CMSC 313 Fall2009
Exam 3
December 18, 2009

Name_________________________________Score _________________

UMBC Username ____

Notes:

a. Please write clearly. Unreadable answers receive no credit.

b. There are no intentional syntax errors in any code provided with this exam. If you think you see an error that would affect your answer, please bring it to my attention

Multiple Choice (2 points each)

Write the letter of the BEST answer on the line provided for each question.

1. _______ When you run two programs (processes) on the same Linux machine, why do you not have to worry about them using the same physical memory?

A. The programmers who wrote the programs knew to avoid using the same addresses and were careful not to do so.

B. The virtual addresses used by the processes are translated to non-overlapping physical addresses.

C. Only one program really runs at a time, and the physical memory is saved/restored as part of each context switch.

D. The linker carefully lays out address spaces to avoid overlap of physical memory

2. _______ The most important job of a linker is

A. code optimizing
B. symbol resolution and relocation
C. providing information for the debugger
D. matching function prototypes with function definitions

3. _______ In the memory hierarchy, level ***k*** is considered a cache for level ***k+1*** because

A. it's smaller and faster
B. it's bigger and slower
C. it's always in memory
D. it's always on the CPU chip

4. _______ A "page fault" occurs when

A. your process makes a system call to the kernel
B. your process access a virtual page that is not in physical memory
C. your process calls a C library function
D. your process accesses memory that belongs to a different process

5. _______ Which of the following techniques is the fastest for organizing the heap's free list?

A. explicit free list
B. implicit free list
C. segregated free lists
D. they're all about the same

6. _______ In a heap memory allocator, "internal fragmentation" refers to

A. wasted space within an allocated block
B. overhead (headers, footers, etc) within a free block
C. free, but unusable blocks within the heap
D. "holes" in the heap

7. _______ A zombie process is created when

A. a child process terminates, but its parent process continues to run
B. a parent process terminates, but its child process continues to run
C. when a process is sent the SIGZOMBIE signal
D. the user types "control-Z" while the process is running

8. _______ A signal handler is

A. a function in the kernel that sends signals
B. a function in the kernel that executes when a process receives a signal
C. a user function that sends signals
D. a user function that executes when a process receives a signal

9._______ Suppose a function declares a local variable named **`my_int`** of type **`int`**. Which of the following (if any) is/are dangerous in C?

A. Returning **`&my_int`**
B. Assigning the value **`my_int`** to a global variable
C. Printing the address of **`my_int`** to the screen
D. None of the above

10._______The term "symbol reference" refers to

A. using the name of variable or function
B. defining and initializing a local variable
C. passing a parameter to a function by reference
D. defining and initializing a global variable

11. ______With respect to disk storage, the term "rotational latency" refers to

A. The number of sectors that pass under the read/write head on each revolution
B. The time required to wait for the desired sector to pass under the read/write head
C. The time it takes for the disk to make one complete revolution
D. The time required to wait for the desired track to pass under the read/write head

12. ______ The system call **`execl( )`** is used to

A. create a child process
B. load and execute a program
C. execute special code in the kernel
D. switch to "kernel mode"

13. ______ RAM is considered to be "volatile memory" because

A. its state is occasionally changed without warning
B. its state is lost when power is removed
C. its state fluctuates, but is always recoverable
D. none of the above

14. Symbols and Linking (18 points)

Consider the following three files, main.c, fib.c, and bignat.c, then answer the questions on the following page. Note that the functionality of the code is irrelevant to the question, so do not spend lots of time determining what the code does.

```
/* main.c */

void fib (int n);

int main (int argc, char** argv)
{
     int n = 0;
     sscanf(argv[1], "%d", &n);
     fib(n);
}


/* fib.c */

#define N 16
static unsigned int ring[3][N];

static void print_bignat(unsigned int* a)
{
     int i;
     for (i = N-1; i >= 0; i--)
          /* print a[i] as unsigned int */
          printf("%u ", a[i]);

     printf("\n");
}

void fib (int n)
{
     int i, carry;
     from_int(N, 0, ring[0]); /* fib(0) = 0 */
     from_int(N, 1, ring[1]); /* fib(1) = 1 */
     for (i = 0; i <= n-2; i++) {
          carry =
               plus(N, ring[i%3], ring[(i+1)%3], ring[(i+2)%3]);
          if (carry) {
               printf("Overflow at fib(%d)\n", i+2);
               exit(0);
          }
     }
     print_bignat(ring[n%3]);
}
```

Furthermore assume that a file `bignat.c` defines functions `plus` and `from_int` with the prototypes

```
int plus (int n, unsigned int* a, unsigned int* b, unsigned int* c);
void from_int (int n, unsigned int k, unsigned int* a);
```

A. (12 points) Fill in the following tables by stating for each name whether it is local or global, and whether it is strong or weak. Cross out any box in the table that does not apply. For example, cross out the first box in a line if the symbol is not in the symbol table, or cross out the second box in a line if the symbol is not global (and therefore neither weak nor strong). Recall that in C, external functions do not need to be declared.

`main.c`

Symbol	Local or Global	Strong or Weak
fib		
main		

`fib.c`

Symbol	Local or Global	Strong or Weak
ring		
print_bignat		
fib		
plus		

B. (6 points) Now assume that the file `bignat.c` is compiled and contained in a static library in archive format, `bignat.a` with global symbols `plus` and `from_int`.

For each of the following gcc commands, state if it

(A) compiles and links correctly, or
(B) linking fails due to undefined references, or
(C) linking fails due to multiple definitions .

Command Result (A, B, or C)

`gcc -o fib main.c fib.c bignat.a` **Result:** ___________

`gcc -o fib bignat.a main.c fib.c` **Result:** ___________

`gcc -o fib fib.c main.c bignat.a` **Result:** ___________

15. Heap Memory Allocation (15 points)

Consider an allocator with the following specification:

a. Uses a single explicit free list.
b. All memory blocks have a size that is a multiple of 8 bytes and is at least 16 bytes
c. All headers, footers, and pointers are 4 bytes in size
d. Headers consist of the block size in the upper 29 bits, a bit indicating if the block is allocated in the lowest bit (bit 0), and a bit indicating if the previous block is allocated in the second lowest bit (bit 1).
e. Allocated blocks consist of a header and a payload (no footer)
f. 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.
g. All freed blocks are immediately coalesced.
h. The heap starts with 0 bytes, never shrinks, and only grows large enough to satisfy memory requests.
i. 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.
j. When a block is split, the lower part of the block becomes the allocated part and the upper part becomes the new free block.
k. Any newly created free block (whether it comes 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.
l. All searches for free blocks start at the head of the list and walk through the list in order.
m. 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.

Below you are given a series of memory requests as they might appear in a user's program. 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 row of boxes, where each box is a single block on the heap, and the bottom of the heap is the left-most box. Simulate the calls to malloc( ) or free( ) on the left by marking each block in the corresponding row. 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.

16a	32f				

A. (6 points) Simulate the memory requests using **FIRST FIT** for block allocation

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

B. (6 points) Simulate the memory requests using **BEST FIT** for block allocation

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

C. (3 points) Mr. Frey's friend, Bob, believes he has discovered a new strategy for allocating blocks. His strategy is to use the LARGEST free block available to fulfill the memory allocation request. **<u>Using no more than two sentences</u>**, describe one possible advantage of Bob's new strategy.

16A. Dynamic Memory Coding -- 15 points

Consider a heap memory allocator that uses an ***implicit free list***. Each memory block, either allocated or free, has a size that is a multiple of eight (8) bytes. Thus, only the 29 higher order bits in the header and footer are needed to record the block size (in bytes), which includes the header and footer. The usage of the remaining 3 lower order bits is as follows:

bit 0 indicates the use of the current block: 1 for allocated, 0 for free.
bit 1 indicates the use of the previous adjacent block: 1 for allocated, 0 for free.
bit 2 is unused and is always set to be 0.

Five helper routines are defined to facilitate the implementation of **`free(void *p)`.** The functionality of each routine is explained in the comment above the function definition. **Write the letter** of the code that correctly completes the function on the line provided in the function body.

```
/* given a pointer p to an allocated block, i.e., p is a
     pointer returned by some previous malloc()/realloc() call;
     returns the pointer to the header of the block
*/
void * header(void* p)
{
     void *ptr;
     ____________________;
     return ptr;
}
     A. ptr = p - 1
     B. ptr = (void *)((int *)p - 1)
     C. ptr = (void *)((int *)p - 4)


/* given a pointer to a valid block header or footer,
     returns the size of the block
*/
int size(void *hp)
{
     int result;
     ____________________________;
     return result;
}

     A. result = (*hp) & (˜7)
     B. result = ((*(char *)hp) & (˜5)) << 2
     C. result = (*(int *)hp) & (˜7)
```

```
/* given a pointer p to an allocated block, i.e. p is
     a pointer returned by some previous malloc()/realloc() call;
     returns the pointer to the footer of the block
*/
void * footer(void *p)
{
     void *ptr;
     ________________________;
     return ptr;
}
     A. ptr = p + size(header(p)) - 8
     B. ptr = p + size(header(p)) - 4
     C. ptr = (int *)p + size(header(p)) - 2


/* given a pointer to a valid block header or footer,
     returns the usage of the current block,
     1 for allocated, 0 for free
*/
int allocated(void *hp)
{
     int result;
     ______________________;
     return result;
}

     A. result = (*(int *)hp) & 1
     B. result = (*(int *hp) & 0
     C. result = (*(int *)hp) | 1

/* given a pointer to a valid block header,
    returns the pointer to the header of previous block in memory
*/
void * prev(void *hp)
{
     void *ptr;
     __________________________;
     return ptr;
}
     A. ptr = hp - size(hp)
     B. ptr = hp - size(hp - 4)
     C. ptr = hp - size(hp - 4) + 4
```

16B. Dynamic Memory Coding -- 15 points

This problem tests your understanding of pointer arithmetic, pointer dereferencing, and malloc implementation. Jimmy J. Johnsom has implemented a simple explicit-list allocator. You may assume that his implementation follows the usual restrictions that you had to comply with in Project 6 such as the 8-byte alignment rule.

The following is a description of JJJ's block structure:

HDR	PAYLOAD	FTR

- HDR - Header of the block (4 bytes)
- PAYLOAD - Payload of the block (arbitrary size)
- FTR - Footer of the block (4 bytes)

The size of the **payload** of each block is stored in the header and the footer of the block. Since there is an 8-byte alignment requirement, the least significant of the 3 unused bits is used to indicate whether the block is free (0) or allocated (1).

For this problem, you can assume that:

- `sizeof(int) == 4 bytes`
- `sizeof(char) == 1 byte`
- `sizeof(short) == 2 bytes`
- `sizeof(long) == 4 bytes`
- The size of any pointer (e.g. `char *`) is 4 bytes.

Note that JJJ is working on a 32-bit machine. Also, assume that the block pointer `bp` points to the first byte of the payload.

Your task is to help JJJ compute the correct payload size (using the function `getPayloadSize()`), by indicating which of the following implementations of getHeader maro are correct. For each of the proposed solutions listed below, fill in the blank with either **C** for correct, or **I** for incorrect.

```
#define GET_SIZE(p) (GET_HDR(p) & ~0x7)
/* get_payload_size returns the actual size of payload.
     bp is pointing to the first byte of a block
     returned from JJJ'ss malloc() */

int get_payload_size(void *bp)
{
     return (int)(GET_SIZE(bp));
}

1. __________ #define GET_HDR(p) (*(int *)((int *)(p) - 1))
2. __________ #define GET_HDR(p) (*(int *)((char *)(p) - 1))
3. __________ #define GET_HDR(p) (*(int *)((char **)(p) - 1))
4. __________ #define GET_HDR(p) (*(char *)((int)(p) - 1)
5. __________ #define GET_HDR(p) (*(long *)((long *)(p) - 1)
6. __________ #define GET_HDR(p) (*(int *)((int)(p) - 4))
7. __________ #define GET_HDR(p) (*(int *)((short)(p) - 2))
8. __________ #define GET_HDR(p) (*(short *)((int *)(p) - 1))
```

17A. Process Control (18 points)

Carefully read the C code below, then answer the questions which follow

```
void handler(int sig) {
     printf("H\n");
     exit(0);
}

int main() {
     pid_t pid1, pid2;
     signal(SIGUSR1,handler);
     pid1 = fork();
     if (pid1 == 0) {
          pid2 = fork();
          printf("A\n");
          if (pid2 == 0) {
               printf("B\n");
               exit(0);
          }
          printf("C\n");
          kill(pid2,SIGUSR1);
          exit(0);
     }
     waitpid(pid1, NULL, 0)
     printf("D\n");
     exit(0);
}
```

Mark the top of each column that represents a valid possible output of this program with 'Yes' and each column which is impossible with 'No'

A	A	A	A	A	A
A	C	B	A	A	A
B	D	H	B	C	C
C	H	C	C	B	H
D		D	H	D	D
			D		

17B. Process Control (8 points)

Consider the following C program, with line numbers:

```
1 int main() {
2    int counter = 0;
3    int pid;
4
5    while (counter < 4 && !(pid = fork())) {
6         counter += 2;
7         printf("%d", counter);
8    }
9
10   if (counter > 0) {
11        printf("%d", counter);
12   }
13
14   if (pid) {
15        waitpid(pid, NULL, 0);
16        counter += 3;
17        printf("%d", counter);
18   }
29 }
```

Use the following assumptions to answer the questions:
• All processes run to completion and no system calls will fail.
• printf() immediately prints to the screen before returning.
• Logical operators such as && evaluate their operands from left to right and only evaluate the smallest number of operands necessary to determine the result. (short-circuit evaluation)

A. List all possible outputs of the program in the following blanks.
(You might not use all the blanks.)

______________________________ ______________________________

______________________________ ______________________________

______________________________ ______________________________

______________________________ ______________________________

______________________________ ______________________________

B. If we modified line 10 of the code to change the > comparison to >=, it would cause the program flow to print out zero counter values. With this change, how many possible outputs are there? (Just give a number, you do not need to list them all.)

NEW NUMBER OF POSSIBLE OUTPUTS = __________

17C. Signals (8 points)

Consider the following C program:

```
void handler1(int sig) {
     printf("Phantom\n");
     exit(0);
}

int main()
{
     pid_t pid1;
     signal(SIGUSR1, handler1);
     if((pid1 = fork()) == 0) {
          printf("Ghost\n");
          exit(0);
     }

     kill(pid1, SIGUSR1);
     printf("Ninja\n");
     return 0;
}
```

Use the following assumptions to answer the questions:
• All processes run to completion and no system calls will fail.
• printf() immediately prints to the screen before returning.

Mark each column that represents a valid possible output of this program with ‘Yes’ and each column which is impossible with ‘No’.

Phantom	**Ninja**	**Ghost**	**Ninja**	**Ninja**
Ninja	**Phantom**	**Ninja**	**Ghost**	**Phantom**
	Ghost	**Phantom**		**Ninja**

18. Locality (9 points)

The three functions below (`init1`, `init2`, and `init3`) refer to the definitions in the box in the upper-left. All perform the same operation -- initializing an array of PERSONs -- with varying degrees of spatial locality. Order the functions with respect to the spatial locality enjoyed by each. I.e. The function which best applies the concept of spatial locality should be listed first, the worst should be listed last.

```
#define N 1000
#define SPACE ' '

typedef struct {
     char initials[3];
     char ID[3];
} PERSON;

PERSON people[N];
```

```
void init1(PERSON *p, int n)
{
  int k, j;

  for (k = 0; k < n; k++) {
     for (j = 0; j < 3; j++)
          p[k].initials[j] = SPACE;
     for (j = 0; j < 3; j++)
          p[k].ID[j] = SPACE;
  }
}
```

```
void init2 (PERSON *p, int n)
{
  int k, j;

  for (k = 0; k < n; k++) {
     for (j = 0; j < 3; j++) {
          p[k].initials[j] = SPACE;
          p[k].ID[j]= SPACE;
     }
  }
}
```

```
void init3(PERSON *p, int n)
{
  int k, j;

  for (j = 0; j < 3; j++) {
     for (k = 0; k < n; k++)
          p[k].initials[j] = SPACE;
     for (k = 0; k < n; k++)
          p[k].ID[j] = SPACE;
  }
}
```

Write the function names on the lines provided in the order in which they best consider the concept of spatial locality

a. ____

b. ____

c. ____