Lab 12

Published

Part A

[Step 1]

1. Download **func.o** and create an assembly file called func.S. You will compile these two files together using the following command: aarch64-linux-gnu-gcc -o func func.o func.S

2. You will be writing all of your functions for this lab in func.S

[Step 2]

1. Write the assembly function CopyIntArray: void CopyIntArray(char *dst, const char *src, int num_elements);

2. This function will copy all the integers from the array src into the array dst. Your function should copy the data one integer at a time. The total number of integers to copy is given my num_elements.

RESTRICTIONS

a) You may not use any external functions.

b) You must copy 4 bytes at a time for each integer.

c) You may not use any globals

[Step 3]

1. Write the assembly function Modulo: unsigned int Modulo(unsigned int dividend, unsigned int divisor);

2. This function will mimic: return dividend % divisor

3. However, there is no modulo instruction for ARMv8, so you will have to do it yourself.

4. You will need to return the remainder after division. Do not concern yourself with the "unsigned". It is there to simplify your modulo function.

RESTRICTIONS

a) You may not call any external functions

b) You may not use any globals

[Step 4]

1. Write the assembly function GetRandom: int GetRandom(int min, int max);

2. This function will return a random number between [min, max]

3. This function will need to perform the following formula: return (rand() % (max-min+1)) + min

4. Notice that there are two functions that you'll need to call here: YOUR Modulo function and the rand() function. Remember that rand() takes no parameters but returns a large integer.

5. When you're finished, return a random number between [min, max]

RESTRICTIONS

a) You must use your Modulo function

b) You must use the rand function

c) You may not use any other external function

d) You may not use any globals

[Step 5]

1. Write the assembly function MatrixMultiply: void MatrixMultiply(int dst[3], int matrix[9], int vec[3]);

2. This function will multiply matrix by vector and store the result in dst.

3. All operands to this function are integer arrays.

RESTRICTIONS

a) You may not use any external functions

b) You may not use any globals

c) You must use a loop to multiply the matrix[] with the vec[].

[Step 6]

1. Write the assembly function CopyData: void CopyData(Data *dst, const Data *src, int *num_elements, int type);

2. This function requires the use of the Data structure which looks like:

```
struct Data {
      char *data;
      int size;
```

```
        short type;
};
```

3. Notice that the Data structure has a "type" field. You will copy those elements from src into dst if and only if the type parameter matches the type in the Data structure.

4. num_elements first provides a pointer to the size of the src array. Since you may or may not copy all of the src into dst, you need to update the value *num_elements with the actual number of elements you copied to dst.

5. You will see that inside of the Data structure, there is a data pointer and a size. You may assume that the destination has provided enough room for the size in the memory pointed by the pointer.

6. You must copy the data from the src->data to the dst->data byte-by-byte. You may write another function for this step, if you wish.

RESTRICTIONS

a) You may not use any external functions
b) You may not use any global variables
c) Use good structure management, including using .equ directives

[Step 7: Testing]

1. Test your code by typing: aarch64-linux-gnu-gcc -o func func.o func.S

2. Then type: ./func 1

3. This will execute all of your functions and show you what passed and what failed.

4. Run this testing code more than once! It generates random values, so in order to test a wide ranging set of parameters, run this testing program at least 10 times.

Part B

[Files]

1. Download the testing code **mmu.o** and create an assembly file called mmu.S.

2. All of your code will go into mmu.S

3. You will see that there are 7 functions you will need to write in mmu.S: InitMMU, ReleaseMMU, FindEntry, Map, Unmap, Read, and Write.

4. To keep things simplified, all of your data (address and values) will be 8-bytes. You can verify this by looking at how these functions are prototyped below.

[MMU]

1. You will be designing the functions of a simplified MMU. Typically, an MMU would have multiple levels of pages (4 to 5 for 64-bit systems!), but you'll be designing a single level of page tables that hold a virtual address and a physical address.

2. You will need to create two global, uninitialized variables: p_tab and p_ent. p_tab stores a memory address where the table starts, and p_ent stores the total number of entries. Both of these will be 64-bit integers.

3. All of your functions must be .global. For example, InitMMU needs to be specified as .global InitMMU, otherwise the mmu.o file will not be able to find your functions!

4. Two functions for managing memory on the heap have been written for you: long *New(int bytes) and void Delete(long *ptr). You pass New the number of bytes you want on the heap and it returns you a memory address to those bytes. Delete takes a pointer that you created with New and deletes that memory off of the heap (the OS reclaims that memory).

[InitMMU function]

```
long *InitMMU(int pages);
```

1. InitMMU will take the number of entries that your page table can store. You will need to make sure that the number of entries is "aligned" by 16, meaning that it is a multiple of 16. For example, if I pass 17 to your function, you need to round it up to 32.

2. After you find the number of entries, you will need to create space for it. Each entry takes 16 bytes (8 for the virtual address and 8 for the physical address). Pass the number of BYTES when you call the New() function. The New() function will return a memory address to the top of your table.

3. Your function will then return the address received from New(). This is how mmu.o navigates your page table.

[ReleaseMMU function]

```
void ReleaseMMU();
```

1. ReleaseMMU will free the memory allocated by InitMMU. Make sure that your p_tab pointer has been properly created by returning from this function if p_tab is 0.

2. If you find out that p_tab is a valid pointer, pass it and call the Delete function.

[FindEntry function]

```
long *FindEntry(long virtual_address);
```

1. Write the FindEntry assembly function. This function takes a virtual address as its parameter. The purpose of this function is to return the memory address where the virtual address has been found. If nothing was found, return 0 (xzr).

2. Remember that your MMU only stores the first 56-bits of the virtual and physical addresses, so 0xfeedbeef and 0xfeedbe12 both map to the same virtual address since they'll look like 0xfeedbe00 in your tables.

3. All of your virtual to physical translations are stored in p_tab. So, iterate through p_tab until you find the virtual address (remember only the first 56 bits). When you do, return the memory location where you found the virtual address.

4. If you could not find the virtual address, return 0 (xzr).

[Map function]

```
void Map(long virtual_address, long physical_address);
```

1. Write the Map assembly function. This function takes a virtual address and a physical address.

2. It first searches p_tab to see if the virtual address has already been mapped. If yes, simply overwrite the physical address with the new physical address (the first 56-bits of it, anyway). If no, find a blank entry (where the virtual address is 0), and store both the virtual address and physical address in the table.use FindEntry to see if the virtual address has been mapped already.

3. We talk about storing the first 56-bits only. This means that if I call Map(0xdeadbeef, 0xfeedbeef), then it will store 0xdeadbe00 as the virtual address and 0xfeedbe00 as the physical address. You are still storing 64-bits, but the last 8-bits are 0s.

4. Use FindEntry to see if the virtual address has been mapped already.

5. If no blank entries are available, your Map function will do nothing.

6. You will notice that the checks you make allow for multiple virtual addresses map to the same physical address, this is perfectly OK. This is how actual MMUs work too.

[Unmap function]

```
void Unmap(long virtual_address);
```

1. Write the Unmap assembly function. This function takes a virtual address, which may or may not have the last 8-bits as 0, but remember that you store only the first 56-bits of the address. So, keep this in mind when searching through your table for the address.

2. You will find the entry containing the given virtual address. When you find that entry, set the virtual address to 0. This means that the entry is invalid and may be used for a subsequent call to Map.

3. If you did NOT find any entry matching the virtual address, just return from Unmap.

[Write function]

```
void Write(long virtual_address, long value);
```

1. Write the Write assembly function. This function takes a virtual address and a value.

2. You will need to find the physical address that the virtual address maps to (i.e. FindEntry)

3. If you did not find an entry, do nothing.

4. If you found the entry, remember the physical address is only the first 56-bits. So, take the last 8-bits of the passed virtual address and OR it with the first 56-bits of the physical address. This will give you your full address.

5. Write the passed value to the full address.

[Read function]

```
long Read(long virtual_address);
```

1. Write the Read assembly function. This function takes a virtual address and returns the value at the physical address.

2. You will need to find the physical address that the virtual address maps to (i.e. FindEntry).

3. If you did not find an entry, return 111.

4. Remember, your p_tab only stores the first 56-bits of the physical address.The last 8-bits comes from the last 8-bits of the virtual address passed to this function.

5. Read the value at the full address and return it.

[Restrictions]

1. Remember you'll be using ONLY two global variables (global, uninitialized variables), so store them in the appropriate section.

2. Everything will be using 64-bit values, including the values passed to / from your Read/Write functions.

3. Any local variables that you need must be located on the stack.

[Testing and Hints]

1. It might be helpful to first write the functions in C++. The functions themselves are not very involved, but the translation to assembly might be difficult for you. However, we will not grade any C++ version of the functions above, so you are not required to write them first in C++.

2. mmu.o will run a simple testing program to see if your functions are working at first glance. Compile this with your assembly file by typing: `aarch64-linux-gnu-gcc -o mmu mmu.o mmu.S`

3. The testing program does NOT test all possible error conditions, so you must use good debugging skills to make sure that you're assembly program is correct.

4. Since you will be calling other functions in your code, make sure you are up-to-speed on which registers you must save and which are free to be destroyed. For example, if you set x0 to 22, and then you call FindEntry from your Map function, don't expect x0 to still be 22!

5. It is very tempting to want to think of pointer arithmetic in assembly, however this is a C++ thing only. For example long *ptr = 0; ptr ++; Now, ptr is 8 because it moved it one sizeof(long). However, in assembly, this doesn't exist. For example: adr x0, my_ptr, add x0, x0, 1. This will add only 1 byte to the address of my_ptr, not 8. Data sizes are controlled by you--either by the instruction you choose or the amount you add to an address.

[You are finished with this lab!]

Submit your completed func.S as func.txt and completed mmu.S as mmu.txt.

Points	150
Submitting	a file upload
File Types	txt

Due	For	Available from	Until
Nov 17, 2017 at 7am	Everyone	Nov 10, 2017 at 7:58am	Nov 17, 2017 at 7am

Lab 12

Criteria	Ratings					Pts
PartA: CopyData function -Uses good structure management (.equ) -Copies src[].data into dst[].data byte-by-byte (do not copy the pointer!) -Updates dst[].type and dst[].size properly	**25.0 pts Full Marks**	**20.0 pts Minor Problems**	**15.0 pts Some Problems**	**5.0 pts Major Problems**	**0.0 pts No Marks**	25.0 pts
PartA: CopyIntArray function -Copies 4 bytes at a time	**10.0 pts Full Marks**	**7.0 pts Minor Problems**	**5.0 pts Some Problems**	**3.0 pts Major Problems**	**0.0 pts No Marks**	10.0 pts
PartA: Modulo function -Uses integer instructions to determine %	**15.0 pts Full Marks**	**12.0 pts Minor Problems**	**8.0 pts Some Problems**	**4.0 pts Major Problems**	**0.0 pts No Marks**	15.0 pts
PartA: GetRandom function -Properly saves caller saved registers -Properly preserves callee saved registers	**15.0 pts Full Marks**	**12.0 pts Minor Problems**	**8.0 pts Some Problems**	**4.0 pts Major Problems**	**0.0 pts No Marks**	15.0 pts
PartA: MatrixMultiply function -Student used a loop to multiply the matrix with the vector.	**10.0 pts Full Marks**	**5.0 pts Some Problems**	**0.0 pts No Marks**			10.0 pts
PartB: InitMMU -Rounds UP entries to a multiple of 16 -Stores the table pointer and # of entries in global, uninitialized integers. -Calls New() with the appropriate # of bytes. -Stores pt_tab and pt_ent, which are in the appropriate section.	**15.0 pts Full Marks**	**8.0 pts Some Problems**	**0.0 pts No Marks**			15.0 pts
PartB: ReleaseMMU -Calls Delete() with the proper pointer.	**5.0 pts Full Marks**	**0.0 pts No Marks**				5.0 pts
PartB: FindEntry -Returns NULL if no entry found -Returns the proper entry if found -Uses a mask properly	**15.0 pts Full Marks**	**8.0 pts Some Problems**	**0.0 pts No Marks**			15.0 pts
PartB: Map -Properly strips 56-bits from the virtual and physical addresses -Uses FindEntry to avoid duplicating entries	**20.0 pts Full Marks**	**16.0 pts Minor Problems**	**11.0 pts Some Problems**	**6.0 pts Major Problems**	**0.0 pts No Marks**	20.0 pts

Criteria	Ratings					Pts
PartB: Unmap -Properly calls FindEntry to find the entry to unmap -Unmaps by zero'ing the virtual address field	**10.0 pts** **Full Marks**	**0.0 pts** **No Marks**				10.0 pts
PartB: Read -Properly translates virtual address into a physical address -Appends the proper 8-bits from the virtual address into the physical address before reading.	**15.0 pts** **Full Marks**	**8.0 pts** **Some Problems**	**0.0 pts** **No Marks**			15.0 pts
PartB: Write -Properly translates the virtual address into a physical address. -Uses FindEntry to find the translation -Properly appends the last 8-bits from the virtual address onto the physical address before writing.	**15.0 pts** **Full Marks**	**11.0 pts** **Minor Problems**	**7.0 pts** **Some Problems**	**3.0 pts** **Major Problems**	**0.0 pts** **No Marks**	15.0 pts
						Total Points: 170.0