CS107
Spring 2008

Handout 21S
April 30, 2008

CS107 Practice Midterm Solution

Problem 1: San Francisco Fine Dining

Consider the following **`struct`** definitions:

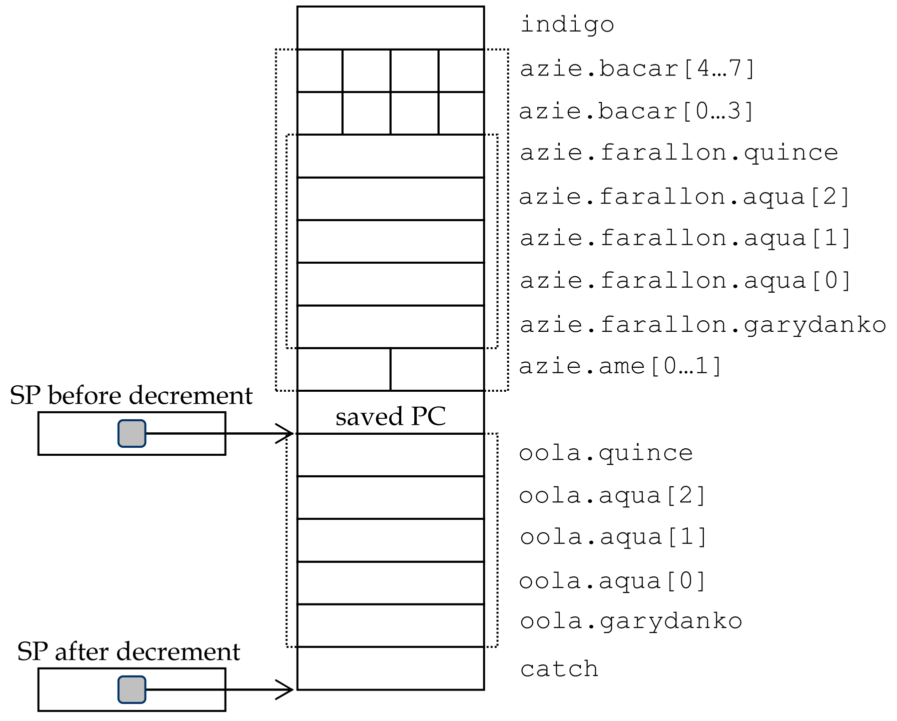
```
typedef struct {           typedef struct {
   int **garydanko;           short ame[2];
   int aqua[3];               appetizer farallon;
   char *quince;              char bacar[8];
} appetizer;               } dessert;

dessert *dinnerisserved(short *boulevard, appetizer *jardiniere);
int *bonappetit(dessert azie, char **indigo)
{
   appetizer oola;
   dessert *catch;

line 1     azie.bacar[azie.ame[2]] += catch->farallon.aqua[4];
line 2     ((appetizer *)(((dessert *)(&oola.quince))->farallon.garydanko))->quince = 0;
line 3     return (*dinnerisserved((short *) &indigo, &oola)).farallon.aqua;
}
```

Generate code for the entire **`bonappetit`** function. Be clear about what assembly code corresponds to what line. You have this and the next page for your code.

The full activation record layout is large as far as CS107 activation records go. The picture is here, and the code is on the next page.



prior to line 1

```
SP = SP - 24;           // make space for local variables
```

line 1

```
R1 = M[SP];             // load catch
R2 = M[R1 + 24];        // load four-byte figure overlaying bacar[0…3]
R3 =.2 M[SP + 32];      // load azie.ame[2]
R4 = SP + 52;           // load &azie.bacar[0]
R5 = R4 + R3;           // synthesize &azie.bacar[azie.ame[2]]
R6 =.1 M[R5];           // load old value
R7 = R6 + R2;           // compute result of self-addition
M[R5] =.1 R7;           // flush new value back to
```

line 2

```
R1 = M[SP + 24];        // load the garydanko field from the dessert at SP + 20
M[R1 + 16] = 0;         // ooo, there's an appetizer there! zero out the quince field
```

line 3

```
R1 = SP + 60;           // compute &indigo
R2 = SP + 4;            // compute &oola
SP = SP - 8;            // make space for dinnerisserver parameters
M[SP] = R1;             // initialize boulevard
M[SP + 4] = R2;         // initialize jardinière
CALL <dinnerisserved>
SP = SP + 8;            // clean up params
RV = RV + 8;            // RV was populated with a dessert *,
                        // update with RV->farallon.aqua
```

after line 3

```
SP = SP + 24;           // clean up locals
RET;
```

Problem 2: Breaking Larger Data Images Into Packets

```
void *packetize(const void *image, int size, int packetSize)
{
   void *list;
   void **currp = &list;
   int numBytesProcessed = 0;

   while (numBytesProcessed < size) {
      if (size - numBytesProcessed < packetSize)
         packetSize = size - numBytesProcessed; // can only happen on last iter
      void *newNode = malloc(packetSize + sizeof(void *));
      memcpy(newNode, (char *) image + numBytesProcessed, packetSize);
      numBytesProcessed += packetSize;
      *currp = newNode;
      currp = (void **)((char *) newNode + packetSize);
   }

   *currp = NULL;
   return list;
}
```

Problem 3: The C `multiset`

```
a. void MultiSetNew(multiset *ms, int elemSize, int numBuckets,
                    MultiSetHashFunction hash, MultiSetCompareFunction compare,
                    MultiSetFreeFunction free)
   {
      HashSetNew(&ms->elements, elemSize + sizeof(int), numBuckets,
                 hash, compare, free);
      ms->elemSize = elemSize;
      ms->free = free;
   }

   /**
    * Function: MultiSetDispose
    * ------------------------
    * Disposes of all previously stored client elements by calling
    * HashSetDispose.
    */

   void MultiSetDispose(multiset *ms)
   {
      HashSetDispose(&ms->elements);
   }


b. void MultiSetEnter(multiset *ms, const void *elem)
   {
      void *found = HashSetLookup(&ms->mappings, elem);
      if (found == NULL) {
         char pair[ms->elemSize + sizeof(int)];
         memcpy(pair, elem, ms->elemSize);
         *(int *)(pair + ms->elemSize) = 1;
         HashSetEnter(&ms->mappings, pair);
      } else {
         if (ms->free != NULL) ms->free(found);
         memcpy(found, elem, ms->elemSize);
         (*(int *)((char *) found + ms->elemSize))++;
      }
   }


c. typedef struct {
      MultiSetMapFunction originalMap;
      void *originalAuxData;
      int elemSize;
   } mapHelper;

   static void ApplyMultiMapFunctionWithAuxData(void *elem, void *auxData)
   {
      mapHelper *helper = auxData;
      int multiplicity = *(int *)((char *) elem + helper->elemSize);
      helper->originalMap(elem, multiplicity, helper->originalAuxData);
   }

   void MultiSetMap(multiset *ms, MultiSetMapFunction map, void *auxData)
   {
      mapHelper helper = { map, auxData, ms->elemSize };
      HashSetMap(&ms->elements, ApplyMultiMapFunctionWithAuxData, &helper);
   }
```

Problem 4: The Queen Of Parking Infractions (5 points)

```
typedef struct {
   const char *licensePlate;
   int numTickets;
} maxTicketsStruct;

static void IdentifyContender(void *elem, int count, void *auxData)
{
   maxTicketsStruct *ticketData = auxData;
   if (count > ticketData->numTickets) {
      ticketData->numTickets = count;
      ticketData->licensePlate = elem; // cast not needed, but fine to have it
   }
}

void FindQueenOfParkingInfractions(multiset *ms, char licensePlateOfQueen[])
{
   maxTicketsStruct ticketData = { NULL, 0 };
   MultiSetMap(ms, IdentifyContender, &ticketData);
   strcpy(licensePlateOfQueen, ticketData.licensePlate);
}
```

Problem 5: Short Answers

For each of the following questions, we expect short, insightful answers, writing code or functions only when necessary. Clarity and accuracy of explanations are key.

a. Assuming all instructions and pointers are 4 bytes, explain why instructions of the form **`M[x] = M[y] + M[z]`**, where **`x`**, **`y`**, and **`z`** are legitimate but otherwise arbitrary memory addresses, aren't included in the instruction set.

   If **`x`**, **`y`**, and **`z`** are legitimate but otherwise arbitrary memory address, then the instruction would need 32 bits to encode any one of them. Since the entire instruction is confined by a 32-bit limit, there's no way to pack information about three address and the opcode for addition into the instruction.

b. Our activation record model wedges the return address information below the function arguments and above the local parameters. Why not pack all variables together, and place this return address at the top or the bottom of the activation record?

   The return address can't be placed on the bottom because the calling function doesn't know where the bottom of the full layout is. The caller knows nothing of what locals are allocated by the callee's instruction list.

   The return address could, in most cases, be placed at the top since the callee function generally knows what all of the parameters are and how large everything is. The key exception—the reason that this as a convention could not be adopted—is that some functions, such as **`printf`** and **`scanf`**, take a variable number of arguments. In those situations, no information about the number and size of the parameters is around, so it can't reliably understand where the stored **`PC`** would be.

c. Write a short function called **`IsLittleEndian`**, which returns true if and only if the computer architecture is little endian—that is, if the least significant byte of a multi-byte figure is stored at the lowest address instead of the highest one.

Cute little function (there are many other answers):

```
static bool IsLittleEndian()
{
   int one = 1;
   return ((*(char *)&one) == 1);
}
```