Finals

1. Cache Memories - 20
2. Dynamic Memory Alloc. - 20
3. Virtual Memory - 20
4. ECF - 20
5. Linking + Relocation:
   7.1 - 7.4
   ________________
   100
1. Dynamic Memory Allocator [10 points]

**Allocator properties:**
1. **Double word** (8 bytes) aligned.
2. **Explicit free list** is used for free block organization.
3. All blocks have a **header** of size 4 bytes and a **footer** of size 4 bytes.
4. **Free blocks** have **prev** and **next** pointers of size 4 **bytes** each.
5. **bit 0** (least significant bit) in the header indicates the use of the current block:
   a. 1 for allocated
   b. 0 for free
6. **zero-sized payloads** are not allowed.
7. **Allocated block size** = `sizeof(header) + sizeof(payload) +` 
   `sizeof(padding) + sizeof(footer)`
8. **Free block size** = `sizeof(header) + sizeof(prev) + sizeof(next) + sizeof(payload) +` 
   `sizeof(padding) + sizeof(footer)`

Please answer the following questions regarding this allocator:

A. Minimum block size = $\frac{16 \text{ bytes}}{32}$

B. Maximum block size = $2 - 8$

C. For the following memory allocation, what is the size of the payload and padding that will be used in the allocated block?

You should **assume** the following:

a. A free block of size **32 bytes** is chosen by the allocator to satisfy the below malloc request.

b. The header of this block is at the memory location **0x8090A0B0**

```
char *p = malloc(8);
```

Payload = 8 bytes

Padding = 16 bytes
D. The contents of the header (and the footer) of a block in the allocator is \( \text{0x000000A9} \).

a. Is the block allocated or free? \( \text{alloc} \)

b. What is the size of the block (in decimal)? \( 168 \text{ bytes} \)

c. Is the contents of the header of this block valid with respect to this allocator? \( \text{Remember:} \) For a block to be valid with respect to an allocator, its size should satisfy the alignment requirement of the allocator.

\[ \frac{168}{8} = 21 \]

YES (OR) NO

2. Cache Hits or Misses? [10 points]

Consider the following \textbf{matrix transpose} function.

```c
void matrix_transpose (int dst[2][2], int src[2][2]) {
    int i, j;
    for (i = 0; i < 2; i++) {
        for (j = 0; j < 2; j++) {
            dst[j][i] = src[i][j];
        }
    }
}
```

Assume this code runs on a machine with the following properties:

- `sizeof(int) == 4`.
- The `src` array starts at address 0 and the `dst` array starts at address 16 (decimal).
- There is a single L1 data cache that is **direct-mapped** with a **block size of 8 bytes**.
- The cache has a **total size of 16 data bytes** and the cache is **initially empty**.
- Accesses to the src and dst arrays are the only sources of read and write misses, respectively.

A. For each row and col, indicate whether the access to \( \text{src[row][col]} \) and \( \text{dst[row][col]} \) is a \textbf{hit (h)} or a \textbf{miss (m)}. For example, reading \( \text{src[0][0]} \) is a miss and writing \( \text{dst[0][0]} \) is also a miss.

<table>
<thead>
<tr>
<th>src array</th>
<th>Column 0</th>
<th>Column 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dst array</th>
<th>Column 0</th>
<th>Column 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B. Repeat part A for a cache with a \textbf{total size of 32 data bytes}.

<table>
<thead>
<tr>
<th>src array</th>
<th>Column 0</th>
<th>Column 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dst array</th>
<th>Column 0</th>
<th>Column 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3. Interrupts, Faults, and System calls [10 points]

For the following exceptions, specify the type of the exception.
The three types of possible events are:
1. Interrupts
2. System Calls
3. Faults

<table>
<thead>
<tr>
<th>No.</th>
<th>Event</th>
<th>Exception Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Accessing memory at virtual address 0x00000000</td>
<td>Fault</td>
</tr>
<tr>
<td>2.</td>
<td>Exiting a C program by calling exit()</td>
<td>System Call</td>
</tr>
<tr>
<td>3.</td>
<td>Pressing CTRL + Z at the keyboard</td>
<td>Int</td>
</tr>
<tr>
<td>4.</td>
<td>Reading some data from a text file using read()</td>
<td>Syst. Call.</td>
</tr>
<tr>
<td>5.</td>
<td>Moving the mouse to click on a desktop icon</td>
<td>Interrupt</td>
</tr>
</tbody>
</table>

4. Linking [20 points]

Consider the three files fact.h and main.c as shown below and answer the questions that follow.

NOTE: Line numbers are provided only for the purposes of answering this question and they are NOT a part of the source files.

```c
#include guarded

1. #ifndef FACT_H
2. #define FACT_H
3. extern unsigned long long int fact(unsigned long int);
4. #endif
```
1. #include "fact.h"
2. extern int g_num_ops;
3. unsigned long long int fact(unsigned long int n) {
4.   unsigned long long int result = 1;
5.   g_num_ops++;
6.   while (n > 1) {
7.     result *= n;
8.     --n;
9.   }
10.  return result;
11. }

main.c
1. #include "fact.h"
2. 
3. #define N 10
4. int g_num_ops = 0;
5. 
6. int main() {
7.   unsigned long long int fact_res[10];
8.   int i;
9.   for (i = 0; i < N; ++i) {
10.      fact_res[i] = fact(i);
11.   }
12. }

The final executable a.out is produced by running the following command:

% gcc fact.c main.c -m32
Questions:

A. What are the files in which the variable `g_num_ops` is declared?

B. What are the files in which the variable `g_num_ops` is defined?

C. Which variables in `fact.c` need relocation?

D. Which variables in `main.c` need relocation?

E. What will happen if the variable `g_num_ops` in `main.c` is made static?

F. What type of object file is `a.out`?
   - `a.out` is generated using the command `gcc sum.c main.c -m32`
   i. Relocatable Object File
   ii. Executable Object File

G. Which variables in `main.c` are stored in the data segment?

H. Which variables in `fact.c` are stored in the data segment?

I. Which variables in `main.c` are stored in the runtime stack segment?

J. Which variables in `fact.c` are stored in the runtime stack segment?
5. Virtual Memory [20 points]

Consider the **page table** (part of it), the state of a 4-entry fully *associative* TLB and the assumptions about memory organization as shown in tables below.

<table>
<thead>
<tr>
<th>VPN</th>
<th>PTE</th>
<th>VPN</th>
<th>PTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0XFF</td>
<td>0X0712</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0XFE</td>
<td>0X0F33</td>
<td>0X0F</td>
<td>0X0F0D</td>
</tr>
<tr>
<td>0XFD</td>
<td>0X0314</td>
<td>0X0E</td>
<td>0X0701</td>
</tr>
<tr>
<td>0XFC</td>
<td>0X0728</td>
<td>0X0D</td>
<td>0X0F1B</td>
</tr>
<tr>
<td>0XFB</td>
<td>0X0737</td>
<td>0X0C</td>
<td>0X0F22</td>
</tr>
<tr>
<td><strong>0XFA</strong></td>
<td><strong>0X0712</strong></td>
<td>0X0B</td>
<td>0X071A</td>
</tr>
<tr>
<td><strong>0XF9</strong></td>
<td><strong>0X0727</strong></td>
<td>0X0A</td>
<td>0X0A06</td>
</tr>
<tr>
<td>0XF8</td>
<td>0X0039</td>
<td>0X09</td>
<td>0X0A3E</td>
</tr>
<tr>
<td>0XF7</td>
<td>0X071F</td>
<td>0X08</td>
<td>0X0F34</td>
</tr>
<tr>
<td>0XF6</td>
<td>0X0F29</td>
<td>0X07</td>
<td>0X0613</td>
</tr>
<tr>
<td>0XF5</td>
<td>0X0070A</td>
<td>0X06</td>
<td>0X0017</td>
</tr>
<tr>
<td>0XF4</td>
<td>0X0F0F</td>
<td>0X05</td>
<td>0X0738</td>
</tr>
<tr>
<td>0XF3</td>
<td>0X071E</td>
<td><strong>0X04</strong></td>
<td>0X0F2D</td>
</tr>
<tr>
<td>0XF2</td>
<td>0X0604</td>
<td>0X03</td>
<td>0X060E</td>
</tr>
<tr>
<td>0XF1</td>
<td>0X031D</td>
<td>0X02</td>
<td>0X0016</td>
</tr>
<tr>
<td>0XF0</td>
<td>0X072B</td>
<td>0X01</td>
<td>0X0F10</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>0X00</td>
<td>0X0608</td>
</tr>
</tbody>
</table>
**TABLE 1**

**ASSUMPTIONS**

<table>
<thead>
<tr>
<th>64 KB physical address space</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 KB virtual address space</td>
</tr>
<tr>
<td>256 byte pages</td>
</tr>
<tr>
<td>Fully associative TLB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VPN</th>
<th>PTE</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0X08</td>
<td>0X0F34</td>
<td>1</td>
</tr>
<tr>
<td>0XFA</td>
<td>0X0F92</td>
<td>0</td>
</tr>
<tr>
<td>0X14</td>
<td>0X073A</td>
<td>1</td>
</tr>
<tr>
<td>0X0E</td>
<td>0X0701</td>
<td>1</td>
</tr>
</tbody>
</table>

A page table entry (PTE) is 16 bits with following break-up:

- **PTE[7:0]** - physical page number (PPN)
- **PTE[8]** - valid bit (V)
- **PTE[9]** - read permission (R)
- **PTE[10]** - write permission (W)
- **PTE[11]** - Supervisor mode (S)
- **PTE[15:12]** - always zero

A. How many bits are required for virtual page number (VPN) for:
   a. the assumptions in table 1
   b. the assumptions in table 1, but instead of 64KB physical address space, we have 128KB physical address space
   c. the assumptions in table 1, but instead of 64KB virtual address space, we have 32KB virtual address space

\[ 2^{7} = 128 \]
B. Given the assumptions in Table 1, how many pages does the entire page table occupy?

C. If possible, convert the following virtual addresses (VA) into their corresponding physical addresses (PA).

Mark the access as:
- **Page Fault**: if the page is on disk, but not in main memory
- **Illegal Access**: if the access does not have desired permission

Assume the process runs in **user mode** and there are no unallocated pages for this process.

Mark only one box for each access.

<table>
<thead>
<tr>
<th>Access</th>
<th>Physical Address</th>
<th>Page Fault</th>
<th>Illegal Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VA=0x04FB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VA=0xFA80</td>
<td>0x1280</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VA=0x1406</td>
<td>0x3A06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VA=0x0220</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VA=0xF1F2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VA=0x01F0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

D. Assume the TLB state shown in the table above when processor issues a read request to virtual address 0xF00D. Modify the TLB state to what it looks like after this access is successful. Make the change to TLB diagram above, not below.
6. Signals [10 points]

The following faulty piece of code tries to count the number of times the Ctrl+C key is pressed by the user:

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int counter = 0;

void SIGINT_handler(int signo)
{
    if(signo == SIGINT)
    {
        counter++;
        sleep(10);
    }
}

int main (void)
{
    struct sigaction sigint_action;
    memset(&sigint_action, 0, sizeof(sigint_action));
    sigint_action.sa_handler = SIGINT_handler;
    sigaction(SIGINT, &sigint_action, NULL);

    printf("Entering while loop\n");

    while(1)
    {
        //Do nothing
    }

    return 0;
}
```
A user starts the program and waits until the statement "Entering while loop" is printed out and then starts pressing Ctrl+C 5 times within the next 10 seconds.

(a) What is the value of counter 50 seconds after the statement "Entering while loop" is printed?

(b) Explain why that value is lower or higher than expected?

7. Cache Miss Rate Analysis [10 points]

You are writing a new 3D game that you hope will earn you fame and fortune. You are currently working on a function to blank the screen buffer before drawing the next frame. The screen you are working with is a $4 \times 4$ array of pixels. The machine you are working on has a 32 bytes direct-mapped cache with 16-byte lines. The C structures you are using are as follows:

```c
struct pixel {
    char r;
    char g;
    char b;
    char a;
};
```

```c
struct pixel, buffer[4][4];
int i, j;
```

Assume the following:
- `sizeof(char) == 1` and `sizeof(int) == 4`.
- `buffer` begins at memory address 0.
- The cache is initially empty.
- The only memory accesses are to the entries of the array buffer.
- Variables `i` and `j` are stored in registers.

What percentage of writes in the following code will miss in the cache?
for (j = 0; j < 4; j++) {
    for (i = 0; i < 4; i++) {
        buffer[i][j].r = 0;
        buffer[i][j].g = 0;
        buffer[i][j].b = 0;
        buffer[i][j].a = 0;
    }
}

What percentage of writes in the following code will miss in the cache?

for (i = 0; i < 4; i++) {
    for (j = 0; j < 4; j++) {
        buffer[i][j].r = 0;
        buffer[i][j].g = 0;
        buffer[i][j].b = 0;
        buffer[i][j].a = 0;
    }
}

8. Heap Memory [10 points]

Consider an allocator with the following properties.

Allocator properties:
1. Single word (4 bytes) aligned.
2. Implicit free list is used for free block organization.
3. Allocator uses best-fit policy for allocating new blocks.
4. All blocks have a header of size 4 bytes.
5. bit 0 (least significant bit) in the header indicates the use of the current block:
   a. 1 for allocated
   b. 0 for free
6. block size = sizeof (header) + sizeof (payload) + sizeof (padding)

Given the contents of the heap shown in FIGURE 1, show the new contents of the heap in FIGURE 2 after a call to char *p = malloc(4); is executed. i.e. Your answers should be given as hexadecimal values.
Note that the address grows from **bottom up**, i.e. the heap starts at the address 0x8049000 and grows upwards. The first block is stored starting at the address 0x8049000.

What is the value of **pointer variable p** if the call to malloc succeeds? 0x804901c

**FIGURE 1**

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Value in Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x804901c</td>
<td>0xFE670031</td>
</tr>
<tr>
<td>0x8049018</td>
<td>0x00000008</td>
</tr>
<tr>
<td>0x8049014</td>
<td>0x12CD00AB</td>
</tr>
<tr>
<td>0x8049010</td>
<td>0x35670011</td>
</tr>
<tr>
<td>0x804900c</td>
<td>0xA1B2C3D4</td>
</tr>
<tr>
<td>0x8049008</td>
<td>0x00000010</td>
</tr>
<tr>
<td>0x8049004</td>
<td>0x12345678</td>
</tr>
</tbody>
</table>

Start of heap => 0x8049000 0x00000009

**FIGURE 2**

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Value in Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x804901c</td>
<td>s</td>
</tr>
<tr>
<td>0x8049018</td>
<td>0x00000009</td>
</tr>
<tr>
<td>0x8049014</td>
<td>S</td>
</tr>
<tr>
<td>0x8049010</td>
<td>S</td>
</tr>
<tr>
<td>0x804900c</td>
<td>S</td>
</tr>
<tr>
<td>0x8049008</td>
<td>S</td>
</tr>
<tr>
<td>0x8049004</td>
<td>S</td>
</tr>
</tbody>
</table>

Start of heap => 0x8049000

Good luck! :)
header = 4 bytes = 32 bits

max size = \( \frac{32}{2} - 1 \)

\[ = \left(2^{32} \right) \left(2 - 1 \right) - 7 \]

\[ = 2^{32} - 8 \]
0x00 00 00 00 A9 \Rightarrow 32 \text{ bits}

<table>
<thead>
<tr>
<th>b2</th>
<th>b1</th>
<th>b0</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>S00</td>
<td>S01</td>
</tr>
<tr>
<td>---</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>16</td>
<td>S10</td>
<td>S11</td>
</tr>
<tr>
<td>24</td>
<td>d00</td>
<td>d01</td>
</tr>
<tr>
<td></td>
<td>d10</td>
<td>d11</td>
</tr>
</tbody>
</table>

- S00
- d00
- S01
- d10
<table>
<thead>
<tr>
<th>Event</th>
<th>Asyn/Syn</th>
<th>Inext/Icur</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt</td>
<td>Asyn</td>
<td>Inext</td>
</tr>
<tr>
<td>Sys call</td>
<td>Syn</td>
<td>Inext+</td>
</tr>
<tr>
<td>Fault</td>
<td>Syn</td>
<td>Icurr</td>
</tr>
</tbody>
</table>

Icurr: Abort routine
1 PTE $\rightarrow$ 2 bytes

Total PTEs = $2^8$ (256)

1 VP $\rightarrow$ 1 PTE

8 VPs $\rightarrow$ $2^8$ PTEs

1 PTE $\rightarrow$ 2 bytes

$2^8$ PTEs $\rightarrow$ $2^8 \times 2$ bytes

$2^9$ bytes $\rightarrow$ PT

512 bytes

256 bytes

256 bytes

512 bytes

To store the entire PT.
Read 0x104 FB (VA)

PTE: 0x0F2D

PPN

PA = 0x2D FB

PPN 2D

b11 b10 b9 b8

S = 1 WR V

VA = 0xFA 8D

VPN

PTE = 0x0F02

PA = PPN PPPD

= 0x12 80

O1111 SWR V

Ox07 12

PPN
VA = 0x14D6

VPN | VPO

PTE = 0x073A

0111
8 WRV

PA = 0x3A06
Cache

<table>
<thead>
<tr>
<th>b00</th>
<th>b01</th>
<th>b02</th>
<th>b03</th>
</tr>
</thead>
<tbody>
<tr>
<td>b10</td>
<td>b11</td>
<td>b12</td>
<td>b13</td>
</tr>
<tr>
<td>b20</td>
<td>b21</td>
<td>b22</td>
<td>b23</td>
</tr>
<tr>
<td>b30</td>
<td>b31</td>
<td>b32</td>
<td>b33</td>
</tr>
</tbody>
</table>

bo0 = \( \frac{1}{4} \) misses.
bo1 = \( \frac{1}{4} \) misses

\( \text{avg} \ b_{ij} \Rightarrow \frac{1}{4} \) misses

25\% miss rate

\[
\begin{align*}
\text{bo0} & \rightarrow \frac{1}{4} \\
\text{bo1} & \rightarrow \frac{0}{4} \\
\text{bo2} & \rightarrow \frac{0}{4} \\
\text{bo3} & \rightarrow \frac{0}{4} \\
\frac{1}{16} \times 100 &= 6.25\%
\end{align*}
\]