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

Real-Time Systems / Real-Time Operating Systems

EE445M/EE380L.12, Fall 2020

Final Exam Solutions

Date: December 12, 2020

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 calculators or any electronic devices other than your laptop/PC (turn cell phones off).
- 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	10	
Problem 3	20	
Problem 4	20	
Problem 5	15	
Problem 6	10	
Problem 7	15	
Total	100	

Problem 1 (10 points): Deadlock

Assume the following *OS_Wait_Or* implementation from the midterm solutions:

```
Sema4Type* OS_Wait_Or(Sema4Type* semaA, Sema4Type* semaB) {
 Sema4Type *semaRes;
 DisableInterrupts();
 while ((*semaA) <= 0 && (*semaB) <= 0) {
   EnableInterrupts(); DisableInterrupts();
 }
 if ((*semaA) > 0) {
   semaRes = semaA;
 } else {
   semaRes = semaB;
 }
 (*semaRes) = (*semaRes) – 1;
  EnableInterrupts();
 return semaRes;
}
```

Given the following program using *OS_Wait_Or*:

```
OS_InitSemaphore(&A,1);
OS_InitSemaphore(&B,1);
OS_InitSemaphore(&C,1);
OS_InitSemaphore(&D,1);
```

```
TaskA(){
 sema4* t1;
 sema4* t2;
 t1= OS_Wait_OR(&A,&B);
 t2= OS_Wait_OR(&B,&C);
 ...

 OS_Signal(t2);
 OS_Signal(t1);
}
```

```
TaskB(){
 sema4* t1;
 sema4* t2;
 t1= OS_Wait_OR(&A,&C);
 t2= OS_Wait_OR(&C,&D);
 ...

 OS_Signal(t2);
 OS_Signal(t1);
}
```

```
TaskC(){
 sema4* t1;
 sema4* t2;
 t1= OS_Wait_OR(&D,&A);
 t2= OS_Wait_OR(&B,&C);
 ...

 OS_Signal(t2);
 OS_Signal(t1);
}
```

Is there any possible interleaving that could cause deadlock with this program? If yes, please show an example that could be a deadlock. If not, justify your answer.

Since the OS_Wait_Or prioritizes the first argument over the second, there is no deadlock, task A or task B will grab sema A, leaving either sema B or sema C available for at least one task to continue past the second OS_Wait.

If order of semas were reversed, or sema A were not available for some other reason (another task grabbing it), there would be a deadlock sequence:

First, Task A holds semaphore B, then Task B holds semaphore C, then Task C holds semaphore D.

B→C→D→(B or C) circular wait.

Problem 2 (10 points): Scheduling

Given the following three foreground threads in your system:

```
ThreadA(){
  while(1) {
    OS_Wait(&Sema);
    countA++;
  }
}
```

```
ThreadB(){
  while(1) {
    OS_Wait(&Sema);
    countB++;
  }
}
```

```
ThreadC() {
  while(1) {
    OS_Signal(&Sema);
    OS_Signal(&Sema);
    countC++;
  }
}
```

Is there any OS realization (scheduling strategy, semaphore implementation, priority assignment) that ensures that all three threads run an equal number of times, i.e. that the three counting variables *countA*, *countB* and *countC* are equal or at most off by one at all times? If so, what is that OS setup? If not, why not? You can assume that the semaphore and all counting variables are initialized to zero.

It can not be round-robin since otherwise whatever thread runs after C would pick up all the semaphores.

With priorities, thread C must have lower priority than A or B or it would run forever and A/B would never run. Priorities also means blocking semaphores.

Finally, threads A and B must have same priorities, with round-robin scheduling among them or else one of the two would always pick up both semaphores whenever C notifies.

Problem 3 (20 points): Heap and ELF loader

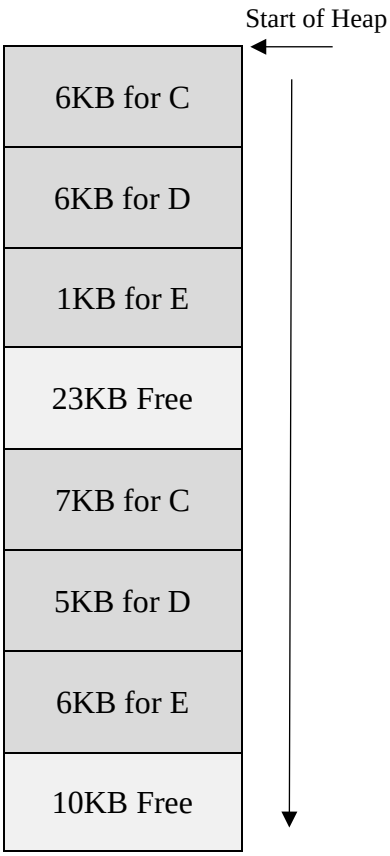
Assume that six ELF executables files with code and data segment sizes as shown below are stored on your SD card and launched from your interpreter in the order *A*, *B*, *C*, *D* and *E*, where the ELF loader uses different heap allocation routines for code and data. *Heap_Alloc* allocates memory in 1KB granularity from start to end, i.e. an allocated block is always placed at the bottom of its chosen free space. Available heap size is 64KB in total and there is no metadata overhead.

Size of code and data segment for each program:

Program	*Code size*	*Data size*
A	6 KB	11KB
B	6 KB	13KB
C	6 KB	7 KB
D	6 KB	5 KB
E	6 KB	1 KB

ELF loader:

```
int exec_elf(…) { int sr;
  …
  sr = StartCritical()
  code = Heap_AllocCode(h->codeSize);
  data = Heap_AllocData(h->dataSize);
  EndCritical(sr);
  …
}
```



Given the heap state as shown on the right:

a) What heap allocation algorithms (first/best/worst fit) are used for code and data? Are they different? How do you know, i.e. explain your reasoning.

Code is best fit, and Data is worst fit.

As the first programs, A must have been allocated into the first 17kB of memory no matter what algorithm, but is already released now. The only way C's data could have ended up where it is (36kB from bottom) is if B was loaded while A was still in memory. Given that C's code is at location 0, A must have been unloaded but B still in memory when C was loaded. At that point, there were 17KB of free space on the bottom, and 28KB of free space on the top of the heap. Then, code must have been first fit or best fit, and data worst fit.

Then, D is loaded, leaving 5kB and 16kB of free space. Since E's worst-fit data ends up in the lower part, however, B must have been released at that point, such that free space was 24kB and 16kB instead. Since E's code ended up in the 16kB part, code must be best fit.

b) What was the program execution order flow, i.e. specify the sequence and order in which programs started and ended to reach this heap state, e.g. in the form *Start A*, *End A*, …

See above: Start A, Start B, End A, Start C, Start D, End B, Start E

c) Suppose you launch program *E* multiple times and there are no other programs loaded. How many instances of *E* can be loaded into memory simultaneously? Assume that there are enough resource other than the heap.

64kB/7kB = maximum 9 instances of E

d) Now suppose you want to load 50 instances of program *E* into memory simultaneously. Is this possible? If no, why not. If yes, explain how to enable this with 64KB of heap.

The code segment can be shared and needs to be loaded only once:

*6kB + 50 * 1kb = 56kB*

Problem 4 (20 points): File Systems

In a filesystem, disk accesses are expensive and can degrade performance. Given a disk with 512 byte blocks and the following trace of file accesses made by an application:

a1: Read File A, Byte 634
a2: Write File A, Byte 634
a3: Read File B, Byte 128
a4: Read File A, Byte 42
a5: Write File B, Byte 128
a6: Read File C, Byte 522
a7: Write File C, Byte 1096
a8: Write File A, Byte 42

a) How many disk accesses are performed by file systems that use an indexed and linked allocation as discussed in class? Assume that the directory and index table are both already loaded into memory, but no other caching of data is performed between accesses.

Indexed File System:	Linked File System:
Every read is one access. *Every write is read-modify-write.* *Total of 12 accesses.*	*17 accesses. Traverse list one every access:* *a1: read A1, read A2* *a2: read A1, read A2, write A2* *a3: read B1* *a4: read A1* *a5: read and write B1* *a6: read C1, read C2* *a7: read C1, read C2, read C3, write C3* *a8: read and write A1*

b) Now assume an implementation of a file system in which you have created a small file system cache in memory that remembers the 3 last blocks from the disk that you have accessed. Suppose that the cache replaces blocks in a First In First Out manner. How many disk accesses are performed for the indexed and linked file systems? Assume that modified (dirty) blocks in the cache are only written back to disk when they are evicted from the cache, and that linked list traversals are always started from the beginning.

Indexed File System:	Linked File System:
7 accesses: *a1: read A2 and cache* *a2: update A2 in cache* *a3: read B1 and cache* *a4: read A1 and cache* *a5: update B1 in cache* *a6: read C2, cache and write-back A2* *a7: read C3, cache and write-back B1* *a8: update A1 in cache*	*9 accesses:* *a1: read and cache A1 & A2* *a2: hit A1 and update A2 in cache* *a3: read and cache B1* *a4: hit A1* *a5: update B1 in cache* *a6: read and cache C1 & C2, write A2* *a7: hit C1 & C2, read and cache C3, write B1* *a8: read and cache A1,replace C1*

c) To have persistence in case of crashes, a periodic writeback of dirty blocks in the cache must be factored in. Assume that after the fifth access in the trace, we perform a periodic writeback of all dirty elements in the cache. How does this affect the number of disk accesses for the index and linked file systems?

Indexed File System:	Linked File System:
7 accesses as before: *a1: read A2 and cache* *a2: update A2 in cache* *a3: read B1 and cache* *a4: read A1 and cache* *a5: update B1 in cache* *-> write back A2 and B1* *a6: read C2 and cache, replace A2* *a7: read C3 and cache, replace B1* *a8: update A1 in cache*	*9 accesses as before:* *a1: read and cache A1 & A2* *a2: hit A1 and update A2 in cache* *a3: read and cache B1* *a4: hit A1* *a5: update B1 in cache* *-> write back A2 and B1* *a6: read and cache C1 & C2, replace A1, A2* *a7: hit C1 & C2, read and cache C3* *a8: read and cache A1,replace C1*

d) How does the write back frequency affect file system performance and reliability?

Less frequent write-backs increase likelihood of data losses in case of crashes.

More frequent write-backs decrease performance, as blocks may be written back more often than they need to be, e.g. in case of successive write updates of the same cached block.

Problem 5 (15 points): Virtual Memory

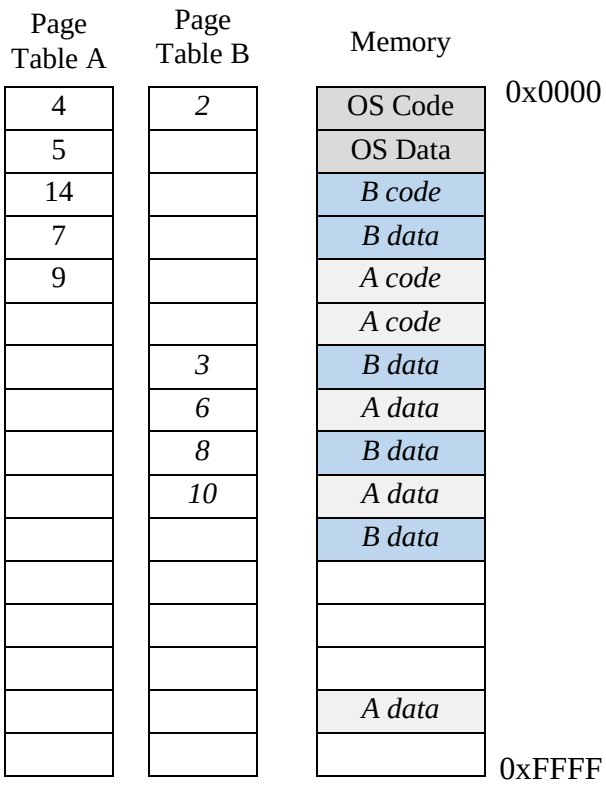
Assume a computer with a 16-bit memory address space using virtual memory with 4kB pages. Given the two ELF executable files containing code and data segments with sizes and load addresses as indicated, the partial state of main memory, and the page table of the process executing program *A* after it has been loaded into memory. No other process is loaded at this point:

Program A:

	Load address	*Size*
Code	0x0000	6kB
Data	0x2000	12kB

Program B:

	Load address	*Size*
Code	0x0000	4kB
Data	0x6000	13kB



a) What is size of each page table per process?

64kB/4kB = 16 pages, needing 4 bits to encode page number/frame, 12 bits for offset.

Hence, every page table entry is 1 byte (assuming not more than 4 additional bits for meta-data).

*As such, page table is 16 * 1B = 16B.*

b) Show the location of the code and data segments for process A in memory. Is there any internal or external fragmentation? What is the largest program (code and data size) that can be loaded?

*There is no external fragmentation, there are 9 unallocated pages and the largest program (code+data) that can be loaded is 9 * 4kB = 36kB.*

There is internal fragmentation, however. A's code segment takes up two pages, with 2kB wasted space in the second page.

c) Now assume that program *B* is also loaded into memory. Fill the page table for the corresponding process and show the updated memory state after *B* has been loaded. Assume that memory is allocated in a first available fashion from bottom to top. Is there any external or internal fragmentation now? What is the largest program that can now be loaded?

*Still no external fragmentation, 4 pages * 4kB = 16kB can still be allocated.*

Additional internal fragmentation for B's data segment not fitting evenly into 4 pages.

Problem 6 (10 points): Relocation

For each of the following functions written in assembly, is the code position-independent? If not, indicate which part of the code is not and what needs to be done to relocate it at load time.

a)
```
funcA
  LDR R1,=count
  LDR R0,[R1]
  ADD R0,R0,#1
  SVC #42
  BX  LR
```

Not position-independent, the count variable is accessed using a hard-coded address.

Requires patching/dynamic linking of count's address for load-time relocation.

b)
```
funcB
  ADD R1,R9,#32
  LDR R0,[R1]
  ADD R0,R0,#1
  BL  OS_Sleep
  BX  LR
```

Not position-independent.

Data accesses are, assuming that R9 is initialized properly.

However, the call to OS_Sleep uses a hard-coded branch offset. Requires patching/dynamic linking during load-time-relocation to adjust branch offset. This may fail if OS_Sleep is too far away and not reachable via a regular branch.

Problem 7 (15 points): Networking

Assume that two computers are directly connected in a local Ethernet network running at a raw physical layer bit rate of 10Mbit/s.

a) What is the maximum achievable bandwidth for transmitting data from one machine to the other at the Ethernet link layer? Assume that no other machines are transmitting, i.e. there are no collisions.

In the best case, an Ethernet frame can at maximum have a payload of 1500 bytes within a total frame length of 1526 bytes.

*Effective bandwidth = 1500 bytes / 1526 bytes * 10Mbit/s = 9.83Mbit/s = 1.23MB/s*

b) Now assume that we are running a TCP/IP protocol over the local network. What is the maximum achievable bandwidth for transmitting data from one machine to the other at the IP layer? Again, assume that no other machines are transmitting, i.e. there are no collisions and that the IP protocol does not use any options.

IP header adds 20 bytes overhead per packet, leaving only 1480 bytes for real payload:

*Bandwidth = 1480 bytes / 1526 bytes * 10Mbit/s = 9.67Mbit/s = 1.21MB/s*

c) Finally, what is the maximum achievable bandwidth for transmitting data from one machine to the other over a UDP and TCP connection? Assume that the TCP connection is already established and that there are no buffering, windowing or acknowledgment delays.

UPD adds 8 byte overhead, TCP adds 20 bytes:

*UDP: 1472 bytes / 1526 bytes * 10Mbit/s = 9.65Mbit/s = 1.21MB/s*

*TCP: 1460 bytes / 1526 bytes * 10Mbit/s = 9.57Mbit/s = 1.19MB/s*