Andrew login ID:____

Full Name:____

Section:____

15-213/18-243, Spring 2011

Final Exam

Tuesday, May 3, 2011

Instructions:

- Make sure that your exam is not missing any sheets, then write your Andrew login ID, full name, and section on the front.
- This exam is closed book and closed notes. A notes sheet is attached to the back.
- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 200 points.
- The problems are of varying difficulty. The point value of each problem is indicated. Good luck!

1 (24):
2 (14):
3 (20):
4 (14):
5 (14):
6 (10):
7 (14):
8 (25):
9 (15):
10 (14):
11 (14):
12 (14):
13 (8):
TOTAL (200):

Problem 1. (24 points):

Multiple choice.

Write the correct answer for each question in the following table:

1	2	3	4	5	6	7	8	9	10	11	12
13	14	15	16	17	18	19	20	21	22	23	24

1. Which of the following is a legitimate difference between IA-32 and x86-64?

   (a) Buffer overflow exploits are impossible under x86-64.
   (b) IA-32 has caller- and callee-saved register conventions, while x86-64 does not.
   (c) Under x86-64, any instructions that take 32-bit operands are illegal.
   [✓ (d)] None of the above.

2. Which of the following is the best justification for using the middle bits of an address as the set index into a cache rather than the most significant bits?

   (a) Indexing with the most significant bits would necessitate a smaller cache than is possible with middle-bit indexing, resulting in generally worse cache performance.
   (b) It is impossible to design a system that uses the most significant bits of an address as the set index.
   (c) The process of determining whether a cache access will result in a hit or a miss is faster using middle-bit indexing.
   [✓ (d)] A program with good spatial locality is likely to make more efficient use of the cache with middle-bit indexing than with high-bit indexing.

3. Which of the following is not true about POSIX-style signals?

   (a) Certain signals cannot be blocked.
   (b) A process can send a signal to itself.
   [✓ (c)] A signal handler executing as the result of a received signal can never be interrupted by another incoming signal.
   (d) Signals can only be delivered when returning from system mode.

4. Which of the following is not a benefit of virtual memory?

   (a) It allows the virtual address space to be larger than the physical address space

   (b) No process can accidentally access the memory of another process

   [✓ (c)] The TLB is more effective since without it dereferencing a virtual address now requires two or more memory accesses

   (d) Different processes can have overlapping virtual address spaces without conflict

5. Which of the following is a difference between blocking and ignoring a signal?

   (a) Once a blocked signal is unblocked, it will be handled by the process. A signal that comes while it is being ignored will never be handled.

   (b) SIGSTOP and SIGINT can be ignored, but not blocked.

   (c) Ignoring a signal only causes it to have no effect, while blocking a signal returns the signal to its sender.

   (d) None of the above

6. Where is the first argument to a function located in 32-bit assembly code, immediately after the `call` instruction is executed?

   (a) %ebp + 0x4

   (b) %ebp - 0x4

   (c) %esp + 0x4

   (d) %exp - 0x4

7. Consider the following piece of code, where out.txt's contents are "abc":

```
int main(int argc, char** argv)
{
    int fd = open("out.txt", O_RDWR);
    char str[] = "xyz";
    char c;

    write(fd1, str, 1);
    read(fd1, &c, 1);
    write(fd1, &c, 1);
    return 0;
}
```

   What is the contents of out.txt after the code is run? Assume all system calls succeed.

   (a) xbb
   (b) xba
   (c) xac
   (d) boat

8. Which of the following is the *best* reason to choose FastCGI over CGI?

   (a) Superior support by web servers like Apache
   (b) Lower process creation costs
   (c) Lower process communication costs
   (d) Better process locality (all tasks can be executed locally)

9. Which of the following system calls can fail due to a network failure?

   (a) `socket(...)`
   (b) `listen(...)`
   (c) `bind(...)`
   (d) `gethostbyname(...)`

10. Which of the following are copied on fork and preserved on exec?

   (a) Global variables.
   (b) File descriptor tables.
   (c) Open file entry structs.
   (d) None of the above.

11. Why would the kernel designer opt for a 2-level page table when a full 2-level page table takes up more memory than a full 1-level page table?

    (a) 2-level tables can translate virtual addresses faster.
    (b) 2-level tables can reference more memory than 1 level tables.
    (c) Most of the time, a 2-level page table will take up less memory than a 1 level page table.
    (d) They wouldn't. Adding more tables offers no advantages.

12. What section of memory holds the assembly for `printf`?

    (a) Stack
    (b) Kernel memory
    (c) Shared libraries
    (d) Heap

13. Every thread has its own __________.

    (a) Heap
    (b) Global values
    (c) Stack
    (d) Text data

14. Why is `gethostbyname` not thread safe?

    (a) Only one thread at a time can do a DNS lookup
    (b) It doesn't have a mutex around it
    (c) It returns a pointer to global shared memory
    (d) It shares instructions with other threads

15. If a page table on a 32-bit system is 2KB in size, how many entries does it contain?

    (a) 2048
    (b) 1024
    (c) 512
    (d) 256

16. What is the function of the TLB?

    (a) Caches data
    (b) Caches instructions
    (c) Caches translation of virtual addresses
    (d) Translates physical addresses to virtual addresses

17. What is distinctive about superscalar processors?

    (a) Can run at frequencies over 3.5GHz
    (b) Can address over 4GB of memory
    (c) Can perform more than one instruction per cycle
    (d) Can have more than 2 levels of cache
    (e) Have more than one core per processor

18. True/False: When requested to send 20 bytes over a network socket, execution will block until all 20 bytes have been sent.

    (a) True
    (b) False

19. True/False: When `printf` returns, the programmer cannot be guaranteed that the data has appeared on the user's terminal.

    (a) True
    (b) False

20. Which of the following tools would you first use to debug an application which is exiting with the error "Segmentation fault"?

    (a) `gdb`
    (b) `strace`
    (c) `strings`
    (d) `objdump`

21. Which of the following tools would you first use to debug a network application that never appears to accept any connections?

    (a) `gdb`
    (b) `strace`
    (c) `objdump`
    (d) `valgrind`

22. Which of the following tools would you first use to debug an application which is exiting with a `glibc error:  double free detected`?

    (a) `gdb`
    (b) `strace`
    (c) `wireshark`
    (d) `valgrind`

23. A 256-byte 4-way set associative cache with 16 byte blocks has

    (a) 4 sets
    (b) 16 sets
    (c) 64 sets
    (d) No sets

24. Imagine a floating point format with no sign bit, one exponent bit, and one fraction bit. Which of the following is not a number?

    (a) 00
    (b) 01
    (c) 10
    (d) 11
    (e) None of the above

Problem 2. (14 points):

Stack discipline.

Consider the following C code and assembly code for two mutually recursive functions:

```
int even(unsigned int n)     0x080483e4 <even+0>:    push   %ebp
{                            0x080483e5 <even+1>:    mov    %esp,%ebp
    if(!n)                   0x080483e7 <even+3>:    sub    $0x8,%esp
    {                        0x080483ea <even+6>:    cmpl   $0x0,0x8(%ebp)
        return 1;            0x080483ee <even+10>:   jne    0x80483f9 <even+21>
    }                        0x080483f0 <even+12>:   movl   $0x1,-0x4(%ebp)
                             0x080483f7 <even+19>:   jmp    0x804840a <even+38>
    return odd(n - 1);       0x080483f9 <even+21>:   mov    0x8(%ebp),%eax
}                            0x080483fc <even+24>:   sub    $0x1,%eax
                             0x080483ff <even+27>:   mov    %eax,(%esp)
                             0x08048402 <even+30>:   call   0x804840f <odd>
                             0x08048407 <even+35>:   mov    %eax,-0x4(%ebp)
                             0x0804840a <even+38>:   mov    -0x4(%ebp),%eax
                             0x0804840d <even+41>:   leave
                             0x0804840e <even+42>:   ret

int odd(unsigned int n)      0x0804840f <odd+0>:     push   %ebp
{                            0x08048410 <odd+1>:     mov    %esp,%ebp
    if(!n)                   0x08048412 <odd+3>:     sub    $0x8,%esp
    {                        0x08048415 <odd+6>:     cmpl   $0x0,0x8(%ebp)
        return 0;            0x08048419 <odd+10>:    jne    0x8048424 <odd+21>
    }                        0x0804841b <odd+12>:    movl   $0x0,-0x4(%ebp)
                             0x08048422 <odd+19>:    jmp    0x8048435 <odd+38>
    return even(n - 1);      0x08048424 <odd+21>:    mov    0x8(%ebp),%eax
}                            0x08048427 <odd+24>:    sub    $0x1,%eax
                             0x0804842a <odd+27>:    mov    %eax,(%esp)
                             0x0804842d <odd+30>:    call   0x80483e4 <even>
                             0x08048432 <odd+35>:    mov    %eax,-0x4(%ebp)
                             0x08048435 <odd+38>:    mov    -0x4(%ebp),%eax
                             0x08048438 <odd+41>:    leave
                             0x08048439 <odd+42>:    ret
```

Imagine that a program makes the procedure call `even(3)`. Also imagine that prior to the invocation, the value of %esp is `0xffff1000`—that is, `0xffff1000` is the value of %esp *immediately before* the execution of the `call` instruction.

A. Note that the call `even(3)` will result in the following function invocations: `even(3)`, `odd(2)`, `even(1)`, and `odd(0)`. Using the provided code and your knowledge of IA32 stack discipline, fill in the stack diagram with the values that would be present immediately before the execution of the `ret` instruction for `odd(0)`. Cross out each blank for which there is insufficient information to complete.

```
+---------------------------------+
|                                 |  0xffff1004
+---------------------------------+
|                                 |  0xffff1000
+---------------------------------+
|                                 |  0xffff0ffc
+---------------------------------+
|                                 |  0xffff0ff8
+---------------------------------+
|                                 |  0xffff0ff4
+---------------------------------+
|                                 |  0xffff0ff0
+---------------------------------+
|                                 |  0xffff0fec
+---------------------------------+
|                                 |  0xffff0fe8
+---------------------------------+
|                                 |  0xffff0fe4
+---------------------------------+
|                                 |  0xffff0fe0
+---------------------------------+
|                                 |  0xffff0fdc
+---------------------------------+
|                                 |  0xffff0fd8
+---------------------------------+
|                                 |  0xffff0fd4
+---------------------------------+
|                                 |  0xffff0fd0
+---------------------------------+
|                                 |  0xffff0fcc
+---------------------------------+
|                                 |  0xffff0fc8
+---------------------------------+
|                                 |  0xffff0fc4
+---------------------------------+
|                                 |  0xffff0fc0
+---------------------------------+
```

B. What are the values of %esp and %ebp immediately before the execution of the `ret` instruction for `odd(0)`?

Problem 3. (20 points):

Assembly/C translation.

Consider the following C code and assembly code for a curiously-named function:

```
typedef struct node
{
    void *data;
    struct node *next;
} node_t;

node_t *lmao(node_t *n, int f(node_t *))
{
    node_t *a, *b;

    if(_____!n______){
        return NULL;
    }

    a = __lmao(n->next, f)__;

    if(__f(n)____){


        b = ____malloc(sizeof(node_t))____;
        b->data = n->data;
        b->next = ____a__;
        return b;
    }

    return ________a_________;
}
```

```
0x4005d0:  mov    %rbx,-0x18(%rsp)
0x4005d5:  mov    %rbp,-0x10(%rsp)
0x4005da:  xor    %eax,%eax
0x4005dc:  mov    %r12,-0x8(%rsp)
0x4005e1:  sub    $0x18,%rsp
0x4005e5:  test   %rdi,%rdi
0x4005e8:  mov    %rdi,%rbx
0x4005eb:  mov    %rsi,%rbp
0x4005ee:  je     0x40061e <lmao+78>
0x4005f0:  mov    0x8(%rdi),%rdi
0x4005f4:  callq  0x4005d0 <lmao>
0x4005f9:  mov    %rbx,%rdi
0x4005fc:  mov    %rax,%r12
0x4005ff:  callq  *%rbp
0x400601:  mov    %eax,%edx
0x400603:  mov    %r12,%rax
0x400606:  test   %edx,%edx
0x400608:  jle    0x40061e <lmao+78>
0x40060a:  mov    $0x10,%edi
0x40060f:  callq  0x400498 <malloc>
0x400614:  mov    (%rbx),%rdx
0x400617:  mov    %r12,0x8(%rax)
0x40061b:  mov    %rdx,(%rax)
0x40061e:  mov    (%rsp),%rbx
0x400622:  mov    0x8(%rsp),%rbp
0x400627:  mov    0x10(%rsp),%r12
0x40062c:  add    $0x18,%rsp
0x400630:  retq
```

Using your knowledge of C and assembly, fill in the blanks in the C code for `lmao` with the appropriate expressions. (Note: `0x400498` is the address of the C standard library function `malloc`.)

Problem 4. (14 points):

Process control.

Consider the following C program:

```
int main()
{
    pid_t pid;
    int status, counter = 4;

    while(counter > 0)
    {
        pid = fork();

        if(pid)
        {
            counter /= 2;
        }
        else
        {
            printf("%d", counter);  /* (1) */
            break;
        }
    }

    if(pid)
    {
        waitpid(-1, &status, 0);
        counter += WEXITSTATUS(status);

        waitpid(-1, &status, 0);
        counter += WEXITSTATUS(status);

        printf(";%d", counter);     /* (2) */
    }

    return counter;
}
```

Use the following assumptions to answer the questions:

- All processes run to completion, and no system calls fail.
- `printf` is atomic and calls `fflush(stdout)` after printing its argument(s) but before returning.

For each question, there may be more blanks than necessary.

A. List every individual digit that can be emitted by a call to `printf`. Include any digits that can be printed along with the semicolon by the `printf` annotated with `(2)`. For example, if `1521;3` were a possible output of the program, the solutions would include `1`, `2`, `3`, and `5`.

__________ __________ __________ __________

__________ __________ __________ __________

B. Notice that the `printf` annotated with `(2)` emits a semicolon in addition to a digit. List all of the digit sequences that can be printed *before* the semicolon is emitted. For example, if `1521;3` were a possible output of the program, `1521` would be one solution.

__________ __________ __________ __________

__________ __________ __________ __________

__________ __________ __________ __________

C. Now list all of the digit sequences that can be printed *after* the semicolon is emitted.

__________ __________ __________ __________

__________ __________ __________ __________

Problem 5. (14 points):

Concurrency.

Consider the following implementation of reader writer locks. A reader writer lock is a concurrency mechanism that allows either multiple readers to have access to a critical section or a single writer.

```
struct rwlock {
    sem_t *sem; int readers; int writers;
};

void rwlock_init(struct rwlock *lock)
{
    sem_init(&lock->sem, 1);
    lock->readers = 0;
    lock->writers = 0;
}

void readlock(struct rwlock *lock)
{
    while(1) {
        sem_wait(lock->sem);
        if(lock->writers == 0) {
            lock->readers++; break;
        }
        sem_post(lock->sem);
    }
}

void writelock(struct rwlock *lock)
{
    while(1) {
        sem_wait(lock->sem);
        if(lock->readers == 0 && lock->writers == 0) {
            lock->writers = 1; break;
        }
        sem_post(lock->sem);
    }
}

void unlock(struct rwlock *lock)
{
    sem_wait(lock->sem);
    if(lock->readers > 0)
        lock->readers--;
    else
        lock->writers--;
    sem_post(lock->sem);
}
```

A. What is the problem with the above implementation?

B. Starvation is a problem where one thread, or kind of thread (think reader or writer), is unable to acquire a resource. After fixing the previous problem, is starvation possible? How?

Problem 6. (10 points):

File I/O

The following problems refer to a file called `numbers.txt`, with contents the ASCII string `0123456789`. You may assume calls to read() are atomic with respect to each other. The following file, `read_and_print_one.h`, is compiled with each of the following code files.

```
#ifndef READ_AND_PRINT_ONE
#define READ_AND_PRINT_ONE
#include <stdio.h>
#include <unistd.h>

static inline void read_and_print_one(int fd) {
    char c;
    read(fd, &c, 1);
    printf("%c", c); fflush(stdout);
}
#ENDIF
```

A. Consider the following code:

```
#include "read_and_print_one.h"
#include <stdlib.h>
#include <fcntl.h>

int main() {
  int file1 = open("numbers.txt", O_RDONLY);
  int file2;
  int file3 = open("numbers.txt", O_RDONLY);
  file2 = dup2(file3, file2);

  read_and_print_one(file1);
  read_and_print_one(file2);
  read_and_print_one(file3);
  read_and_print_one(file2);
  read_and_print_one(file1);
  read_and_print_one(file3);

  return 0;
}
```

List all possible outputs of the above code.

B. Consider the following code:

```
#include "read_and_print_one.h"
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
  int file1;
  int file2;
  int file3;
  int pid;

  file1 = open("numbers.txt", O_RDONLY);
  file3 = open("numbers.txt", O_RDONLY);

  file2 = dup2(file3, file2);

  read_and_print_one(file1);
  read_and_print_one(file2);

  pid = fork();

  if (!pid) {
    read_and_print_one(file3);
    close(file3);
    file3 = open("numbers.txt", O_RDONLY);
    read_and_print_one(file3);
  } else {
    wait(NULL);
    read_and_print_one(file3);
    read_and_print_one(file2);
    read_and_print_one(file1);
  }

  read_and_print_one(file3);

  return 0;
}
```

List all possible outputs of the above code.

Problem 7. (14 points):

Deadlocks and Dreadlocks

Two threads (X and Y) access shared variables A and B protected by mutex_a and mutex_b respectively. Assume all variable are declared and initialized correctly.

```
Thread X                 Thread Y
P(&mutex_a);             P(&mutex_b);
A += 10;                 B += 10;
P(&mutex_b);             P(&mutex_a);
B += 20;                 A += 20;
V(&mutex_b);             V(&mutex_a);
A += 30;                 B += 30;
V(&mutex_a);             V(&mutex_b);
```

A. Show an execution of the threads resulting in a deadlock. Show the execution steps as follows

Thread X	**Thread Y**
$P(\&mutex\_a)$	
$A+=10$	
$P(\&mutex\_b)$	
	$P(\&mutex\_b)$
$\ldots$	
	$\ldots$

Answer:

B. There are different approaches to solve the deadlock problem. Modify the code above to show **two** approaches to prevent deadlocks. You can declare new mutex variables if required. Do not change the order or amount of the increments to A and B. Rather, change the locking behavior around them. The final values of A and B must still be guaranteed to be incremented by 60.

   Answer:

Problem 8. (25 points):

Thread Safety

A fellow 213 student works on cutting edge research finding prime numbers. He wants to speed up his code by making it multi-threaded. He is running into some issues while implementing a thread safe version of the `next_prime` function and asks for your help.

```
struct big_number *next_prime(struct big_number current_prime) {
  static struct big_number next;

  next = current_prime;
  addOne(next);
  while (isNotPrime(next)) {
    addOne(next);
  }

  return &next;
}

struct big_number *ts_next_prime(struct big_number current_prime) {
  return next_prime(current_prime);
}
```

A. Why is the function `ts_next_prime` thread-unsafe?

Answer:

B. Assume the mutex guarding the call to `next_prime` is initialized correctly in the following code.

```
struct big_number *ts_next_prime(struct big_number current_prime) {
  struct big_number *value_ptr;

  sem_wait(&mutex);
  value_ptr = next_prime(current_prime);
  sem_post(&mutex);

  return value_ptr;
}
```

The following modification to the function is still not thread safe. Explain why, and show an example execution with two threads showing the problem?

Show the execution steps as follows

Thread 1	**Thread 2**
$sem\_wait(\&mutex)$	
	$sem\_wait(\&mutex);$
$value\_ptr = next\_prime(current\_prime)$	
$\ldots$	
	$\ldots$

Answer:

Thread 1	**Thread 2**

C. Fill in the blanks below to fix `ts_next_prime`.

```
struct big_number *ts_next_prime(struct big_number current_prime) {
  struct big_number *value_ptr;

  struct big_number *ret_ptr = ____________________________________;
  sem_wait(&mutex);
  value_ptr = next_prime(current_prime);
  ____________________________________;
  sem_post(&mutex);

  return ret_ptr;
}
```

Why does this fix work? :

D. One disadvantage of using a thread-safe `ts_next_prime` as opposed to `next_prime` is higher overhead. List the overheads.

Answer:

E. Is the final version of your function `ts_next_prime` reentrant too?
Circle your answer: Yes No

Problem 9. (15 points):

Structure alignment

Consider the following C struct.

```
struct st1_t {
  char a;
  char b;
  char c;
};

struct st2_t {
  st1_t d;
  st1_t e;
  st1_t *f;
  short g;
  char h;
  double i;
  long j;
};
```

A. Show how the `st1_t` struct above would appear on a 32 bit Linux system. Label the bytes that belong to the various fields with their names and clearly markthe end of the struct. Use hatch marks to indicate bytes that are allocated in the struct but are not used.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

B. Show how the `st2_t` struct above would appear on a 32 bit Linux system. Label the bytes that belong to the various fields with their names and clearly markthe end of the struct. Use hatch marks to indicate bytes that are allocated in the struct but are not used.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Problem 10. (14 points):

Floating point

Use the following 8-bit floating point representation: 1 sign bit, 3 exponent bits, and 4 fraction bits.

A. What is the bias?

B. What is the smallest positive value that can be represented?

C. What is the largest positive denormalized number?

D. What is the representation of negative infinity?

E. Fill in the following table. Use round-to-even. Value should be written to decimal

Bits	Decimal Value
`0 000 0000`	`0`
`1 010 0000`	
	`-13`
	`1/16`
`0 111 1111`	
	`15/256`

Problem 11. (14 points):

Signals.

Consider the following C program:

```
int counter = 0;

void handler1(int sig) {
    printf("%d", counter);
    kill(getpid(), SIGUSR2);
}

void handler2(int sig) {
    counter = 5;
    printf("%d", counter);
}

int main(int argc, char *argv[])
{
    int pid;

    signal(SIGUSR1, handler1);
    signal(SIGUSR2, handler2);

    if ((pid = fork())) {
        kill(pid, SIGUSR1);
    } else {
        counter++;
        printf("%d", counter);
    }

    return 0;
}
```

Using the following assumptions, list all possible outputs of the code:

- All processes run to completion and no system calls will fail
- `printf()` is atomic and calls `fflush(stdout)` after printing argument(s) but before returning

Problem 12. (12 points):

Address translation. This problem deals with virtual memory address translation using a multi-level page table, in particular the 2-level page table for a 32-bit Intel system with 4 KByte pages tables. Assume all processes are running under Supervisor mode. The following diagrams are direct from the Intel System Programmers guide and should be used on this problem:

The contents of the relevant sections of memory are shown on this page. All numbers are given in **hexadecimal**. Any memory not shown can be assumed to be zero. The Page Directory Base Address is `0x0045d000`.

For each of the following problems, perform the virtual to physical address translation. If an error occurs at any point in the address translation process that would prevent the system from performing the lookup, then indicate this by circling FAILURE and noting the physical address of the table entry that caused the failure.

For example, if you were to detect that the present bit in the PDE is set to zero, then you would leave the PTE address in (b) empty, and circle FAILURE in (c), noting the physical address of the offending PDE.

Address	Contents
000c3020	345ab236
000c3080	345ab237
000c332f	08e4523f
000c3400	93c2ed98
000c3cbc	34abd237
000c3ff0	93c2ed99
000c4020	8e56e237
000c432f	33345237
000c4400	43457292
000c4cbc	385ed293
000c4ff0	c3726292
0045d000	000c3292
0045d028	000c4297
0045d032	0df2a292
0045d0a0	000c3297
0045d3ff	0df2a236
0045d9fc	0df2a237
0df2a000	deded000
0df2a080	bc3de239
0df2a3fc	000c4296
0df2a4a0	00324236
0df2a4fc	df72c9a6
0df2b080	01f008c3
0df2bff0	000c5112

TLB

Index	Tag	Frame Number	Valid
0	0x03506	0x98f8a	1
	0x27f4a	0x34abe	0
1	0x1f7ee	0x95cbc	0
	0x2a064	0x72954	1
2	0x1f7f0	0x95ede	0
	0x2005d	0xaa402	0
3	0x3fc2e	0x2029e	1
	0x3df82	0xff644	0

1. Read from virtual address `0x9fd28c10`. Scratch space:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(a) (TLB Hit) Physical address is: 0x

OR

(b) Physical address of PDE: 0x

(c) Physical address of PTE: 0x

(d) (SUCCESS) The physical address accessed is: 0x

OR

(FAILURE) The physical address of the table entry causing the failure is: 0x

2. Read from virtual address `0x0d4182c0`. Scratch space:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(a) (TLB Hit) Physical address is: 0x

OR

(b) Physical address of PDE: 0x

(c) Physical address of PTE: 0x

(d) (SUCCESS) The physical address accessed is: 0x

OR

(FAILURE) The physical address of the table entry causing the failure is: 0x

3. Read from virtual address `0x0a32fcd0`.
   Scratch space:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

   (a) (TLB Hit) Physical address is: 0x

   OR

   (b) Physical address of PDE: 0x

   (c) Physical address of PTE: 0x

   (d) (SUCCESS) The physical address accessed is: 0x

   OR

   (FAILURE) The physical address of the table entry causing the failure is: 0x

Problem 13. (8 points):

Networks.

Consider a multi-threaded proxy that handles requests concurrently and a single-threaded proxy that handles requests serially.

A. Under which circumstances would the multi-threaded proxy perform better than the single-threaded proxy?

B. Under which circumstances would the single-threaded proxy perform no worse than the multi-threaded proxy?