Andrew login ID (please print in BLOCK capital letters): | | | | | | | | | | | | |

Full Name: _____

Recitation Section (or TA): _____

15-213/18-243, Fall 2009

Final Exam

Monday, Dec 14, 2009

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name, Andrew login ID, and recitation section (A–H) on the front.
- The exam has a maximum score of 141 points.
- This exam is OPEN BOOK. You may use any books or notes you like. *No calculators or other electronic devices are allowed.*
- Please make sure we can read your andrew ID. It needs to be in block capital letters.

Multiple Choice	1 (18):
Virtual Memory?	2 (18):
Stack	3 (16):
Signals	4 (9):
Assembly	5 (12):
Network Programming	6 (12):
Floats and Ints	7 (10):
Processes and Threads	8 (12):
Synchronization	9 (14):
Concurrency	10 (10):
File I/O	11 (10):
Extra Credit	12 and 13 (0):
	TOTAL (141):

Problem 1. Multiple Choice (18 points):

A. Which of these uses of caching is not crucial to program performance?

(a) Caching portions of physical memory
(b) Caching virtual memory pages
(c) Caching virtual addresses
(d) Caching virtual address translations
(e) None of the above (i.e., they are all crucial)

B. For which values can X not be equal to Z in the code below (circle all that apply):

```
int X = CONSTANT;
float Y = X;
int Z = Y;
```

(a) For large positive values of CONSTANT (e.g., >1,000,000,000)
(b) For large negative values of CONSTANT (e.g., > -100)
(c) For small positive values of CONSTANT (e.g., < 100)
(d) For small negative values of CONSTANT (e.g., < -1,000,000,000)
(e) None of the above (i.e., X==Z in all of these cases)

C. What is the maximum number of page faults per second that can be serviced in a system with a disk that has the following characteristics: 10,000 RPM rotation speed (6ms per full revolution), average seek time of 7ms, 1000 sectors per track. (Assume that all in-memory pages that get replaced are clean.)

(a) 50
(b) 100
(c) 77
(d) Not enough information to determine the answer

D. If a parent process forks a child process, to which resources might they need to synchronize their access to prevent unexpected behavior?

(a) file descriptors
(b) malloc'ed memory
(c) stack
(d) None of the above

E. Which of the following is not a situation that results in a signal being sent to a process?

(a) A process terminates
(b) A process accesses an invalid memory address
(c) A new connection arrives on a listening socket
(d) A divide by zero
(e) None of the above (i.e., all result in a signal being sent)

F. Mr. Fred says that, if one of a process's memory addresses is bigger than a second one, then its corresponding value must appear before the second one's value in physical memory. True or False?

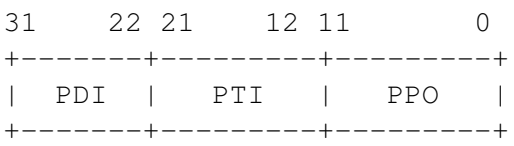
(a) True.
(b) False.

Problem 2. Virtual Memory? (18 points):

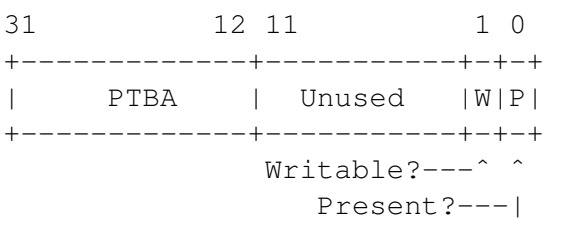
In this question you will write macros that will can be used to implement virtual memory on an x86 32-bit machine. You won't be writing any virtual memory code, just some helpful macros.

Here is the layout for our VM structure with 32-bit virtual and physical addresses, and 4kb pages:

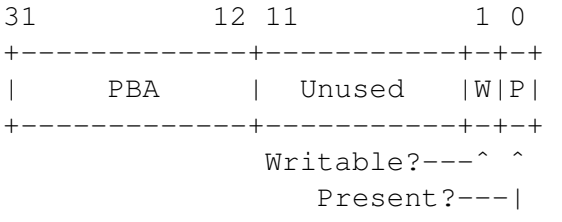
Virtual addresses are structured as such:



A Page Directory Entry (PDE) is structured as such:



A Page Table Entry (PTE) is structured as such:



You do not need to know how these values connect for this question, just know where each value is bitwise.

Each of these should easily fit on a single line. Do not make any assumptions about the type of the values passed to these macros!

```
/* Given a virtual address, returns the Physical Page Offset. */
#define VA_GET_PPO(virtual)


/* Given a virtual address, returns the page table index. */
#define VA_GET_PTI(virtual)


/* Given a virtual address, returns the page directory index. */
#define VA_GET_PDI(virtual)


/* Given a page directory entry, returns the page table base address. */
#define PDE_GET_PTBA(pde)


/* Given a page table entry, returns the page base address. */
#define PTE_GET_PBA(pte)


/* Returns one if this page table entry is present. */
#define IS_PRESENT(pte)


/* Returns one if this page table entry is writable. */
#define IS_WRITABLE(pte)


/* Returns a new Page Table Entry with the present bit set to the
 * value in Present (either 1 or 0) */
#define SET_PRESENT(pte, pres)


/* Returns a new Page Table Entry with the writable bit set to the
 * value in Writable (either 1 or 0) */
#define SET_WRITABLE(pte, write)
```

As a closing note: you just wrote the hardest part of virtual memory translations: congratulations!

Problem 3. The Stack Question (16 points):

Answer the following questions about x86 stack convention (in 32-bit mode):

A. How does the `call` instruction modify the stack?

B. How does the `leave` instruction modify the stack?

C. How does the `ret` instruction modify the stack?

D. How are arguments passed to a function?

E. How are return values passed back to a calling function?

F. Please draw a stack region that details how a function would call

```
printf("I lost %d %s.\n", 5, "marbles");
```

The first argument to `printf` (the format string) is located at `0xcafebeef` and the string `"marbles"` is located at `0xbeefbabe`. Please draw the stack area modified/created during this function call from the argument build area to the top of the stack as it exists just after the call instruction.

G. Why is this code potentially harmful? (Hint: it has to do with the stack!)

```
int fd = accept(server,&clientAddr,&clientlen);
pthread_create(&tid,NULL,handle_request,(void *)&fd);
```

Confession time: Did you do that on proxylab?

Problem 4. Signals (9 points):

Consider the C code below. Assume that no errors prevent any processes from running to completion and that a process terminated by an uncaught signal has an exit status of 0.

```
int count = 0;

void killhandler(int sig){
  printf("SIGKILL received\n");
  return;
}

void childhandler(int sig){
  int status;
  wait(&status);
  count += WEXITSTATUS(status);
  return;
}

main(){
  int i; // for loop iterator
  pid_t pid[3]; // pids of child processes

  Signal(SIGKILL, killhandler);
  Signal(SIGCHLD, childhandler);

  // Fork 3 child processes
  for(i=0; i<3; i++){
    pid[i] = fork();
    if(!pid[i]){ // If child process
      Signal(SIGKILL, SIG_DFL);
      exit(5);
    }
  }

  // Parent process only
  for(i=0; i<3; i++){
    kill(pid[i], SIGKILL);
  }
  sleep(5);
  printf("count = %d\n", count);
  exit(0);
}
```

A. What is the maximum number of times "SIGKILL received" could be printed?

B. **List** all possible **values** of count that could be printed:

Problem 5. Assembly – Leaping to Conclusions (12 points):

Your classmate Meddie Tartan is trying to write a more advanced proxy. Her goal is to write a cache that keeps track of several different things about the request, so that the proxy can better match clients' requests. She wants to keep track of it in a struct as follows:

```
struct cache_entry {
    char *url;
    char *browser_signature;
    char *http_referrer;
    char *content;
}
```

Since cache entries are stored in a global data structure, once the information is parsed out of the information the server sent, it needs to be copied into a `malloc`ed area that will stay around after the present stack frame goes away. Since each member of the cache entry struct is a string of arbitrary length, `malloc` needs to be called many times in the course of building the entry.

Since she remembered having to handle error cases appropriately during proxylab, Meddie's first idea was to write code that called `malloc` each time, and if any of them fail, return an error code, like so:

```
struct cache_entry *allocate_entry(int url_length, int signature_length,
                                   int referrer_length, int content_length)
{
    struct cache_entry *entry = malloc(sizeof(struct cache_entry));
    if (!entry)
        return NULL;
    entry->url = malloc(url_length+1); /* +1 for null terminator */
    if (!entry->url)
        return NULL;
    entry->browser_signature = malloc(signature_length+1);
    if (!entry->browser_signature)
        return NULL;
    entry->http_referrer = malloc(referrer_length+1);
    if (!entry->http_referrer)
        return NULL;
    entry->content = malloc(content_length+1);
    if (!entry->content)
        return NULL;
    return entry;
}
```

A. Briefly explain (one sentence) what can go wrong with the above code.

After Meddie Tartan realizes her mistake, she finds a fancy proxy binary that claims to do correctly what she wants her program to do. However, its source code was not available, so she tries disassembling it to see what the control flow looks like. She deduced a basic skeleton, as follows, of what this proxy's version of the same function does, but needs your help filling in the specifics (she thinks one or more uses of `goto` might be involved).

```
struct cache_entry *allocate_entry(int url_length, int signature_length,
                                   int referrer_length, int content_length)
{
    struct cache_entry *entry = malloc(sizeof(struct cache_entry));
    if (!entry)

        _____________________
    entry->url = malloc(url_length+1); /* +1 for null terminator */
    if (!entry->url)

        _____________________
    entry->browser_signature = malloc(signature_length+1);
    if (!entry->browser_signature)

        _____________________
    entry->http_referrer = malloc(referrer_length+1);
    if (!entry->http_referrer)

        _____________________
    entry->content = malloc(content_length+1);
    if (!entry->content)

        _____________________

    _____________________

    _____________________

    _____________________

    _____________________

    _____________________

    _____________________

    _____________________

    _____________________

    _____________________

    _____________________

    _____________________
}
```

B. Fill in the blanks above, using the assembly code on the next page.

```
000000000040058c <allocate_entry>:
  40058c:       48 89 5c 24 d8          mov    %rbx,-0x28(%rsp)
  400591:       48 89 6c 24 e0          mov    %rbp,-0x20(%rsp)
  400596:       4c 89 64 24 e8          mov    %r12,-0x18(%rsp)
  40059b:       4c 89 6c 24 f0          mov    %r13,-0x10(%rsp)
  4005a0:       4c 89 74 24 f8          mov    %r14,-0x8(%rsp)
  4005a5:       48 83 ec 28             sub    $0x28,%rsp
  4005a9:       89 fd                   mov    %edi,%ebp  # 1st argument
  4005ab:       41 89 f4                mov    %esi,%r12d # 2nd argument
  4005ae:       41 89 d5                mov    %edx,%r13d # 3rd argument
  4005b1:       41 89 ce                mov    %ecx,%r14d # 4th argument
  4005b4:       bf 20 00 00 00          mov    $0x20,%edi
  4005b9:       e8 aa fe ff ff          callq  400468 <malloc@plt>
  4005be:       48 89 c3                mov    %rax,%rbx
  4005c1:       48 85 c0                test   %rax,%rax
  4005c4:       74 7a                   je     400640 <allocate_entry+0xb4>
  4005c6:       8d 7d 01                lea    0x1(%rbp),%edi
  4005c9:       48 63 ff                movslq %edi,%rdi
  4005cc:       e8 97 fe ff ff          callq  400468 <malloc@plt>
  4005d1:       48 89 03                mov    %rax,(%rbx)
  4005d4:       48 85 c0                test   %rax,%rax
  4005d7:       74 5a                   je     400633 <allocate_entry+0xa7>
  4005d9:       41 8d 7c 24 01          lea    0x1(%r12),%edi
  4005de:       48 63 ff                movslq %edi,%rdi
  4005e1:       e8 82 fe ff ff          callq  400468 <malloc@plt>
  4005e6:       48 89 43 08             mov    %rax,0x8(%rbx)
  4005ea:       48 85 c0                test   %rax,%rax
  4005ed:       74 3c                   je     40062b <allocate_entry+0x9f>
  4005ef:       41 8d 7d 01             lea    0x1(%r13),%edi
  4005f3:       48 63 ff                movslq %edi,%rdi
  4005f6:       e8 6d fe ff ff          callq  400468 <malloc@plt>
  4005fb:       48 89 43 10             mov    %rax,0x10(%rbx)
  4005ff:       48 85 c0                test   %rax,%rax
  400602:       74 1e                   je     400622 <allocate_entry+0x96>
  400604:       41 8d 7e 01             lea    0x1(%r14),%edi
  400608:       48 63 ff                movslq %edi,%rdi
  40060b:       e8 58 fe ff ff          callq  400468 <malloc@plt>
  400610:       48 89 43 18             mov    %rax,0x18(%rbx)
  400614:       48 85 c0                test   %rax,%rax
  400617:       75 27                   jne    400640 <allocate_entry+0xb4>
  400619:       48 8b 7b 10             mov    0x10(%rbx),%rdi
  40061d:       e8 66 fe ff ff          callq  400488 <free@plt>
  400622:       48 8b 7b 08             mov    0x8(%rbx),%rdi
  400626:       e8 5d fe ff ff          callq  400488 <free@plt>
  40062b:       48 8b 3b                mov    (%rbx),%rdi
  40062e:       e8 55 fe ff ff          callq  400488 <free@plt>
  400633:       48 89 df                mov    %rbx,%rdi
  400636:       e8 4d fe ff ff          callq  400488 <free@plt>
  40063b:       ba 00 00 00 00          mov    $0x0,%rax
  400640:       48 8b 1c 24             mov    (%rsp),%rbx
  400644:       48 8b 6c 24 08          mov    0x8(%rsp),%rbp
  400649:       4c 8b 64 24 10          mov    0x10(%rsp),%r12
  40064e:       4c 8b 6c 24 18          mov    0x18(%rsp),%r13
  400653:       4c 8b 74 24 20          mov    0x20(%rsp),%r14
  400658:       48 83 c4 28             add    $0x28,%rsp
  40065c:       c3                      retq
```

Problem 6. Network Programming (12 points):

A. Consider a logging daemon, whose purpose is simply to accept messages from clients and log them to a file. (This is sometimes used in secure networks to ensure that, even if a cracker breaks into one computer, he can't erase his traces from the logs because they are stored elsewhere.)

In the following table, please list the system and/or library calls that each side of the connection would make. Each row should contain the name of a single system or library call, placed in the appropriate column. Please list the calls in the order they are called, not the order they return.

Please only include network-related operations (`accept`, `bind`, `close`, `connect`, `listen`, `read`, `socket`, and `write`).

You should make the following assumptions:

- The server finishes initializing before the client starts.
- The server only serves a single client.
- The client already knows the server's IP address.
- The client only sends one message before closing the connection.

Server:	**Client:**

B. Consider a simple network protocol for a webcam, in which a client connects to the server, and the server sends back 4096 bytes of image data. Suppose the client contains the following snippet of code to receive data from the server:

```
#define IMAGE_SIZE 4096
...
char *image_buf = malloc(IMAGE_SIZE);
if (!image_buf) {
  fprintf(stderr, "Out of memory.\n");
  exit(1);
}
...
int err = recv(server_fd, image_buf, IMAGE_SIZE, 0);
if (err < 0) {
  fprintf(stderr, "Error reading from server: %s\n", strerror(errno));
  exit(1);
}
...
/* BUG: Even if malloc and recv succeed, sometimes the data in image_buf
   only partially matches the image the server sent.  -- hqbovik */
```

What is the most likely cause of the bug?

Problem 7. Floats and Ints (10 points):

Conversion from float to int is a notoriously expensive operation. Not all processors do this in hardware. A clever technique uses the normalization step in double-precision floats. Imagine if you could normalize a (non-negative) floating point number so that the low-order significand bit represents 2^0. Then the significand would be an integer.

Assume a double where, after taking the exponent into account, the low-order bit represents 1 (2^0). What does the "implied leading 1" of the significand represent? _____

Now, assume a double x is known to be in the range $0 <= x < 2^{32}$. To force the low-order bit of x to represent 1 (2^0), you should (circle one:)

- *add* or
- *multiply*

by (fill in the blank:) _____

After this operation, you can store the double to memory and read the (circle one:)

- *upper* 32 bits (the end with the sign bit == 0), or
- *lower* 32 bits (the other end)

as an unsigned integer to get the integer value of the original double.

This algorithm will round (circle one:)

- toward zero,
- up,
- down,
- to the nearest even.

Hints:

- double-precision significand has 52 bits
- double-precision exponent has 11 bits
- double-precision sign-bit is (of course) 1 bit

Problem 8. Processes and Threads (12 points):

Consider the C code below. Assume all system calls are successful and that all processes run to completion.

```
#include <stdlib.h>
#define NUM_FORKS 4
char array[NUM_FORKS+2];
int pos = 0;
char outs[9] = {'1','1','8','5','2','2','4','1','3'};

void work(void* id){
  int index = (*((int*)id))*2;
  char writeMe = outs[index];
  array[pos++] = writeMe;
}

int main(){
  char three = '3';
  int i;
  int pid[NUM_FORKS];

  for(i = 0; i<NUM_FORKS; i++){
    if(!(pid[i] = fork())){
      work((void*)(&i));
      exit(0);
    }
    waitpid(pid[i], NULL, 0);
  }

  array[pos++] =  three;
  array[pos] = '\0';

  printf("%s", array);

  exit(0);
}
```

A. What is the output to the terminal?

B. Is there a race condition?

```
#include<stdlib.h>
#include<pthread.h>
#define NUM_THREADS 4
char array[NUM_THREADS+2];
int pos = 0;
char outs[9] = {'1','1','8','5','2','2','4','1','3'};

void* work(void* id){
  int index = (*((int*)id))*2;
  char writeMe = outs[index];
  array[pos++] = writeMe;
  return NULL;
}
int main(){
  char three = '3';
  int i;
  pthread_t tid[NUM_THREADS];

  for(i = 0; i<NUM_THREADS; i++){
    pthread_create(&(tid[i]), NULL, work, (void*)(&i));
    pthread_join(tid[i], NULL);
  }

  array[pos++] = three;
  array[pos] = '\0';

  printf("%s", array);

  exit(0);
}
```

C. What is the output to the terminal?

D. Is there a race condition?

Problem 9. Synchronization (14 points):

For each situation, state the one primitive that, when used correctly around the relevant critical section, prevents race conditions and results in the most concurrency. When more than one primitive will work with equal concurrency, give the primitive that is simplest, as defined below. If your answer is a semaphore, you *must specify its initial value*.

Your response to each question will be exactly one of the following primitives, listed here in order from simplest to most complex:

none needed, `mutex`, `condvar`, `semaphore(n)`, `rwlock`.

You may assume that all relevant information has been given to you. For example, if it is not explicitly stated that a thread writes to a variable, then there is no possible race condition involving writes. No additional logic or variables may be added to the programs; you are only wrapping critical sections with concurrency primitives.

Consider the following situations:

A. Two threads read from a global variable. _____

B. Two threads increment a global counter. _____

C. Two hundred threads search through a regular linked list of integers; one thread occasionally adds to the tail of the list. _____

D. Two hundred threads search through a regular linked list of integers; one thread occasionally removes and frees nodes from the list. _____

E. At most seven threads may be within the critical section simultaneously. _____

F. One thread waits, blocked, for events that occur about once every minute; it is acceptable for an event to be missed if a later event is handled. _____

G. One thread waits, blocked, for events that may occur at any time and are inserted into a queue when they do occur; it is unacceptable for any event to be missed. _____

Problem 10. Concurrency – Show me the money (10 points):

Consider the following code which is used to manage accounts. You can assume that integer overflows do NOT occur.

```
struct account
{
    int balance; // in dollars
    sem_t sem;   // mutex for this account, initialized to 1
};

struct account accounts[NUM_ACCOUNTS];

// transfers a dollar from one account to another
int transfer_dollar(int id_from, int id_to)
{
    // check if input ids are valid
    if (id_from < 0 || id_from >= NUM_ACCOUNTS ||
        id_to   < 0 || id_to   >= NUM_ACCOUNTS) {
        return -1;
    }

    // Lock accounts
    P(&accounts[id_from].sem);
    P(&accounts[id_to].sem);

    // make sure there is money to transfer
    if (accounts[id_from].balance == 0) {
        V(&accounts[id_to].sem);
        V(&accounts[id_from].sem);
        return -1;
    }

    // do transfer
    accounts[id_from].balance--;
    accounts[id_to].balance++;

    // Unlock accounts
    V(&accounts[id_to].sem);
    V(&accounts[id_from].sem);

    return 0;
}
```

A. Assume that there is only a single thread. Give input parameters which would cause the $transfer\_dollar$ function to deadlock.

B. Now assume that multiple threads can call the $transfer\_dollar$ function. Describe what is wrong with the locking scheme and then describe a possible solution.

Problem 11. File I/O (10 points):

Part A

Consider the following code:

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

  char buf[3] = "ab";
  int r = open("file.txt", O_RDONLY);
  int r1, r2, pid;

  r1 = dup(r);

  read(r, buf, 1);

  if((pid=fork())==0) {
    r1 = open("file.txt", O_RDONLY);
  }
  else{
    waitpid(pid, NULL, 0);
  }
  read(r1, buf+1, 1);

  printf("%s", buf);
  return 0;
}
```

Assume that the disk file `file.txt` contains the string of bytes `15213` . Also assume that all system calls succeed.

What will be the output when this code is compiled and run? _____

Part B

Consider the following code:

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

  char buf[4] = "abc";

  int r = open("file.txt", O_RDWR);
  int r1 = 0;
  int r2 = open("file.txt", O_RDWR);

  dup2(r, r1);

  read(r, buf, 1);

  read(r2, buf+1, 2);

  write(r, buf, 3);

  read(r2, buf, 1);

  write(r1, buf, 1);

  return 0;
}
```

r	r1	r2	buf	file.txt

Assume that the disk file `file.txt` originally contains the string of bytes `12345` . Also assume that all system calls succeed.

What will `file.txt` contain when this code is compiled and run? _____

Note: To help you, we have provided space to keep track of the program state. We recommend that you write in the values of the file position pointers corresponding to `r`, `r1`, and `r2`, and the characters in `buf` and `file.txt`, after each program step.

Problem 12. Extra Credit (4 points):

This problem is Extra Credit; do not attempt it until you have finished all other questions on the exam. This question is based on knowledge this class does not cover, and you are not expected to know how to solve it.

This problem deals with a tricky problem with GCC when run with high levels of optimization. This code in particular is compiled with

```
$ gcc -O3 input.c
```

One of your friends who hasn't taken 213 comes to you with a program, wanting your help. They tell you that they have been debugging it for hours, finally removing all their intricate code and just putting a single printf statement inside their loop. They show you this relevant code:

```
short a = 1024;
short b;

for(b=1000;;b++){
    if(a+b < 0){
        printf("Overflow!, stopping\n");
        break;
    }
    printf("%d ",a+b);
}
```

Their code never breaks and runs in an infinite loop. You, being a 213 student of course immediately ask to see the assembly dump:

```
08048380 <main>:
 8048380: 8d 4c 24 04           lea    0x4(%esp),%ecx
 8048384: 83 e4 f0              and    $0xfffffff0,%esp
 8048387: ff 71 fc              pushl  0xfffffffc(%ecx)
 804838a: 55                    push   %ebp
 804838b: 89 e5                 mov    %esp,%ebp
 804838d: 53                    push   %ebx
 804838e: bb e8 07 00 00        mov    $0x7e8,%ebx
 8048393: 51                    push   %ecx
 8048394: 83 ec 10              sub    $0x10,%esp
 8048397: 89 5c 24 04           mov    %ebx,0x4(%esp)
 804839b: 83 c3 01              add    $0x1,%ebx
 804839e: c7 04 24 70 84 04 08  movl   $0x8048470,(%esp)
 80483a5: e8 2e ff ff ff        call   80482d8 <printf@plt>
 80483aa: eb eb                 jmp    8048397 <main+0x17>
 80483ac: 90                    nop
 80483ad: 90                    nop
 80483ae: 90                    nop
 80483af: 90                    nop
```

A. From the programmer's point of view, what is wrong with this assembly code?

B. Why do you think gcc did this? (hint: we never mentioned this in class)

C. Please write the assembly code necessary to achieve the behavior intended by the programmer, and tell us where you would insert the code.

Problem 13. Extra Credit (up to 2 points):

A. Which instructor is a serious musician?

B. Which instructor is far too concerned with the success of University of Michigan sports teams?

C. How many of the course staff have first and last names that start with the same letter?

D. How many bits in a virtual address on a 32-bit architecture?

E. Draw a picture of the course staff (including special guest, Goger Ganberg) on board a boat (sailing away on post-semester vacation)... any picture will do, but the picture that makes us laugh most wins!