CS252 Homework 1

Name: Maxime Kotur

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”
__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.
Data: Initialized Global variables
Heap: Memory returned when calling malloc/new It grows upwards. Dynamic memory allocation
Bss: Uninitialized global variables. They are initialized to zeroes.
Text: Instructions that the program runs
Stack: It stores local variables and return addresses. It grows downwards.

3. Enumerate and describe the contents of an inode

- Mode - Read, Write, Execute. Also tells if file is directory, device, symbolic link
- Owners - Userid, Groupid
- Major number – Determines the devices
- Minor number –It determines what file it refers to inside the device.
- Time Stamps - Creation time, Access Time, Modification Time.
- Size - Size of file in bytes
- Reference Count – Reference count with the number of times the i-node appears in a directory (hard links). It Increases every time file is added to a directory. The file the i-node represents will be removed when the reference count reaches 0.

- Direct block – There are 12 of them in the structure. Points directly to the block.
- Single indirect – There are 3 of them. Points to a block table that has 256 entry's.
- Double indirect – Points to a page table of 256 entries which then points to another page table of 256
- Triple Indirect - Points to a page table of 256 entries which then points to another page table of 256 that points to another page of 256 bytes

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

- Memory Leaks - Memory leaks are objects in memory that are no longer in use by the program but that are not freed.
- Premature Free - A premature free is caused when an object that is still in use by the program is freed.
- Double Free- It is caused by freeing an object that has already been freed.
- Wild Frees - Wild frees happen when a program attempts to free a pointer in memory that was not returned by malloc.
- Memory Smashing - Memory Smashing happens when less memory is allocated than the memory that will be used.

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

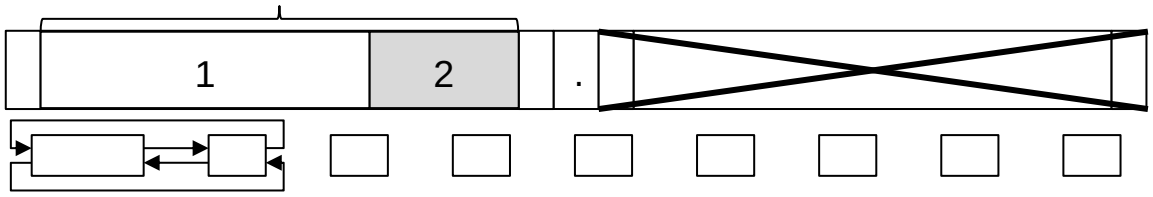
- I-Node – It uniquely identifies a file in the computer. An I-nodes is made of two parts:
    - Major number – Determines the devices
    - Minor number – It determines what file it refers to inside the device.
- Open Mode – How the file was opened: Read Only, Read Write, Append
- Offset – The next read or write operation will start at this offset in the file. Each read/write operation increases the offset by the number of bytes read/written.
- Reference Count – It is increased by the number of file descriptors that point to this Open File Object. When file is closed, the reference count is decremented. When the reference count reaches 0 the Open File Object is removed. The reference count is initially 1 and it is increased after fork() or calls like dup and dup2.

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.

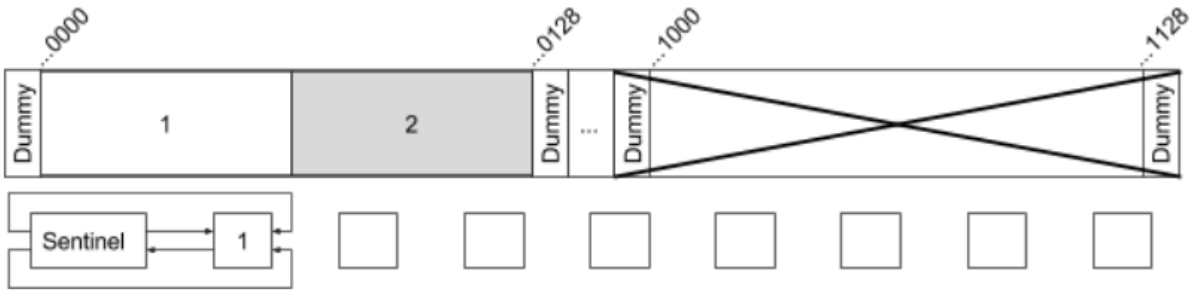
Example Diagram:



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

Question 1

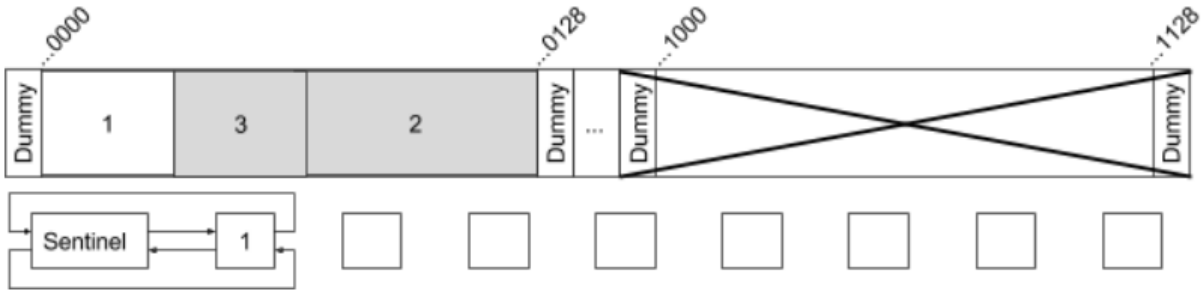
This is the heap before the operation:



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.



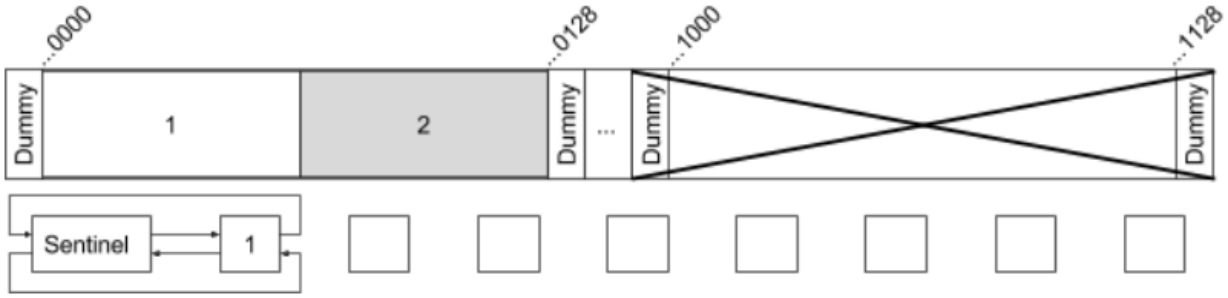
Block	Offset	Size	Left Size	Allocated
1	...000	32	16	0
2	...0064	64	32	1
3	...0032	32	32	1
	...			

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

It returns 10032 + sizeof(BoundaryTag)

Question 2

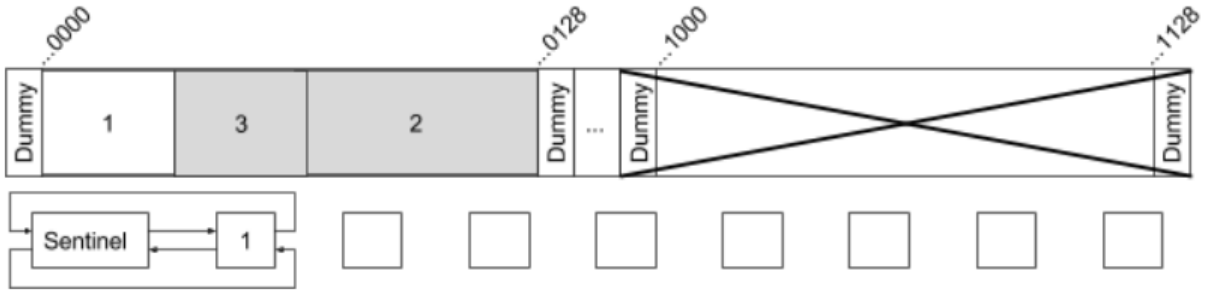
This is the heap before the operation:



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.

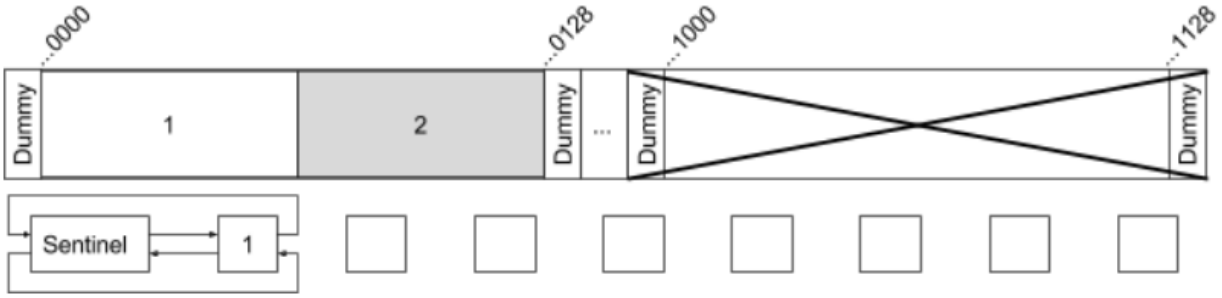


Block	Offset	Size	Left Size	Allocated
1	...0000	32	16	0
2	...0064	64	32	1
3	...0032	32	32	1
	...			

What is the value returned by the operation above assuming the beginning of the allocable memory in the first arena is address 10000?
It returns 10032 + sizeof(BoundaryTag)

Question 3

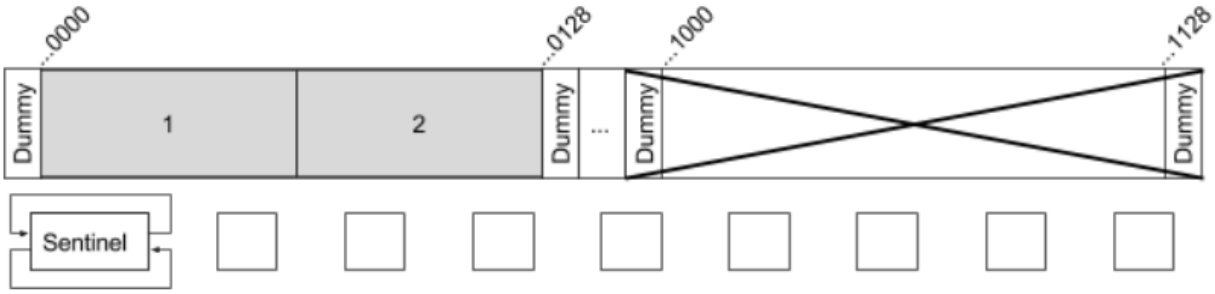
This is the heap before the operation:



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.

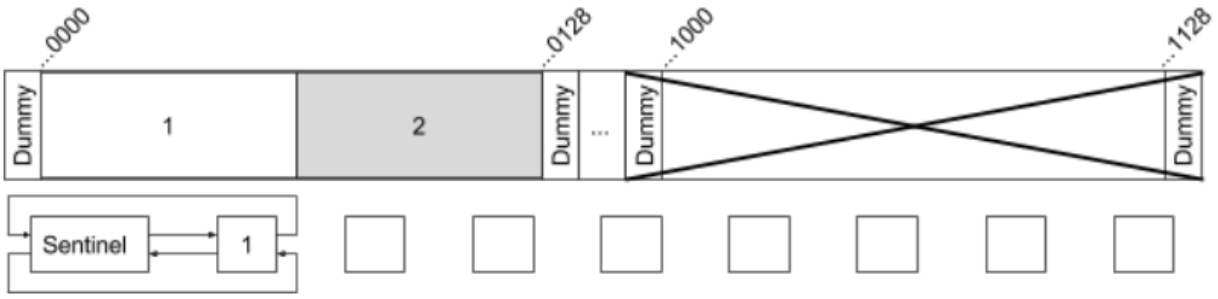


Block	Offset	Size	Left Size	Allocated
1	...0000	64	16	1
2	...0064	64	64	1
	...			
	...			

What is the value returned by the operation above assuming the beginning of the allocable memory in the first arena is address 10000?
It returns 10000 + sizeof(BoundaryTag)

Question 4

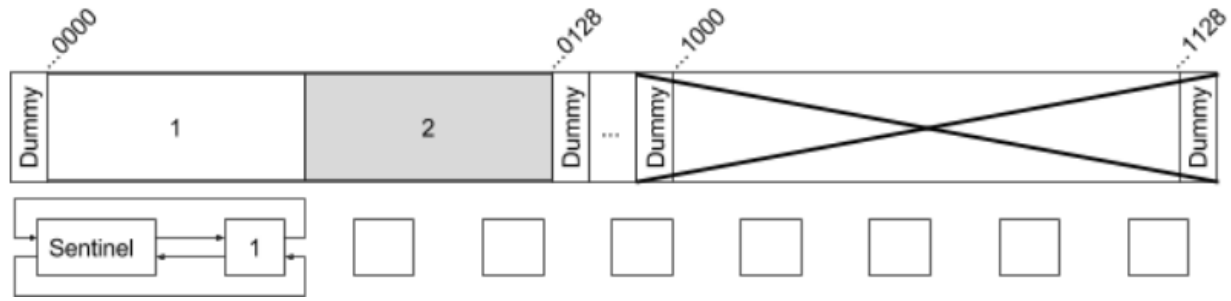
This is the heap before the operation:



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.

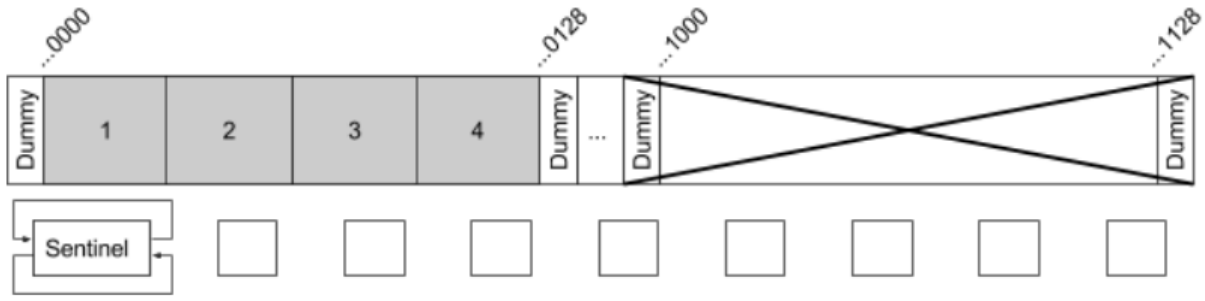


Block	Offset	Size	Left Size	Allocated
1	...0000	64	16	0
2	...0064	64	64	1
	...			
	...			

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).
It would return NULL as we have an error.

Question 5

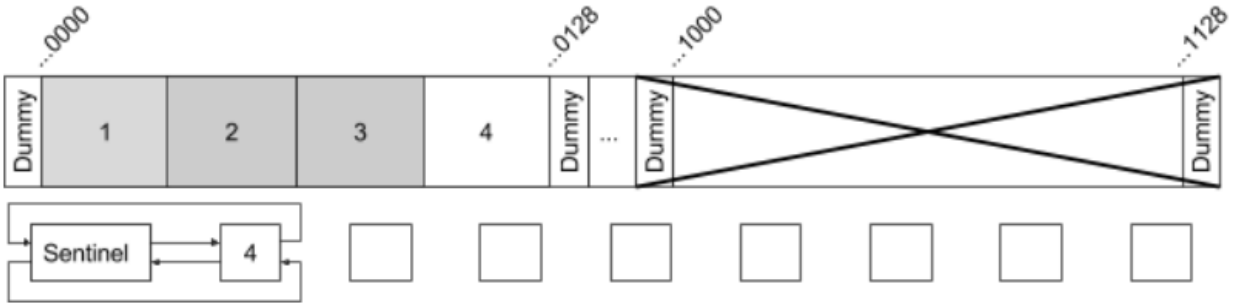
This is the heap before the operation:



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
```

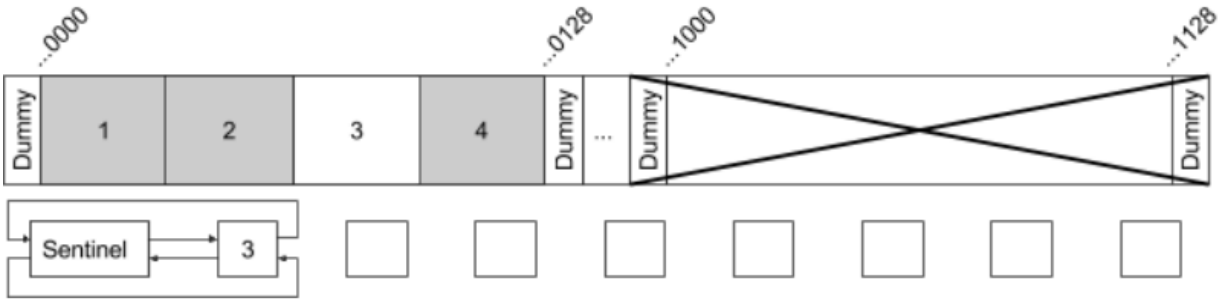
Draw the heap after the operation above and fill up the table.



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

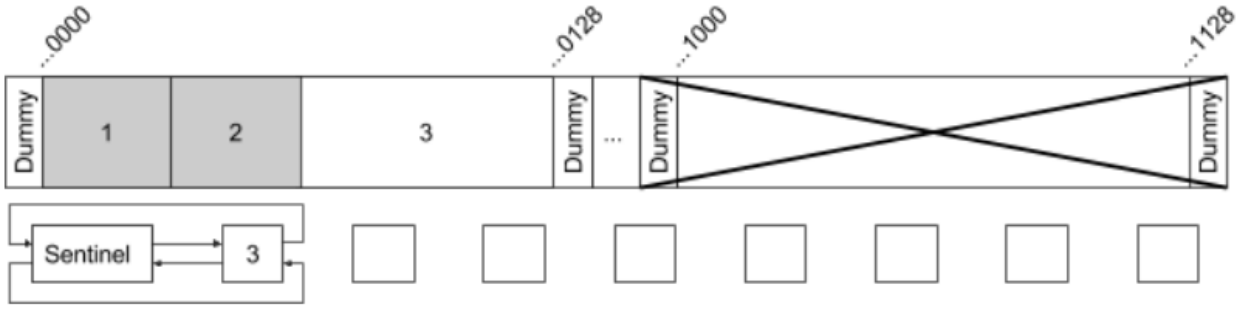
This is the heap before the operation:



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
```

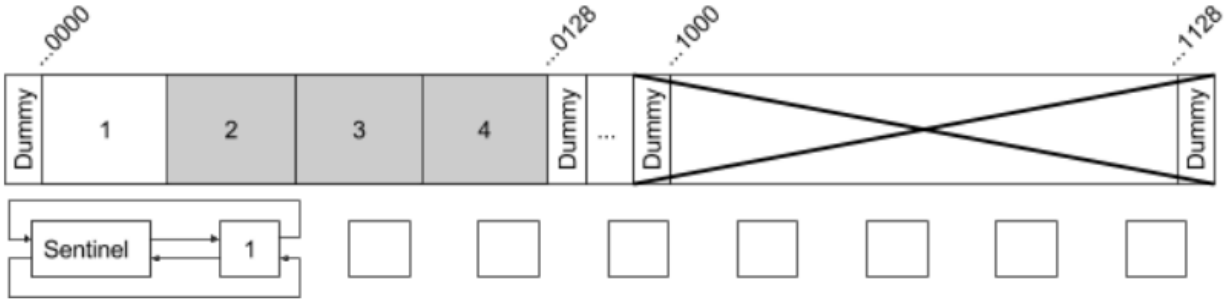
Draw the heap after the operation above and fill up the table.



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

Question 7

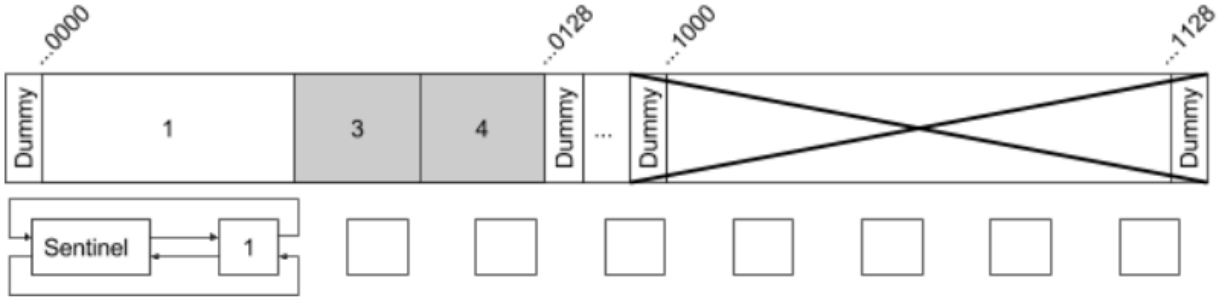
This is the heap before the operation:



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
```

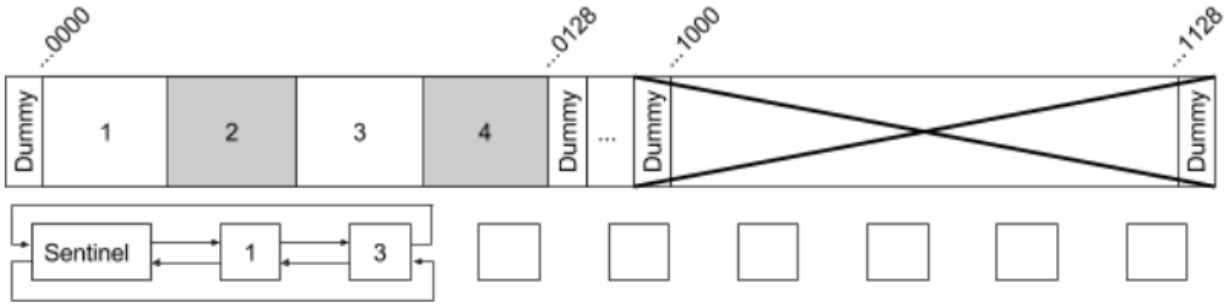
Draw the heap after the operation above and fill up the table.



Block	Offset	Size	Left Size	Allocated
1	...0000	64	16	0
3	...0064	32	64	1
4	...0096	32	32	1
	...			

Question 8

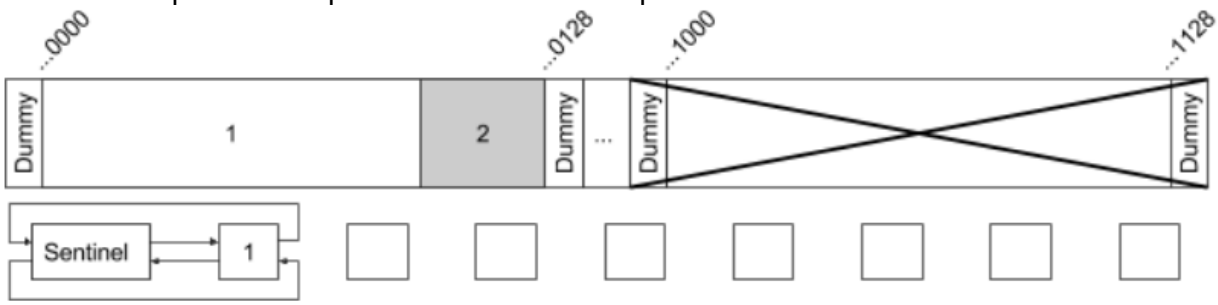
This is the heap before the operation:



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>

du -h * | sort -hr | head -n $limit
```

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

#Usage: mem_proc <pid>

```
while :
do
      cat /proc/<pid>/mem
      sleep 5
done
```

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 -q -E '^(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) The current behavior of this program is the behavior described above. argv[0] is the name of the executable. But it is also the first argument so if we call exec ourselves we can specify the binary and then just iterate over the args.
b) Not applicable
c) Not applicable

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) It is not.
b) The parent will print before the child due to the child process needing to sleep
c) The return value of the fork call should be set to a value and the parent should call waitpid to wait for the child process and then it can print.

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) It is not
b) The parent process writes 100000 bytes to the pipe. However we know that linux pipes have a fixed buffer size which can lead to the parent blocking and waiting for another process to read from the pipe before it can continue to write.
c) The writes should occur after the child process has started reading. Thus while the parent is writing the bytes in, we have the child reading them out.

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) {
  if(numberOfSimpleCommands == 0) {
      prompt();
      return;
  }

int tmpin=dup(0);
  int tmpout=dup(1);
  int tmperr=dup(2);

  int fdin;
  int fdout;
  int fderr;


          if (_inFileName) {
                  fdin = open(_inFileName->c_str(), O_RDONLY);
          }
          else {
                  fdin = dup(tmpin);
          }
  int ret;
   for ( int i = 0; i < command->numberOfSimpleCommand; i++) {
dup2(fdin,0);
    close(fdin);
```

```
    if (i ==  _simpleCommandsArray.size() - 1) {
      if (_outFileName) {
        if (_append) {
          fdout = open(_outFileName->c_str(), O_WRONLY | O_CREAT | O_APPEND,
0666);
        }
        else {
          fdout = open(_outFileName->c_str(), O_WRONLY | O_CREAT | O_TRUNC, 0666);
        }
      }
      else {
           fdout = dup(tmpout);
      }
      if (_errFileName) {
       if (_append) {
          fderr = open(_errFileName->c_str(), O_WRONLY | O_CREAT | O_APPEND, 0666);
        }
        else {
          fderr = open(_errFileName->c_str(), O_WRONLY | O_CREAT | O_TRUNC, 0666);
        }
      }
      else {
        fderr = dup(tmperr);
      }
    }
    else {
      int fdpipe[2];
                    pipe(fdpipe);
                    fdout=fdpipe[1];
                    fdin=fdpipe[0];
    }
    dup2(fdout,1);
    close(fdout);
    dup2(fderr,2);
    close(fderr);
    ret = fork();
    if (ret == 0) {
      std::vector<char*> stl;
      stl.reserve(_simpleCommandsArray[i]->_argumentsArray.size());
      for (unsigned j = 0; j < _simpleCommandsArray[i]->_argumentsArray.size(); j++) {
        const char * arg = _simpleCommandsArray[i]->_argumentsArray[j]->c_str();
        stl.push_back(const_cast<char *>(arg));
      }
```

```
      stl.push_back(NULL);
      char* const* rem = stl.data();
      execvp( _simpleCommandsArray[i]->_argumentsArray[0]->c_str(), rem);
      perror("execvp");

      _exit(1);
    }
    else if (ret < 0) {
      // There was an error in fork
      perror("fork");
      _exit(2);
    }
  }

  dup2(tmpin, 0);
  dup2(tmpout, 1);
  dup2(tmperr, 2);
  close(tmpin);
  close(tmpout);
  close(tmperr);
    // Clear to prepare for next command
  clear();

  // Print new prompt
  if (isatty(0)) {
    Shell::prompt();
  }

}
```

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

```
std::vector<std::string> allpaths;
bool compareFunction (std::string *a, std::string *b) {
  return *a<*b;
}

bool comparePrefixes (std::string a, std::string b) {
  return a<b;
}
…
IN INSERT ARGUMENT:
else if (strchr((*argument).c_str(), '*') || strchr((*argument).c_str(), '?')) {
    std::string arg = *argument;
    wild_card(arg);
    if (args == _argumentsArray.size()) {
      _argumentsArray.push_back(argument);
    }
    return;
  }
…
```

void expandWildcardsIfNecessary(char * arg) {

```
std::string original = arg;
  for (unsigned int i = 0; i < arg.size(); i++) {
    if (arg[i] == '*') {
      arg.insert(i, 1, '.');
      i++;
    }
    if (arg[i] == '.') {
      arg.insert(i, 1, '\\');
      i++;
    }
    if (arg[i] == '?') {
      arg[i] = '.';
    }
  }
  arg.insert(0, 1, '^');
  arg.push_back('$');
  regex_t reg;
  std::vector<std::string *> args;
```

```
int expbuf = regcomp( &reg, arg.c_str(), REG_EXTENDED | REG_NOSUB);
if (expbuf < 0) {
  perror("regcomp");
  return;
}

if (original[0] == '/') {
  unsigned int i = 1;
  int index = 0;
  while (original[i] != '*') {
    if (original[i] == '/') {
      index = i;
    }
    i++;
  }
  std::string prefix = original.substr(0, index);
  i = index + 1;
  // std::cout << "first prefix is : " << prefix << '\n';
  std::vector<std::string> prefixes;
  prefixes.push_back(prefix);

  std::string suff = "";
  std::vector<std::string> suffixes;
  while (i < original.size()) {
    if (original[i] == '/') {
      // std::cout << "suffix is: " << suff << '\n';
      suffixes.push_back(suff);
      suff = "";
      i++;
      continue;
    }
    suff += original[i];
    i++;
  }
  suffixes.push_back(suff);
  // std::cout << "suffix is: " << suff << '\n';
  for (unsigned int i = 0; i < suffixes.size(); i++) {
    prefixes = wild_Expansions(prefixes, suffixes[i]);
  }
  std::sort(prefixes.begin(), prefixes.end(), comparePrefixes);
  for (unsigned int i = 0; i < prefixes.size(); i++) {
    std::string * addin = new std::string(prefixes[i]);
    _argumentsArray.push_back(addin);
```

```
    }
    return;
  }
  DIR *dir = opendir(".");
  if (dir == NULL) {
    perror("opendir");
    return;
  }
  struct dirent * ent;
  regmatch_t match;
  while ((ent = readdir(dir)) != NULL) {
          if (regexec(&reg, ent->d_name, 1, &match, 0) == 0) {
      if (original[0] != '.' && std::string(ent->d_name)[0] != '.') {
        // std::cout << "matches is :" << std::string(ent->d_name) << '\n';
        std::string * temp = new std::string(ent->d_name);
        args.push_back(temp);
      }
      else if (original[0] == '.') {
        std::string * temp = new std::string(ent->d_name);
        args.push_back(temp);
      }
    }
  }

  std::sort(args.begin(), args.end(), compareFunction);
  for (unsigned int i = 0; i < args.size(); i++) {
    _argumentsArray.push_back(args[i]);
  }

  regfree(&reg);
  closedir(dir);
  return;
}
std::vector<std::string> SimpleCommand::wild_Expansions(std::vector<std::string>
prefixes, std::string suffix) {
  std::vector<std::string> new_prefixes;
  for (unsigned int i = 0; i < suffix.size(); i++) {
    if (suffix[i] == '*') {
      suffix.insert(i, 1, '.');
      i++;
    }
    if (suffix[i] == '.') {
```

```
    suffix.insert(i, 1, '\\');
    i++;
  }
  if (suffix[i] == '?') {
    suffix[i] = '.';
  }
}
suffix.insert(0, 1, '^');
suffix.push_back('$');
for (unsigned int i = 0; i < prefixes.size(); i++) {
  regex_t reg;
  int expbuf = regcomp( &reg, suffix.c_str(), REG_EXTENDED | REG_NOSUB);
  if (expbuf < 0) {
    prefixes.erase(prefixes.begin() + i);
    i--;
    continue;
  }
  // std::cout << "path : " << prefixes[i] << '\n';
  if (prefixes[i] == "") {
    prefixes[i] = "/";
  }
  // std::cout << "reg : " << suffix << '\n';
  DIR *dir = opendir(prefixes[i].c_str());
  if (dir == NULL) {
    prefixes.erase(prefixes.begin() + i);
    i--;
    continue;
  }
  struct dirent * ent;
  regmatch_t match;
  while ((ent = readdir(dir)) != NULL) {
    if (regexec(&reg, ent->d_name, 1, &match, 0) == 0) {
      // std::cout << "matches is :" << std::string(ent->d_name) << '\n';
      if (suffix[0] != '.' && std::string(ent->d_name)[0] != '.') {
        // std::cout << "matches is :" << std::string(ent->d_name) << '\n';
        std::string t = prefixes[i];
        if (prefixes[i] != "/") {
          t += "/";
        }
        t += std::string(ent->d_name);
```

```
        new_prefixes.push_back(t);
      }
      else if (suffix[0] == '.') {
        std::string t = prefixes[i];
        if (prefixes[i] != "/") {
          t += "/";
        }
        t += std::string(ent->d_name);
        new_prefixes.push_back(t);
      }
    }
  }
  regfree(&reg);
  closedir(dir);
 }
 return new_prefixes;
}
```

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 < 3) {
      fprinft(stderr “usage: ls grep arg1 | sort < arg2 >> arg3”);
      exit(1);
   }
   int tempin = dup(0);
   int tempout = dup(1);
   int fdin;
   int fdout;
   int ret;
   int fdpipe[2];

   if(pipe(fdpipe) == -1) {
      perror(“pipe”);
      exit(1);
   }
```

```
  fdin = open(argv[2], O_RDONLY);
  if(fdin < 0) {
     perror("open inputfile);
     exit(1);
  }
  dup2(fdin, 0);
  close(fdin);

  dup2(fdpipe [1], STDOUT_FILENO);
  close(fdpipe [1]);

  ret = fork();
  if(ret < 0) {
     perror("fork");
     exit(1);
  }
  if(ret == 0){
     close(fdpipe [0];
     char *args = args[3];
     args[0] = "grep";
     args[1] = argv[1];
     args[2] = NULL;

     execvp( args[0], args);
     perror("execvp")
     _exit(1);
  }
  outfd = open(args[3], O_WRONLY | O_APPEND | O_CREAT, 0664);
  if(outfd < 0) {
    perror("open outfile");
    exit(1);
  }
  dup2(outfd, 1);
  close(outfd);

  dup2(fdpipe [0], 0);
  close(fdpipe [0]);
ret = fork();
  if(ret < 0) {
     perror("fork");
     exit(1);
  }
  if(ret == 0){
     close(fdpipe [1];
     char *args = args[2];
     args[0] = "sort";
     args[1] = NULL;

     execvp( args[0], args);
     perror("execvp")
```

```
        _exit(1);
    }

    dup2(tempin, 0);
    dup2(tempout, 1);
    close(tempin);
    close(tempout);

    waitpid(ret, 0, 0);
    return 0;
}
```