The University of Texas at Austin
Department of Electrical and Computer Engineering

Real-Time Systems / Real-Time Operating Systems

ECE445M/ECE380L.12, Spring 2023

Final Exam Solutions

Date: April 28, 2023

UT EID: ________________

Printed Name: ____
Last, First

Your signature is your promise that you have not cheated and will not cheat on this exam, nor will you help others to cheat on this exam:

Signature: ____

Instructions:

- Open book, open notes and open web.
- No electronic devices other than your laptop/PC (cell phones off and stowed away).
- You are allowed to access any resource on the internet, but no electronic communication other than with instructors.
- Please be sure that your answers to all questions (and all supporting work that is required) are contained in the space (boxes) provided.
- For all questions, unless otherwise stated, find the most efficient (time, resources) solution.

Problem 1	10	
Problem 2	20	
Problem 3	30	
Problem 4	15	
Problem 5	25	
Total	100	

*Name:*________________________________

Problem 1 (10 points): Miscellaneous

a) Suppose you are asked to design a new robot car using your RTOS and a PID controller. How would you define the error-term for your PID controller?

The error term in a control system is general the deviation of the desired from the actual (measured or estimated) state of the variable being controlled.

For a self-driving robot, there are many different ways to define the variable to control, e.g. distance to the closest wall (error term = difference between actual and desired distance), or angle to the wall (error term = difference between actual and desired angle).

b) What are two things you could do to minimize the impact of noise when using the GP2Y0A21YK IR sensors with your robot car?

The first step is to avoid noise in the filter input in the first place, e.g. by shielding the cables from any noise sources (such as motors).

Removing noise otherwise requires filtering the input signal. For an analog sensor like the IR, we always need an analog low-pass filter in the front just to avoid sample aliasing. This filter will also remove high-frequency noise, but will let any low-frequency noise that is within the sampled range through.

We can complement this with digital filers, such as a median filter that can remove specific sources of noise (such as spikes or outliers) without otherwise affecting the signal.

*Name:*______________________________

Problem 2 (20 points): OS Kernel

Below is the *SVC_Handler*, *OS_Kill()*, and *OS_AddThread()* in a multithreaded, round-robin RTOS with spinlock semaphores and dynamic process loading via a heap, where applications running on the OS trigger calls to *OS_Id()*, *OS_Kill()*, *OS_Sleep()*, *OS_Time()* and *OS_AddThread()* via SVC traps. Assume that *allocateTCB()* and *allocateStack()* do all the proper allocations and initializations associated with thread TCB and stack creation. If unsure, write down any of your assumptions.

```
SVC_Handler
  LDR  R12,[SP, #24]
  LDRH R12,[R12,#-2]
  BIC  R12, #0xFF00
  LDM  SP, {R0-R3}

  PUSH {LR}
  LDR  LR, =Return
  CMP  R12, #0
  BEQ  OS_Id
  CMP  R12, #1
  BEQ  OS_Kill
  CMP  R12, #2
  BEQ  OS_Sleep
  CMP  R12, #3
  BEQ  OS_Time
  CMP  R12, #4
  BEQ  OS_AddThread

Return
  POP  {LR}
  STR  R0, [SP]
  BX   LR
```

```
// Kill thread
OS_Kill() {
  OS_bWait(&mutex); // lock OS kernel

  // Decrement threads in process
  PCB_t* parent = runPt->pcb;
  parent->numThreads--;
  if (!parent->numThreads){
     // Kill process if last thread is killed
     Heap_Free(parent->data);
     parent->id = -1; // Mark parent as unused
  }

  // Remove thread from list
  runPt->prev->next = runPt->next;
  runPt->next->prev = runPt->prev;

  OS_bSignal(&mutex);
  OS_Suspend();
}
```

```
// Add thread, return 0 if unsuccessful
int OS_AddThread(void(*task)(void), uint32_t stackSize) {
   OS_bWait(&mutex); // lock OS kernel
   TCB_t *tcb = allocateTCB();
   If (tcb == NULL) {
      OS_bSignal(&mutex);
      return 0;
   }
   uint32_t *sp = allocateStack(task);
   if (sp == NULL) {
      OS_bSignal(&mutex);
      return 0;
   }
   enqueue(runPt, tcb); // Place TCB in running linked list

   OS_bSignal(&mutex);
   return 1;
}
```

*Name:*______________________________

There may be bugs associated with the code above. Please list all bugs, if any, and how they might be fixed. If there are multiple different solutions, list all possible ways to fix a bug.

The code has four bugs:

1) The OS kernel makes calls to OS routines (OS_Kill() and OS_AddThread()} that use mutex semaphores from within the SVC handler. Calling OS_bWait() from within an interrupt handler will lead to the system locking up.

Possible solutions:
- *Replace the use of mutex semaphores to create mutual exclusion within the OS kernel with Enable/DisableInterrupts()*
- *Use dynamic linking of the application code to implement calls to mutex-based OS routines instead of SVC traps.*

2) The OS_Kill() routine does not free the process code segment when the process gets killed, i.e. when the last thread is killed.

Solution: need to call Heap_Free() on the process' code segment together with freeing the data segment.

3) OS_Kill() does not de-allocate the TCB.

Solution: need to do something like runPt->id = -1;

4) OS_AddThread() does not initialize the stack pointer in the TCB

Solution: need to set tcb->sp = sp in OS_AddThread()

There could be other bugs, but those will all depend on the assumptions being made. E.g. technically, the TCB also will need to be de-allocated if allocateStack() fails, but one can assume that if TCB allocation succeeded, stack allocation will, too, i.e. something major is wrong then. Note that we also assume here that allocateTCB() will perform the initialization of the pcb pointer in the TCB (based on the runPt);

Problem 3 (30 points): Heap

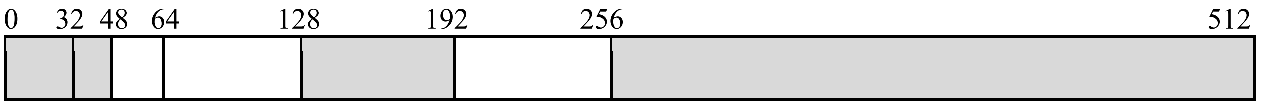
a) Assume a 512 byte heap implemented using the algorithm discussed in class, in the lecture notes and book, consisting of blocks with headers and tails for each block indicating the block size and negative sizes indicating free space. Memory is 4-byte word aligned. Given the sequence of *Heap_Malloc()* and *Heap_Free()* calls on the right, show the final state of the heap at the end of the sequence for each al1location scheme.

```
a = Heap_Malloc(14);
b = Heap_Malloc(4);
c = Heap_Malloc(3);
e = Heap_Malloc(3);
Heap_Free(a);
Heap_Free(c);
d = Heap_Malloc(4);
```

Initial Heap	First Fit	Best Fit	Worst Fit
-126	*1*	*-4*	*-4*
	d		
	1		
	-1		
	-1	*-4*	*-4*
	1	*1*	*1*
	B1	*b*	*b*
	1	*1*	*1*
	-1	*1*	*-1*
		d	
	-1	*1*	*-1*
	1	*1*	*1*
	e	*e*	*e*
	1	*1*	*1*
	-111	*-111*	*1*
			d
			1
			-108
…	…	…	…
-126	*111*	*111*	*-108*

*Name:*______________________________

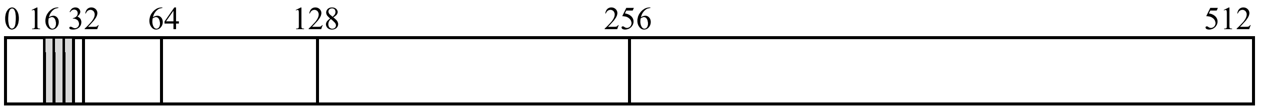
b) Now assume that the 512 byte heap is implemented using the Knuth's Buddy Allocation algorithm with the smallest allocation size of 4 bytes. Given the following heap state (where grey are allocated blocks and white are free ones), list a sequence of *Heap_Malloc*/*Heap_Free* commands that could result in this heap.



One possible sequence:

a= Heap_Malloc(17…32); // anything in that range
b =Heap_Malloc(1…16);
c =Heap_Malloc(33…64);
d =Heap_Malloc(33…64);
Heap_Free(c);
e = Heap_Malloc(129…256); // this can be at any point after 'a' has been allocated

c) What would Knuth's Buddy Allocation Algorithm final heap state look like for the sequence from a)? You can assume that the algorithm stores necessary meta-data to keep track of heap state outside of the heap array. Not accounting for meta-data overhead, which implementation or allocation scheme has the most and least amount of internal or external fragmentation?



Blocks above are: b (16..19), d (20..23), e (24..27)

All heaps waste 1 byte for block 'e', i.e. have the same internal fragmentation not including overhead for meta-data. Buddy allocation in general has higher internal fragmentation for anything that is not a power of two.

The largest allocatable block with Buddy Allocation is 256 bytes. The largest allocatable block with first fit and best fit is 444 bytes, with worst fit it is 432 bytes.

In all cases, with 3 words allocated, the available free space is 500 bytes not counting meta-data overhead (452 or 460 bytes with overhead for the regular heap), i.e. roughly the same.

As such, first/best fit have the least, buddy allocation the most external fragmentation for this example. Buddy allocation in general has less external fragmentation on average since it uses fixed-size (power of two) buckets, but not in this specific example.

Problem 4 (15 points): File System

a) For an SD Card with 64 GiB of memory, how many 512-byte blocks would you need to store the FAT?

*64GiB = 64 * 2^30 bytes = 2^27 * 512 byte blocks, i.e. a FAT with 2^27 = entries.*

To address 2^27 blocks, each FAT entry must hold at least 27 bits, i.e. 4 bytes.

*As such, FAT size = 2^29 bytes = 2^20 * 512 byte blocks.*

b) The FAT on the right was recovered for a SD Card, but the directory was corrupted. Fill in the directory with the starting blocks and the lengths of each file and the free blocks list. You can assume that the free block list is larger than any file.

Directory:

File Name	Starting Block	File Size (in blocks)
File 1	*2*	*3*
File 2	*3*	*2*
File 3	*5*	*5*
File 4	*16*	*6*
File 5	*15*	*2*
Free	*6*	*12*

FAT:

0	X
1	X
2	4
3	10
4	8
5	12
6	14
7	21
8	-
9	-
10	-
11	-
12	13
13	20
14	17
15	9
16	7
17	18
18	19
19	24
20	11
21	22
22	23
23	31
24	25
25	26
26	27
27	28
28	29
29	30
30	-
31	-

Problem 5 (25 points): System Design

a) Assuming you have a round robin scheduler with a time slice of 2 ms with 5 tasks. Suppose you want to log data to your SD card in Task 1. The amount of data to log is 24 MB and it takes 6000 time slices of running Task 1 to collect the data. The data is not written to disk by Task 1 until all 24 MB have been collected. Assuming a disk with 512 byte blocks and an SPI bus running at a clock rate of 4 MHz using single-block transfers only, what is the amount of time needed to finish logging (i.e. collecting and writing) the data in ms? Assume zero command-response delay (NCR=0). Please show your work.

*Time to collect the data is 6000 round robin time slices = 6000 * 2ms * 5 = 60,000 ms.*

*24MB = 24,000,000 bytes = 46,875 blocks * 512 bytes, i.e. 46,875 single-block disk writes.*

4Mbit/s equals 2us/byte for the SPI. Each single-block write takes 6+1+512+3= 522 bytes.

*Thus, writing 24MB to disk takes 46,875 * 522 * 2us = 48,937.5 ms. This is assuming that the task will not get interrupted while writing. Otherwise the time is multiplied by 5x.*

Total time: 60,000 ms + 48,937.5 ms = 108,937.5 ms.

b) Now assume that we want to do remote logging to a second TM4C connected via Ethernet. After collecting the data, the first TM4C sends the log via the network and the second TM4C writes it to their SD card. The receiver TM4C first buffers all data before writing it to disk. Assuming your SPI is running at 4 MHz and an Ethernet physical layer baud rate of 10 Mbits/s, what is the total time needed to log the data? You can assume that there are no other machines on the Ethernet network and zero SPI command response delay. Please show your work.

An Ethernet frame fits 1,500 bytes, so sending 24MB requires 16,000 frames.

Each Ethernet frame has 7+1+12+2+4 = 26 bytes overhead, i.e. 1,526 bytes total.

*At 10Mbit/s, sending 24MB takes 16,000 * 1,526 * 8 / 10^4 = 19,532.8 ms*

Total time = 60,000 ms + 19,532.8 ms + 48,937.5 ms = 128,470.3 ms

c) What is the total time needed to log the data if we change from Ethernet to using a CAN running at 1Mbit/s with 11 bit IDs assuming no stuffing is needed. Please show your work.

A CAN frame fits 8 bytes, so 24MB requires 3,000,000 frames.

Each CAN frame has 11 + 64 + 36 = 111 bits.

*At 1Mbit/s, sending 24MB takes 3,000,000 * 111 / 10^3 = 333,000 ms.*

Total time = 60,000 ms + 333,000 ms + 48,937.5 ms = 441,937.5 ms.

d) Now assume that the first TM4C will start collecting the next batch of data as soon as it is finished sending the previous one. What is the maximum rate at which data can be logged using Ethernet versus CAN? What is the maximum rate at which data can be logged assuming DMA is used on both TM4Cs to send and receive data, i.e. to copy the data from memory to the Ethernet or CAN controller and vice versa?

With two TM4Cs, we can perform writing to disk in parallel with collecting the next batch of data, but data transmission over the network is still serial, i.e. accumulative.

I.e. the time per data batch is transmission time + max(data collection, disk write).

For Ethernet: 19,532.8 ms + max(60,000 ms, 48,937.5 ms) = 79,532.8 ms per 24MB for a rate of 24,000,000 bytes / 79.5328 s ≈ 300 kB/s.

For CAN: 393,000 ms per 24MB for a rate of appr. 61 kB/s.

Using DMA support, we can free the CPU to also overlap the transmission to happen in parallel to data collection and disk write (for different batches, i.e. in a pipelined fashion), i.e. the time per batch is the max(transmission, data collection, disk write), and the rates would be:

For Ethernet: 60s per 24MB = 400 kB/s (system is bottlenecked by data collection)

For CAN: 333s per 24MB ≈ 72 kB/s (bottlenecked by CAN).