CS252 Homework 1

Name:__ Zheying Lu _____

Answer this practice exam and turn it in pdf format using the following command in data:

```
turnin -c cs252 -p hw1 hw1.pdf
```

Part 1. True False Questions

Answer True/False (T/F)

_T___ The loader is also called “Runtime Linker”
__F__ COFF is a format for executable files
__F__ The command "chmod 440 file" makes a file readable and writable by user, group, and others.
__T__ 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.
When assuming a 64-bit architecture, a program sees memory as an array of bytes that goes from address 0 to 2^64-1
It has Text part as instructions that the program runs, data as initialized global variables, BSS as uninitialized global variables, heap as memory returned, stack as place to store local variables.

3. Enumerate and describe the contents of an inode
An inode has information about a file and what blocks make the file.

4. Enumerate the 5 Memory Allocation Errors and describe them.
Memory leaks:
Are objects in memory that are no longer in use by the program but that are not freed
Premature frees:
Is caused when an object that is still in use by the program is freed
Memory Smashing:
Happens when less memory is allocated than the memory that will be used
Wild Frees:
Happen when a program attempts to free a pointer in memory that was not returned by malloc
Double frees:
Is caused by freeing an object that is already free

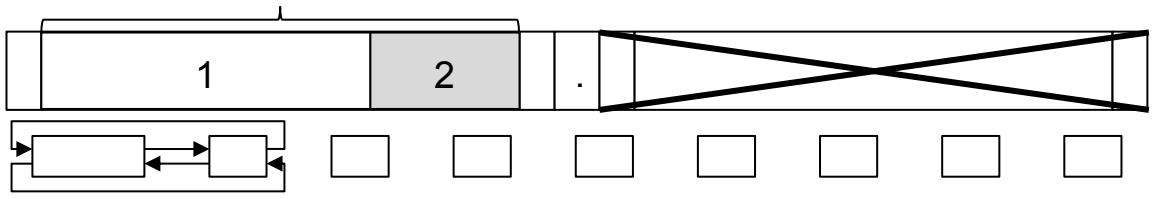
5. List and explain the attributes of an Open File Object.
It is stored in the kernel. It contains the state of an open file: I-Node/Open Mode/Offset/Reference Count
Node:
uniquely identifies a file in the computer
Open Mode:
How the file was opened: Read Only, Read Write, Append
Offset:
The next read or write operation will start at this offset in the file
Reference Count:
It is increased by the 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.

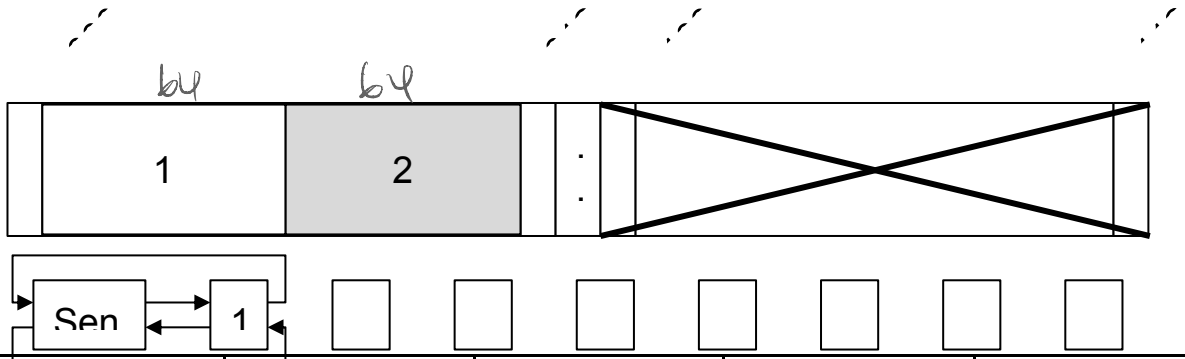
Example Diagram:



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

Question 1

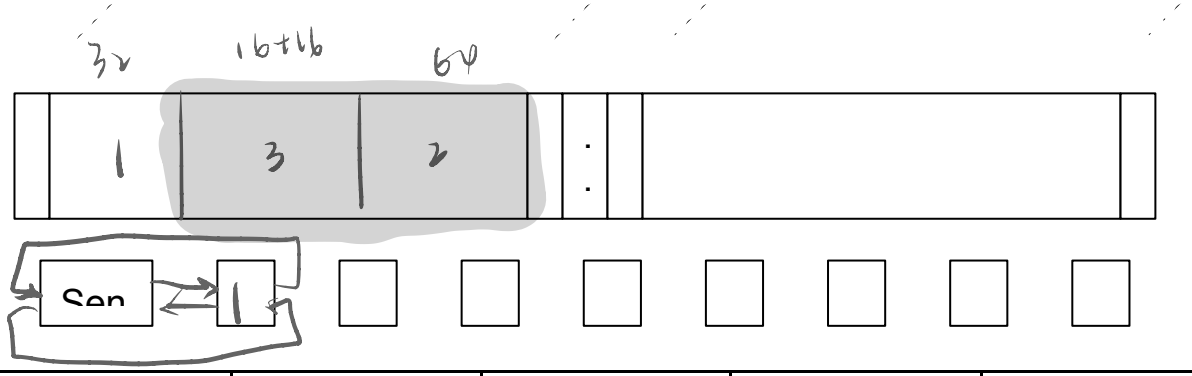
This is the heap before the operation:



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.



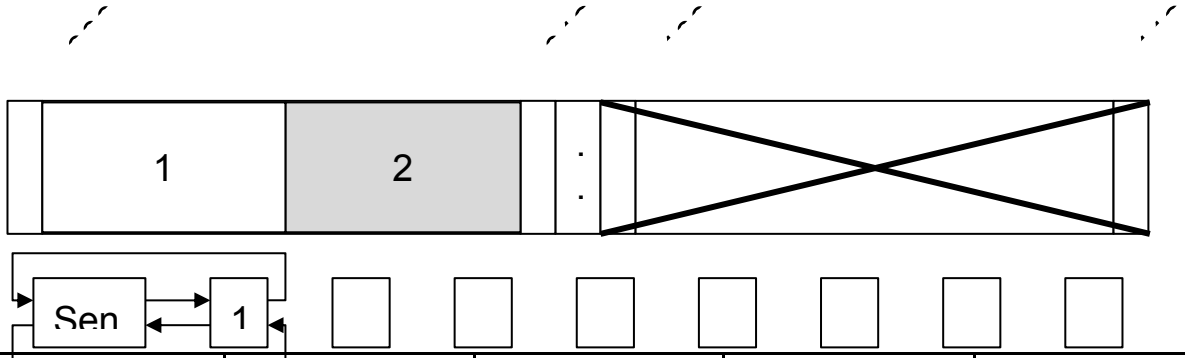
Block	Offset	Size	Left Size	Allocated
1	... 0000	32	16	0
2	... 0064	32	32	1
3	... 0032	64	32	1
	...			

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

10032 + sizeof (boundary tag)

Question 2

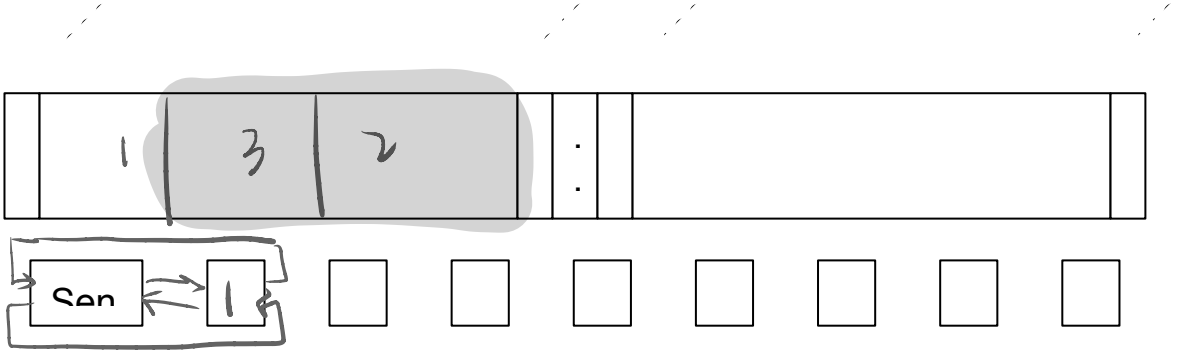
This is the heap before the operation:



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.



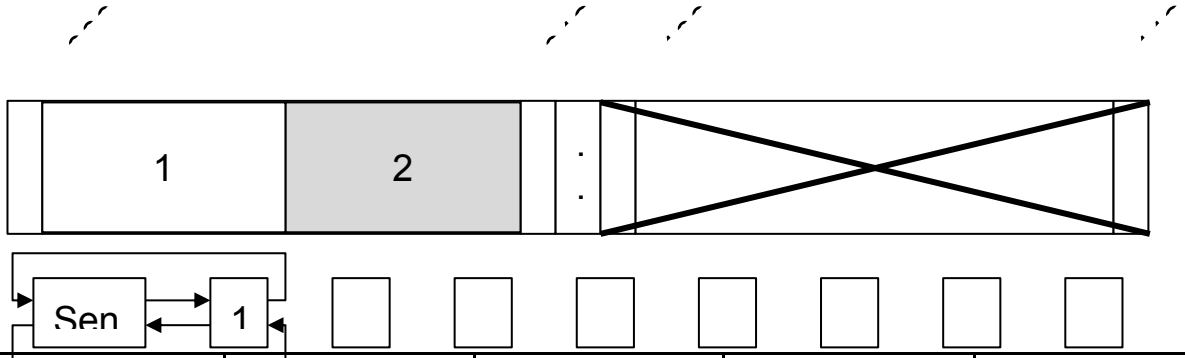
Block	Offset	Size	Left Size	Allocated
1	... 0000	32	16	0
2	... 0064	32	32	1
3	... 0032	64	32	1
	...			

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

10032 + sizeof (boundary tag)

Question 3

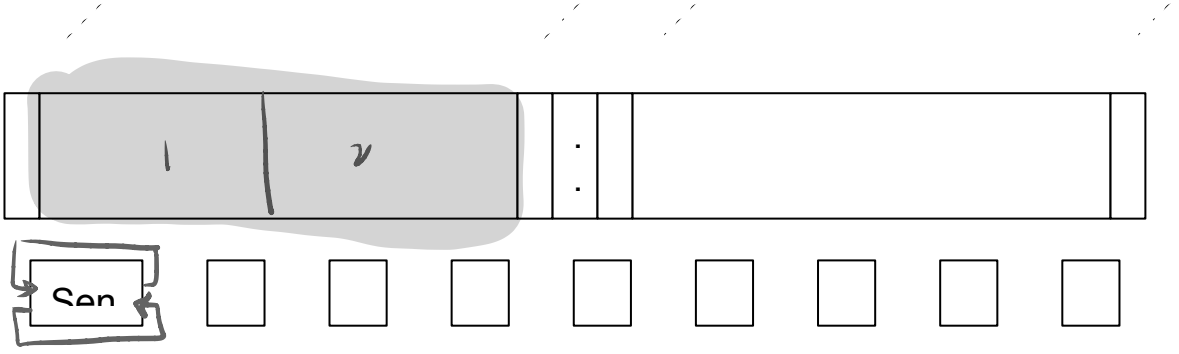
This is the heap before the operation:



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.



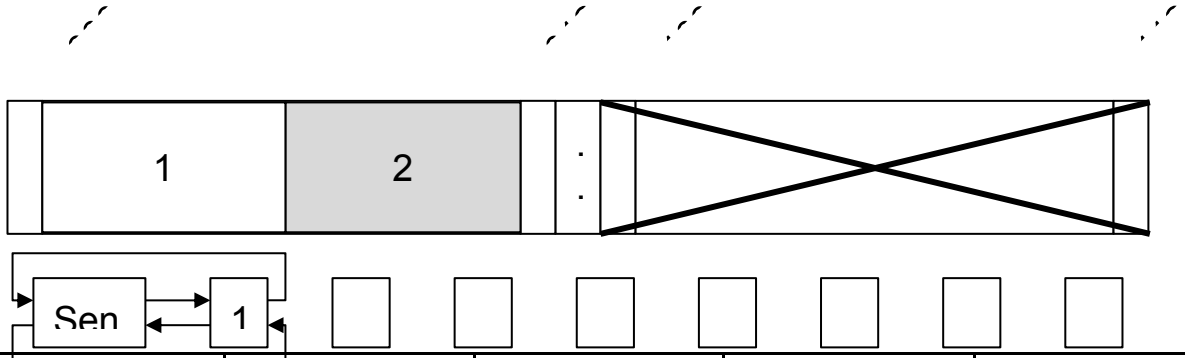
Block	Offset	Size	Left Size	Allocated
1	... 0000	64	16	1
2	... 0064	64	64	1
	...			
	...			

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

10000 + sizeof (boundary tag)

Question 4

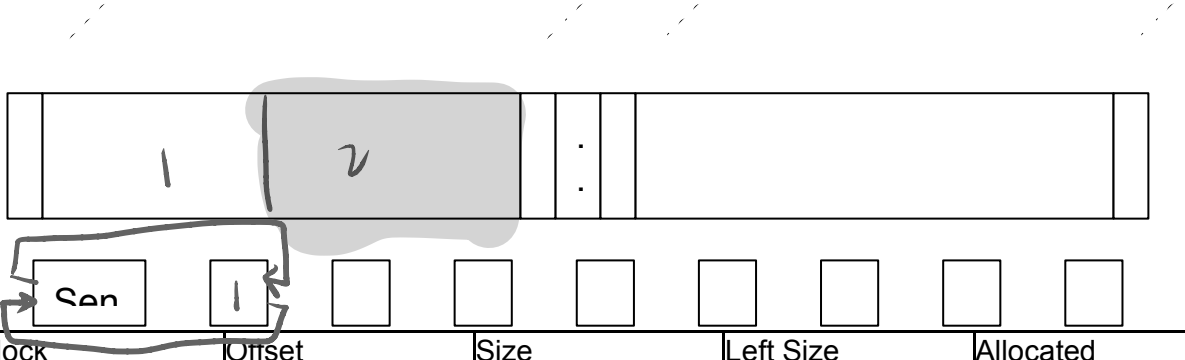
This is the heap before the operation:



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.



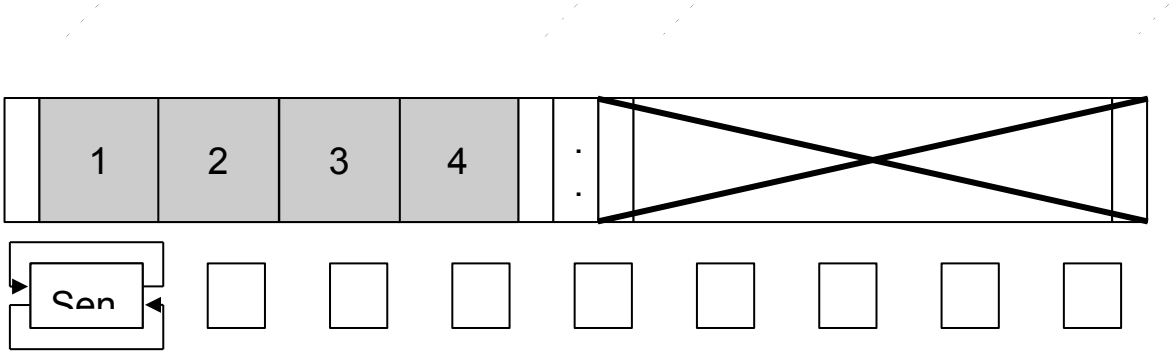
Block	Offset	Size	Left Size	Allocated
1	... 0000	64	16	0
2	... 0064	64	64	1
	...			
	...			

What is the value returned by the operation above assuming the beginning of the allocable memory in the first arena is address 10000? (Assume max heap size is 128bytes).

Null

Question 5

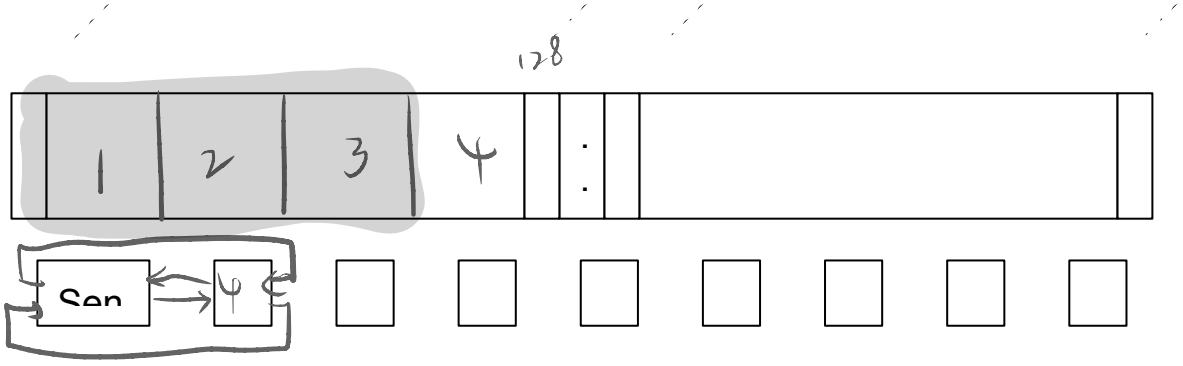
This is the heap before the operation:



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
```

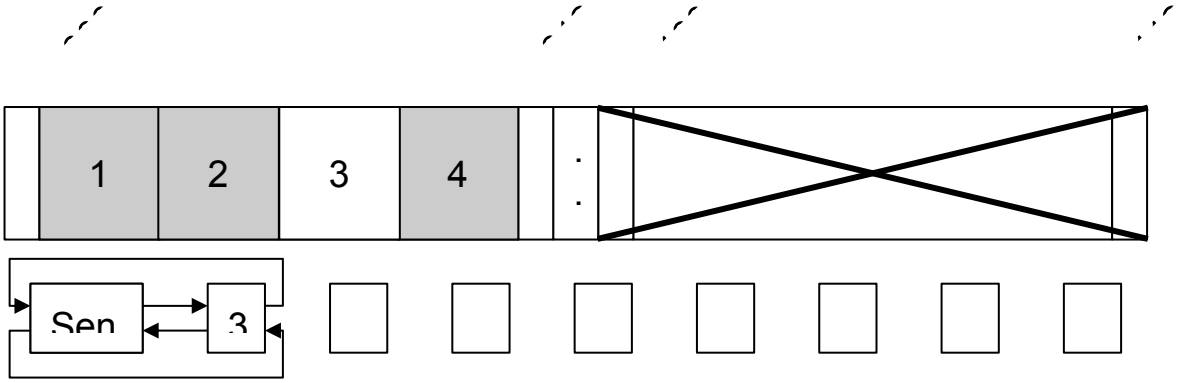
Draw the heap after the operation above and fill up the table.



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

Question 6

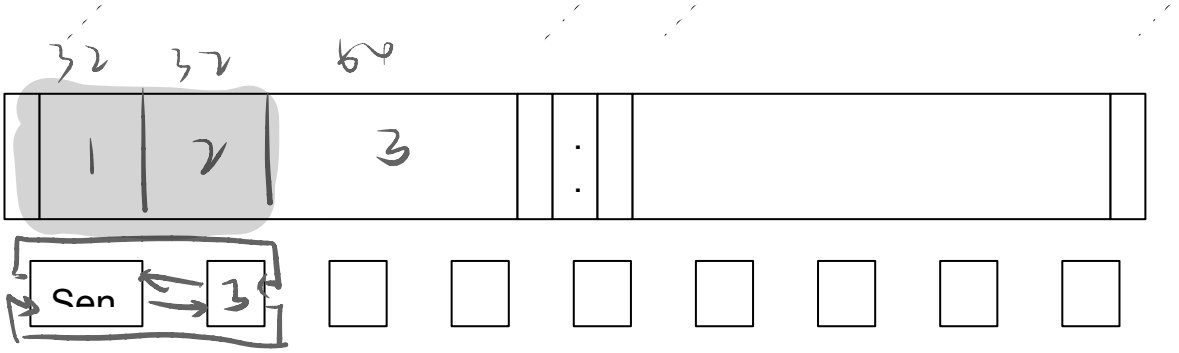
This is the heap before the operation:



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
```

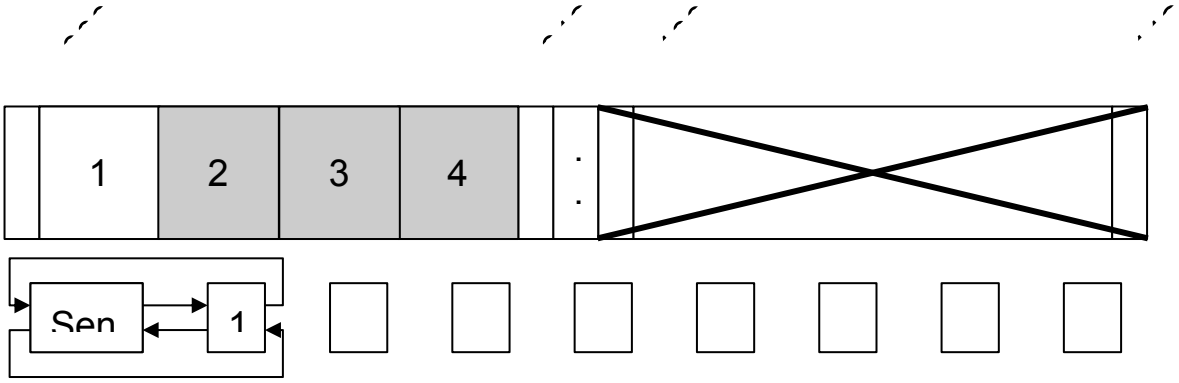
Draw the heap after the operation above and fill up the table.



Block	Offset	Size	Left Size	Allocated
1	... 0000	32	16	1
2	... 0032	32	32	1
3	... 0064	64	32	0
	...			

Question 7

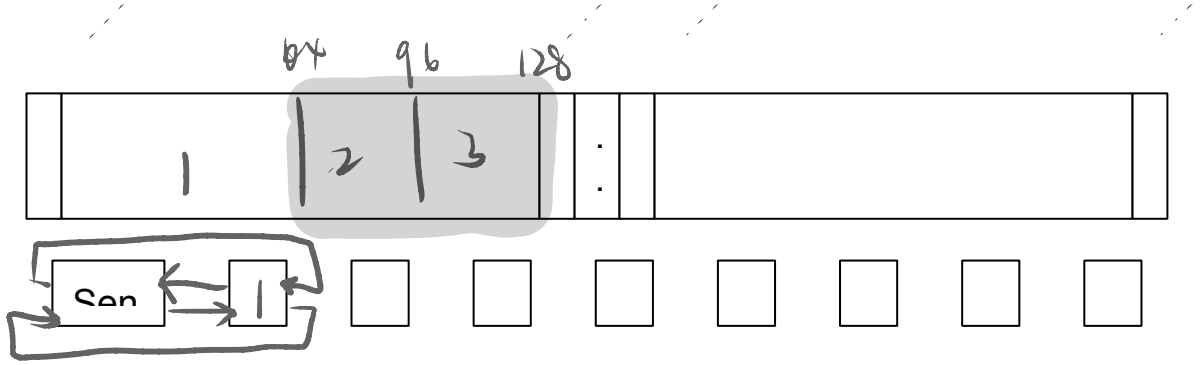
This is the heap before the operation:



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
```

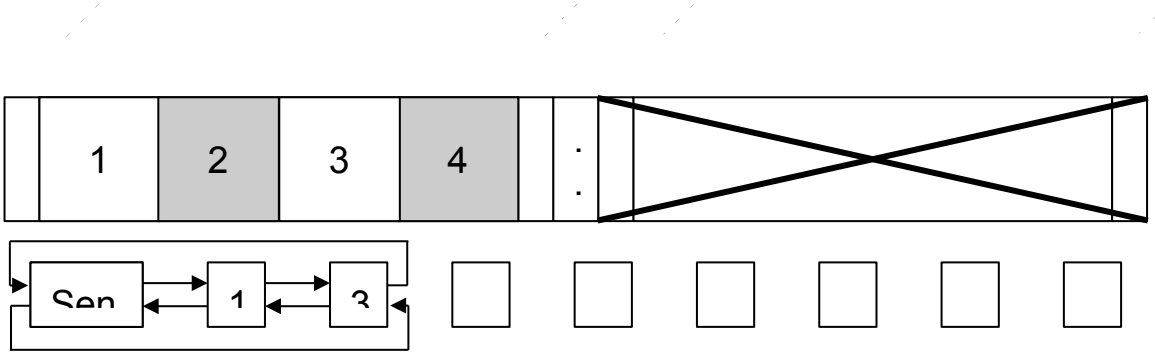
Draw the heap after the operation above and fill up the table.



Block	Offset	Size	Left Size	Allocated
1	... 0000	64	16	0
2	... 0064	32	64	1
3	... 0096	32	32	1
	...			

Question 8

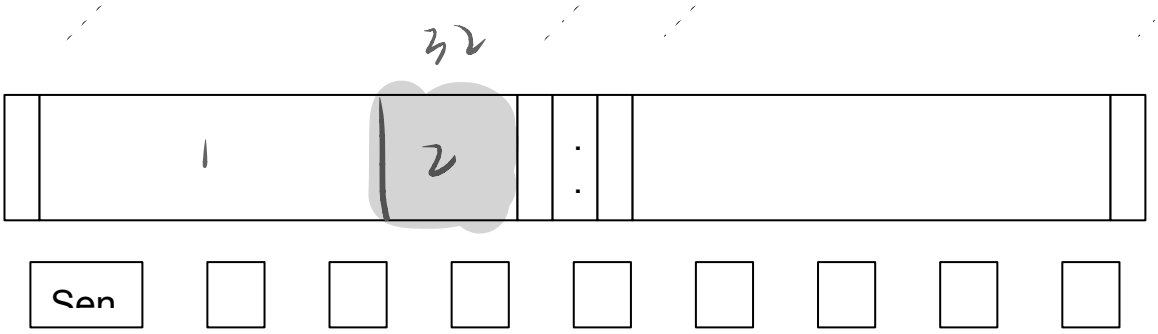
This is the heap before the operation:



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
1	... 0000	96	16	0
2	... 0096	32	96	1
	...			
	...			

Shell Scripting

1. Print the names of the files below the current directory that are larger than <limit> bytes.

```
# Usage: large_files <limit>
```

```
quota -s | tail -1 | awk '{print $2}'
```

3. Write a shell script that loops forever and prints every 5 secs the physical memory used by a process.

#Usage: mem_proc <pid>

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() {
    echo $1 | grep -eq '^(1)?\([0-9]{3}\) [0-9]{3} [0-9]{4}$'
    if [ $? -eq 0 ]; then
        echo "valid"
    else
        echo "invalid"
    fi
```

```
}
```

Unix System Calls

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

a) **Does the program's behavior match the specified behavior?**
b) **If not, why is it different?**
c) **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.

5. Who am I?

Prints the contents of `argv` to the terminal

a) Is the current behavior of this program the behavior described above?
b) If not, why is it different?
c) 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);
}
```

a) Yes

6. 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
```

a) Is the current behavior of this program the behavior described above?
b) If not, why is it different?
c) 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");
    }
}
```

(a) No
(b) Because the parent will usually print before the child
(c) The parent should call waitpid to wait for the child.

7. XL Pipeline

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

a) Is the current behavior of this program the behavior described above?
b) If not, why is it different?
c) 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("");
    }
}
```

(a) No

(b) Because linux pipe has a fixed buffer size. But the parent process writes 100,000 bytes to the pipe

(c) Move the writing process after the child proces

8. From your shell project write the simplified function "execute" that will execute the command passed as argument. Each command is made of multiple SimpleCommands that communicate with pipes. simpleCommand[0] will take the input from file "input" and it will pass its output to simpleCommand[1] and so on. The output of the last SimpleCommand will be passed to the file in "output" if any. Then it will wait until the last command finishes. If "input" or "output" are NULL then use the default input/output.

```
typedef struct SimpleCommand {
  const char * arguments[]; // Command and arguments of this SimpleCommand. Last
argument is NULL.
};

typedef struct Command {
   int numberOfSimpleCommands;     // Number of simple commands
  SimpleCommand simpleCommand[]; // Array of simpleCommands
  const char * input;                    // Input file or NULL if default input
  const char * output;                   // Output file or NULL if default output
 int background;
};

void execute( Command * command) {
      int tmpin = dup(0);
      int tmpout = dup(1);
      int tmperr = dup(2);
   int fdin;
     if ( input != NULL){fdin= open (input, O_CREAT| O_WRONLY |
                                          O_TRUNC, 0664) }
       else {fdin = dup(tmpin)}
   for ( int i = 0; i < command->numberOfSimpleCommand; i++) {
       dup2 (fdin,0);
       close(fdin);
        if (i == numsimplecommands-1) {
            if (output != NULL) {
                  fdout = open (output, O_CREAT| O_WRONLY |
                                          O_TRUNC, 0664) }
            else { fdout=dup(tmpout) }
        }
```

```
    else {
        int fdpipe[2];
        pipe(fdpipe);
        fdout = fdpipe[1];
        fdin = fdpipe[0];
    }
    dup2(fdout, 1);
    close(fdout);
    ret = fork();
    if (ret == 0) {
        perror("execvp");
        _exit(1);
    }
}
dup2(tmpin, 0);
dup2(tmpout, 1);
close(tmpin);
close(tmpout);
if (!background) {
    waitpid(ret, NULL, 0);
}

}
```

9. From your shell project write the function that does the wildcard expansion. Assume

that the function wildcardToRegularExpression is given.

```
int maxEntries = 20;
int nEntries = 0;
char ** array;

void expandWildcardsIfNecessary(char * arg)
{
  if (strchr((char*)arg->c_str(), '*') == NULL &&
      strchr((char*)arg->c_str(), '?') == NULL) {
    Command::_currSimpleCommand->insertArgument(arg);
    return;
  }

  char* reg = (char*) malloc(2*strlen(arg)+10);
  char* a = arg;
  char* r = reg;
  *r = '^'; r++;
  while (*a) {
    if (*a == '*') { *r = '.'; r++; *r = '*'; r++; }
    else if (*a == '?') { *r = '.'; r++; }
    else if (*a == '.') { *r = '\\'; r++; *r = '.'; r++; }
    else { *r = *a; r++; }
    a++;
  }
  *r = '$'; r++; r = 0;
  regex_t regex;
  int expbuf = regcomp(&regex, reg, REG_EXTENDED|REG_NOSUB);
  if (expbuf) { perror("compile"); return; }
  free(reg);
  struct dirent *ent;
  regmatch_t match;
  while ((ent = readdir(dir)) != NULL) {
    if (!regexec(&regex, ent->d_name, 1, &match, 0)) {
      if (*temp) {
        if (ent->d_type == DT_DIR) {
          char* nPrefix = (char*) malloc(150);
          if (!strcmp(toOpen, ".")) nPrefix = strdup(ent->d_name);
          else if (!strcmp(toOpen, "/")) sprintf(nPrefix, "%s%s",
              toOpen, ent->d_name);
          else sprintf(nPrefix, "%s/%s", toOpen, ent->d_name);
          expandWildcards(nPrefix, (*temp == '/') ? ++temp : temp);
          free(nPrefix);
        }
      } else {
        if (nEntries == maxEntries)
          maxEntries *= 2;
          array = (char**) realloc(array, maxEntries * sizeof(char*));
          assert(array != NULL);
      }
      char * argument = (char*) malloc(100);
      argument[0] = '\0';
      if (prefix) sprintf(argument,
          "%s%s", prefix, ent->d_name);
      if (ent->d_name[0] == '.') {
        if (*suffix == '.') {
          array[nEntries] = (argument[0]
              != '\0') ? strdup(argument) :
            strdup(ent->d_name);
          nEntries++;
        } else {
          array[nEntries] = (argument[0]
              != '\0') ? strdup(argument) :
            strdup(ent->d_name);
          nEntries++;
        }
        free(argument);
      }
    }
  }
  regfree(&regex);
  closedir(dir);
}
```

10. (20 pts.) Write a program "grepsort arg1 arg2 arg3" that implements the command "grep arg1 | sort < arg2 >> arg3". The program should not return until the command

finishes. "arg1", "arg2", and "arg3" are passed as arguments to the program. Example of the usage is **"grepsort hello infile outfile"**. This command will print the entries in file **infile** that contain the string **hello** and will append the output sorted to file **outfile**. Do error checking. Notice that the output is appended to arg3.

```
int main( int argc, char ** argv)
{
    if (argc<4){
        fprintf(stderr, "%s", "error");
        exit(1);}
    int dfin = dup(0);
    int dfout = dup(1);
    int dferr = dup(2);
    int fdpipe[2];
    dup2(dfin,0);
    dup2(fdpipe[1],1);
    dup2(fderr,2);
    int pid = fork();
    if (pid==0) {
        close(fdpipe[0]);
        close(fdpipe[1]);
        close(dfin);
        close(dfout);
        close(dferr);
        exit(1);
    }
```

```
}
```