Lecture 12: Midterm review

Vikram Iyer

Adapted from material by Blake Hannaford

Logistics and announcements

- Vikram's Friday OH canceled this week (for grading and DRS exam)
- Midterm this Friday 5/10, 90 min
- **Location MOR 230 at 12:30pm (normal lecture room and time)**
  - DRS Exam location: ECE 345 (Lab), one of the TAs will proctor
- Closed book/notes except **1 sheet of (8.5"x11") paper, single side**
- Topics
  - C programming
  - Number representation and logical operators
  - Pointers
  - Schedulers
  - Interrupts and ADCs

3 Pointers (15 pts)

For the following declarations and statements,

```
// declarations

int i,j,*k;
char *x3,y3,*z3;
char *a3 = "Hello My Name is Albert";

// assignment statements
i = 517;
j = i+3;
k = &j;
x3 = a3+strlen(a3)-14;
y3 = 'x';
z3 = a3 + 17;
```

```
print_str(z3);

strlen(x3);

*k+10;

y3+1;

x3*-7;
```

3 Pointers (15 pts)

For the following declarations and statements,

```
// declarations

int i,j,*k;
char *x3,y3,*z3;
char *a3 = "Hello My Name is Albert";

// assignment statements
i = 517;
j = i+3;
k = &j;
x3 = a3+strlen(a3)-14;
y3 = 'x';
z3 = a3 + 17;
```

"Hello My Name is Albert"
^ ^
*a3 *a3+17

```
print_str(z3);
```
Prints: "Albert"

```
strlen(x3);

*k+10;
```
Dereference k (get the value stored there) and then do +10

```
y3+1;

x3*-7;
```
ERROR can't multiply pointers

1 C Programming (15 pts)

1.1 10pts

Write a C function to use bitwise operators to count how many bits are zero in a 16 bit int.

1 C Programming (15 pts)

1.1 10pts

Write a C function to use bitwise operators to count how many bits are zero in a 16 bit int.

```
int bitcount(int x) {
    int j,k;
    k=0;
    for(j=0;j<16;j++){
        if(!(x&(1<<j)))
            k++;
    }
    return(k);
}
```

General approach: go through each of the bits. Create another variable to assemble the output, and for each '0' we increment it.

j is our loop iterator
k stores our output

For each iteration shift a 1 "j" places (counting up) which creates a mask with a. 1 sliding across the whole number. Use this to check the bit and then increment the counter

If the bit is a 1 then append a 1 to the output (shift K left and or the LSB with 1)

1 C Programming (15 pts)

1.1 10pts

Write a C function to slide all of the ones in a 16-bit unsigned integer to the lsb end of the word. Examples:

```
0101010101010101  --->  0000000011111111
0100001000011000  --->  0000000000001111
1010000000000000  --->  0000000000000011
0000000000010100  --->  0000000000000011
```

1 C Programming (15 pts)

1.1 10pts

Write a C function to slide all of the ones in a 16-bit unsigned integer to the lsb end of the word. Examples:

```
0101010101010101  --->  0000000011111111
0100001000011000  --->  0000000000001111
1010000000000000  --->  0000000000000011
0000000000010100  --->  0000000000000011
```

```
unsigned int slide(unsigned int x) {
unsigned int i,j,k;
    k = 0;
    for(i=0;i<16;i++) {
        j = 1<<(15-i);
        if(j & x)
            k = (k<<1)|1;
    }
    return(k);
}
```

General approach: go through each of the bits. Create another variable to assemble the output, and for each ‘1’ we find in x append another ‘1’ to the output.

i is our loop iterator
k stores our output

j is a mask with a single bit set to ‘1’ where this gets shifted in each iteration of the loop. We use this mask to check the bit

If the bit is a 1 then append a 1 to the output (shift K left and or the LSB with 1)

2 Logical Operators in C (15 pts)

2.1 (10 pts)

Evaluate the following expressions:

```
#define A   01010101
#define B   11110000
#define C   00111010
#define D   01010101
#define E   00010001

unsigned char a,b,c,d;

a =   A | B | ~C ;
b =  (B & C) << 2 ;
c =  (B && !C) | C ;
d =  (A << 1) | D;
e =  (B & C & A) >> 4 ;
```

2.2 (5 pts.)

Convert A, B, C, D to hex;

A	
B	
C	
D	
E	

2 Logical Operators in C (15 pts)

2.1 (10 pts)

Evaluate the following expressions:

```
#define A   01010101
#define B   11110000
#define C   00111010
#define D   01010101
#define E   00010001

unsigned char a,b,c,d;

a =   A | B | ~C ;
b =  (B & C) << 2 ;
c =  (B && !C) | C ;
d =  (A << 1) | D;
e =  (B & C & A) >> 4 ;
```

2.2 (5 pts.)

Convert A, B, C, D to hex;

A	0x55
B	0xF0
C	0x3A
D	0x55
E	0x11

Every 4 binary bits maps to 1 hex digit. Use the table to look up the correct hex digits

"0x" indicates a hex value

Base 10	Base 2	Base 16
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

2 Logical Operators in C (15 pts)

2.1 (10 pts)

Evaluate the following expressions:

```
#define A   01010101
#define B   11110000
#define C   00111010
#define D   01010101
#define E   00010001

unsigned char a,b,c,d;

a =   A | B | ~C ;
b =  (B & C) << 2 ;
c =  (B && !C) | C ;
d =  (A << 1) | D;
e =  (B & C & A) >> 4 ;
```

Give answers in binary:

a								
b								
c								
d								
e								

2 Logical Operators in C (15 pts)

2.1 (10 pts)

Evaluate the following expressions:

```
#define A   01010101
#define B   11110000
#define C   00111010
#define D   01010101
#define E   00010001

unsigned char a,b,c,d;

a =   A | B | ~C ;
b =  (B & C) << 2 ;
c =  (B && !C) | C ;
d =  (A << 1) | D;
e =  (B & C & A) >> 4 ;
```

Give answers in binary:

a	1	1	1	1	0	1	0	1
b	1	1	0	0	0	0	0	0
c	0	0	1	1	1	0	1	0
d	1	1	1	1	1	1	1	1
e	0	0	0	0	0	0	0	1

a: first flip the bits of c, and then check each bit position. Because this is an or, if any bit is a '1' the output will be a '1'

b: do the B&C first. Only 2 bits are both 1, then shift the result left by 2

c: These are LOGICAL operators. Nonzero numbers evaluate to TRUE so !C is evaluated first and is !(TRUE) = FALSE. Anding this with anything makes the result false. FALSE is all zeros, TRUE is a 1 in the LSB (00000001). Since this is FALSE = 0, bitwise OR with '0' doesn't do anything, it gives you back C.

4 Schedulers (15 pts.)

Consider the following three tasks with a *non-preemptive*, *round robin* scheduler WITH synchronization. As we discussed in class: Consider P, and D to be *specifications* and C a resource requirement. After charting the schedule, are P and D satisfied?

- **Task A** C = 233μsec, P=1ms, D=0.5ms.
- **Task B** C = 100μsec, P=2ms, D=2ms.

```
Task_B() {
   compute for 100mu sec.;
   sleep(2);
   }
```

- **Task C** C = 233μsec, P=1ms, D=1ms.

```
Task_C() {
  static i=0;
  compute for 233mu sec.;
  if(i++ > 1) { halt_me(); }
   }
```

Make the following assumptions:

- Functions `sleep(n)` and `halt_me()` exist as described in class.
- A static variable is initialized only once, before the function is executed for the first time.
- The scheduler takes 100μsec of CPU time.
- Assume scheduler starts for the first time at $t = 0\mu$sec.
- As in class, make each horizontal line represent 333μsec.

Complete the schedule analysis through at least 7 milliseconds.
(chart on next page)

Prefix	Postfix

int i=0;
if(**++i**> 0)
  Serial.println("True");
else
  Serial.println(i);
	int i=0;
	if(**i++**> 0)
	  Serial.println("True");
	else
	  Serial.println(i);

OUTPUT: >>> True	OUTPUT: >>> 1

4 Schedulers (15 pts.)

Scheduler absorbs extra time so that tasks always start on the next OS tick (e.g. at 1ms, 2ms, etc).

Consider the following three tasks with a *non-preemptive*, *round robin* scheduler WITH synchronization. As we discussed in class: Consider P, and D to be *specifications* and C a resource requirement. After charting the schedule, are P and D satisfied?

- Task A C = 233μsec, P=1ms, D=0.5ms.
- Task B C = 100μsec, P=2ms, D=2ms.

D = deadline, task needs to finish within the first 0.5 ms of period

P = period
Task runs once in every 2ms block

```
Task_B() {
  compute for 100mu sec.;
  sleep(2);
  }
```

- Task C C = 233μsec, P=1ms, D=1ms.

```
Task_C() {
  static i=0;
  compute for 233mu sec.;
  if(i++ > 1) { halt_me(); }
  }
```

Important detail: This is a postfix operator (i++), so the statement will be executed with the current value of a variable and then the value gets incremented. In prefix (++i), the value of the variable will get incremented first and then the statement will get executed.

Nonblocking delay function implemented with a counter. Convention: counter updates right before each time tick, function runs if sleep_count=0

Make the following assumptions:

- Functions `sleep(n)` and `halt_me()` exist as described in class.
- A static variable is initialized only once, before the function is executed for the first time.
- The scheduler takes 100μsec of CPU time.
- Assume scheduler starts for the first time at $t = 0\mu$sec.
- As in class, make each horizontal line represent 333μsec.

Stop running task after this is called

If this were -100 us then we start Task A at 0ms

Complete the schedule analysis through at least 7 milliseconds. (chart on next page)

Prefix	Postfix

int i=0;
if(**++i**> 0)
Serial.println("True");
else
Serial.println(i);
	int i=0;
	if(**i++**> 0)
	Serial.println("True");
	else
	Serial.println(i);

OUTPUT: >>> True	OUTPUT: >>> 1

Steps for filling out diagrams:

1. Label every 3rd box as an OS time tick (1 ms, 2 ms, etc)
2. Check the scheduler start time (t=-100 us, t=0ms)
3. Shade in the box for Task A and then return to the scheduler for 100 us
4. Immediately after the scheduler runs for 100us, call Task B. Return to the scheduler after completing it.
5. Immediately after the scheduler runs for 100us, call Task C. Return to the scheduler after completing it.
6. Update the sleep variables and if synchronized, wait until the next OS tick.
   - Assume that if a sleep counter reaches 0 at an OS tick (e.g. t=2 ms) then the task will run during the following cycle (between t=2ms and t=3ms)
   - For a non-preemptive scheduler, if a task runs past a deadline D or the tasks take longer than 1 OS tick (e.g. say Task C goes past 1 ms), a synchronized scheduler will wait until 2 ms to before it can run Task A again
7. If a timing violation occurs (deadline is not met), make a note of it on the diagram but assume this does not affect how it runs (e.g. if Task B violated deadline go ahead and continue on to Task C)
8. Make sure to fill out the worksheet all the way until the time tick we said (e.g. 7 ms)

	Scheduler	Task A	Task B	Task C
0 ms				
1 ms				
2 ms				
3 ms				
4 ms				

Task A C = 233μsec, P=1ms, D=0.5ms.

Task B C = 100μsec, P=2ms, D=2ms.

```
Task_B() {
   compute for 100mu sec.;
   sleep(2);
   }
```

Task C C = 233μsec, P=1ms, D=1ms.

```
Task_C() {
  static i=0;
  compute for 233mu sec.;
  if(i++ > 1) { halt_me(); }
   }
```

	Scheduler	Task A	Task B	Task C
4 ms				
5 ms				
6 ms				
7 ms				
8 ms				

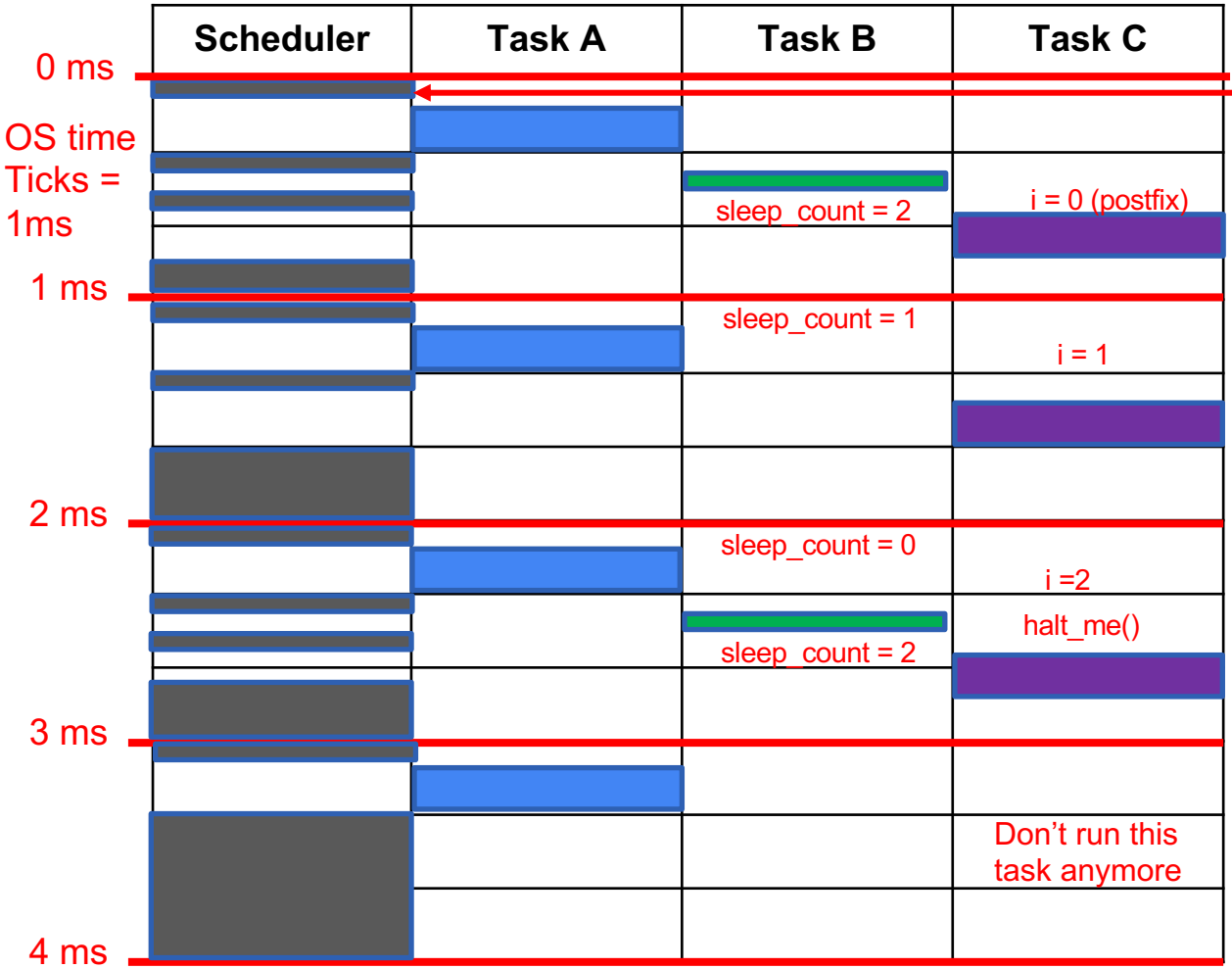
Task A C = 233μsec, P=1ms, D=0.5ms.

Task B C = 100μsec, P=2ms, D=2ms.

```
Task_B() {
   compute for 100mu sec.;
   sleep(2);
   }
```

Task C C = 233μsec, P=1ms, D=1ms.

```
Task_C() {
  static i=0;
  compute for 233mu sec.;
  if(i++ > 1) { halt_me(); }
   }
```



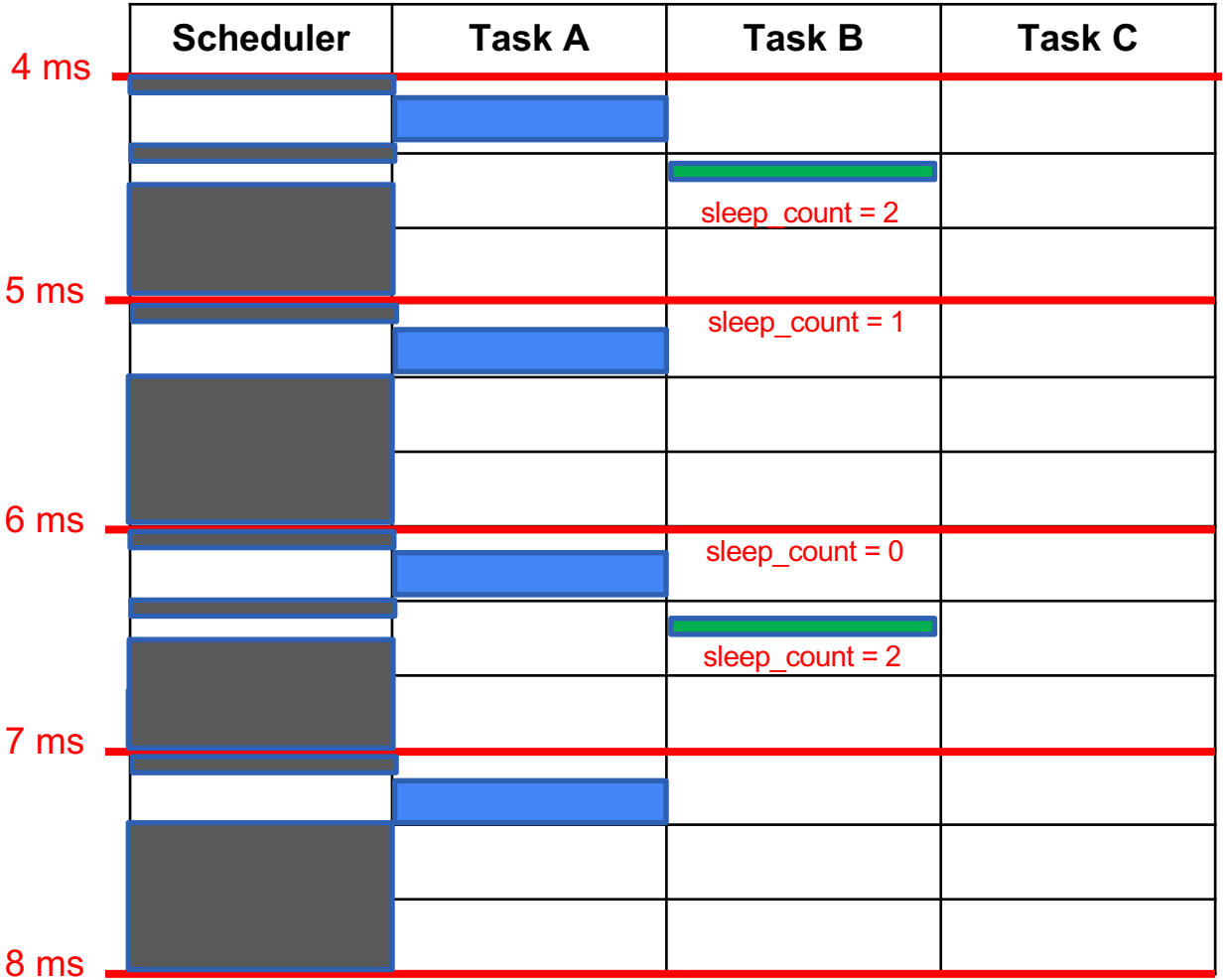
Task A C = 233μsec, P=1ms, D=0.5ms.

Task B C = 100μsec, P=2ms, D=2ms.

```
Task_B() {
   compute for 100mu sec.;
   sleep(2);
   }
```

Task C C = 233μsec, P=1ms, D=1ms.

```
Task_C() {
  static i=0;
  compute for 233mu sec.;
  if(i++ > 1) { halt_me(); }
   }
```



Task A C = 233μsec, P=1ms, D=0.5ms.

Task B C = 100μsec, P=2ms, D=2ms.

```
Task_B() {
   compute for 100mu sec.;
   sleep(2);
   }
```

Task C C = 233μsec, P=1ms, D=1ms.

```
Task_C() {
  static i=0;
  compute for 233mu sec.;
  if(i++ > 1) { halt_me(); }
   }
```

4 Process Context and Interrupts (10 pts)

4.1 (Process Context 5pts)

A "context switch" occurs each time a function is called or exited. Briefly explain why a "context switch" takes up both CPU time and memory. Do all context switches use memory?

4 Process Context and Interrupts (10 pts)

4.1 (Process Context 5pts)

A "context switch" occurs each time a function is called or exited. Briefly explain why a "context switch" takes up both CPU time and memory. Do all context switches use memory?

When a function is called, the CPU must save context information (i.e. PC, and registers) on the stack. This takes both CPU time and memory. When the function exits, CPU time is required to restore the registers, but memory is freed up as the data are ``popped'' off of the stack. So, function calls use up memory, but function returns free up memory.

For questions like this you just need to know the general concepts we went over in lecture, we won't ask for exact details on what happens for a particular chip like the Arduino

What are some problems or potential issues to be careful of when working with interrupts? List at least 3.

If the ADC Value on a 10-bit ADC in a 3.3V system is 362, what is the voltage on the pin?

What are some problems or potential issues to be careful of when working with interrupts? List at least 3.

From lecture on interrupts

- Very hard to debug/can break debugging tools
- Interrupts can override global variables
- ISR takes over processor, can override even a pre-emptive scheduler
- Can take up CPU time and disrupt time critical tasks
- Plays around with the stack

Think of this as a unit conversion. Note 1024 comes from the 10 bits. These can encode 2^n=1024 values, and the voltage range of 3.3 is split up into these increments

If the ADC Value on a 10-bit ADC in a 3.3V system is 362, what is the voltage on the pin?

$$ADC\ Value = \frac{(Voltage\ on\ Pin(mV)) * (Max.\ ADC\ Value)}{(System\ Voltage(mV))}$$

or

$$ADC\ Value = \frac{(Voltage\ on\ Pin(mV)) * 1024}{3300}$$

To convert an `analogRead` value to voltage, should we divide by 1023 or 1024?

I think the key here is to remember that an ADC conversion represents a range of values with a step size of `5V/1024 = 0.0048828125V`. So if `analogRead` returns 0, this is really a range of 0V to 0.0048828125V, and 1 is a range of 0.0048828125V to 0.009765625V, *etc.* In that regard, we would want to divide analogRead by 1024 and if analogRead returns 1023, 1023/1024 * 5V = 4.9951171875V **to** 5V.