Full Name:_______________________________________

Final Exam, Spring 2018 Date: 5/15

Instructions:

- Final exam takes 100 minutes. Read through all the problems and complete the easy ones first.
- This exam is **closed book**, except that you may bring a single doube-sided page of prepared note.

1 (xx/20)	2 (xx/25)	3 (xx/30)	4 (xx/25)	Bonus (xx/10)	Total (xxx/100+10)

1 Basic C and Machine instructions (20 points):

Answer the following multiple-choice questions. Circle *all* answers that apply. Each question is 5-points.

A. Suppose the content of %eax is `0x12345678`. When running on the x86-64 machine (Little Endian), after successfully executing instruction `movl %eax, (%ecx)`, what is the byte value stored at the address given by %ecx?

1. 0x12
2. 0x01
3. 0x78
4. 0x08
5. None of the above

B. Which following instruction is **not** a valid x86-84 instruction?

1. `mov $10, %eax`
2. `mov (%eax), %ebx`
3. `mov %eax, %rip`
4. `mov (%eax), (%ebx)`
5. `mov (%eax, %ebx, 8), %ecx`
6. `mov (%eax, %ebx, 10), %ecx`

C. What is the output of the following code snippet?

```
int a[3] = {1, 2, 3};
int *b;
b = a;
b++;
print("%d", *b);
```

1. 1
2. 2
3. 3
4. Segmentation fault
5. None of the above.

D. What is the output of the following code snippet?

```
int a[3] = {1, 2, 3};
int *b;
b = a;
(*b)++;
print("%d", *b);
```

1. 1
2. 2
3. 3
4. Segmentation fault
5. None of the above.

2 threads, processes, mallocs (25 points):

Answer the following multiple-choice questions. Circle *all* answers that apply. Each question is 5-points.

A. Which of the following statements are true?

1. The same virtual address in different processes always refers to the same physical memory location.
2. Segmentation fault *always* occurs whenever there is an out-of-bound array access.
3. The OS is responsible for populating entries in a process' page table.
4. The MMU (memory management unit) hardware is responsible for populating the entries in a process' page table.
5. The OS is responsible for traversing a multi-level page table to find the physical page number for a virtual address.
6. The MMU is responsible for traversing a multi-level page table to find the physical page number for a virtual address.

B. What is the printout of the following program? (Assume `fork` and `waitpid` return successfully).

```
int counter = 100;
void* doWork(void *a) {
  counter++;
  printf("%d ", counter);
}
void main() {
  pid_t pid = fork();
  if (pid == 0) {
    doWork();
  } else {
    waitpid(pid);
    doWork();
 }
}
```

1. 101 101
2. 101 102
3. 102 101
4. 102 102
5. None of the above.

C. What is the printout of the following program? (Assume `pthread_create` and `pthread_join` return successfully).

```
int counter = 100;
void* doWork(void *a) {
  counter++;
  printf("%d ", counter);
}
void main() {
  pthread_t pid;
  pthread_create(&pid, NULL, doWork, NULL);
  pthread_join(pid, NULL);
  doWork();
}
```

1. 101 101
2. 101 102
3. 102 101
4. 102 102
5. None of the above.

D. Which of the following statements are true w.r.t. `malloc`?

1. Every call to `malloc` results in the memory allocator making a syscall (e.g. `sbrk`) to request memory from OS.
2. `malloc` returns failure *if and only if* the memory allocator does not have free blocks.
3. When using the implicit-list design, `malloc` tends to traverse more blocks than when using the explicit-list design.
4. Your lab4 implementation works correctly when multiple threads concurrently call `malloc`.
5. None of the above.

E. Suppose we have a malloc implementation based on the implicit-list design with the following header (and footer) definition. Let us assume there is no further padding between the header and the payload. Suppose you are to implement `void free(void *p)`. Which of the following statement returns a pointer to the header of block to be freed?

```
typedef struct {
   int size;
  int allocated;
} header_t;
```

1. `(header_t *)((char *)p-1)`
2. `((header_t *)p)-1`
3. `(header_t *)((char *)p+1)`
4. `((header_t *)p)+1`
5. `(header_t *)p`
6. `(header_t *)((char *)p - sizeof(header_t))`
7. `(header_t *)((char *)p + sizeof(header_t))`
8. None of the above.

3 Virtual vs. Physical Addresses, Paging (30 points)

Each of the following questions is worth 5 points.

Consider a hypothetical machine with 16-bit virtual and physical addresses. Its virtual memory system supports a 2-level page table hierarchy with a page size of 64 bytes.

A. Given that the page size is 64 bytes, how many pages are there in the virtual or physical address space? How many bits are required to uniquely identify each virtual or physical page?

1. 2^9 pages, 9-bit.
2. 2^{10} pages, 10-bit
3. 2^{11} pages, 11-bit
4. 2^{12} pages, 12-bit
5. None of the above.

B. Suppose each page table entry (PTE) is 2 bytes in size. How many PTEs can be stored in one page?

1. 2
2. 16
3. 32
4. 64
5. None of the above.

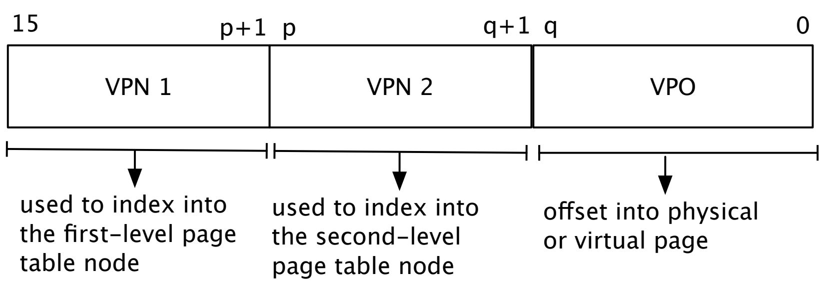

Figure 1: The breakdown of a 2-byte virtual address for traversing a two-level page table.

C. Based on your answer for question B., what should be the address offsets for p and q, as illustrated in Figure 1?

1. p=10, q=6
2. p=10, q=5
3. p=11, q=6
4. p=11, q=5
5. None of the above.

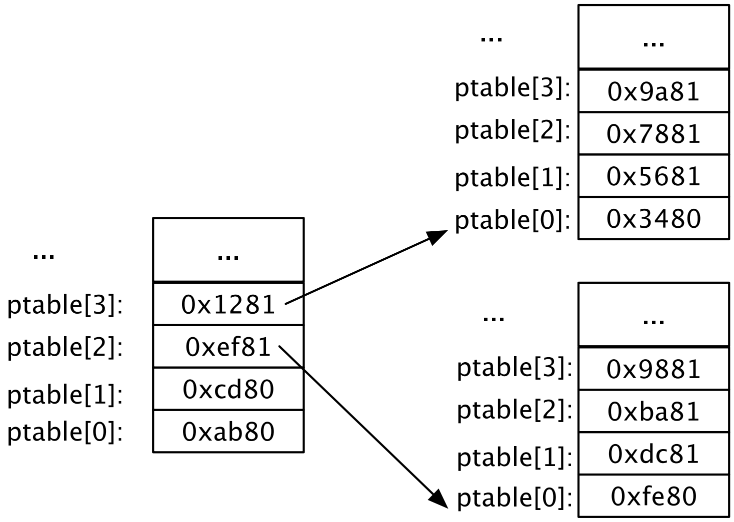

Figure 2: An example 2-level page table. Each PTE is 2-byte in size. The higher order x-bits of a PTE represent a page number. The lowest order bit of a PTE indicates its validity. All portions of the page table node marked with "..." correspond to invalid PTEs.

D. For each valid 2-byte PTE, its highest order x bits represent a page number (where x is your answer in the question A.) What kind of page number is stored in a valid PTE?

1. physical page number only.
2. virtual page number only.
3. either physical or virtual page number.
4. None of the above.

E. Based on the page table shown in Figure 2, what is the translated physical address for virtual address `0x10F6`? (Note that the lowest order bit of a PTE indicates the PTE's validity.)

F. Based on the page table shown in Figure 2, what is the translated physical address for virtual address `0x1836`? (Note that the lowest order bit of a PTE indicates the PTE's validity.)

4 Concurrent programming (25 points)

Ben Bitdiddle has implemented a stack data structure that exposes the API calls of pushing an integer to the back of the stack and poping an integer from the back of the stack. His first implementation is shown below.

```
#define MAX_LEN 2
typedef struct {
   int data[MAX_LEN];
   int len;
} stack_t;

void stack_init(stack_t *s) {
   for (int i = 0; i < MAX_LEN; i++) {
      s->data[i] = 0;
   }
   s->len = 0;
}

void stack_push(stack-t *s, int v) {
L1: int l = ++s->len; // increment s->len, assign new value to l
    if (l <= MAX_LEN) {
        s->data[l-1] = v;
    }
}

void stack_pop(stack_t *s, int *v) {
L2: int l = s->len--;  // decrement s->len, assign old value to l
    if (l > 0) {
        *v = s->data[l-1];
    }
}
```

The corresponding machine code for the C statement at L1 is

```
mov    0x8(%rdi),%eax       ; memory location 0x8(%rdi) stores s->len
add    0x1,%eax             ; %eax stores the value of local variable l
mov    %eax,0x8(%rdi)
```

The corresponding machine code for the C statement at L2 is

```
mov    0x8(%rdi),%edx      ; memory location 0x8(%rdi) stores s->len,
                           ; %edx stores local variable l
lea    -0x1(%rdx),%eax     ;
mov    %eax,0x8(%rdi)
```

For each of the following questions, circle *all* answers that apply. Each question is worth 5 points.

The symbol $\star$ indicates "don't care" for a value.

A. Suppose stack has just been initialized via `stack_init(s)`. Suppose 2 threads concurrently access this stack: thread-1 calls `stack_push(s, 1)`, and thread-2 calls `stack_push(s, 2)`. What are the state of the stack after both threads finish?

1. `s->data={1,2}, s->len=2`
2. `s->data={2,1}, s->len=2`
3. `s->data={1,⋆}, s->len=1`
4. `s->data={2,⋆}, s->len=1`
5. `s->data={⋆,⋆}, s->len=0`
6. None of the above.

B. Suppose the state of stack `s` is: `s->data={1,0}, s->len=1`. Suppose 2 threads concurrently access this stack: thread-1 calls `stack_push(s, 2)`, thread-2 calls `stack_pop(s, &val)`. What are the state of the stack after both threads finish?

1. `s->data={1,2}, s->len=2`
2. `s->data={2,1}, s->len=2`
3. `s->data={1,⋆}, s->len=1`
4. `s->data={2,⋆}, s->len=1`
5. `s->data={⋆,⋆}, s->len=0`
6. None of the above.

In Ben Bitbiddle's second implementation of the stack, he adds locking primitives to synchronize access. His second implementation is shown as follows:

```
#define MAX_LEN 2
typedef struct {
int data[MAX_LEN];
int len;
pthread_mutext_t mu;
} stack_t;

void stack_init(stack_t *s) {
   for (int i = 0; i < MAX_LEN; i++) {
      s->data[i] = 0;
   }
   pthread_mutex_init(&s->mu);
   s->len = 0;
}

void stack_push(stack-t *s, int v) {
   pthread_mutex_lock(&s->mu);
   int l = ++s->len; // increment s->len, assign new value to l
   pthread_mutex_unlock(&s->mu);
   if (l <= MAX_LEN) {
      s->data[l-1] = v;
   }
}

void stack_pop(stack_t *s, int *v) {
   pthread_mutex_lock(&s->mu);
   int l = s->len--;  // decrement s->len, assign old value of to l
   pthread_mutex_unlock(&s->mu);
   if (l > 0) {
      *v = s->data[l-1];
   }
}
```

C. Suppose stack has just been initialized via `stack_init(s)`. Suppose 2 threads concurrently access this stack: thread-1 calls `stack_push(s, 1)`, and thread-2 calls `stack_push(s, 2)`. What are the state of the stack after both threads finish? (Note: this is the same question as (A), except you are answering this question w.r.t. Ben's second implementation).

1. `s->data={1,2}, s->len=2`
2. `s->data={2,1}, s->len=2`
3. `s->data={1,*}, s->len=1`
4. `s->data={2,*}, s->len=1`
5. `s->data={*,*}, s->len=0`
6. None of the above.

D. Suppose the state of stack `s` is: `s->data={1,0}, s->len=1`. Suppose 2 threads concurrently access this stack: thread-1 calls `stack_push(s, 2)`, thread-2 calls `stack_pop(s, &val)`. What could be the potential value of `val` when the return of `stack_pop(s, &val)` in thread-2?

1. 0
2. 1
3. 2
4. None of the above.

E. Does Ben's second implementation correctly synchronize his stack implementation? Answer "yes" or "no". If your answer is "no", please give a correct implementation.

Bonus question (10 points): In Ben's first and second implementation, `stack_push` does not store a value in the queue if the queue is already full (similarly, `stack_pop` does not store any value in `*v` if the queue is empty). Suppose Ben Bitbiddle would like to implement a "blocking stack". In this new implementation, `stack_push` blocks the calling thread if the queue is full and waits to store the value when the queue is not full. Similarly, `stack_pop` blocks the calling thread if the queue is empty and waits to pop a value off when the queue becomes non-empty. Please give your implementation for the blocking queue.

—END of Final—-