Dynamic memory allocation, counting semaphores

1) [20 pts] The following functions (modified from Valvano p. 122) implement a fixed sized block memory allocation scheme. Use a TI-RTOS GateMutexPri object to make Heap_Allocate() and Heap_Release() reentrant and free of shared data issues (for Tasks only). In addition, make Heap_Allocate() block whenever there is no free memory available, rather than returning NULL. Freeing memory should unblock any Task waiting for free memory. Indicate how your semaphore(s) must be initialized. You do not need to modify Heap_Init(), which is meant to be called before any Tasks start.

```
#define SIZE 80 // size of memory block
#define NUM 5   // number of memory blocks available for allocation
#define NULL 0  // empty pointer
char *FreePt;         // shared globals
char Heap[SIZE*NUM];

void Heap_Init(void) {
  char *pt;
  FreePt = &Heap[0];
  for(pt = &Heap[0]; pt != &Heap[SIZE*(NUM-1)]; pt += SIZE) {
    *(char**)pt = pt + SIZE;
  }
  *(char**)pt = NULL;
}

void *Heap_Allocate(void) {
  char *pt;
  pt = FreePt;       // accesses to globals highlighted
  if (pt != NULL) {
    FreePt = *(char**)pt;
  }
  return(pt);
}

void Heap_Release(void *pt) {
  char *oldFreePt;
  oldFreePt = FreePt;
  FreePt = (char*)pt;
  *(char**)pt = oldFreePt;
}
```

(solution on the next page)

For mutually exclusive access to the heap globals, create a GateMutexPri object called heapGate.
For blocking a Task when no free memory is available, create a counting semaphore called heapSem and initialize it to count = 5 (the number of memory blocks available for allocation). Unblocking order depends on the application (OK if you leave it unspecified).

```
void *Heap_Allocate(void) {
  char *pt;
  IArg key;
  Semaphore_pend(heapSem, BIOS_WAIT_FOREVER); //block if no free mem
  key = GateMutexPri_enter(heapGate);
  pt = FreePt;       // accesses to globals highlighted
  if (pt != NULL) { // this condition may be removed (always true)
    FreePt = *(char**)pt;
  }
  GateMutexPri_leave(heapGate, key);
  return(pt);
}

void Heap_Release(void *pt) {
  char *oldFreePt;
  IArg key;
  key = GateMutexPri_enter(heapGate);
  oldFreePt = FreePt;
  FreePt = (char*)pt;
  *(char**)pt = oldFreePt;
  GateMutexPri_leave(heapGate, key);
  Semaphore_post(heapSem);                // release 1 memory block
}
```

Real-time benchmarks

2) [25 pts] To give developers an idea of the performance of TI-RTOS, a set of benchmarks is included for several CPU architectures and compilers. The Cortex-M4F CCS benchmarks are given to you in PDF form together with this HW assignment. The benchmarks are explained in Appendix B of the TI-RTOS User's Guide. Using this information, answer the following. Express timing results in clock cycles and in microseconds given a 40 MHz CPU clock. Assume no code other than TI-RTOS disables interrupts anywhere.

```
void Hwi1(UArg arg0) {
  // Hwi latency is from the interrupt to this point in the code
  <clear interrupt flag>; // assume this takes 4 clock cycles
  Semaphore_post(sem0);
}

void Task1(UArg arg0, UArg arg1) {
  <init>;
  while (1) {
    Semaphore_pend(sem0, BIOS_WAIT_FOREVER);
    // Task latency is from the interrupt to this point in the code
    <run periodic code>;
  }
}
```

a) [10 pts] What is the latency of the highest-priority, TI-RTOS dispatched Hwi? Note that this is not the same as the interrupt latency.

Hwi latency is from the interrupt to the start of the Hwi function. Its components are the interrupt latency (mainly the longest time with interrupts disabled) and Hwi dispatcher prolog (execution time of the Hwi dispatcher until the user Hwi function is called).

From the TI-RTOS User's Guide:

Interrupt Latency

The Interrupt Latency benchmark is the maximum number of cycles during which the SYS/BIOS kernel disables maskable interrupts. Interrupts are disabled in order to modify data shared across multiple threads. SYS/BIOS minimizes this time as much as possible to allow the fastest possible interrupt response time.

The Interrupt Latency is measured within the context of a SYS/BIOS application containing SYS/BIOS API calls that internally disable interrupts. A timer is triggered to interrupt on each instruction boundary within the application, and the latency is calculated to be the time, in cycles, to vector to the first instruction of the interrupt dispatcher. Note that this is not the time to the first instruction of the user's ISR.

Interrupt prolog for calling C function. This is the execution time from when an interrupt occurs until the user's C function is called.

Hwi latency = Interrupt latency + Hwi dispatcher prolog = 135 + 113 = <u>248 cycles</u>

Hwi latency = (248 cycles) / (40 MHz) = <u>6.2 µs</u>

b) [15 pts] What is the latency of the highest priority Task that blocks on a signaling semaphore? Assume only a single Hwi exists in the system: the one that posts to the semaphore to unblock the highest-priority task. The signaling code is below.

There are two approaches to estimating the task latency:

1. **Account for the semaphore post and task switch (in Hwi epilog) separately.**
2. **Use the "Hardware Interrupt to Blocked Task" benchmark (see figure B-1).**

Hardware interrupt to blocked task. This is a measurement of the elapsed time from the start of an ISR that posts a semaphore, to the execution of first instruction in the higher-priority blocked task, as shown in Figure B–1.

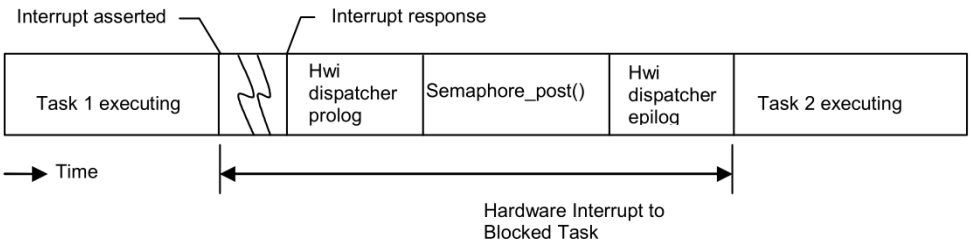

Figure B–1 Hardware Interrupt to Blocked Task

Approach #1:
Task latency = Hwi latency + time to clear interrupt flag + Post Semaphore No Task Switch + Hwi dispatcher epilog = 248 + 4 + 151 + 208 = <u>611 cycles</u>
Task latency = (611 cycles) / (40 MHz) = <u>15.3 µs</u>

Approach #2:
Task latency = Interrupt latency + time to clear interrupt flag + Hardware Interrupt to Blocked Task = 135 + 4 + 488 = <u>627 cycles</u>
Task latency = (627 cycles) / (40 MHz) = <u>15.7 µs</u>

The interrupt latency has been added separately because the User's Guide isn't clear exactly what type of measurement "Hardware Interrupt to Blocked Task" is. It may or may not include the interrupt latency and the Hwi dispatcher prolog. The easiest way to measure this parameter is by triggering an interrupt in software (read timer; trigger interrupt; read timer after semaphore pend in the unblocked task; subtract timer readings). This measurement will not include the interrupt latency, but will include the Hwi dispatcher prolog. The close match to Approach #1 indicates that this is indeed the method used.

Grading note: The answer is acceptable even if the interrupt latency (135 cycles) is not taken into account, or if the Hwi dispatcher prolog is added in. For approach #2, the "time to clear interrupt flag" may also be omitted. This is due to the ambiguous explanation in the TI-RTOS User's Guide.

Button debouncing

3) [28 pts] Determine the output gButtons of the Lab 0 button debouncing routine for the following two sequences of inputs (for one button). A step in the sequence is the argument of one call to ButtonDebounce() executing every 5 ms. Assume the first value in the sequence has been recurring for a long time. What button event does each sequence represent?

   a) [8 pts] ...0, 1, 0, 0, 0

      Output: 0, 0, 0, 0, 0

This sequence is a <u>glitch</u> in the button input, potentially caused by EMI. Superficially it looks like the button is pressed momentarily, but it is normally not possible to press a button for only 5 ms.

The easiest way to analyze this is by running ButtonDebounce() in test code. Alternatively, one could manually keep track of state[0]: Increment it by 5 for every input = 1, decrement by 2 for every input = 0. Restrict to the range 0...10. 0 = change state to 'not pressed,' 10 = change state to 'pressed.'

state[0] = 0, 5, 3, 1, 0

At no point did piState[0] reach 10, the button pressed state.

   b) [8 pts] ...0, 1, 1, 0, 0, 0, 0, 0, 1, 1

      Output: 0, 0, 1, 1, 1, 1, 1, 0, 0, 1

      state[0] = 0, 5, 10, 8, 6, 4, 2, 0, 5, 10

This sequence is <u>two button presses</u> (switch closure events) occurring in rapid succession, but still recognized by the debouncing routine as separate.

   c) [12 pts] Suppose you decided that part (b) should represent a single button press event. Which of the tuning constants in buttons.h would you modify? Fast button press response takes priority over fast button release response. Explain your reasoning.

The tuning constants in buttons.h are BUTTON_SAMPLES_RELEASED (default 5) and BUTTON_SAMPLES_PRESSED (default 2). These represent how many stable samples are required to determine the debounced state of the button (5 samples for button released, 2 samples for button pressed). Since our priority is fast button press response, we should <u>increase</u> BUTTON_SAMPLES_RELEASED, which <u>lengthens the ramp time from pressed to not pressed</u>. In this case, increasing BUTTON_SAMPLES_RELEASED to <u>6</u> is sufficient, and changes the output to a single button press:

state[0] = 0, 6, 12, 10, 8, 6, 4, 2, 8, 12
Output: 0, 0, 1, 1, 1, 1, 1, 1, 1, 1

Timer I/O

4) [27 pts] A 24-bit timer is being used in capture mode to measure the period of an input waveform. The following ISR places the measured period into a global variable. When the period of the input waveform decreases below 16 μs, you notice occasional erroneous period measurements that are too large.

```
uint32_t period; // measured period in clock cycles

void TimerCaptureISR1(void) { // ISR for capture interrupts
    static uint32_t last_count = 0;
    uint32_t count;

    <clear timer capture interrupt flag>;
    count = <read captured timer count>;
    period = (count - last_count) & 0xffffff;
    last_count = count;
}
```

a) [15 pts] What is the most likely cause of these errors? How much do you expect an erroneous measurement to deviate from the actual period? Assume the ISR execution time is much less than 16 μs.

16 μs is a relatively short period, which is also equal to the relative deadline of TimerCaptureISR1(). When the relative deadline shortens below a certain point, the erroneous period measurements are most likely due to missing the deadline and skipping one interrupt. Since the capture interrupts are periodic, by skipping one interrupt, the measurement is twice the actual period.

b) [12 pts] How could the valid period measurement range be extended down to 8 μs? Propose two approaches to try (not necessarily guaranteed to work). You may change the hardware settings, or even add (inexpensive) hardware.

1. **Increase the priority of TimerCaptureISR1(). Since the ISR execution time is much less than 16 μs, the culprit is latency. It may be possible to achieve lower latency by increasing the ISR priority, but this may cause another ISR to miss deadlines.**
2. **High latency could also be caused by a critical section that disables interrupts. In such a case, optimize the critical section that causes high latency.**
3. **Pass the input signal through a frequency divider (prescaler). A single flip-flop is sufficient for frequency division by 2, enough to measure the 8 μs period. Some microcontrollers feature built-in prescalers for timer inputs. Software can then take the frequency division into account when calculating the period.**
4. **Install a second microcontroller dedicated to period measurement. This is also a solution in the case that TimerCaptureISR1() causes too high a CPU load for the shortest period input waveform. The acquired data will need to be transferred to the main microcontroller.**

Grading note: Accept other plausible approaches.