Part I: Short Questions

1. Assume we divide the heap into fixed-sized units of size $S$. Also assume that we never get a request for a unit larger than $S$. Thus, for any request you receive, you will return a single block of size $S$, if you have any free blocks. What kind of fragmentation can occur in this situation, and why is this bad?

2. Describe a piece of hardware the the OS uses to implement memory management. Why is that hardware needed?

3. Assume we manage a heap in a best-fit manner. What does best-fit management try to do? What kind of fragmentation can occur in this situation, and why is this bad?

4. Why is stack-based memory management usually much faster than heap-based memory management? If it’s so much faster, why do we ever use the heap?

5. Assume we have a system that performs dynamic relocation of processes in physical memory. A process that has just started needs 2000 total bytes of memory in its address space. The OS currently has two free regions: the first between physical addresses of 1000 and 2000, and the second between physical addresses 5000 and 7000. What value would end up in the base register for this process, and what value would be in the bounds register? (write down any assumptions that you make about the bounds register)

6. Describe the LRU memory management policy. What kind of hardware support do we need to implement “perfect” LRU management efficiently?

7. In a system with pure paging, assume we have a 32-bit address space, and a 4 KB page size.
   a) How many bits of an address specify the logical page number (a.k.a. the virtual page number), and how many bits specify the offset?
   b) Let’s say we are translating the logical address 0x00010033; if each logical page is mapped to a physical page that is a single page number higher (i.e., logical page 10 is mapped to physical page 11, logical page 11 is mapped to physical page 12), what is the final translated physical address?

8. Someone has written new memory allocator to replace the standard malloc()/free() implementation. It works as follows: one half of available memory is divided into fixed-sized units of 4KB, and the other half is managed by a best-fit free list. If an allocation request is less than or equal to 4KB and there is space in the fixed-sized half, a 4KB unit is allocated from the fixed-sized half; otherwise, the best-fit algorithm is used over the other half of memory, and the requested size is returned (if space is available). Assuming 32KB of total memory is available, what is the shortest series of allocation requests that lead to all of memory getting allocated? What is the minimum amount of memory requested that will lead to all of memory being allocated? What type of fragmentation occurs with the new allocator?

9. For the following question, please circle all answers that apply. A translation lookaside buffer (TLB) is generally used to:
   a) translate virtual page numbers into physical page numbers
   b) translate physical page numbers into virtual page numbers
   c) make segmentation have the benefits of a pure paging approach
   d) translate the addresses generated by loads
   e) translate the addresses generated by stores
   f) translate the addresses generated by instruction fetches
   g) remove the need for a full-sized page table
   h) make translations happen quickly
Part II: Longer Questions

1. Staying In-Bounds.

You are dealing with a system that performs static relocation. In static relocation, a loader rewrites the addresses of a process as it is getting loaded into the system so as to “relocate” the address space of that process to an arbitrary address in physical memory. In this system, all programs are compiled as if they will get loaded at address 1000. Then, when the loader is “loading a process”, it must re-write any addresses within the program in order to generate addresses at the correct offset in physical memory.

```assembly
load 0(R1), R2  # loads value at address ‘R1 + 0’ into R2
add R2, 5, R2   # add 5 to R2
store 0(R1), R2 # store value at address ‘R1 + 0’ back into R2
```

a): Assuming that the process gets loaded at physical address 2500, how would the loader re-write the statements above so as to provide proper static relocation?

b): Let’s say we want to implement some additional checks in our static relocation scheme. Specifically, we want to make sure that all addresses generated by the process do not extend beyond its address space. If an address is outside of the limit, the program should just be forced to exit. What would we have to do before each load and store instruction in order to guarantee that they stay within the address space of the process?

c): In contrast with static relocation, dynamic relocation is a hardware approach to relocating the address space of a process in physical memory. What hardware is required to implement dynamic relocation?

d): If you contrast software-based static relocation with the extra checks (as described in this question in part (b)) to traditional hardware-based dynamic relocation, are they equivalent, or does one approach give you more capabilities than the other? Explain.

2. Your Losing Your Memory (20 points)

In this problem, you have to design a virtual memory system for TinyOS, an operating system for a handheld device. Unlike a traditional modern OS, you don’t have a lot of memory to fool around with, and so saving space is important.

For this question, assume that your TinyOS is running on a handheld with 1K pages, and on this handheld in particular, there is 128KB of physical memory. Assume addresses (virtual and physical) are 20 bits long.

a) Let’s say we want to use a single linear array as the page table of a process. Assume that each page table entry needs, in addition to any translation information, 2 bits for protection information, and 4 other bits for miscellaneous stuff. How much memory will the page table occupy?

b) Now let’s try a different structure to track translations, called MiniTable. MiniTable is a linear array that is proportional to the size of physical memory, with one entry per physical page. Each entry specifies which process has the page mapped (i.e, a process ID), as well as the virtual page number of the page that maps to this physical page. How much space does a MiniTable occupy? (assume that a PID is 6 bits in length)

c) One problem with MiniTable is the time overhead to perform a translation. How much time does a virtual to physical translation lookup take in MiniTable, as compared to the linear array described in part (a)? Please state your assumptions.

d) Another problem with MiniTable is that it does not easily allow processes to share a page. Describe why this is, and how you might fix MiniTable to allow sharing.