CS 452 Second Exam

Colby Blair

Due April 4th, 2012

Grade: _____

1

a. F
b. T
c. T
d. F
e. T

2

(a) Variables allocated in the **bss** section.

- In languages like C, the compiler typicall adds code to the beginning (in our case, to the program memory) that initializes variables found in the bss section to 0, NULL, etc. Variables found in the bss section are typically globals or static variables, seen through the life of the program.

(b) Variables allocated in the **data** section

- Variables in the data section are initialized variables. For Harvard archetictures like ours, each time the variable is referenced, a load in program memory must be inserted to bind the variable to a register. Each time the variable is assigned to, a store must be inserted into program memory

(c) Local variables declared (but not initialized) as static

- At the begging of program memory, code is inserted to initialize static variables to 0, NULL, etc.

(d) an interrupt service routine

- Enabling interrupts will cause the compiler to create an interrupt vector/table at the beginning of program memory. The values in this table are usually the memory addresses for the respective interrupt handler functions.

3

The *round-robin* for our RTOS is a **non-preemptive** scheduling algorithm. It is the simplest possible algorithm for running multiple tasks. It calls the first task, lets it run until it finishes, then calls the second, etc. Since tasks are essentially unmanaged, **starvation** is possible, and tasks with higher prority can end up waiting on lower priority tasks.

By comparison, *multi-user round-robin* algorithms preemptively give each task even time slices. If a task is not finished, it is interrupted and will resume on the next round. It is a starvation free algorithm, but it also has no task priorities.

4

The primary point of the tick counter in the RTOS is to create a adjustable time unit in which to base preemptive actions (interrupts, context switches) on.

5

Having a **fast time tick** means that context switctes happen fast. A a pro, **Higher priority tasks wait less** for the next time tick, in which case they get to switch with a lower priority tasks and run. As a con, context switches can happen more often, which means the CPU has much more overhead doing context switch operations, and less actual task execution is done.

Having a **slow time tick** means just the opposite. As a pro, there are **less context switches and thus less overhead**. As a con, **higher priority tasks** have to wait longer before they can switch with a lower priority task.

6

Disadvantages to disabling interrupts while tasks are in critical section:

- If a task goes into an infinite loop (i.e. logic issues or waiting on IO), than the entire operating system is frozen.
- If the user forgets to renable interrupts, than the operating system may be frozen, or scheduling will not function properly.

7

```
int SEMAPHORE1 = TRUE; /*available*/

int CHECK_SEMAPHORE_INTERVAL = 3; /*seconds*/

/*
 * This function will make the callee wait until the semaphore is ready,
 * and then reserves the semaphore for the callee. The callee must free
 * it when it is done with its critical section.
 */
void P(int semaphore)
{
        while(semaphore != TRUE)
        {
                delay(CHECK_SEMAPHORE_INTERVAL);
        }

        /*
          * we execute the semaphore reservation as quickly as possible
          * after it is freed, to avoid any mutex issues by another task
          * doing the same thing.
          */

        semaphore = FALSE; /*reserved*/

        return;
}
```

8

Things that UIK has to preserve during a context switch are:

- The Stack Pointer register
- The General Purpose register
- The Status register
- Program Counter

9

UIK Design

The task stack, PCB, and more can be seen coded as follows:

```
#include <inttypes.h>
#include <avr/io.h>
#include <avr/pgmspace.h>


#define MAX_TASKS 100

/*Task State Type*/
enum task_state_t
{
        TREADY,
        TRUNNING,
        TWAITING
};


struct process_control_block
{
        /*
         * Task pointer - function pointer to the task,
         * since our tasks are functions.
         */
        void (*task_p)();

        /*Task state - see task_state enum type*/
        enum task_state_t task_state;

        /*Task/Process ID*/
        uint64_t tid;

        /*Program Counter*/
        uint64_t pc;

        /*Memory Limits*/
        uint64_t mem_ub;        /*upper bound*/
        uint64_t mem_lb;        /*lower bound*/

        /*Registers*/
        /*var types match the bit width of registers*/
        uint8_t stat_reg;       /*Status Register*/
        uint8_t gen_reg;        /*General Purpose Register*/

        /*
```

```
        * Opened Files - not implemented, since we don't
        * have a file system
        */

};

/*Task Stack - add elements here*/
struct process_control_block TASK_STACK[MAX_TASKS] PROGMEM;

/*
 * Task Stack Pointer - initialize to the beginning of
 * the stack.
 */
struct process_control_block *STACK_POINTER = TASK_STACK;
```

9.1 Task Stack(s)

I'm starting out with one task stack. It is specified in the same spot as the process control block (PCB). I've declared it with PROGMEM, so it lives in program memory as opposed to data memory. Anticipating lots of context switching, this will hopefully keep latency to a minimum, since it won't have to go to data memory for context information. The task stack is **static** in size, but the values for the tasks can be modified. New tasks have to be compiled in.

9.2 Task Control Block

Contained in my Task Control Block is all the standard stuff from our textbook and classnotes: (function) **pointer** to the task, **state/status** (blocked, ready, running), pid (process/task id), program counter, **register** (Status, General Purpose), and **memory limits** (upper and lower bounds).

9.3 Stack Pointer

To access the Stack pointer, we'll have to write some assembly code. I choose to use inline assembly in C.

```
asm volatile (
              ...
             "in      r0, 0x3d          \n\t"    \
             "st      x+, r0             \n\t"    \
             "in      r0, 0x3e          \n\t"    \
             "st      x+, r0             \n\t"    \
              );
```

The Stack Pointer is now in the General Purpose Register, and I can access this way. 0x03d and 0x3E above are the locations of the Stack Pointer in memory. I save this later in a variable in my C code (see code snippet above).

9.4 Context

I save the context of each task (function) in there respective PCB's. The Task Stack and the PCB's are defined in program memory using the AVR PROGMEM macro.

9.5 Program counter register

Saving the Program Counter is tricky, apparently. You have to do some sort of jump, and the PC value is popped on top of the stack. In assembly, an **RCALL** to a relative position of 0. We can then store the value to whatever we like.

To restore the program to the PC, do an assembly ICALL (immediate jump) to our stored PC value.

We can use inline assembly in the C source to do all this. Example:

```
asm volatile("RCALL 0\n\t"
             "nop\n\t"
               ...
             "nop\n\t"
             "ICALL r1\n\t"
             ::);
```