ECE 2035(B) Programming for Hardware/Software Systems
Spring 2013 Exam Three April 10\textsuperscript{th} 2013

Name:

Q1: /14

Q2: /12

Q3: /8

Q4: /8

Total: /42
For functional call related questions, let's assume the following convention is used when implementing function calls (**exactly the same as in exam 2, repeated below**):

1. The stack grows downwards unless specified otherwise.
2. Frame pointer is not used, so it doesn't need to be preserved and restored.
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 the input parameters on the stack, and updates the stack pointer accordingly (**right on top of the input parameters**).
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, which includes the return address, return value, and local variables.
7. caller is responsible of deallocating the entire activation frame (of the callee) when the callee returns.
8. The activation frame of a function is populated in the following order:
   1) input parameters (**allocates only if there are any input parameters**)
   2) return address
   3) return value (**allocates only if the function does return a value**)
   4) local variables (if any)

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

9. The local variables are allocated in the same order as they were defined.
10. Stack pointer points to the next available slot on the stack.
**Question 1 stack and heap (14 pts)**

Suppose we have the following program written in C.

```c
int foo(int n) {
    int *y, i;
    y = (int *)malloc(n*1024);
    for(i=0;i<n*1024/4;i++) {
        y[i] = i;
    }
    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.

Part A: 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.

4pts

*Each call to foo(1024) will consume 1024*1024 (heap) + 20 bytes (stack).*

\[
1 \text{ (code)} + 1 \text{ (main stack)} + x \times (1024 \times 1024 + 20) / 1024^2 \leq 4096 \text{ (address space)} \\
x < 4093.9. \text{ So } x \text{ can only go as high as 4093 before the program depletes all usable memory.}
\]

Part B: Now suppose the above program has built-in support of garbage collection. And the garbage collector runs periodically. In order to keep memory usage of the above program capped by 100Mbytes, how often does the garbage collector need to
run? Answer this question using the number of iterations in the for loop in main.
4pts
1 (code) + 1 (main stack) + x*(1024*1024+20)/1024^2 \leq 100 (address space)
x < 97.998
So x needs to be at most 97, or smaller.

Part C: suppose the program now changes to: 6pts

```c
int foo(int n) {
    int *y, i;
    y = (int *)malloc(n*1024);
    for(i=0;i<n*1024/4;i++) {
        y[i] = i;
    }
    if(n<10000) {
        foo(n+1);
    }
    return 0;
}

int main() {
    int i;
    foo(1024);
    return 0;
}
```

Does the program still have the memory leak problem (that its memory usage will grow continuously)? 3pts

Yes. It allocates \( n \times 1024 \) bytes at every recursive function call, and does not deallocate.

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

No. The allocated memory are alive throughout the execute of the program, so it will not be reclaimed by any garbage collector. This is a programming error that garbage collector cannot help.

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

This question does not apply.
Question 2 heap management (12 pts)

For this question, the management of the heap takes a different scheme than the sample problem that was discussed in class. So pay attention here.

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 aps 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. 3 pts

$2004, 2136, 2088, 2040, 2080$

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. 3pts

$2024(12), 2116(16), 2052(24)$

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? 3pts

$2116$

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 free, you do not need to concatenate them. 3pts

*Only 2004 will.*
**Question 3: 8 points**

In this question, we assume that garbage collection does not exist, and programmers need to manually manage memory usages via ‘malloc’ and ‘free’. Each allocated memory region is preceded by a one-word header indicating the size (number of bytes) of the region, which will be used by ‘free’ to determine how many bytes need to be returned to the free list. We also assume that heap memory will be continuously allocated without any gaps starting from the beginning of the heap (of course, when the heap is not fragmented).

Part A: What is the consequence if a pointer to a stack-allocated object is passed to the function `free()`? 4pts

*A segment of the stack space will be put on the free list. A later malloc may use this for memory allocation, which will destroy the proper content on the stack and most likely cause the program to crash.*

Part B: Given the following code:

```c
main() {
    int *x, *y, i, sum;
    x = (int *) malloc(400);
    y = (int *) malloc(400);
    for (i=0; i<=100; i++) {
        x[i] = i;
    }
    for (i=0; i<100; i++) {
        y[i] = 2*i;
    }
    sum = 0;
    for (i=0; i<100; i++) {
        sum = sum + y[i] + x[i];
    }
    free(x);
    free(y);

    return 0;
}
```

The above code has a heap management problem. What is it? 4pts

The first for loop goes over the boundary of the allocated 400 bytes for array `x`. According to the assumptions, it will actually overwrite the header word of the array `y`. When `free(y)` is executed, it will use wrong size information (should be 400 bytes, but now updated to 100), which will free less than what was allocated.
Question 4: Project 1, 8pts

Project 1 baseline MIPS code is listed below

.data
TileGrid: .alloc 25
Solution: .alloc 25

.text
Puzzle: addi $1, $0, TileGrid
swi 563 # Create Tile Puzzle
andi $6, $2, 0xF # $6: Lx
sli $6, $6, 2 # $6: 4*Lx
srl $7, $2, 4 # $7: Ly
mult $6, $7
mflo $8 # $8: 4*Lx*Ly
addi $1, $0, 0 # $1: I
add $3, $8, $0 # $3 = 4*Lx*Ly

ClrSol: addi $3, $3, -4 # dec $3
sw $0, Solution($3) # clear Solution entry
bne $3, $0, ClrSol # if not finished clearing Solution, loop

Loop: slt $3, $1, $0 # is I < 0?
bne $3, $0, Exit # if so, exit loop
slt $3, $1, $8 # is I < TileGrid size?
beq $3, $0, Exit # if not, exit loop
addi $29, $29, -4 # push return address
sw $31, 0($29) # ($31) on stack
jal RotateUntilFit # input: I ($1), return R in $10
lw $31, 0($29) # pop return address
addi $29, $29, 4 # from stack

# if rotated successfully (i.e., rotation amt (R) < 4)
# then set Solution[I] = R, inc I, and continue Loop
# else backtrack until find fit at new index I and continue Loop w/ I++

slti $3, $10, 4 # is R < 4?
beq $3, $0, Back # if not (there is no rotation for which TG[I]
# will match its neighbors, so backtrack)
sw $10, Solution($1) # else (R<4): set Solution[I] = R and loop
j Continue

Back: addi $29, $29, -4 # push return address
sw $31, 0($29) # ($31) on stack
jal Backtrack # return R in $10
lw $31, 0($29) # pop return address
addi $29, $29, 4 # from stack

Continue: addi $1, $1, 4 # I++
j Loop

Exit: addi $4, $0, Solution
swi 569 # Apply Rotations
jr $31

RotateUntilFit: lw $10, Solution($1) # initial rotation amount
addi $5, $1, TileGrid # address of current tile
# check fit
div $1, $6 # I / Lx*4
mflo $2 # row number
mfhi $4 # col number * 4

RotLoop: beq $4, $0, WestOK # if column 0, don't check West
lb $14, 2($5) # current tile's West triangle
lb $15, -4($5) # tile to West's East triangle
bne $14, $15, NoFit # if adjacent tiles don't match, NoFit

WestOK: beq $2, $0, NorthOK # if row 0, don't check North
lb $14, 3($5) # current tile's North triangle
sub $3, $5, $6 # addr of tile to North
lb $15, 1($3) # tile to North's South triangle
bne $14, $15, NoFit  # if adjacent tiles don't match, NoFit
NorthOK:
  addi $14, $4, 4  # col num + 4
  div $14, $6  # col num + 4 / Lx * 4
  mfhi $14  # col num + 4 % Lx * 4
  beq $14, $0, EastOK  # if on rightmost col, don't check East
  lb $14, 0($5)  # current tile's East triangle
  lb $15, 4($5)  # tile to East's East triangle
  beq $14, $15, EastOK  # current tile East=member of East nbr
  lb $15, 5($5)  # tile to East's South triangle
  beq $14, $15, EastOK  # current tile East=member of East nbr
  lb $15, 6($5)  # tile to East's West triangle
  beq $14, $15, EastOK  # current tile East=member of East nbr
  lb $15, 7($5)  # tile to East's North triangle
  bne $14, $15, NoFit  # current tile's East not in nbr, NoFit
EastOK:
  addi $14, $2, 1  # row number + 1
  beq $14, $7, SouthOK  # if bottom row, don't check South
  lb $14, 1($5)  # current tile's South triangle
  add $3, $5, $6  # addr of tile to South
  lb $15, 0($3)  # tile to South's East triangle
  beq $14, $15, SouthOK  # current tile South=member of S nbr
  lb $15, 1($3)  # tile to South's South triangle
  beq $14, $15, SouthOK  # current tile South=member of S nbr
  lb $15, 2($3)  # tile to South's West triangle
  beq $14, $15, SouthOK  # current tile South=member of S nbr
  lb $15, 3($3)  # tile to South's North triangle
  bne $14, $15, NoFit  # current tile's S not in nbr, NoFit
SouthOK:
  sw $10, Solution($1)  # Solution[I] = rotation amt
  j Return  # return w/ $10 = current rotation amt
NoFit:
  slti $3, $10, 4  # is rot amt < 4?
  beq $3, $0, Return  # if not, Return
  addi $10, $10, 1  # inc rotation amt
  # Rotate Tile
  lw $19, 0($5)  # load in TileGrid[I]
  sll $19, $19, 8  # shift it left by 8
  lb $20, 3($5)  # extract byte 3
  or $19, $19, $20  # move byte 3 to byte 0
  sw $19, 0($5)  # store result back to TileGrid[I]
  j RotLoop  # continue looping
Return:
  jr $31
Backtrack:
  slt $3, $1, $0  # is I < 0?
  bne $3, $0, ExitBackTrk  # if so, stop backtracking
  sw $0, Solution($1)  # Solution[I] = 0 (reset)
  addi $1, $1, -4  # go back to previous tile
  lw $10, Solution($1)  # load in starting rot amt Solution[I-1]
  addi $10, $10, 1  # inc rotation amt
  # Rotate Tile
  addi $5, $1, TileGrid  # address of current tile
  lw $19, 0($5)  # load in TileGrid[I]
  sll $19, $19, 8  # shift it left by 8
  lb $20, 3($5)  # extract byte 3
  or $19, $19, $20  # move byte 3 to byte 0
  sw $19, 0($5)  # store result back to TileGrid[I]
  sw $10, Solution($1)  # store rotation amt in Solution
  addi $29, $29, -4  # push return address
  sw $31, 0($29)  # on stack
  jal RotateUntilFit  # return R in $10
  lw $31, 0($29)  # pop return address
  addi $29, $29, 4  # from stack
  # if rotated successfully (i.e., rotation amt (R) < 4)
  # then set Solution[I] = R and return
  # else keep backtracking until find fit at new index I
  slti $3, $10, 4  # is R < 4?
  beq $3, $0, Backtrack  # if not (there is no rotation for which
  # TG[I] will match its nbrs, backtrack
  sw $10, Solution($1)  # else (R<4): set Solution[I] = R, ret
ExitBackTrk:
  jr $31
Part A: Typical practice is to save return address on the activation frame on the stack. Why does function RotateUntilFit **not** need to save the return address on the stack? Why does function Backtrack need to save and restore its return address using the stack? 4 pts

*This is because RotateUntilFit does not call any other functions, so $31$ will not be modified during its execution, and consequently, there is no need to save and restore $31$ inside RotateUntilFit.*

Part B: When tracing back, the baseline code always traces back by 1 tile, which makes it very efficient in determining which tile to track back. If you are to explore the trade-off between (1) trace back to a more effective tile so that the algorithm can check less tiles, and (2) spend less instructions in determining which tile to track back; what other options would you like to take? 4 pts

*One option would be this: when a track-back is decided, the algorithm will first go backward in the same row, but when the algorithm reaches the left-most tile on that row, it does not track back to the ‘previous’ tile (which would be the right-most tile on the previous row), instead, it tracks back to the tile that is right above. This will allow the algorithm to skip an entire row (well, one less), and goes back to work on a tile that needs to be re-examined.*