Midterm Sample Answer
ECE 454F 2008: Computer Systems Programming
Date: Tuesday, Oct 28, 2008 3 p.m. - 5 p.m.

Instructor: Cristiana Amza
Department of Electrical and Computer Engineering
University of Toronto

Problem number	Maximum Score	Your Score
1	5	
2	10	
3	10	
4	20	
5	30	
6	25	
total	100	

This exam is open textbook and open lecture notes. You have two hours to complete the exam. Use of computing and/or communicating devices is NOT permitted. You should not need any such devices. You can use a basic calculator if you feel it is absolutely necessary.

Work independently. Do not remove any sheets from this test book. Answer all questions in the space provided. No additional sheets are permitted. Scratch space is available at the end of the exam.

Write your name and student number in the space below. Do the same on the top of each sheet of this exam book.

Your Student Number	
Your First Name	
Your Last Name	

Student Number: Name:

Problem 1. Basic Facts. (5 Points)

```
typedef struct
{
  char c[3];
} Obj;
```

Assume a program that you are writing is supposed to access N objects in sequence e.g., in a loop. The objects can either be *statically* allocated (as an array of objects) or dynamically allocated (as an array of pointers to objects). Furthermore, the dynamic allocation can use either an *implicit* list allocator or an *explicit* list allocator. Assume an empty heap initially and a first-fit policy for the dynamic allocators. Please order the three approaches above from best to worst in terms of estimated cache behavior of the program. Justify your answer briefly by stating any assumptions about the cache line size and sketching or stating the structure of allocated blocks (you can assume and use any variation of the techniques above learned in class for block allocation).
Answer:
Best: Statically allocated. 2nd: Implict list allocator. Worst: Explicit list allocator.

Problem 2. Profiling and Speedup. (10 Points)

Part A (5 points)

Explain the following profile data obtained using gprof on the n-body simulation application. Note: You have to explain the meaning of the numbers and the organization of the output. What conclusions can you draw from this profile data? Think of why we used gprof in the first place when drawing conclusions.

```
Flat profile sample: ..................
%      cumulative self              self     total
time   seconds    seconds    calls   s/call   s/call  name
96.33   1076.18   1076.18   278456   0.00     0.00  World::update()
 0.96   1086.85    10.67  212849617  0.00     0.00  ThreadManagerSerial::doSerialWork()
 0.63   1093.86     7.01        1    7.01    39.30  OpenGLWindow::runWindowLoop
 (char const*, int, int, bool (*)(), void (*)(SDL_KeyboardEvent*))
..................

Call graph sample:

index % time    self  children    called     name
..................
             1076.18    0.09  278456/278456   worldUpdate() [5]
[6]     96.3 1076.18    0.09  278456        World::update() [6]
                0.07    0.01  278240/278241   World::calculateTimeDeltaMilliseconds()[27]
                0.01    0.00    5338/5538
World::resetForegroundElement(World::ForegroundElement*) [38]
                0.00    0.00     149/149      World::updateCallsPerSecond(int) [47]
                0.00    0.00    4052/4287     World::radiusForMass(float) [139]
                0.00    0.00     642/642
```

Student Number: Name:

`World::swapForegroundElement(World::ForegroundElement*,World::ForegroundElement*)[140]`
`.....................`

Please also use the space on the next page if needed.
Answer:
You should provide a basic explanation of the profile data. For example, from the flat profile we see that 96.33% of the application running time is spent in the function World::update. In seconds this translates to 1076.18. This function is being called 278456 times and the time per call is in terms of milliseconds or less. Similarly for the other 2 functions in the flat profile.

The call graph profile is for the function World::update. This function is being called from worldUpdate (the parent) and it calls a number of other functions, such as World::calculateTimeDeltaMilliseconds, World::updateCallsPerSeconds, World::radiusForMass, World::swapForegroundElement (these are the children). As in the previous case, the application spends 96.33% in World::update, meaning 1076.18 s. Next you should say how much time is spent in each child. You should also explain the called column. From the called column we can see that worldUpdate is the only parent of World::update and there are 278456 calls to this function. The same for the children.

Conclusions: From the flat profile we see that World::update is the bottleneck of the application. From the call graph profile we notice that the bottleneck is actually inside this function and not in its children.

Part B (5 Points)

Assume that by profiling the code of our game application with gprof you get that 80% of the time is spent in a loop which is updating the physics of the game world, while 20% of the time is spent in frame rendering. Therefore you focus all your efforts only on improving the update world function through code optimizations such as loop unrolling, use of blocking for the update loop for cache hit improvements and loop parallelization. What is the best speedup you can get for the game application as a whole ?
Answer:
By Amdahl's Law, if $1/s$ of the program is sequential, then you can never get a speedup better than s.
For this question, $1/s = 0.2$, so the best speed up is 5.

Problem 3. Performance Optimization. (10 Points)

The following problem concerns optimizing codes for maximum performance on an Intel Pentium III. Recall the following performance characteristics of the functional units for this machine:

Operation	Latency	Issue Time
Integer Add	1	1
Integer Multiply	4	1
Integer Divide	36	36
Floating Point Add	3	1
Floating Point Multiply	5	2
Floating Point Divide	38	38
Load or Store (Cache Hit)	1	1

You've just joined a programming team that is trying to develop the world's fastest factorial routine. Starting with recursive factorial, they've converted the code to use iterations as follows:

```
int fact(int n) {
  int i;
  int result = 1;

  for (i = n; i > 0; i--)
    result = result * i;

  return result;
}
```

By doing so, they have reduced the number of cycles per element (CPE) for the function from around 63 to 4 (really!). Still, they would like to do better.

One of the programmers heard about loop unrolling. He generated the following code:

```
int fact_u2(int n) {
  int i;
  int result = 1;

  for (i = n; i > 1; i-=2) {
    result = (result * i) * (i-1);
  }

  return result;
}
```

(a) (5 points) Please compute the ideal CPE for his code and explain the performance of his code.
Answer: 4CPE. Performance is limited by the 4 cycle latency of integer multiplication.

(b) (5 points) You modify the line inside the loop to read:

```
result = result * (i * (i-1));
```

Please compute the ideal CPE for your code and explain the performance of your code.
Answer: 2CPE. The multiplication i * (i-1) can overlap with the multiplication by result from the previous iteration.

Problem 4. Cache Miss Rates. (20 Points)

After watching the presidential election you decide to start a business in developing software for electronic voting. The software will run on a machine with a 1024-byte direct-mapped data cache with 64 byte blocks.

You are implementing a prototype of your software that assumes that there are 7 candidates. The C-structures you are using are:

```
struct vote {
    int candidates[7];
    char valid;
};

struct vote vote_array[16][16];
register int i, j, k;
```

You have to decide between two alternative implementations of the routine that initializes the array `vote_array`. You want to choose the one with the better cache performance.

You can assume:

- `sizeof(int) = 4`
- `sizeof(char) = 1`
- `vote_array` begins at memory address 0
- The cache is initially empty.
- The only memory accesses are to the entries of the array `vote_array`. Variables `i`, `j` and `k` are stored in registers.

A. What percentage of the writes in the following code will miss in the cache?

```
for (i=0; i<16; i++){
    for (j=0; j<16; j++) {
```

```
        vote_array[i][j].valid=0;
    }
}

for (i=0; i<16; i++){
    for (j=0; j<16; j++) {
        for (k=0; k<7; k++) {
            vote_array[i][j].candidates[k] = 0;
        }
    }
}
```

Miss rate in the first loop: _____ %

Answer: (16x8)/(16x16) = 1/2 = 50%

Miss rate in the second loop: _____ %

Answer: (16x8)/(16x16x7)=1/14=7.14%

Overall miss rate for writes to `vote_array`: _____ %

Answer: (16x16)/(16x16x8)=1/8 = 12.5 %

B. What percentage of the writes in the following code will miss in the cache?

```
for (i=0; i<16; i++){
    for (j=0; j<16; j++) {
        for (k=0; k<7; k++) {
            vote_array[i][j].candidates[k] = 0;
        }
        vote_array[i][j].valid=0;
    }
}
```

Miss rate for writes to `vote_array`: _____ %

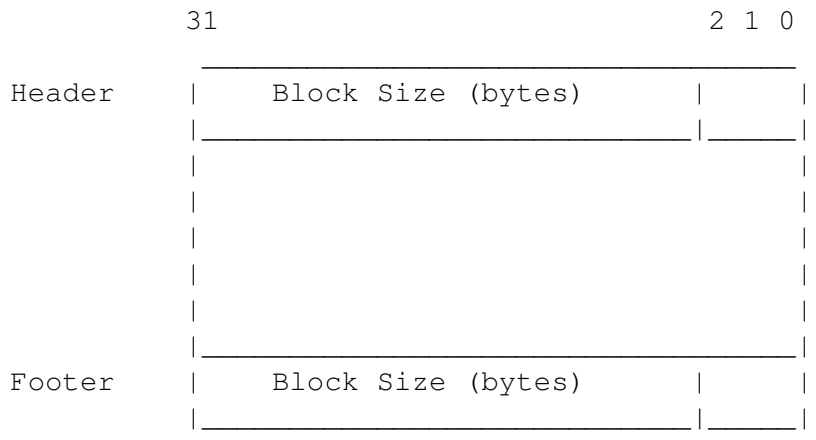
Answer:(16x8)/(16x16x8)=1/16=6.25%

Student Number: Name:

Problem 5. Dynamic Memory Allocation (30 Points)

Consider an allocator that uses an implicit free list.

The layout of each free memory block is as follows:



Each memory block, either allocated or free, has a size that is a multiple of eight bytes. Thus, only the 29 higher order bits in the header and footer are needed to record block size, which includes the header (and footer for free blocks). 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.

Part A. (12 points)

Consider the allocator that uses implicit free lists. We make the following space optimization: *Allocated* blocks do not have a footer. *Free* blocks still have the footer and the rest of the block structure is the same as before.

Using the space optimization above and given the contents of the heap shown on the left, we executed `malloc(5) and free(0x400b010)` in sequential order.

Please show the new contents of the heap after `malloc(5)` is executed in the 2nd table, and show the new contents of the heap after `free(0x400b010)` is executed in the 3rd table. Your answers should be given as hex values. Update only the necessary boundary tags. Note that the address grows from bottom up. Assume that the allocator uses a *first fit* allocation policy.

Student Number: Name:

Address	Content
0x400b028	0x00000012
0x400b024	0x400b621c
0x400b020	0x400b620c
0x400b01c	0x00000012
0x400b018	0x400b512c
0x400b014	0x400b511c
0x400b010	0x400b601c
0x400b00c	0x00000011
0x400b008	0x0000000a
0x400b004	0x0000000a
0x400b000	0x400b511c
0x400affc	0x00000009

Student Number: Name:

Answer for after malloc(5):

Address	Content
0x400b028	0x00000012
0x400b024	0x400b621c
0x400b020	0x400b620c
0x400b01c	**0x00000013**
0x400b018	0x400b512c
0x400b014	0x400b511c
0x400b010	0x400b601c
0x400b00c	0x00000011
0x400b008	0x0000000a
0x400b004	0x0000000a
0x400b000	0x400b511c
0x400affc	0x00000009

Student Number: Name:

Answer for for free(0x400b010):

Address	Content
0x400b028	0x00000012
0x400b024	0x400b621c
0x400b020	0x400b620c
0x400b01c	**0x00000011**
0x400b018	**0x0000001a**
0x400b014	0x400b511c
0x400b010	0x400b601c
0x400b00c	0x00000011
0x400b008	0x0000000a
0x400b004	**0x0000001a**
0x400b000	0x400b511c
0x400affc	0x00000009

Part B. (18 points)

Assume that you want to extend the previous implicit allocator to improve its performance. You would like to reduce allocation time by maintaining an explicit doubly-linked free list. You also want to improve its memory utilization by using the footer only when a block is free. You decide that a first-fit search algorithm is sufficient. You may assume that: `sizeof(void *) is 4`.

(a) (4 points) Given that the block size must be a multiple of 8 bytes, what is the minimum block size allowable under this scheme?
Answer: 16 bytes (4 for header, 4 for footer, 4 x 2 for next and prev pointers)

(b) (4 points) Given that the block size must be a multiple of 8 bytes, determine the amount of memory (in bytes), wasted due to internal fragmentation, for each of the following four allocation requests. You should also specify the sources of internal fragmentation.

malloc(1)
Answer: 4 for header, 1 for data, 11 for padding
malloc(5)
Answer: 4 for header, 5 for data, 7 for padding
malloc(12)
Answer: 4 for header, 12 for data, no padding
malloc(13)
Answer: 4 for header, 13 for data, 7 padding
Internal fragmentation: 25 bytes (11 for malloc(1) + 7 for malloc(5) + 0 for malloc(12) + 7 for malloc(13))

In order to further improve the performance of your allocator, you decide to try to implement an explicit binary tree data structure to enable a fast best-fit search through the free blocks. Each free block within the tree must now maintain a pointer to each of its children, and to its parent.

(c) (4 points) Assuming that the block size must still be a multiple of 8 bytes, what is the minimum block size allowable under this new scheme?
Answer: 24 bytes (4 for header, 4 for footer, 4 x 3 for pointers + 4 for padding)

(d) (6 points) Comment on the effect that this allocator has on memory utilization when compared to the previous explicit linked list first-fit allocator. You should discuss any opposing tensions (trade-offs) that might exist.
Answer: The effect on memory utilization will depend on the allocation requests. This allocator will, on average, reduce the amount of external fragmentation because of the best-fit replacement policy; however, it suffers from higher internal fragmentation for small (less than 20 bytes) memory allocation requests.

Problem 6. Parallel Programming (25 Points + 5 Bonus Points)

Part A (5 points)

Consider the following code, which we want to parallelize.

```
do i = 1, n
   a [2*i] = b[i] + c[i]
   d[i] = a [2*i+1]
enddo
```

Is parallelization possible by just splitting the loop iterations on several processors ? Please explain briefly.
Answer: Yes, no dependency carried by this loop.

Part B (20 points + 5 bonus points)

Consider the following code.

(a). (5 points) Explain what dependencies there are in the following code:

```
for (i = 0; i < 100000; i++)
   a[i + 1000] = a[i] + 1;
```

```
Answer:
Dependences between a[0], a[1000], a[2000] ...
                    a[1], a[1001], a[2001] ...
Dependence distance is 1000
```

(b). (15 points) Rewrite the loop above to allow parallel execution and point out which part of the code can be parallelized.

```
 Use the "#pragma omp parallel for" notation.
```

```
Answer 1: (not ideal - inner loop, not outer loop is parallelized):

for (i = 0; i < 100; i++)
#pragma omp parallel for
for (j=i*1000; j < (i+1)*1000; j++)
  a[j+1000] = a[j] + 1;
```

```
Answer 2:
```

```
#pragma omp parallel for private(stride)
for (i=1; i<100; i++){
    stride = i*1000;
    for(j=0; j<1000; j++)
        a[stride+j] = a[j] + i;
}

This manages to parallelize the outer loop, which is more
efficient. The way it works is:
for i 1000...1999 a is incremented by 1
for i 2000...2999 a is incremented by 2 etc.

This is how the original algorithm works.

Answer 3 (also parallelizes the outer loop,
 but poor cache locality and high risk of false sharing):

#pragma omp parallel for private j
for (i = 0; i < 1000; ++i)
   for (j = 0; j < 100; ++j)
       a[i+(j+1)*1000] = a[i+j*1000] + 1;
```

(c). (5 bonus points) Rewrite your parallel code if necessary to optimize both its cache performance and parallelism.