Jonathan W. Valvano First Name: _______________ Last Name:____________________
May 13, 2017, 9am-12n Closed book part

(25) Question 1. For each definition, select the term that best describes it. Not all words will be used. Place the corresponding numbers into the boxes.

1. Aging
2. Aliasing
3. Anti-Reset-Windup
4. Atomic
5. Bank-Switched Memory
6. Big Endian
7. Board Support Package
8. Bounded Waiting
9. Brushed DC motor
10. Burst DMA
11. Central Limit Theorem
12. Content Addressable Memory
13. Cooperative Nonpreemptive scheduler
14. Crisp Input
15. Critical Section
16. Cycle Steal DMA
17. Deadlock
18. Dual Address DMA
19. External Fragmentation
20. Firm real time
21. Flash Memory
22. Full Duplex Channel
23. Half Duplex Channel
24. Hard real time
25. Hook
26. Internal Fragmentation
27. Little Endian
28. Little's Formula
29. Minimally Intrusive
30. Mutual Exclusion
31. Nyquist Theorem
32. Path Expression
33. Preemptive scheduler
34. Priority Inversion
35. Random Access Memory
36. Reentrant function
37. Priority Scheduler
38. Pulse width modulation
39. Semaphore initialization
40. Servo
41. Simplex Channel
42. Single Address DMA
43. Slew Rate
44. Soft real time
45. Stabilization
46. Stuff Bits
47. Utilization factor
48. Velocity Factor

40	A DC motor with built-in controller. The microcontroller specifies desired position and the motor adds/subtracts power to move the shaft to that position.
12	A storage device that takes as input the data, and creates as output the address at which this data is located.
47	Throughput (actual number of packets per second) divided by the capacity (maximum capacity the system can handle in packets per second).
20	A system that expects all critical tasks to complete on time. Once a deadline as passed, there is no value to completing the task. However, the consequence of missed deadlines is real but the overall system operates with reduced quality.
24	A system that can guarantee that a process will complete a critical task within a certain specified range. There is an upper bound on the latency between when a task is supposed to be performed and when it is actually performed.
44	A system that implements best effort to execute critical tasks on time, typically using a priority scheduler. Once a deadline as passed, the value of completing the task diminishes over time.

21	A type of memory such that when you perform a write cycle to it, you can cause bits to go from 1 to 0, but not 0 to 1.
13	A scheduler that cannot suspend execution of a thread without the thread's permission. The threads suspend themselves at times convenient for the thread.
7	A set of software routines that abstract the I/O hardware such that the same high-level code can run on multiple computers.
36	A software function that can be started by one thread, interrupted and executed by a second thread.
32	A software technique to guarantee subfunctions within a module are executed in a proper sequence. For example, it forces the user to initialize I/O device before attempting to perform I/O.
1	A technique used in priority schedulers that temporarily increases the priority of low priority treads so they are run occasionally.
10	An I/O synchronization scheme that transfers an entire block of data all at once directly from an input device into memory, or directly from memory to an output device.
46	Method used in CAN to synchronize in conditions when long strings of zeros are sent, or when only strings of ones are sent.
25	An indirect function-call added to a software system that allows the user to attach their programs to run at strategic times. These attachments are created dynamically at run time and do not require recompiling the entire system.
28	The average number of packets in the system is equal to the average arrival rate in packets per second multiplied by the average response time of a packet.
18	Direct memory access that requires two bus cycles to transfer data from source to destination. The first cycle brings data from the source into the DMA controller, and the second sends the data to the destination
31	Used to determine the minimum sampling rate required to faithfully represent a signal in digital form.
41	A communication channel that allows bits (information, error checking, synchronization or overhead) to transfer only in one direction.
15	Locations within a software module, which if an interrupt were to occur at one of these locations, then an error might occur (e.g., data lost, corrupted data, program crash, etc.)
6	Mechanism for storing multiple byte numbers such that the most significant byte exists first in the smallest memory address.
26	Storage that is allocated for the convenience of the operating system but contains no information. This space is wasted.
48	The ratio of the speed at which information travels relative to the speed of light.
30	Thread synchronization where at most one thread at a time is allowed to enter at a time.
4	Software execution that cannot be divided or interrupted.

(5) Question 2. What are the three necessary conditions to cause deadlock?

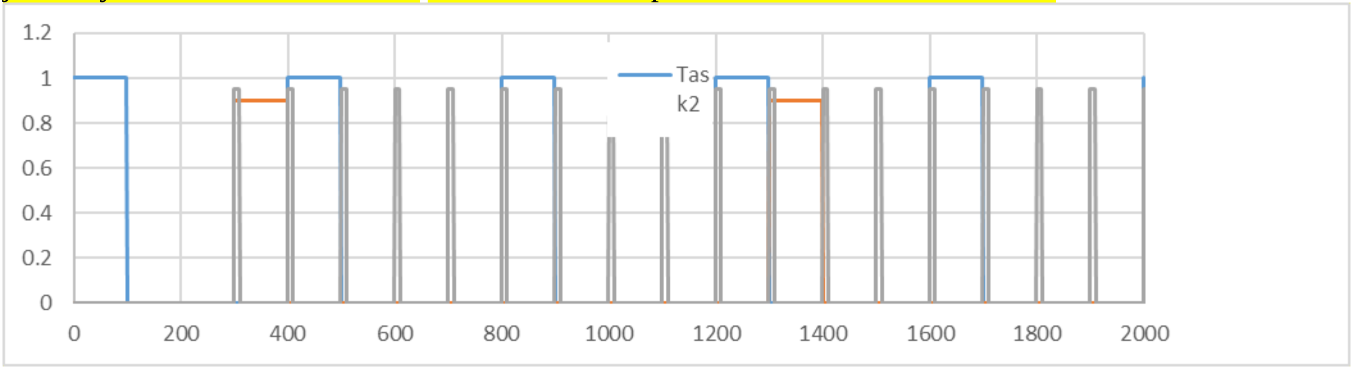
Hold and wait, circular wait, no preemption

(5) Question 3. You are designing a **real-time scheduler** for this system. There are three periodic tasks that have minimal interaction with each other.

Task 1:	Executes every 1000 µs,	execution time varies from 5 to 100µs.
Task 2:	Executes every 400 µs,	execution time varies from 10 to 100µs.
Task 3:	Executes every 100 µs,	execution time varies from 1 to 10µs.

Without actually writing the scheduler, you can determine whether or not a real-time solution is likely. Is it possible to schedule these tasks? If no, prove it. If yes, justify your answer.

(10% + 25% + 10% = 45%, yes it should be possible according to the Rate Monotonic Theorem). However, because the periods of the tasks overlap, there is ACTUALLY NO scheduler that runs without jitter if you were to search for it. Here is one example schedule that does not work



(5) Question 4. Explain how an operating system can implement **position independent data** on the Cortex M. The concept is similar to position independent code, but for data accesses.
R9 as static base (SB) register, must point to base address of data/RW segment. All references use offsets added to R9/SB. The offsets implement position independent data.

```
LDR  r1,[r9,#ofs]
...
LDR  r0,=ofs
ADD  r0,r9,r0
LDR  r0,[r0]
```

(5) Question 5. Give three different reasons for implementing **paging** in a multi-process operating system.
Eliminate external fragmentation, implement virtual memory, and provide protection (another possibility is it simplifies relocation)

(15) Question 6. Write C code for a FIFO queue that can be used to pass 8-bit data between foreground threads. None of the FIFO functions will be called from an interrupt service routine. You must write all of the FIFO code. There will be multiple producers and multiple consumers running in the foreground using a preemptive scheduler accessing this one FIFO. You can define semaphores by adding globals:

```
long semaphore=0;
```

You may call the following two blocking semaphore functions without showing their implementations.

```
void Wait(long *semaPt);
void Signal(long *semaPt);
```

You must use these following private globals. Other than semaphores, you may not add any additional global variables.

```
#define FIFOSIZE 10
uint8_t static volatile PutI;  // index to put next
uint8_t static volatile GetI;  // index to get next
uint8_t static Fifo[FIFOSIZE];
```

Part a) Show the semaphores needed. Use good names

```
long DataRoomLeft;
long DataAvailable;
long Mutex=1; // load time initialization
```

Part b) Show the initialization code that configures the FIFO and initializes the semaphores

```
void Fifo_Init(void){
  Wait(&Mutex);            // this is critical
  PutI = GetI = 0;         // Empty
  DataRoomLeft = FIFOSIZE; // size of queue
  DataAvailable = 0;       // number currently in FIFO
  Signal(&Mutex);          // end of critical section
}
```

Part c) Show the function that stores into the FIFO. A producer thread should block on full.

```
void Fifo_Put(uint8_t data){
  Wait(&DataRoomLeft);    // wait for space
  Wait(&Mutex);           // this is critical
  Fifo[PutI] = data;      // save in FIFO
  PutI = (PutI+1)%SIZE;   // next place to put
  Signal(&Mutex);         // end of critical section
  Signal(&DataAvailable); // one more entry
}
```

Be careful not to switch Wait mutex with other wait doing so causes a deadlock

Part d) Show the function that retrieves from the FIFO. A consumer thread should block on empty.

```
uint8_t Fifo_Get(void){ uint8_t data;
  Wait(&DataAvailable);     // wait for data
  Wait(&Mutex);             // this is critical
  data = Fifo[GetI];        // get data
  GetI = (GetI+1)%SIZE;     // next place to get
  Signal(&Mutex);           // end of critical section
  Signal(&DataRoomLeft);    // more space
  return data;
}
```

Jonathan W. Valvano First Name: _______________ Last Name:_____________________
Open book part

Open book, open notes, calculator (no laptops, phones, devices with screens larger than a TI-89 calculator, devices with wireless communication). Please don't turn in any extra sheets.

(10) Question 7. A **barrier** for a group of threads is a place in the code where the thread must stop and cannot proceed until all other threads reach their barriers. You can define and initialize semaphores by adding globals like this.

```
long semaphore=0;
```

You may call the following two blocking semaphore functions without showing their implementations.

```
void Wait(long *semaPt);
void Signal(long *semaPt);
```

Other than semaphores, you may not add any additional global variables.
Part a) Define the semaphores needed, including their initial values.

```
long s12=0;  // task1 signals task2
long s13=0;  // task1 signals task3
long s23=0;  // task2 signals task3
long s21=0;  // task2 signals task1
long s31=0;  // task3 signals task1
long s32=0;  // task3 signals task2
```

second answer

```
long done1=0;  // task1 is done
long done2=0;  // task2 is done
long done3=0;  // task3 is done
```

Part b) Place a barrier between the **`start`** and **`end`** functions in each thread to implement this three-thread barrier. Basically, the threads will not execute their corresponding **`end`** functions until all threads have executed their **`start`** functions. You may assume this sequence executes just once.

```
void thread1(void){
 start1();

 Signal(&s12);
 Signal(&s13);
 Wait(&s21);
 Wait(&s31);
```

second answer

```
 Signal(&done1);
 Wait(&done3);


 end1();
 OS_Kill();
}
```

```
void thread2(void){
  start2();

 Signal(&s21);
 Signal(&s23);
 Wait(&s12);
 Wait(&s32);
```

second answer

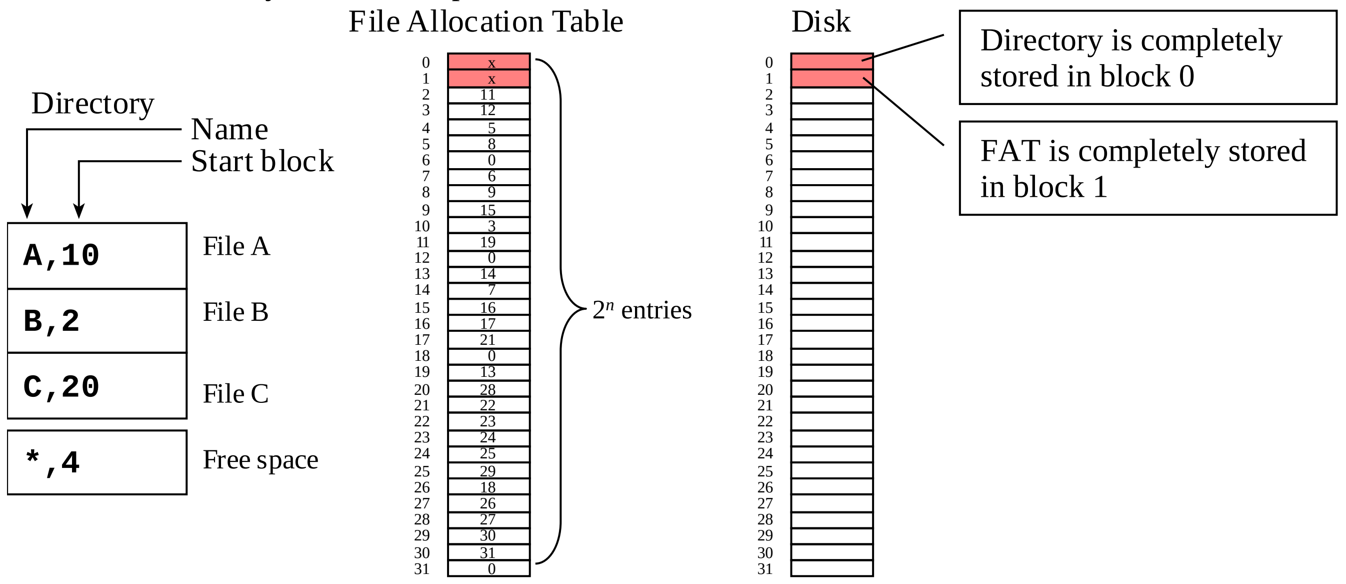
```
 Wait(&done1);
 Signal(&done2);


  end2();
  OS_Kill();
}
```

```
void thread3(void){
  start3();

 Signal(&s31);
 Signal(&s32);
 Wait(&s13);
 Wait(&s23);
```

second answer

```
 Wait(&done2);
 Signal(&done3);


  end3();
  OS_Kill();
}
```

(10) Question 8. Consider a file system that uses a FAT. There are 2^n entries in the FAT, and each entry is 4 bytes (32 bits). Each disk block contains $4*2^n$ bytes, meaning the entire FAT will always fit in one block. Assume the FAT entry size is always 32 bits. It is shown as n=5 in the figure, but n could be larger. The directory is in block zero, the FAT is in block 1. Each directory entry contains a file name, file size, and an index to the first FAT entry for that file. The last entry in the directory contains an index to the first FAT entry for the free space.



Part a) If each directory entry requires 16 bytes of information (file name, file size, and starting FAT index), then what is the maximum number of files that can be stored on this disk? Solve in general for any $n \le 16$ (partial credit: solve for n=5).

$\boxed{2^{n-2} - 1}$

(2^2*2^n bytes in directory)/(2^4 bytes/entry)= 2^{n-2} entries
One entry is free space

E.g., n=5, block size 128 bytes, there are 128/16=8 entries in directory, so 7 files are possible.

Part b) What is the maximum size of this disk including directory and FAT? Solve in general for any n.

There are 2^n entries in the FAT, so 2^n possible blocks
Each block is $4*2^n$ bytes, so max disk is 2^{2n+2} bytes
E.g., n=5, block size 128 bytes, there are 32 entries in FAT,
128*32 bytes in disk = 2048 bytes

2^{2n+2} total bytes or
$(2^n-2)2^{n+2}$ data bytes

(extra credit) Part c) Assuming the FAT entries remain 32 bits, and one block for the FAT, what is the largest value of n possible, such that the entire disk is accessible?

From b) the largest disk is 2^{2n+2} bytes. If the FAT entry is 32 bits, there can be at most 2^{32} blocks. Each block is $4*2^n$ bytes, so the disk can be at most 2^{n+34} bytes.
To make it fit $2^{2n+2} \le 2^{n+34}$, so $2n+2 \le n+34$, so $n \le 32$

$\boxed{n \le 32}$

(20) Question 9. In this question you will implement **blocking semaphores with bounded waiting**. The OS has the following TCB structure, and it cannot be changed.

```
struct TCB {
  long *stackPointer;   // pointer to top of stack
  struct TCB *Next;     // linked list
  long *BlockPt;        // nonzero if blocked on this semaphore
  uint64_t BlockTime;   // time when this thread was blocked
};
typedef struct TCB TCBType;
typedef TCBType * TCBPtr;
TCBPtr RunPt;           // Pointer to tcb of thread currently running
```

The OS uses a signed 32-bit integer for semaphores (`long`), which also cannot be changed. There is an `OS_Time` function that returns the current time as a 64-bit unsigned integer with units of 12.5ns. You may assume this time never rolls over (i.e., the system runs for less than 664 years). The prototype is

```
uint64_t OS_Time(void);
```

This is the ISR thread switch, Program 3.11 in the book, and it cannot be modified

```
SysTick_Handler                  ; 1) Saves R0-R3,R12,LR,PC,PSR
    CPSID   I                    ; 2) Prevent interrupt during switch
    PUSH    {R4-R11}             ; 3) Save remaining regs r4-11
    LDR     R0, =RunPt           ; 4) R0=pointer to RunPt, old thread
    LDR     R1, [R0]             ;    R1 = RunPt
    STR     SP, [R1]             ; 5) Save SP into TCB
    PUSH    {R0,LR}
    BL      Scheduler
    POP     {R0,LR}
    LDR     R1, [R0]             ; 6) R1 = RunPt, new thread
    LDR     SP, [R1]             ; 7) new thread SP; SP = RunPt->sp;
    POP     {R4-R11}             ; 8) restore regs r4-11
    CPSIE   I                    ; 9) tasks run with interrupts enabled
    BX      LR                   ; 10) restore R0-R3,R12,LR,PC,PSR
```

This is the scheduler, and it cannot be modified

Part a) Implement `OS_Wait`, which has the following prototype.

```
void OS_Wait(long *semaPt){
long sr;
  sr= StartCritical();     // make atomic
  (semaPt->Value)--;
  if(semaPt->Value < 0){
    RunPt->BlockPt = semaPt;      // block
    RunPt->BlockTime = OS_Time(); // time this was blocked
    OS_Suspend();                 // this thread stops running
  }
  EndCritical(sr);  // end critical section
}
```

Part b) Implement `OS_Signal`, which has the following prototype.

```
void OS_Signal(long *semaPt){
long sr; TCBPtr pt;                    // search pointer
TCBPtr oldestPt=0;
uint64_t oldestTime=0xffffffffffffff;
  sr= StartCritical();     // Test and set is atomic
  (semaPt->Value)++;
  if(semaPt->Value < 1){
    pt = RunPt->Next;
    while(pt != RunPt){ // look at them all
      if(pt->BlockPt == semaPt){ // blocked on this thread?
        if(pt->BlockTime < oldestTime){
          oldestTime = pt->BlockTime;
          oldestPt = pt;
        }
      }
      pt = pt->Next; // find oldest blocked on this semaphore
    }
    if(oldestPt){
      oldestPt->BlockPt = 0; // wakeup oldest one
    }else{}// crash, something bad happened
  }
  EndCritical(sr);  // end critical section
}
```

Second answer

```
long sr; TCBPtr pt;                    // search pointer
TCBPtr oldestPt=0;
  sr= StartCritical();     // Test and set is atomic
  (semaPt->Value)++;
  if(semaPt->Value < 1){
    pt = RunPt->Next;
    while(pt != RunPt){ // look at them all
      if((pt->BlockPt==semaPt)&&(pt->BlockTime < oldestPt->BlockTime){
        oldestPt = pt;
      }
      pt = pt->Next; // find oldest blocked on this semaphore
    }
    if(oldestPt){
      oldestPt->BlockPt = 0; // wakeup oldest one
    }else{}// crash, something bad happened
  }
  EndCritical(sr);  // end critical section
}
```