5 problems. 50 points total. Closed book, closed notes, no electronic devices. Write your name above. _____

1. [10 points] In GeekOS project 4, if a user thread makes a function call with an argument of size more than 1 page, the thread is terminated. This is because the paging system allows a user thread to access an address at most one page away from the top of its stack.

a. [4 pts]
Describe briefly how you can change the OS to overcome this restriction (i.e., allow arguments of larger than 1 page). You cannot make changes to the user code (user program or compiler).

Solution:
Change the paging system code to allow user to allocate for a virtual page at an address x more than one page away from the top of stack. Can also allocate for the virtual pages in between x and top of stack at the same time.

b. [4 pts]
Describe briefly how you can modify user code to overcome this restriction. You cannot make any change to the OS.

Solution:
Solution 1: Break up argument into smaller chunks, e.g., typecast as needed.
Solution 2: Put argument in decreasing order of addresses, i.e., growing outward from stack top.

−2 pts for using Malloc. (GeekOs does not allow user-mode threads to use Malloc.)

c. [2 pts]
Describe briefly how a small change in hardware can overcome this restriction, with essentially no change to the OS or user code.

Solution:
Have hardware grow the stack towards increasing addresses.
(x86 stack grows towards decreasing addresses; the compiler lays out arguments from low to high addresses.)

Would require modification only to OS and user code that accesses the stack in ways other than push and pop.
For example, `mov esp, 4` would become `mov esp, -4.`

2. [10 points] A byte-addressable single-level paging system has 32-bit virtual addresses, 24-bit physical addresses, page size of 8KB, and LRU replacement policy. Page table entries are aligned to 32-bit addresses (i.e., the least significant 5 bits are zero). Each page table entry has the following fields: page number; 1-bit "present" field; 6 bits for protection, accessed, etc. The present bit is 1 iff the page is mapped to a physical page.

a. [4 pts] Draw the page map table for a process. Show the number of entries, the fields and their sizes.

Solution:
$8K = 2^{13}$ So 13-bit offset within a page.
virtual address: [19-bit vitrual page number, 13-bit offset]
physical address: [11-bit physical page number, 13-bit offset]

The page table has 2^{19} entries. Each entry has a present field (1 bit), physical page number (11 bits), misc (6 bits), and 14 unused bits (to round up the entry size to a multiple of 32 bits).

	present	physical page #	misc	unused
	1	11	6	14
0				
1				
…				
$2^{19} - 1$				

b. [4 pts] A process is allocated four physical pages, numbered 100, 101, 102, 103. The process makes the following sequence of memory accesses (only the virtual page numbers are shown):
0, 1, 0, 0, 1, 1, 0, 4, 4, 0, 3, 3, 0, 6, 7, 7, 6, 1, 1, 0
Initially, virtual page 0 is mapped to physical page 100 and no other virtual page is mapped to a physical page. What is the number of page faults. What is the state of the page table (page number and present fields) at the end.

Solution:

vp#	0	1	0	0	1	1	0	4	4	0	3	3	0	6	7	7	6	1	1	0
fault?		F						F			F			F	F			F		
pp#	100	101						102			103			101	102			103		

6 faults in total.
Page table at end

	present	physical page #	misc	unused
0	1	100		
1	1	103		
…				
6	1	101		
7	1	102		
…				

c. The hardware has a TLB of four entries. Draw the TLB, showing its fields and their sizes. Indicate which part of the TLB is associatively searched.

Solution:

	present	virtual page #	physical page #	misc
	1	19	11y	6
0				
1				
2				
3				

3. [10 points] A resource allocation system that uses the Banker's algorithm for 3 resource types (A, B, C) and 5 users (P0, P1, P2, P3 P4) is currently in the following state. (Alloc: resources held by each user. Max: max need of each user. Req: ongoing request of each user. Avail: free resources.)

	Alloc	Max	Req	Avail		**Solution** Need
	A B C	A B C	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	3 2 2	3 3 2		7 4 3
P1	2 0 0	3 2 2	0 2 1			1 2 2
P2	3 0 2	9 0 2	6 0 0			6 0 0
P3	2 1 1	2 2 2	0 1 0			0 1 1
P4	0 0 2	4 3 3	2 3 0			4 3 1

a. [6 pts] Is the state safe. If you answer yes, give a sequence of process ids that leads to all processes completed. If you answer no, give a sequence of activities that results in a deadlocked state.

Solution:
Yes, the state is safe.
There are many safe sequences. Here is one: < P1, P3, P2, P4, P0 > is a safe sequence.

There is no safe sequence starting with P0.
The state is not deadlocked (obviously, since it is safe).

b. [4 pts] Is there an ongoing request other than P2's that cannot be granted immediately. Justify your answer.

Solution:
Yes.

P4's request cannot be granted in the above state.
Doing so would lead to Avail = [1 0 2] and Need = [2 0 1], which is not a safe state.

4. [10 points]
Implement a counting semaphore *x* using binary semaphores and no other synchronization construct (no atomic read-modify-write, no disabling interrupts, no access to PCBs, no wait/wakeup, etc.) and no busy waiting.
Specifically, supply three chunks of code: one for *x*'s initialization, one for P(*x*), one for V(*x*).

Solution:

See implementation 2 in note 10x:
http://www.cs.umd.edu/users/shankar/412-Notes/10x-countingSemUsingBinarySem.pdf

8 points for giving implementation 1 in note 10x.

5 points for giving a solution that uses busy waiting and is otherwise correct.

5. [10 points]
Let x be a strong binary semaphore (i.e., a thread gets past P(x) if V(x)'s keep happening, even if other threads do P(x)'s.)
Implement x using weak binary semaphores and no other synchronization construct (no atomic read-modify-write, no disabling interrupts, no access to PCBs, no wait/wakeup, etc.) and no busy waiting.
Supply three chunks of code: one for x's initialization, one for P(x), one for V(x).
Assume that x is accessed by at most N user threads with ids 0, 1, …, N−1.
You can use N and the id of the executing thread in your implementation (e.g., thread j calls P(j,x) instead of P(x)).

Solution
I see no way to implement this without using a separate semaphore for each thread to wait on.
With such semaphores, can use the same approach is the bounded-wait test-and-set implementation (see slides or book).

```
Shared by all threads
    WeakBinarySemaphore mutex ← 1;          // protects val, waiting[N]
    WeakBinarySemaphore w[N] ← 0;           // thread j waits on w[j] iff it has to wait on P(x)
    boolean waiting[N] ← 0;                 // waiting[j] true iff thread j is blocked at w[j]
    int  val ← initVal;                     //  initial value of semaphore x

P(i, x) {
  P(mutex);
  if  val < 0
    val ← 0;
    V(mutex)
  else
    waiting[i] ←  true;
    V(mutex);
    P(w[i])
}


V(i, x) {
  P(mutex);
  if  "for every j: waiting[j] is false"
    val ← 1;
    V(mutex)
  else
    let j be next (in modulo-N order) true entry in waiting[N];
    waiting[j] ← false;
    V(mutex);
    V(w[j])
}
```