived as well. A language that allowed the programmer to simply tie resources and data to whatever memory space would take a lot of work out of things. The language could then come with libraries or features to do things like resource management, mutex, semphores, and scheduling easily. It would just require the programmer to identify the addresses where stuff lived.

2

Any functionality that could be kept in standards C could be included in the general purpose O/S. The O/S would have to leave things like interrupts, timers, i/o, special locking operations, and anything else hard dependent up to the developer to add / create. To me, these would be like kernel modules. The O/S should be able to assume that a call to an interrupt, timer, or hardware resource locking does the same thing conceptually, regardless of the actual implemenation.

Although this may not be exactly true, the O/S kernel modules for particular hardware could make the hardware perform to the O/S assumptions. Some efficiency may be lost due to the lack of using the hardware to its full potential, but the result would be an O/S that could run on many types of hardware. This is probably a trade off of any software written for multiple kinds of hardware.

3

Binary semaphores

Binary semaphores, like mutex semaphores, have only two states; taken or not taken. The difference is that binary semaphores cannot be owned. Therefore, they are useful for public resources in the system that need service by tasks. This could be things like hardware or events in an event queue.

Mutex semaphores

Mutex semaphores, like binary semaphores, have two states, but mutex's can be owned. They are useful when a task needs to lock a resource (like a port) that involve operations in its critical section. This can potentially lead to a situation of priority inversion, where a task owns a resource

but is suspended, and the higher priority task is mutexed out of using it. The optimization for this is to write a schedule that can detect these priority inversions / deadlocks.

Counting semaphores

Counting semaphores are typically used to guard resources that are available in discrete quantities. One example of this are used slots in a circular queue.

4

In virtual memory, each task can own pages that map to collections of physical memory somewhere, either loaded more locally (registers or RAM), or more isolated (discs on the system). This can simplify memory management and protection from the tasks's perspective, but adds a lot of overhead in the embedded system world.

Like the RTOS I reported on, simpler memory management is used, where the physical hardware memory maps closer to the address space that tasks see, than virtual memory. This cuts down on overhead, but doesn't allow for things like memory fragmentation. Since embedded system memory is generally pretty low, these lacks of features generally aren't a problem.

5

Priority inversions happen when a task that is of a lower priority holds a resource (i.e. via a mutex semaphore), is suspended by a task of higher priority, and that task is then waiting on that resource to be freed. The higher priority task must essentially hand runtime back over to the lower priority task, or it will hang the system waiting for the locked resource. This nullifies its higher priority.

Solutions include disabling interrupts when a priority inversion prone resource is consumed, but is not favorable, because it can run away with the OS, and disables preemption. Another approach is priority inheritance, which means that when a lower priority tasks create a priority inversion, that lower priority task is temporarily assigned a higher priority so it can finish. Another approach is to create priority thresholding, which allows only tasks with certain priorities to interrupt other tasks. This means that tasks that could priority invert each other should stay in the same group.

6

step	hardware	software
1	program counter (PC) is pushed to stack	
2	address from interrupt vector table jumped to	
3		push general purpose (GP) registers to stack
4		push status register (SR) to stack
5		store the current stack pointer (SP)
6		set SP to new task
7		restore new task SR
8		restore new task GP
9	PC is restored from stack	

7

a)

The problem with the textbfbypass capacitor is that it probably doesn't really buffer the circuit from the spike that the switch might put off. It does decouple the power requirements a bit from the switch by providing a little extra current during dips in power, but probably only takes out small, brief powe spikes.

b)

The symptoms would be still a noticable spike coming from the switch, and if the switch spikes things when the capacitor is fully charged (switched off and on quickly), the power spike might be worse than without the capacitor.

c)

One solution would be to create an RC circuit to ease incoming amperage / power. The resistor in the RC circuit reduces the amperage coming in, and increases the charge time of the capacitor. At first, the voltage over the capacitor to ground is 0V and over the resistor is the source voltage. As the capacitor charges from the incoming current, the voltage increases across it until it matches source. At this point, the voltage over the resistor is 0V.

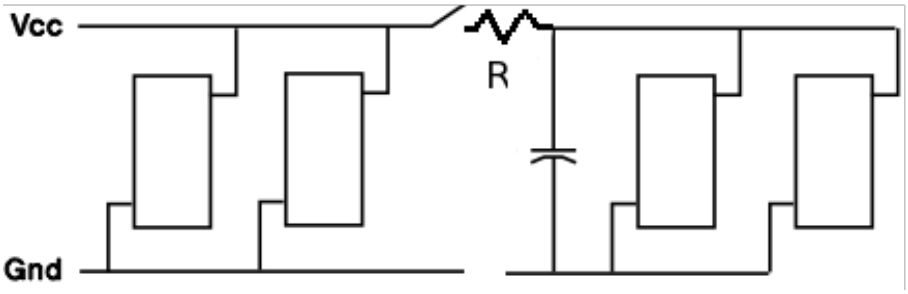

Figure 1: RC circuit solution

The voltage curve measure at the positive side of the capacitory can be determined from the RC time constant. The effect of the gradual voltage increase here is a solution to buffering power spikes from the switch.

8

bitset

```
#include <inttypes.h>
#include <avr/io.h>
#include <util/delay.h>
```

```
#define LEDPORT1 PORTB
#define LEDPORT2 PORTC

/*
 * Sets the LED port bit specified by bitnum to 1.
 * Retains all other values.
*/
void bitset(uint16_t bitnum)
{
        uint8_t bitnum1 = bitnum & 0x0F;
        uint8_t bitnum2 = bitnum >> 8;
        /*
         * Logic OR int 1 shifted bitnum times with the
         * current LEDPORT values
         */
        LEDPORT1 = ~( ~LEDPORT1 | (1 << (bitnum1)) );

        LEDPORT2 = ~( ~LEDPORT2 | (1 << (bitnum2)) );
}
```

bitclr

```
#include <inttypes.h>
#include <avr/io.h>
#include <util/delay.h>

#define LEDPORT1 PORTB
#define LEDPORT2 PORTC

/*
 * Sets the LED port bit specified by bitnum to 0.
 * Retains all other values.
*/
void bitclr(uint16_t bitnum)
{
        uint8_t bitnum1 = bitnum & 0x0F;
        uint8_t bitnum2 = bitnum >> 8;
        /*
         * Logic AND inverse of int 1 shifted bitnum times with the
         * current LEDPORT values
         */
        LEDPORT1 = ~( ~LEDPORT1 & ~(1 << (bitnum1)) );

        LEDPORT2 = ~( ~LEDPORT2 & ~(1 << (bitnum2)) );
}
```

allset

```
#include <inttypes.h>
#include <avr/io.h>
#include <util/delay.h>

#define LEDPORT1 PORTB
#define LEDPORT2 PORTC

/*
 * Sets all the LED port bits to 1
*/
void allset(void)
{
        LEDPORT1 = ~0xFF;
        LEDPORT2 = ~0xFF;
}
```

allclr

```
#include <inttypes.h>
#include <avr/io.h>
#include <util/delay.h>

#define LEDPORT1 PORTB
#define LEDPORT2 PORTC

/*
 * Sets all the LED port bits to 0
*/
void allclr(void)
{
        LEDPORT1 = ~0x00;
        LEDPORT2 = ~0x00;
}
```

setval

```
#include <inttypes.h>
#include <avr/io.h>
#include <util/delay.h>

#define LEDPORT1 PORTB
#define LEDPORT2 PORTC

/*
```

```
 * Sets the LED port to the param val
*/
void setval(uint32_t val)
{
        uint8_t val1 = val & 0x0F;
        uint8_t val2 = val >> 8;

        LEDPORT1 = ~val1;
        LEDPORT2 = ~val2;
}
```

setabs

```
#include <inttypes.h>
#include <avr/io.h>
#include <util/delay.h>

#define LEDPORT1 PORTC
#define LEDPORT2 PORTB

/*
 * Sets the LED port to the abs of param val
*/
void setabs(uint32_t val)
{
        if( (0xE0 & val) == 0xEF)
        {
                val = ~val;
        }

        uint8_t val1 = val & 0x0F;
        uint8_t val2 = val >> 8;

        LEDPORT1 = ~val1;
        LEDPORT2 = ~val2;
}
```

9

a)

The equations for Rate Monotonic response time is given as:

$$R_i = C_i + \sum_{j \epsilon hgp(i)} (\frac{R_i}{T_j}) C_j w_i^{n+1} \tag{1}$$

For processes:

process	T (period)	C (computation time)	Priority
1	6	1.5	1
2	5	2	2
3	4	1	3

We calculate response time as follows:

$$R_1 = \frac{3}{2} \tag{2}$$

$$w_2^0 = 2 \tag{3}$$

$$w_2^1 = 2 + \lceil \frac{2}{6} \rceil \frac{3}{2} = \frac{7}{2} \tag{4}$$

$$w_2^2 = 2 + \lceil \frac{\frac{7}{2}}{6} \rceil \frac{3}{2} = \frac{7}{2} \tag{5}$$

$$R_2 = \frac{7}{2} \tag{6}$$

$$\tag{7}$$

$$w_3^0 = 1 \tag{8}$$

$$w_3^1 = 1 + \lceil \frac{1}{6} \rceil \frac{3}{2} + \lceil \frac{1}{5} \rceil 2 = \frac{9}{2} \tag{9}$$

$$w_3^2 = 1 + \lceil \frac{\frac{9}{2}}{6} \rceil \frac{3}{2} + \lceil \frac{\frac{9}{2}}{5} \rceil 2 = \frac{9}{2} \tag{10}$$

$$w_3^3 = 1 + \lceil \frac{\frac{9}{2}}{6} \rceil \frac{3}{2} + \lceil \frac{\frac{9}{2}}{5} \rceil 2 = \frac{9}{2} \tag{11}$$

$$w_3^4 = 1 + \lceil \frac{\frac{9}{2}}{6} \rceil \frac{3}{2} + \lceil \frac{\frac{9}{2}}{5} \rceil 2 = \frac{9}{2} \tag{12}$$

$$w_3^5 = 1 + \lceil \frac{\frac{9}{2}}{6} \rceil \frac{3}{2} + \lceil \frac{\frac{9}{2}}{5} \rceil 2 = \frac{9}{2} \tag{13}$$

$$\tag{14}$$

$$R_3 = \frac{9}{2} \tag{15}$$

So we can rewrite the table as follows:

process	T (period)	C (computation time)	Response	Utilization
1	6	1.5	1.5	.25
2	5	2	3.5	.50
3	4	1	4.5	.25

This gives us a **combined utilization** of 1.0. This is above the threshold of **.78** for three processes, but the process set **will meet its deadlines**. This means that the task set **does have a feasible static priority assignment**.

b)

A calculated above, the **load / combined utilization** is 1.0.

c)

Considering the table again:

process	T (period)	C (computation time)	Response	Utilization
1	6	1.5	1.5	.25
2	5	2	3.5	.50
3	4	1	4.5	.25

These lead to the following Earliest Deadline First schedule:

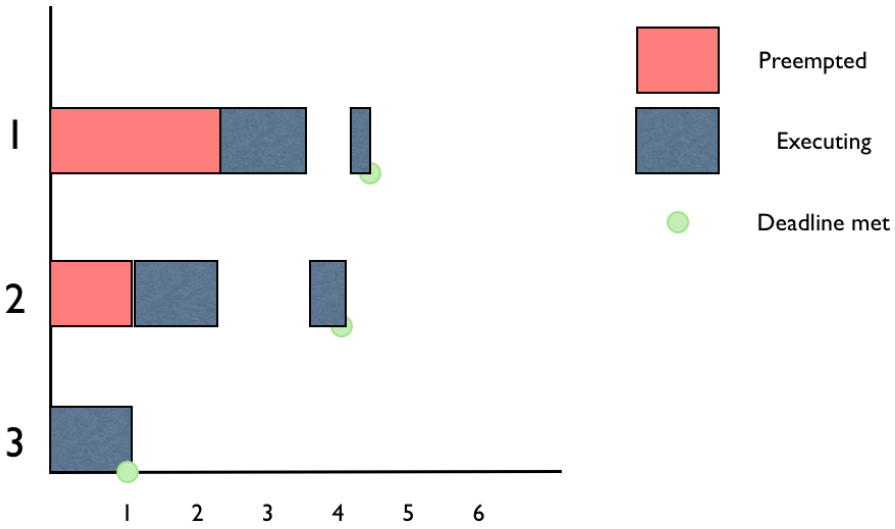

Figure 2: Earliest Deadline First schedule