Andrew login ID: _____

Full Name: _____

CS 15-213, Fall 2001

Final Exam

December 13, 2001

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name and Andrew login ID on the front.
- 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 120 points.
- This exam is OPEN BOOK. You may use any books or notes you like. You may use a calculator, but no laptops or other wireless devices. Good luck!

1 (20):	
2 (10):	
3 (10):	
4 (8):	
5 (12):	
6 (6):	
7 (14):	
8 (10):	
9 (16):	
10 (14):	
TOTAL (120):	

Problem 1. (20 points):

We are running programs on a machine with the following characteristics:

- Values of type `int` are 32 bits. They are represented in two's complement, and they are right shifted arithmetically. Values of type `unsigned` are 32 bits.
- Values of type `float` are represented using the 32-bit IEEE floating point format, while values of type `double` use the 64-bit IEEE floating point format.

We generate arbitrary values `x`, `y`, and `z`, and convert them to other forms as follows:

```
/* Create some arbitrary values */
int x = random();
int y = random();
int z = random();
/* Convert to other forms */
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
double   dx = (double) x;
double   dy = (double) y;
double   dz = (double) z;
```

For each of the following C expressions, you are to indicate whether or not the expression *always* yields 1. If so, circle "Y". If not, circle "N". You will be graded on each problem as follows:

- If you circle no value, you get 0 points.
- If you circle the right value, you get 2 points.
- If you circle the wrong value, you get -1 points (so don't just guess wildly).

Expression	Always True?
`(x<y) == (-x>-y)`	Y N
`((x+y)<<4) + y-x == 17*y+15*x`	Y N
`~x+~y+1 == ~(x+y)`	Y N
`ux-uy == -(y-x)`	Y N
`(x >= 0) || (x < ux)`	Y N
`((x >> 1) << 1) <= x`	Y N
`(double)(float) x == (double) x`	Y N
`dx + dy == (double) (y+x)`	Y N
`dx + dy + dz == dz + dy + dx`	Y N
`dx * dy * dz == dz * dy * dx`	Y N

Problem 2. (10 points):

A C function `looper` and the assembly code it compiles to on an IA-32 machine running Linux/GAS is shown below:

```
looper:
  pushl %ebp
  movl %esp,%ebp
  pushl %esi
  pushl %ebx
  movl 8(%ebp),%ebx
  movl 12(%ebp),%esi
  xorl %edx,%edx
  xorl %ecx,%ecx
  cmpl %ebx,%edx
  jge .L25
.L27:
  movl (%esi,%ecx,4),%eax
  cmpl %edx,%eax
  jle .L28
  movl %eax,%edx
.L28:
  incl %edx
  incl %ecx
  cmpl %ebx,%ecx
  jl .L27
.L25:
  movl %edx,%eax
  popl %ebx
  popl %esi
  movl %ebp,%esp
  popl %ebp
  ret
```

```
int looper(int n, int *a) {
  int i;
  int x = _________________;

  for(i = _______________;

      ___________________;

      i++) {

    if (_____________________)

       x = ____________________;


    ______________________;
  }

  return x;
}
```

Based on the assembly code, fill in the blanks in the C source code.

Notes:

- You may only use the C variable names `n`, `a`, `i` and `x`, not register names.
- Use array notation in showing accesses or updates to elements of `a`.

Problem 3. (10 points):

Consider the following incomplete definition of a C struct along with the incomplete code for a function `func` given below.

```
typedef struct node {

  ________________ x;

  ________________ y;

  struct node *next;

  struct node *prev;

} node_t;
```

```
node_t n;

void func() {

  node_t *m;

  m = ________________________;

  m->y /= 16;

  return;
}
```

When this C code was compiled on an IA-32 machine running Linux, the following assembly code was generated for function `func`.

```
func:
  pushl %ebp
  movl n+12,%eax
  movl 16(%eax),%eax
  movl %esp,%ebp
  movl %ebp,%esp
  shrw $4,8(%eax)
  popl %ebp
  ret
```

Given these code fragments, fill in the blanks in the C code given above. Note that there is a unique answer.

The types must be chosen from the following table, assuming the sizes and alignment given.

Type	Size (bytes)	Alignment (bytes)
char	1	1
short	2	2
unsigned short	2	2
int	4	4
unsigned int	4	4
double	8	4

Problem 4. (8 points):

Consider the source code below, where `M` and `N` are constants declared with `#define`.

```
int array1[M][N];
int array2[N][M];

void copy(int i, int j)
{
    array1[i][j] = array2[j][i];
}
```

Suppose the above code generates the following assembly code:

```
copy:
  pushl %ebp
  movl %esp,%ebp
  pushl %ebx
  movl 8(%ebp),%ecx
  movl 12(%ebp),%eax
  leal 0(,%eax,4),%ebx
  leal 0(,%ecx,8),%edx
  subl %ecx,%edx
  addl %ebx,%eax
  sall $2,%eax
  movl array2(%eax,%ecx,4),%eax
  movl %eax,array1(%ebx,%edx,4)
  popl %ebx
  movl %ebp,%esp
  popl %ebp
  ret
```

What are the values of `M` and `N`?

`M` =

`N` =

The following problem will test your understanding of the runtime stack. You are given the following declarations on an x86 architecture:

```
struct file_spec {
  int fs_tag, parent_dir, size;
};

struct l_node {
  int tag;
  struct l_node *next;
};

struct l_node get_list_head()
{
  /* some irrelevant code */
}

void make_alias(..., ..., ..., ..., ...)
{
  /* some more irrelevant code */
}

void save_file(struct file_spec *file, int len, char *descriptor)
{
  int result = 0;
  struct l_node root = get_list_head();

  make_alias(..., ..., ..., ..., ...);

  /* yet more irrelevant code */
}
```

On the next page, you have the diagram of the stack immediately before the `call` assembly instruction for `make_alias()` in `save_file()` is executed. Argument `descriptor` given to `save_file()` is stored at `0xbffff5c0`. You can make the following assumptions:

- The function `make_alias()` takes exactly five arguments.
- The allocation order of local variables on the stack corresponds to their definition order in the source code.
- The compiler does not insert any additional unused space on the stack apart from unused space required for alignment restrictions of variables.
- No registers (apart from `%ebp`) are being saved on the stack.

Feel free to make comments or notes in the third column of the table - they will not be graded.

Address	Numeric Value	Comments/Description
0xbffff5c0	0xbffff7a0	
0xbffff5bc	0x000feedb	
0xbffff5b8	0xbffff780	
0xbffff5b4	0x080459ec	
0xbffff5b0	0xbffff630	
0xbffff5ac	0x00000000	
0xbffff5a8	0x83045c30	
0xbffff5a4	0x00000045	
0xbffff5a0	0xbffff5ac	
0xbffff59c	0x83045c30	
0xbffff598	0x00000045	
0xbffff594	0xbffff780	
0xbffff590	0x000feedb	
0xbffff58c	0xbffff7a0	

Problem 5. (12 points):

A. Give the current value of the frame pointer (machine register `%ebp`).

B. The declaration of `make_alias()` is missing the types of its parameters. Give the types in the order they would appear in the source code. The names of the parameters do not matter.

C. List the arguments passed to `make_alias()` in `save_file()`, in the order they would appear in the source code.

```
make_alias(        ,        ,        ,        ,        );
```

Problem 6. (6 points):

The following table gives the parameters for a number of different caches, where m is the number of physical address bits, C is the cache size (number of data bytes), B is the block size in bytes, and E is the number of lines per set. For each cache, determine the number of cache sets (S), tag bits (t), set index bits (s), and block offset bits (b).

Cache	m	C	B	E	S	t	s	b
1.	32	1024	4	4				
2.	32	1024	4	256				
3.	32	1024	8	1				
4.	32	1024	8	128				
5.	32	1024	32	1				
6.	32	1024	32	4				

Problem 7. (14 points):

Consider a direct mapped cache of size 64K with block size of 16 bytes. Furthermore, the cache is write-back and write-allocate. You will calculate the miss rate for the following code using this cache. Remember that `sizeof(int) == 4`. Assume that the cache starts empty and that local variables and computations take place completely within the registers and do not spill onto the stack.

A. Now consider the following code to copy one matrix to another. Assume that the `src` matrix starts at address 0 and that the `dest` matrix follows immediately follows it.

```
void copy_matrix(int dest[ROWS][COLS], int src[ROWS][COLS])
{
    int i, j;

    for (i=0; i<ROWS; i++) {
        for (j=0; j<COLS; j++) {
            dest[i][j] = src[i][j];
        }
    }
}
```

1. What is the cache miss rate if `ROWS = 128` and `COLS = 128`?
   Miss rate = _____%
2. What is the cache miss rate if `ROWS = 128` and `COLS = 192`?
   Miss rate = _____%
3. What is the cache miss rate if `ROWS = 128` and `COLS = 256`?
   Miss rate = _____%

B. Now consider the following two implementations of a horizontal flip and copy of the matrix. Again assume that the `src` matrix starts at address 0 and that the `dest` matrix follows immediately follows it.

```
void copy_n_flip_matrix1(int dest[ROWS][COLS], int src[ROWS][COLS])
{
    int i, j;

    for (i=0; i<ROWS; i++) {
        for (j=0; j<COLS; j++) {
            dest[i][COLS - 1 - j] = src[i][j];
        }
    }
}
```

1. What is the cache miss rate if `ROWS = 128` and `COLS = 128`?
   Miss rate = _____%
2. What is the cache miss rate if `ROWS = 128` and `COLS = 192`?
   Miss rate = _____%

```
void copy_n_flip_matrix2(int dest[ROWS][COLS], int src[ROWS][COLS])
{
    int i, j;

    for (j=0; j<COLS; j++) {
        for (i=0; i<ROWS; i++) {
            dest[i][COLS - 1 - j] = src[i][j];
        }
    }
}
```

1. What is the cache miss rate if `ROWS = 128` and `COLS = 128`?
   Miss rate = _____%
2. What is the cache miss rate if `ROWS = 192` and `COLS = 128`?
   Miss rate = _____%

Problem 8. (10 points):

Consider the following function for computing the product of an array of n integers. We have unrolled the loop by a factor of 3.

```
int aprod(int a[], int n)
{
    int i, x, y, z;
    int r = 1;
    for (i = 0; i < n-2; i+= 3) {
        x = a[i]; y = a[i+1]; z = a[i+2];
        r = r * x * y * z; // Product computation
    }
    for (; i < n; i++)
        r *= a[i];
    return r;
}
```

For the line labeled `Product computation`, we can use parentheses to create 5 different associations of the computation, as follows:

```
        r = ((r * x) * y) * z; // A1
        r = (r * (x * y)) * z; // A2
        r = r * ((x * y) * z); // A3
        r = r * (x * (y * z)); // A4
        r = (r * x) * (y * z); // A5
```

We express the performance of the function in terms of the number of cycles per element (CPE). As described in the book, this measure assumes the run time, measured in clock cycles, for an array of length n is a function of the form $Cn + K$, where C is the CPE.

We measured the 5 versions of the function on an Intel Pentium III. Recall that the integer multiplication operation on this machine has a latency of 4 cycles and an issue time of 1 cycle.

(continued)

The following table shows some values of the CPE, and other values missing. The measured CPE values are those that were actually observed. “Theoretical CPE” means that performance that would be achieved if the only limiting factor were the latency and issue time of the integer multiplier.

Version	Measured CPE	Theoretical CPE
A1	4.00	
A2	2.67	
A3		$4/3 = 1.33$
A4	1.67	
A5		$8/3 = 2.67$

Fill in the missing entries. For the missing values of the measured CPE, you can use the values from other versions that would have the same computational behavior. For the values of the theoretical CPE, you can determine the number of cycles that would be required for an iteration considering only the latency and issue time of the multiplier, and then divide by 3.

Problem 9. (16 points):

This problem tests your understanding of exceptional control flow in C programs. Assume we are running code on a Unix machine. The following problems all concern the value of the variable `counter`.

Part I (6 points)

```
int counter = 0;

int  main()
{
    int i;

    for (i = 0; i < 2; i ++){
        fork();
        counter ++;
        printf("counter = %d\n", counter);
    }

    printf("counter = %d\n", counter);
    return 0;
}
```

A. How many times would the value of `counter` be printed: _____

B. What is the value of `counter` printed in the first line? _____

C. What is the value of `counter` printed in the last line? _____

Part II (6 points)

```
pid_t pid;
int counter = 0;

void handler1(int sig)
{
    counter ++;
    printf("counter = %d\n", counter);
    fflush(stdout);   /* Flushes the printed string to stdout */
    kill(pid, SIGUSR1);
}

void handler2(int sig)
{
    counter += 3;
    printf("counter = %d\n", counter);
    exit(0);
}

main() {
    signal(SIGUSR1, handler1);
    if ((pid = fork()) == 0) {
        signal(SIGUSR1, handler2);
        kill(getppid(), SIGUSR1);
        while(1) {};
    }
    else {
        pid_t p; int status;
        if ((p = wait(&status)) > 0) {
            counter += 2;
            printf("counter = %d\n", counter);
        }
    }
}
```

What is the output of this program?

Part III (4 points)

```
int counter = 0;

void handler(int sig)
{
    counter ++;
}


int  main()
{
    int i;

    signal(SIGCHLD, handler);

    for (i = 0; i < 5; i ++){
        if (fork() == 0){
            exit(0);
        }
    }

    /* wait for all children to die */
    while (wait(NULL) != -1);

    printf("counter = %d\n", counter);
    return 0;
}
```

A. Does the program output the same value of `counter` every time we run it? `Yes` `No`

B. If the answer to A is `Yes`, indicate the value of the `counter` variable. Otherwise, list all possible values of the `counter` variable.

Answer: `counter` = _____

Problem 10. (14 points):

Consider an allocator with the following specification:

- Uses a single explicit free list.
- All memory blocks have a size that is a multiple of 8 bytes and is at least 16 bytes.
- All headers, footers, and pointers are 4 bytes in size
- Headers consist of a size in the upper 29 bits, a bit indicating if the block is allocated in the lowest bit, and a bit indicating if the previous block is allocated in the second lowest bit.
- Allocated blocks consist of a header and a payload (no footer)
- Free blocks consist of a header, two pointers for the next and previous free blocks in the free list, and a footer at the end of the block.
- All freed blocks are immediately coalesced.
- The heap starts with 0 bytes, never shrinks, and only grows large enough to satisfy memory requests.
- The heap contains only allocated blocks and free blocks. There are is no space used for other data or special blocks to mark the beginning and end of the heap.
- When a block is split, the lower part of the block becomes the allocated part and the upper part becomes the new free block.
- Any newly created free block (whether it come from a call to `free`, the upper part of a split block, or the coalescing of several free blocks) is inserted at the beginning of the free list.
- All searches for free blocks start at the head of the list and walk through the list in order.
- If a request can be fulfilled by using a free block, that free block is used. Otherwise the heap is extended only enough to fulfill the request. If there is a free block at the end of the heap, this can be used along with the new heap space to fulfill the request.

A. Simulating Malloc (10 points)

Below you are given a series of memory requests. You are asked to show what the heap looks like after each request is completed using a first fit and a best fit placement policy. The heap is represented as a series of boxes, where each box is a single block on the heap, and the bottom of the heap is the left most box. In each block, you should write the total size (including headers and footers) of the block in bytes and either 'f' or 'a' to mark it as free or allocated, respectively. For example, the following heap contains an allocated block of size 16, followed by a free block of size 32.

16 a	32 f

Assume that the heap is empty before each of the sequences is run. You do not necessarily have to use all the boxes provided for the heap. Some of the boxes are already filled in to help you.

	First Fit						Best Fit					
`ptr1 = malloc(32);`												
`ptr2 = malloc(16);`		24a										
`ptr3 = malloc(16);`												
`ptr4 = malloc(40);`												
`free(ptr3);`									24f			
`free(ptr1);`												
`ptr5 = malloc(16);`												
`free(ptr4);`			24a									
`ptr6 = malloc(48);`								24a				
`free(ptr2);`												

B. Code for Malloc (4 points)

For this part, you are asked to complete some small functions which are used to setup blocks. Each function will be missing a line of code and you are given three choices for this line of code. Circle the choice that completes the function correctly.

Function 1

```
/* Input: void *block: a pointer to a block
 *        unsigned long size: the size of the block,
 *        char alloc: the lower order bit indicates if this block is
 *                    allocated
 *        char palloc: the lower order bit indicates if the previous
 *                     block is allocated
 * Actions: This function will construct a header from the last 3
 *          parameters and place it in the header of the block pointed
 *          to by the first parameter.
 */
void make_header(void *block, unsigned long size, char alloc, char palloc)
{
        long header;

        ________;

        *(long *)block = header;
}
```

A. `header = (size >> 3) | ((alloc & 0x1) << 31) | ((palloc & 0x1) << 30);`
B. `header = (size & ~0x7) | (alloc & 0x1) | ((palloc & 0x1) << 1);`
C. `header = size | alloc | palloc;`

Function 2

```
/* Input: void *block: a pointer to a block
 *        char palloc: the low order bit indicates if the previous
 *                     block is allocated
 * Actions: Sets just the bit in the header indicating if the previous
 *          block is allocated.  Does nothing to the rest of the header.
 */
void set_palloc_in_header(void *block, char palloc)
{
        long curr = *(long *)block;

        __________;
}
```

A. `*(long *)block = (curr & ~0x2) | ((palloc & 0x1) << 1);`
B. `*(long *)block = (curr & ~(0x1 << 30)) | ((palloc & 0x1) << 30);`
C. `*(long *)block = (curr | (0x1 << 30)) & ~((palloc & 0x1) << 30);`