HW1-Fall2018

**CS252 Homework**

**Name:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Dayu Liu\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Answer this practice exam and bring it the day of the midterm exam.

**Part 1. True False Questions**

**Answer True/False (T/F)**

\_T\_\_\_ The loader is also called “Runtime Linker”

\_\_T\_\_ COFF is a format for executable files

\_\_F\_\_ The command "chmod 440  file" makes a file readable and writable by user, group, and others.

\_F\_\_\_ strace is a UNIX command that shows the tree of processes in the system.

\_\_F\_\_ All processes have a parent process.

**Part 2. Short questions.**

2. Enumerate and describe the memory sections of a program.

Text-code instructions

Data - initialized global variables

Bss - uninitialized global variables

Heap - dynamic memory allocated by malloc, new, etc

Stack – local variables

3. Enumerate and describe the contents of an inode

Major numbers: determine the devices

Minor numbers: determine what file is referred to inside the device

Mode: permission of read, write, execute

Owners: userid, groupid

Time stamps: time of creation, access, modification

Size: size of file in bytes

Reference count: number of times the i-node appears in a directory

4. Enumerate the 5 Memory Allocation Errors and describe them.

Premature free – an object is still in use but its been freed

Int \*p = (int \*)malloc(sizeof(int));

\*p = 8;

Free (p);

\*p = 9;

Double free- free an object that has been freed before

Int \*p = (int \*)malloc(sizeof(int));

free (p);

free (p);

Memory leak-objects that are no longer used are not freed

While (1){

Ptr = malloc (100);

}

Memory smashing- not enough memory allocated to be used

Char \* s = malloc(8);

Strcpy (s, “hello world”);

Wild free- Free on non-heap object

Int q;

Int \*p = &q;

Free (p)

5. List and explain the attributes of an Open File Object.

inode: It uniquely identifies a file: major numbers and minor numbers

open mode-how the file is opened: read only, read write, append

offset: The next read or write operation will start at this offset in the file

reference count: number of file descriptors that point to this Open File Object

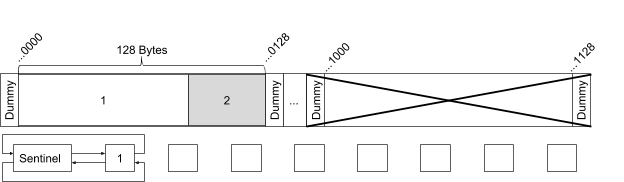
Malloc

Below is a diagram showing current state of a memory allocator like the one implemented in lab

1. The only difference is that the arena size is only 128 bytes, to simplify the arithmetic. All of the data structures are the same as they were in the lab and the code is being run on a 64-bit linux system, like the lab machines or data. The top diagram shows how the blocks are laid out in memory, the lower diagram is a representation of the free list, and the table contains the metadata about each block. Addresses are truncated and given in decimal for simplicity.

In the diagram given below there are two blocks in a single arena. Block 1 is not allocated and is the only node in the free list. Block 2 has been allocated. There is space for a second arena to be allocated if necessary. If the second arena is not required simply cross it out as in the diagram below. For each malloc/free call update the memory space diagram, free list, and table as necessary to contain the state of the allocator after performing the requested operation. For calls to malloc include the value returned by the malloc call.

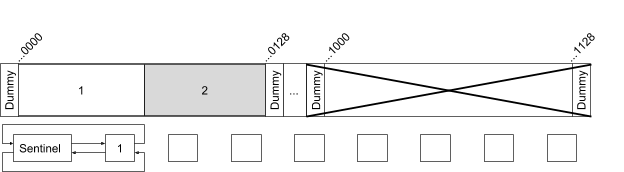
Example Diagram:



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Block | Offset | Size | Left Size | Allocated |
| 1 | ...0000 | 88 | 16 | 0 |
| 2 | ...0088 | 40 | 88 | 1 |
|  | ... |  |  |  |
|  | ... |  |  |  |

Question 1

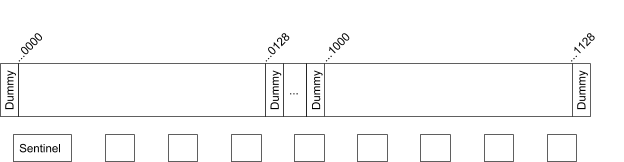
This is the heap before the operation:



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Block | Offset | Size | Left Size | Allocated |
| 1 | ...0000 | 64 | 16 | 0 |
| 2 | ...0064 | 64 | 64 | 1 |
|  | ... |  |  |  |
|  | ... |  |  |  |

malloc(8);

Draw the heap after the operation above and fill up the table.

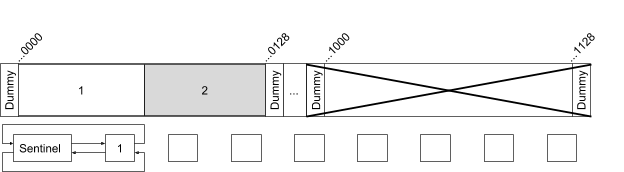


|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Block | Offset | Size | Left Size | Allocated |
|  | ... |  |  |  |
|  | ... |  |  |  |
|  | ... |  |  |  |
|  | ... |  |  |  |

What is the value returned by the operation above assuming the beginning of the allocable memory in the first arena is address 10000?

Question 2

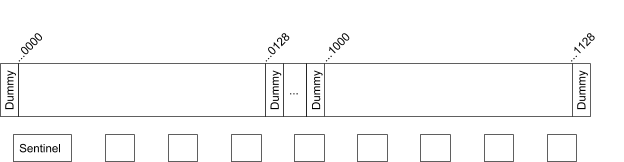
This is the heap before the operation:



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Block | Offset | Size | Left Size | Allocated |
| 1 | ...0000 | 64 | 16 | 0 |
| 2 | ...0064 | 64 | 64 | 1 |
|  | ... |  |  |  |
|  | ... |  |  |  |

malloc(1);

Draw the heap after the operation above and fill up the table.

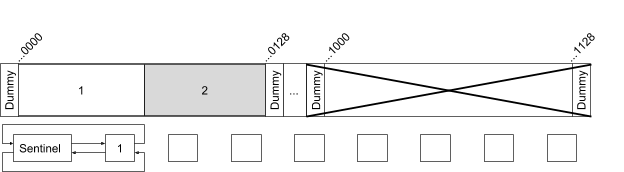


|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Block | Offset | Size | Left Size | Allocated |
|  | ... |  |  |  |
|  | ... |  |  |  |
|  | ... |  |  |  |
|  | ... |  |  |  |

What is the value returned by the operation above assuming the beginning of the allocable memory in the first arena is address 10000?

Question 3

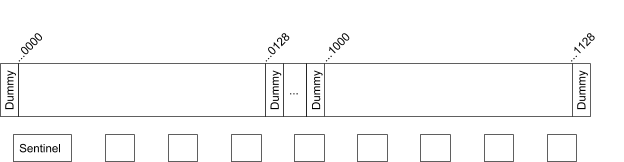
This is the heap before the operation:



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Block | Offset | Size | Left Size | Allocated |
| 1 | ...0000 | 64 | 16 | 0 |
| 2 | ...0064 | 64 | 64 | 1 |
|  | ... |  |  |  |
|  | ... |  |  |  |

malloc(32);

Draw the heap after the operation above and fill up the table.

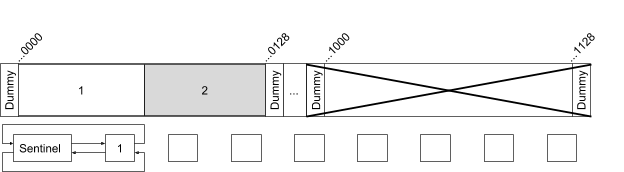


|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Block | Offset | Size | Left Size | Allocated |
|  | ... |  |  |  |
|  | ... |  |  |  |
|  | ... |  |  |  |
|  | ... |  |  |  |

What is the value returned by the operation above assuming the beginning of the allocable memory in the first arena is address 10000?

Question 4

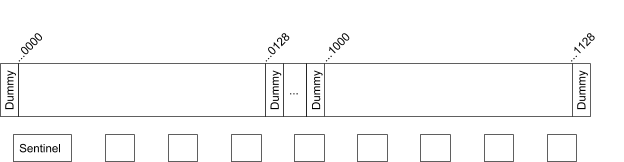
This is the heap before the operation:



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Block | Offset | Size | Left Size | Allocated |
| 1 | ...0000 | 64 | 16 | 0 |
| 2 | ...0064 | 64 | 64 | 1 |
|  | ... |  |  |  |
|  | ... |  |  |  |

malloc(128);

Draw the heap after the operation above and fill up the table.

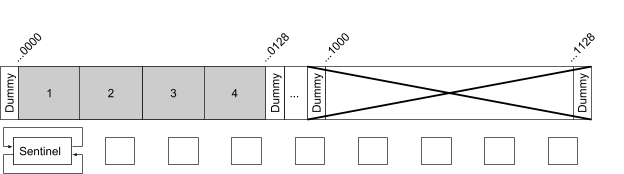


|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Block | Offset | Size | Left Size | Allocated |
|  | ... |  |  |  |
|  | ... |  |  |  |
|  | ... |  |  |  |
|  | ... |  |  |  |

What is the value returned by the operation above assuming the beginning of the allocable memory in the first arena is address 10000?

Question 5

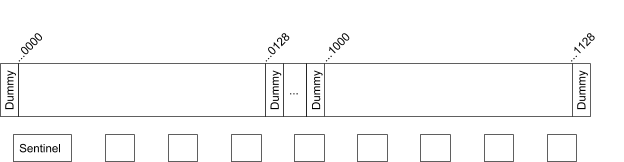
This is the heap before the operation:



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Block | Offset | Size | Left Size | Allocated |
| 1 | ...0000 | 32 | 16 | 1 |
| 2 | ...0032 | 32 | 32 | 1 |
| 3 | ...0064 | 32 | 32 | 1 |
| 4 | ...0096 | 32 | 32 | 1 |

free(...0096) // free block 4

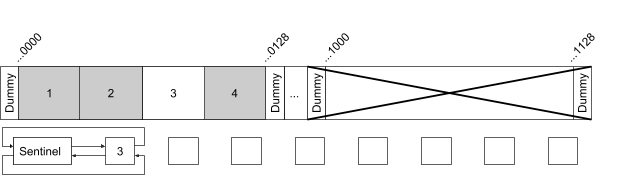
Draw the heap after the operation above and fill up the table.



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Block | Offset | Size | Left Size | Allocated |
|  | ... |  |  |  |
|  | ... |  |  |  |
|  | ... |  |  |  |
|  | ... |  |  |  |

Question 6

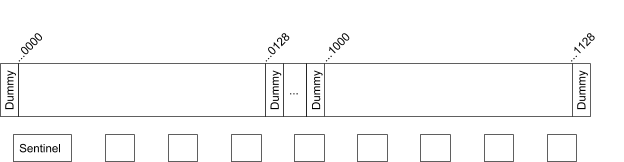
This is the heap before the operation:



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Block | Offset | Size | Left Size | Allocated |
| 1 | ...0000 | 32 | 16 | 1 |
| 2 | ...0032 | 32 | 32 | 1 |
| 3 | ...0064 | 32 | 32 | 0 |
| 4 | ...0096 | 32 | 32 | 1 |

free(...0096) // free block 4

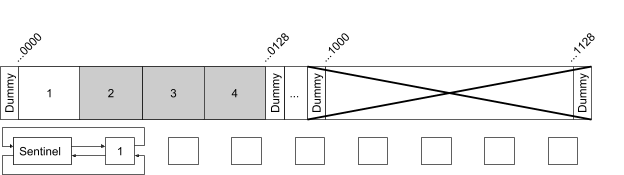
Draw the heap after the operation above and fill up the table.



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Block | Offset | Size | Left Size | Allocated |
|  | ... |  |  |  |
|  | ... |  |  |  |
|  | ... |  |  |  |
|  | ... |  |  |  |

Question 7

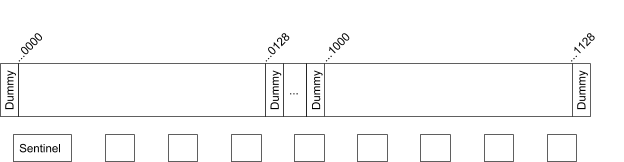
This is the heap before the operation:

****

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Block | Offset | Size | Left Size | Allocated |
| 1 | ...0000 | 32 | 16 | 0 |
| 2 | ...0032 | 32 | 32 | 1 |
| 3 | ...0064 | 32 | 32 | 1 |
| 4 | ...0096 | 32 | 32 | 1 |

free(...0032) // free block 2

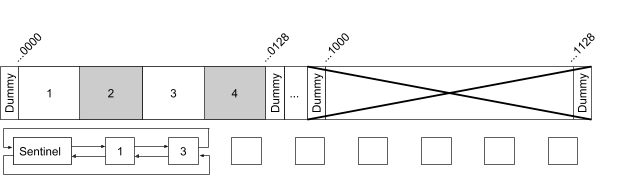
Draw the heap after the operation above and fill up the table.



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Block | Offset | Size | Left Size | Allocated |
|  | ... |  |  |  |
|  | ... |  |  |  |
|  | ... |  |  |  |
|  | ... |  |  |  |

Question 8

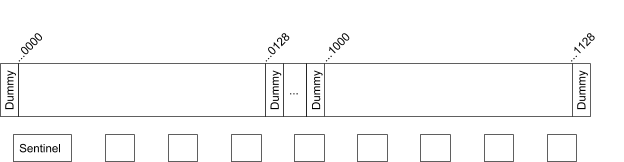
This is the heap before the operation:



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Block | Offset | Size | Left Size | Allocated |
| 1 | ...0000 | 32 | 16 | 0 |
| 2 | ...0032 | 32 | 32 | 1 |
| 3 | ...0064 | 32 | 32 | 0 |
| 4 | ...0096 | 32 | 32 | 1 |

free(...0032) // free block 2

Draw the heap after the operation above and fill up the table.



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Block | Offset | Size | Left Size | Allocated |
|  | ... |  |  |  |
|  | ... |  |  |  |
|  | ... |  |  |  |
|  | ... |  |  |  |

Shell Scripting

**1. Print the current user’s remaining disk quota space as a single human readable number.**

# Usage: quota\_left

**2. Print the limit largest files below the current directory with sizes in human readable format.**

# Usage: large\_files <limit>

**3. Create a backup of the file and every frequency seconds check if there are changes to the original file. If there are changes then recopy the original file to the backup file. The backup file should be a hidden file with “\_backup” appended to the end.**

**ex:**

**myMalloc.c**

**.myMalloc.c\_backup**

# Usage: backup <file> <frequency>

function backup() {

}

**4. Check if a  US phone number is valid: “(765) 123 4567” or “1 (765) 123 4567” with the leading 1 being optional. print “valid” if it is valid or “invalid” otherwise**

**Note: The only valid forms of a phone number for the purposes of this question are the ones described above**

# Usage: valid <number>

function valid() {

}

Unix System Calls

Below are a series of small programs. For each answer the following three questions:

1. **Does the program’s behavior match the specified behavior?**
2. **If not why is it different?**
3. **How can it be changed to work as described?**

You may make a few assumptions about the programs. They were compiled and are running on a 64-bit Linux system like one of the lab machines or data. Also we may assume that any system calls that are being made will be successful. In the case of the program having different behavior than is described this could either be due to some kind of crash/hang, another issue that causes the behavior to be nondeterministic, or simply an bug causing incorrect output.

1. Who am I?

**Prints the contents of argv to the terminal**

1. Is the current behavior of this program the behavior described above?
2. If not, why is it different?
3. How can it be changed to work as described?

|  |
| --- |
| #include<stdio.h> #include<stdlib.h> #include<unistd.h> int main(int argc, char \*\* argv) {    if (argc == 0) {        puts("");        exit(0);    }    printf("%s \n", argv[0]);    execvp(“/proc/self/exe”, ++argv); } |

2. Ready! Set! Go!

**The parent should let its child win the race condition. We expect to see:**

**On your mark!**

**Get set!**

**Go!**

**Child**

**Parent**

1. Is the current behavior of this program the behavior described above?
2. If not, why is it different?
3. How can it be changed to work as described?

|  |
| --- |
| #include<stdio.h> #include<unistd.h> int main(int argc, char \*\* argv) {    puts("On your mark!");    puts("Get set!");    puts("Go!");    if (fork()) {        puts("Parent");    } else {        sleep(1);          puts("Child");    }  } |

3. XL Pipeline

**print 100,000 !’s to the terminal after they have been passed by the parent to the child through the pipe.**

1. Is the current behavior of this program the behavior described above?
2. If not, why is it different?
3. How can it be changed to work as described?

|  |
| --- |
| #include<stdio.h> #include<unistd.h> int main(int argc, char \*\* argv) {    int pipeFd[2];    pipe(pipeFd);    for (int i = 0; i < 100000; i++) {        char c = '!';        write(pipeFd[1], &c, 1);    }    if (!fork()) {        char c;        for (int i = 0; i < 100000; i++) {            read(pipeFd[0], &c, 1);            write(1, &c, 1);        }        puts("");    } } |

4. Slow Author/Patient Reader

**The patient reader prints the string containing the numbers from 0 to 9.**

1. Is the current behavior of this program the behavior described above?
2. If not, why is it different?
3. How can it be changed to work as described?

|  |
| --- |
| #include<stdio.h> #include<stdlib.h> #include<unistd.h> int main(int argc, char \*\* argv) {    int pipeFd[2];    pipe(pipeFd);    if (fork()) {        char s[12];        fgets(s, 11, fdopen(pipeFd[0], "r"));        puts(s);    } else {        char c;        for (int i = 0; i < 10; i++) {            sleep(1);            char c = '0' + (char)i;            write(pipeFd[1], &c, 1);            puts("It'll be worth the wait!");        }        c = '\n';        write(pipeFd[1], &c, 1);    } } |

Published by [Google Drive](https://docs.google.com/)–[Report Abuse](https://docs.google.com/abuse?id=e/2PACX-1vSZuOoyGvknJg6vJLQZIiJPBfTJf93MDgW0kAVEu7H6O3LBBo7TWZ_wN8MLzcF_khHA1zhshvw6HRGW)–Updated automatically every 5 minutes