ECE 2035(A) Programming for Hardware/Software Systems
Fall 2013 Exam Three November 20th 2013

Name:

Q1: /8

Q2: /30

Q3: /30

Q4: /32

Total: /100
For functional call related questions, let’s assume the following convention is used when implementing function calls (which is exactly the same as we discussed in class):

1. The stack grows downwards unless specified otherwise.
2. The stack contains the following items, in the order that they are allocated on the stack:
   - Return address
   - Frame Pointer
   - Input Parameters
   - Return Value
   - Local Variables of the Callee
3. All parameters are passed via the stack (none are passed via registers)
4. Caller only allocates part of the activation frame for the callee before calling 'jal' - it only allocates until the return value (including return value) and updates the stack pointer accordingly (so it points to the return value).
5. Caller pushes input parameters onto the stack in the reverse order as the parameters are declared.
6. Callee is responsible of populating the rest of the activation frame (just the local variables).
7. Callee is responsible of deallocating the local variable part of the activation frame when it returns
8. Caller is responsible of deallocating the rest of the callee’s activation frame.

In addition to the above, we also have the following:

9. The local variables are allocated in the same order as they were declared.
10. Stack pointer points to the last occupied slot on the stack.
Question 1: Function Pointers  8 points

In pthread, pthread_create() has the following prototype:

```c
int pthread_create( pthread_t *thread,
                const pthread_attr_t *attr,
                void (**start_routine) (void *),
                void *arg);
```

The third parameter is a function pointer that points to the function to be executed by the newly created thread.

What’s the name of the function pointer in the above prototype? 2pts

**start_routine**

Given a function pointed by this pointer, what’s its input data type? 3pts

**void** *

Given a function pointed by this pointer, what’s type of its return value? 3pts

**void** *
**Question 2: Concurrency 30 points**

Part 1: True or False, circle your answer: 8 points

1. True  **False** Multi-threaded programs can be executed on multi-core systems only.
2. True  **False** Threads within a process cannot have shared accesses to heaps.
3. **True** False Threads within a process have private stack pointers.
4. **True** False Threads within a process can have the same start routine.

Part 2: Suppose we have the following program: 22 points

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int x = 0;

void *work(void *t) {
    int y = 0, *p;
    p = (int *) t;
    *p = *p + 1;
    y = y + 1;
    printf("%d\n", x);
    printf("%d\n", y);
    return null;
}

int main() {
    pthread_t threads[4];
    pthread_create(&threads[0], NULL, work, (void *) &x);
    pthread_create(&threads[1], NULL, work, (void *) &x);
    pthread_exit();
}
```

What are the possible values that might print for x? (list all the possibilities, though it will be printed twice by the two threads) 11 pts

1, 2

What are the possible values that might print for y? (list all the possibilities, though it will be printed twice by the two threads) 11 pts

1
Question 3 stack and heap (30 pts)

Part A (15 points): Suppose we have the following program written in C.

```c
int foo(int n) {
    int *y, i;
    y = (int *)malloc(n * 1024 * sizeof(int)); //sizeof(int) is 4
    //some usages of y
    return 0;
}

int main() {
    int i;
    for(i=1;i<10000;i++) {
        foo(1024);
    }
    return 0;
}
```

The above program has the classical problem of memory leak: the program keeps allocating more and more memory, and never frees. Eventually, the system will run out of memory.

We have the following assumptions about the above program:

1. The code and global variables take 1Mbytes
2. The stack of `main()` takes 1Mbytes
3. Stack starts from the top of the address space and grows downward
4. Heap starts right on top of the code and global variable, and grows upward
5. The address space is 32-bit, and a process can use up the entire 4GB address space.

1. Where does `malloc()` (pointed by `y`) get the allocation? Answer heap or stack. 5 points

**Heap**

2. During which iteration will the above program deplete all the usable memory? Answer this question using the value of index `i` in the for loop in `main`. 10 pts

<table>
<thead>
<tr>
<th>code and global:</th>
<th>1MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack of <code>main</code>:</td>
<td>1 MB</td>
</tr>
<tr>
<td>stack of each call to <code>foo</code>:</td>
<td>4MB + 24B (1 return address, 1 frame pointer, 1 input, 1 return value, 2 local variables)</td>
</tr>
<tr>
<td>so we have:</td>
<td>1MB + 1MB + n(4M+24B) &lt; 4096MB</td>
</tr>
<tr>
<td>so:</td>
<td>n &lt;1023.49</td>
</tr>
<tr>
<td>so:</td>
<td>we can do at most 1023 iterations.</td>
</tr>
</tbody>
</table>
Part B (15 points): suppose the program now changes to:

```c
int foo(int n) {
    int y[n*1024], i;
    //some usages of y;
    return 0;
}

int main() {
    int i;
    for(i=1;i<10000;i++) {
        foo(1024);
    }
    return 0;
}
```

This program uses the Variable Length Array (VLA) feature of C99.

1. Where does array y get its allocation? Answer Heap or Stack. 5 points

**stack**

2. Does the program still have the memory leak problem (that its memory usage will grow continuously)? 5pts

**NO**

3.A If the answer is yes, will a garbage collector solve the memory leak problem for this program? 5pts

**N/A**

3.B If the answer is no, what is the maximum amount of memory that will be consumed by the above program? 5pts

- **code + global:** 1MB
- **stack of main:** 1MB
- **each call to foo:** 4MB + 20B

**Total consumption:** 6MB + 20B
Question 4 heap management (32 pts)

For this question, automatic garbage collection is used for heap management:

1. The heap contains user allocated objects.
2. The objects can be as simple as a variable or an array, or as complicated as a struct.
3. For each user allocated object, the management scheme will allocate one extra word. This extra word indicates the size of the user object, and is placed right in front of the object in the memory.
4. User programs do not do free(), a garbage collector will put allocated memory regions back to the free list when they will not be used any more.
5. The free list is not maintained within the heap. Instead, it is maintained somewhere else in the address space, in the form of a linked list.

Below is a snapshot of a heap storage. Values that are pointers are denoted with a "$". The heap has been allocated continuously beginning at $2000 with no gaps between objects. The heap pointer is $2148 (which indicates the heap can grow from this address upwards).

<table>
<thead>
<tr>
<th>addr</th>
<th>value</th>
<th>addr</th>
<th>value</th>
<th>addr</th>
<th>value</th>
<th>addr</th>
<th>value</th>
<th>addr</th>
<th>value</th>
<th>addr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>16</td>
<td>2028</td>
<td>8</td>
<td>2056</td>
<td>2152</td>
<td>2084</td>
<td>24</td>
<td>2112</td>
<td>16</td>
<td>2140</td>
<td>2076</td>
</tr>
<tr>
<td>2004</td>
<td>2024</td>
<td>2032</td>
<td>16</td>
<td>2060</td>
<td>32</td>
<td>2088</td>
<td>16</td>
<td>2116</td>
<td>$2148</td>
<td>2144</td>
<td>2020</td>
</tr>
<tr>
<td>2008</td>
<td>44</td>
<td>2036</td>
<td>8</td>
<td>2064</td>
<td>$2020</td>
<td>2092</td>
<td>$2040</td>
<td>2120</td>
<td>28</td>
<td>2148</td>
<td>0</td>
</tr>
<tr>
<td>2012</td>
<td>$2088</td>
<td>2040</td>
<td>16</td>
<td>2068</td>
<td>16</td>
<td>2096</td>
<td>0</td>
<td>2124</td>
<td>0</td>
<td>2152</td>
<td>0</td>
</tr>
<tr>
<td>2016</td>
<td>4</td>
<td>2044</td>
<td>72</td>
<td>2072</td>
<td>36</td>
<td>2100</td>
<td>$2080</td>
<td>2128</td>
<td>64</td>
<td>2156</td>
<td>0</td>
</tr>
<tr>
<td>2020</td>
<td>12</td>
<td>2048</td>
<td>24</td>
<td>2076</td>
<td>4</td>
<td>2104</td>
<td>12</td>
<td>2132</td>
<td>12</td>
<td>2160</td>
<td>0</td>
</tr>
<tr>
<td>2024</td>
<td>$2040</td>
<td>2052</td>
<td>20</td>
<td>2080</td>
<td>2056</td>
<td>2108</td>
<td>8</td>
<td>2136</td>
<td>$2088</td>
<td>2164</td>
<td>0</td>
</tr>
</tbody>
</table>
Part A  Suppose the stack holds a local variable whose value is the memory address $2004$, and register $1$ holds the address $2136$. No other registers or static variable currently hold heap memory addresses. List the addresses of all objects in the heap that are not garbage. 8 pts

\textbf{2004, 2136, 2088, 2040, 2080}

Part B. Create a sorted (by size) free list by scanning the memory for garbage, starting at address $2000$ and inserting each garbage object into the free list in increasing size order. List the base of address of each object (not the address of the one-word header) on the free list (in order). Indicate the size of each object in a pair of parenthesis after its base address. 8 pts

\textbf{2024(12), 2116(16), 2052(24)}

Part C. Based on the free list created in Part B, if an object of size 14 bytes is to be allocated, what address will be returned using a best-fit allocation strategy? 8 pts

\textbf{2116}

Part D. Suppose the activation frame that contains the local variable of address $2004$ is now deallocated. Which address will be reclaimed by the Mark and Sweep garbage collection strategy? If this causes two continuous regions to be freed, you do not need to concatenate them. 8 pts

\textbf{Only 2004 will.}