```
struct T {
      int  x[4];
      int* y;
    };
    struct S* s;
```

Write the SM213 assembly code that is equivalent to the following C statement

```
s->c.x[1] = 0;.
```

Exercise 4

What happens when the following code is compiled (and if it compiles) runs?

```
void gp (void* inv, void** outv) {
  intptr_t  in  = (intptr_t) inv;
  intptr_t* out = (intptr_t*) outv;
  *out = in * *out;
}

void fp (void* inv, void** outv) {
  intptr_t  in  = (intptr_t) inv;
  intptr_t* out = (intptr_t*) outv;
  *out = in + *out;
}

void foo (void** in, void**out, int n, void (*fp) (void*, void**)) {
  for (int i=0; i<n; i++) {
    fp (in[i], out);
  }
}

int main(int argc, char** argv) {
  intptr_t a[] = {2,3,4,5};
  intptr_t v   = 1;
  foo ((void**) a, (void**) &v, 4, gp);
  printf ("%ld\n", v);
}
```

Exercise 5

Consider the following code. Will it introduce a memory leak or a dangling pointer?

```
char* copy (char* from, int n) {
  char* to = malloc (n);
  for (int i=0; i<n; i++)
    to[i] = from[i];
  return to;
}
```

10 **(6 marks)** **Using Function Pointers.** Write a C procedure named `apply` that returns either the minimum or maximum value of a list of non-negative integers, as determined by one of its parameters, a function pointer. If the list is empty it should return -1. Assume the existence of procedures named `min(a,b)` and `max(a,b)` that compare two integers and returns the min or max integer. No other procedures are allowed and `apply` is not permitted to use an `if` statement. For example, the following statement should compute the maximum value in list `a` of length `n`.

```
int m = apply (max, a, n);
```

Write the procedure `apply()`:

11 **(6 marks)** **IO Devices.** For each of the following, (a) explain what it is and (b) state whether the CPU or IO controller determines when it occurs (i.e., initiates it).

11a Programmed IO (PIO):

11b DMA:

11c Interrupts:

3 **(8 marks)** **Dynamic Control Flow.** Give SM213 assembly code for the following C statements. Assume that `i` is a global variable of type `int`.

3a Using a jump table, the statement:

```
switch (i) {
    case 4:
        i = 0;
        break;
    case 6:
        i = 1;
        break;
    default:
        i = 2;
        break;
}
```

3b Where the global variable `int (*bar)(void)` was previously declared, the statement:

```
bar();
```

9 **(6 marks)** **Switch Statements.** There are two ways to implement `switch` statements in machine code. For purposes of this question, lets call them A and B.

9a Describe A, very briefly.

9b Describe B, very briefly.

9c State precisely one situation where A would be preferred over B and why.

9d State precisely one situation where B would be preferred over A and why.

10 **(9 marks)** **IO Devices.** Three key hardware features used to incorporate IO Devices with the CPU and memory are Programmed IO (PIO), Direct Memory Access (DMA) and interrupts.

10a Carefully explain the difference between PIO and DMA; give one advantage of DMA.

10b Demonstrate why interrupts are needed by carefully explaining what programs would have to do differently to perform IO if interrupts didn't exist and what disadvantages this approach would have.

10c Explain how interrupts would be added to the Simple Machine simulator by indicating where the interrupt-handling logic would be added and saying roughly what it would do.

8 (12 marks) Static and Dynamic Procedure Calls.

8a Procedure calls in C are normally static. Method invocations in Java are normally dynamic. Carefully explain the reason why Java uses dynamic method invocation and what benefit this provides to Java programs.

8b Carefully explain an important disadvantage of dynamic invocation in Java or other languages.

8c Demonstrate the use of function pointers in C by writing a procedure called `compute` that:

1. has three arguments: a non-empty array of integers, the size of the array, and a function pointer;
2. computes either the array min or max depending only on the value of the function pointer argument;
3. contains a `for` loop, no `if` statements, and one procedure call (per loop).

Give the C code for `compute`, the two procedures that it uses (i.e., that are passed to it as the value of the function-pointer argument), and two calls to `compute`, one that computes min and the other that computes max (be sure to indicate which is which).

Exercise 4

It prints 120.

Exercise 5

Yes, a memory leak is possible. It can be fixed as follows:

```
char* copy (char* from, int n) {
  char* to = malloc (n);
  for (int i=0; i<n; i++)
    to[i] = from[i];
  return to;
}
void foo (char* x, int n) {
  char* y = copy (x, n);
  printf ("%s", y);
  free (y);
}
```

Exercise 6

```
void getsum (char* buf, int n) {
  int s=0;
  for (int i=0; i<256; i++)
    s += buf[i];
  printf ("%d\n", s);
}

char buf[256];

void ps() {
  int s = async_read (1234, buf, 256, getsum);
}
```

```
int  c = 0;
int* v[2];

void bar (int* ap) {
  if (v[c] == NULL || *ap < *v[c]) {
    if (v[c] != NULL)
      dec(v[c]);
    v[c] = ap;
    inc(v[c]);
  }
  c = (c + 1) % 2;
}

void zot (int a) {
  if (v[c] != NULL && a < *v[c])
    *v[c] = a;
  c = (c + 1) % 2;
}
```

10 **(6 marks)** **Using Function Pointers.** Write a C procedure named `apply` that returns either the minimum or maximum value of a list of non-negative integers, as determined by one of its parameters, a function pointer. If the list is empty it should return -1. Assume the existence of procedures named `min(a,b)` and `max(a,b)` that compare two integers and returns the min or max integer. No other procedures are allowed and `apply` is not permitted to use an `if` statement. For example, the following statement should compute the maximum value in list `a` of length `n`.

```
int m = apply (max, a, n);
```

Write the procedure `apply()`:

```
int apply (int (*f)(int, int), int* a, int n) {
    if (n<=0)
        return -1;   // for 2016W2 this isn't necessary.
    else {
        int v = a[0];
        for (int i=1; i<n; i++)
            v = f(v, a[i]);
        return v;
    }
}
```

11 **(6 marks)** **IO Devices.** For each of the following, (a) explain what it is and (b) state whether the CPU or IO controller determines when it occurs (i.e., initiates it).

11a Programmed IO (PIO):

11b DMA:

11c Interrupts:

12 **(8 marks)** **Threads and Scheduling.** Answer the following questions about threads.

12a Explain briefly (without giving any code) how threads can be used to simplify code that performs asynchronous operations such as communicating with an IO controller.

12b Explain briefly the difference between `uthread_yield` and `uthread_block`.

12c Is it possible for a thread to unblock itself? Explain your answer.

12d Consider a system in which there is at least one thread on the *ready queue* when a thread unblocks. What happens to the unblocked thread? Explain.

13 **(9 marks)** **Synchronization** Consider the following C code that uses mutexes and condition variables. The code is considered to be is correct if both procedures complete successfully when they are called from concurrent threads

CPSC 213, Winter 2015, Term 2 — Extra Questions **Solution**

Date: March 3, 2015; Instructor: Mike Feeley

1 **(8 marks)** **Loops and If.** The following assembly code computes `s = a[0]` where `a` is a global, static array of integers. Modify this code so that it computes the sum of all positive elements of the array where the size of the array is stored in a global int named `n`. Your solution should avoid unnecessary memory accesses where possible (e.g., inside of the loop). You may modify the code in place. Comment every line you add. Hint: notice that you have to add four things: (1) read the value of `n`, (2) turn part of this code into a loop, (3) exit the loop at the right time, and (4) only sum positive numbers; you might want to take these one at a time.

2 **(6 marks)** **Static Control Flow.** Give SM213 assembly code for the following C statements. Assume that `i` is a global variable of type `int`.

2a

```
if (i==0)
    i = 1;
else
    i = 2;
```

```
    ld $i, r0       # r0 = &i
    ld (r0), r1     # r1 = i
    beq r1, L0      # goto L0 if i==0
    ld $2, r2       # t_i = 2 if i !=0
    br L1           # goto L1
L0: ld $1, r2       # t_i = 1 if i==0
L1: st r2, (r0)     # i = t_i
```

2b

```
while (i!=0)
    i -= 1;
```

```
    ld $i, r0       # r0 = &i
    ld (r0), r1     # t_i = i
L0: beq r1, L2      # goto L1 if t_i == 0
    dec r1          # t_i--
    br L0           # goto L0
L1: st r1, (r0)     # i = t_i
```

3 **(8 marks)** **Dynamic Control Flow.** Give SM213 assembly code for the following C statements. Assume that `i` is a global variable of type `int`.

3a Using a jump table, the statement:

```
switch (i) {
    case 4:
        i = 0;
        break;
    case 6:
        i = 1;
        break;
    default:
        i = 2;
        break;
}
```

```
         ld $i, r0         # r0 = &i
         ld (r0), r1       # r1 = i
         ld $-4, r2        # r2 = -4
         add r2, r1        # r1 = i-4
         bgt r1, L0        # goto L0 if i > 4
         beq r1, L0        # goto L0 if i == 4
         br DEFAULT        # goto L0 if i < 4
L0:      ld $-2, r2        # r2 = -2
         add r1, r2        # r2 = i-6
         bgt r2, DEFAULT   # goto DEFAULT if i > 6
         ld $JT, r2        # r2 = JT
         j *(r2, r1, 4)    # goto jt[i-4]
CASE_4:  ld $0, r2         # t_i = 0
         br L1             # goto L1
CASE_6:  ld $1, r2         # t_i = 1
         br L1             # goto L1
DEFAULT: ld $2, r2         # t_i = 2
L1:      st r2, (r0)       # i = t_i

# The Jump Table
JT:      .long CASE_4
         .long DEFAULT
         .long CASE_6
```

3b Where the global variable `int (*bar)(void)` was previously declared, the statement:

```
bar();
```

```
ld $bar, r0   # r0 = &bar
gpc $2, r6    # r6 = return address
j *(r0)       # bar()
```

4 **(8 marks)** **Procedure Calls.** Give SM213 assembly for these statements. Assume the `i` is a global variable of type `int`, that `r5` stores the value of the stack pointer, and that arguments are passed on the stack.

4a

```
int foo (int i, int j) {
    return j;
}
```

```
ld 4(r6), r0   # r0 = j
j (r6)         # return j
```

4b

```
i = foo (1, 2);
```

```
deca r5        # make stack space for arg0
deca r5        # make stack space for arg1
ld $1, r0      # r0 = 1
st r0, 0(r5)   # arg0 = 1
ld $2, r0      # r0 = 2
st r0, 4(r5)   # arg1 = 2
gpc $6, r6     # r6 = return address
j foo          # t_i = foo (1,2)
inca r5        # free stack space for arg1
inca r5        # free stack space for arg0
ld $i, r1      # r1 = &i
st r0, (r1)    # r1 = t_i
```

5 **(12 marks)** Consider the following SM213 assembly code that implements a simple C procedure.

8 (12 marks) Static and Dynamic Procedure Calls.

8a Procedure calls in C are normally static. Method invocations in Java are normally dynamic. Carefully explain the reason why Java uses dynamic method invocation and what benefit this provides to Java programs.

Java's method invocations are dynamic because they read a jump table at runtime to determine which method to call. Which method is called depends on the actual type of the object, since each class has its own jump table.

Dynamic method invocation allows for polymorphic dispatch to occur. Polymorphism is a powerful tool makes it easy to add functionality to existing code by simply extending a class and overriding methods.

8b Carefully explain an important disadvantage of dynamic invocation in Java or other languages.

Every method call has the additional overhead of performing a memory read to determine which method to call. This can affect performance, especially for very short methods where a memory read might consist of a significant amount of the method's execution time.

8c Demonstrate the use of function pointers in C by writing a procedure called `compute` that:

1. has three arguments: a non-empty array of integers, the size of the array, and a function pointer;
2. computes either the array min or max depending only on the value of the function pointer argument;
3. contains a `for` loop, no `if` statements, and one procedure call (per loop).

Give the C code for `compute`, the two procedures that it uses (i.e., that are passed to it as the value of the function-pointer argument), and two calls to `compute`, one that computes min and the other that computes max (be sure to indicate which is which).

```
int compute(int* array, int size, int (*fn)(int, int)) {
    int acc = array[0];
    for (int i = 1; i < size; i++) {
        acc = fn(acc, array[i]);
    }
    return acc;
}

int max(int a, int b) {
  return a > b ? a : b;
}

int min(int a , int b) {
  return a < b ? a : b;
}

int arr[5] = {7, -4, 1, 9, 3};

compute(arr, 5, max); // returns max of array
compute(arr, 5, min); // returns min of array
```

The first thing you should do when asked this type of question is to figure out what your function signature should look like - how many arguments does it have, what types are they, and what type is the return value? Also give your parameters meaningful names.

Make sure you think about **edge cases**. If we weren't told that the array is non-empty, we wouldn't be able to do `int acc = array[0];` without potentially segfaulting or causing undefined behavior.

Search for 'ternary operator' if you're not sure about the syntax of the expressions in `max` and `min`. Knowing shorthand coding tricks like this will save you time when writing code in an exam.

9 **(6 marks)** **Switch Statements.** There are two ways to implement `switch` statements in machine code. For purposes of this question, lets call them A and B.

9a Describe A, very briefly.

A sequence of `if` statements.

9b Describe B, very briefly.

A jump table of labels corresponding to switch-cases.

9c State precisely one situation where A would be preferred over B and why.

If there are very few cases to consider, then the overhead of using a jumptable is higher than a few statements. Reading memory is much slower than executing a conditional branch.

OR: If the case values are spread apart (sparsely populated), there will be a lot of wasted memory in the jumptable because we have to represent a contiguous range of values in a jump table. e.g.

```
switch (i) {
  case 1:
    j = 5;
    break;
  case 1000000:
    j = 10;
    break;
 }
```

would require a jump table with one million elements!

9d State precisely one situation where B would be preferred over A and why.

When you have lots of cases to check and their values are close together (densely populated), the jump table is the best choice. When there are N cases, it takes `O(N)` to test all cases, whereas it will always take `O(1)` with a jump table. If the values are closer together, then we waste less memory creating the jump table.

10 **(9 marks)** **IO Devices.** Three key hardware features used to incorporate IO Devices with the CPU and memory are Programmed IO (PIO), Direct Memory Access (DMA) and interrupts.

10a Carefully explain the difference between PIO and DMA; give one advantage of DMA.

The CPU uses PIO to read or write to an IO device one word at a time. IO Devices use DMA to read or write memory directly, without involving the CPU. An advantage of PIO is that the CPU can use it to transfer data to an IO device, or control it; e.g., to signal the IO device that the CPU wants something, Another advantage is that PIO has lower overhead and lower latency for very small transfers because it avoids the overhead of setting up a DMA. The advantage of DMA is that the transfer occurs asynchronously to the CPU and so the CPU is free to do other things during the transfer. For any transfer larger than 64-128 bytes, DMA typically transfers with lower overhead and latency than PIO.

10b Demonstrate why interrupts are needed by carefully explaining what programs would have to do differently to perform IO if interrupts didn't exist and what disadvantages this approach would have.

If interrupts didn't exist, a program would have to repeatedly *poll* the IO device to determine whether the device had information (e.g., keyboard presses) for it. The disadvantages of polling are that it wastes CPU cycles unnecessarily when the IO device doesn't have information for the CPU. If polling is very frequent, then this overhead is very high.

Infrequent polling may not be a suitable solution either because it increases the latency (i.e., delay) between when an IO device notifies the CPU that it wants its attention, and when the CPU actually notices it. This would increase, for example, the latency of disk and network reads. There is an undesirable tradeoff between latency and CPU 'wastage'.

10c Explain how interrupts would be added to the Simple Machine simulator by indicating where the interrupt-handling logic would be added and saying roughly what it would do.