Midterm Exam Fall 2017 SOLUTION

1. Answer True/False (T/F) (1 pt. each)

_**T**_ data is a memory section
_**T**_ elf is a type of an executable file format
_**F**_ The command "chmod 110 file" makes a file executable by others.
_**F**_ printf() is a system call.
_**F**_ The interrupt vector can be modified in user mode .

2. (5 pts.) Using the following program fill up the table with the corresponding memory sections where the indicated addresses are located

```
int a = 5;
int b[20];
char *q;
int main() {
        int x;
        int *p =(int*)
                malloc(sizeof(int));
       q = “Hello”;
}
```

Address	Section
&a	**data**
&b[0]	**bss**
&x	**stack**
p	**heap**
q	**text / rodata**
&main	**text**
&p	**stack**
b	**bss**
&p[0]	**heap**
&q	**bss**

.5 pts per correct answer

3. (5 pts.) Enumerate the 5 memory allocation error and give a code example of each of them.

a)	**Memory Leak**	**malloc(7);**

b)	**Premature Free**	**int *a = malloc(sizeof(int));**
		free(a);
		***a = 10;**

c)	**Memory Smashing**	**char *a = malloc(1);** **a[1337] = ‘x’;**

d)	Double Free	int *a = malloc(sizeof(int));
		free(a);
		free(a);

e)	Wild Free	free(7);

.5 pts per error and per example

4. (5 pts.) Describe all the steps that a program and the OS will do to execute the following system call

int fd = write(int fd, char * buffer, int length)

1. **User program calls write to write to disk**
2. **Write wrapper in libc generates a software interrupt for the syscall**
3. **OS checks the arguments**
   a. **fd is a valid file descriptor, and opened in write mode**
   b. **[buff, buff + length - 1] is a valid memory range**
4. **OS tells the hard drive to write buffer to disk to fd**
5. **OS puts the current process in the wait state until the disk operation is complete. During this the OS context switches to another process**
6. **Disk completes the write operation, triggering an interrupt**
7. **Interrupt handler puts the original process into the ready state so it can be scheduled again**

5. (5 pts)

a) Explain why a bug such as the following one is difficult to debug:

```
int * array = new int[40];
array[7]=9;
….
delete [] array;
array[2]=5;
```

(3 pts)This bug is a premature free. An error won't be thrown until that value is attempted to be used, potentially much later, making it hard to pinpoint the bug.

b) What can you as a programmer do to make it easy to detect this kind of problems.

(2 pts) Always set a pointer to NULL after freeing it. An error will be thrown exactly when the pointer is improperly dereferenced after freeing.

6. (5 pts) Explain how Multilevel Feedback-Queue Scheduling works.

- **MLFQ is a scheduler that contains multiple queues with different priorities.**
- **The scheduler first schedules ready processes with the highest priority.**
- **Round robin is used within each queue.**
- **Processes are artificially aged the longer they spend in a queue so that processes of low priority get a chance to run. After it is scheduled, its priority is reset to its initial priority.**
- **Interactive processes' priorities are increased if they don't use all their quantum**
- **Processes that have smaller burst times get higher priorities**

7. (5 pts) Describe what each of the following commands and arguments do:

Command	Description

ls -al	`ls : list the contents of the directory (defaults to current directory)`
	`-a : do not ignore entries starting with .`
	`-l : use a long listing format`

cp -R dir1 dir2	`cp : copy dir1 to dir2`
	`-R : copy directories recursively`

rm -rf dir1	`rm : remove dir1`
	`-r : remove the contents of dir1 recursively`
	`-f : ignore nonexistent files and arguments, never prompt`

mkdir -p dir1/dir2	`mkdir : create directory dir2, if it doesn’t exist`
	`-p : if dir1 doesn’t exist, make it as well (no errors)`

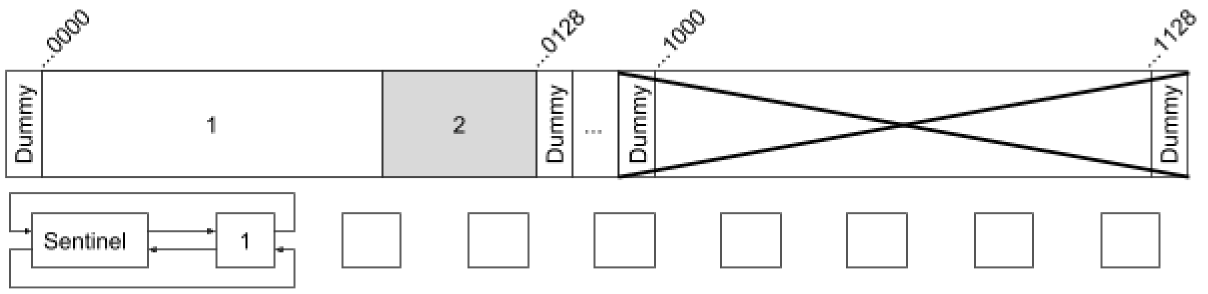
tail -f a.log	`tail : print the last 10 lines of a.log to stdout`
	`-f : output appended data as the file grows`

which gcc	`which : returns the pathname gcc which would be executed in the current environment`

find .	`find : search for files in a directory hierarchy`
	`find . specifically lists every file inside the current directory`

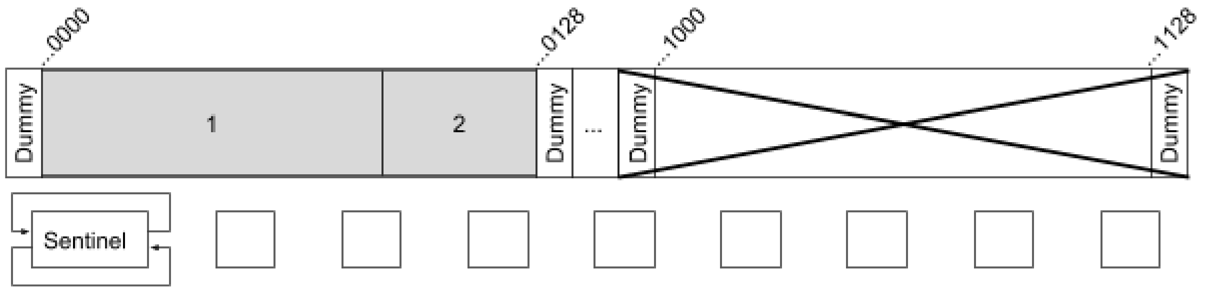
ps -e	`ps : display information about currently active processes`
	`-e : select all processes on the system`

Question 8 (5 points)



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

`malloc(60);`

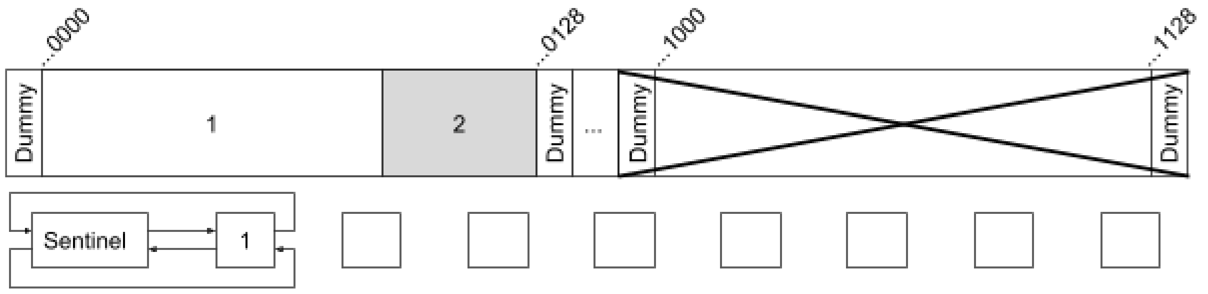


Block	Offset	Size	Left Size	Allocated
1	**...0000**	**88**	**16**	**1**
2	**...0088**	**40**	**88**	**1**

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

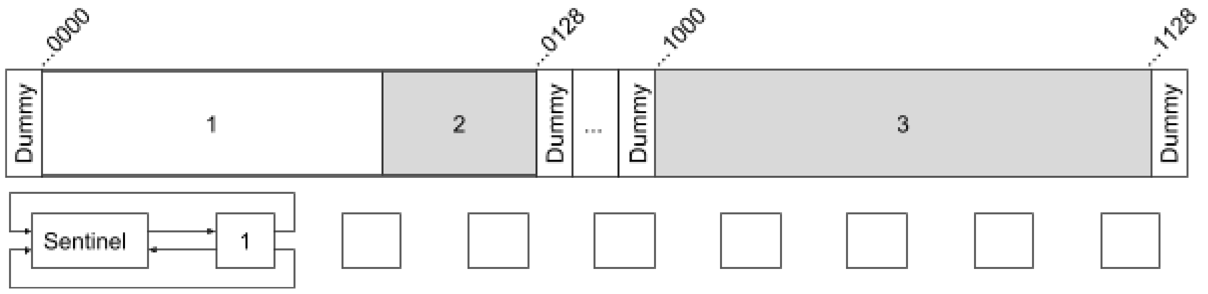
..10000 + sizeof(BoundaryTag) OR
..10016 OR
...0000 + sizeof(BoundaryTag) OR
...0016

Question 9 (5 points)



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

```
malloc(112);
```

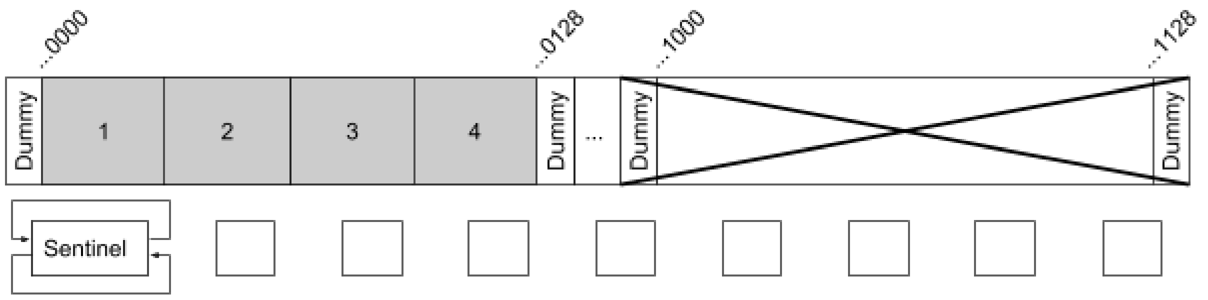


Block	Offset	Size	Left Size	Allocated
1	**...0000**	**88**	**16**	**0**
2	**...0088**	**40**	**88**	**1**
3	**...1000**	**128**	**16**	**1**

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

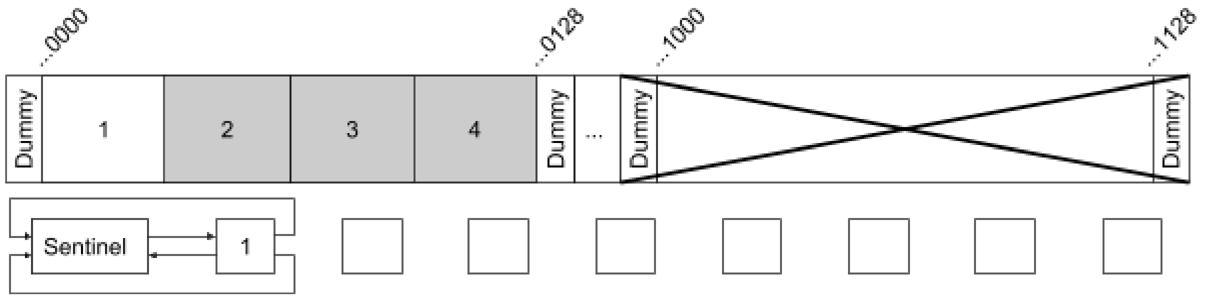
..11000 + sizeof(BoundaryTag) OR
..11016 OR
...1000 + sizeof(BoundaryTag) OR
...1016

Question 10 (5 points)



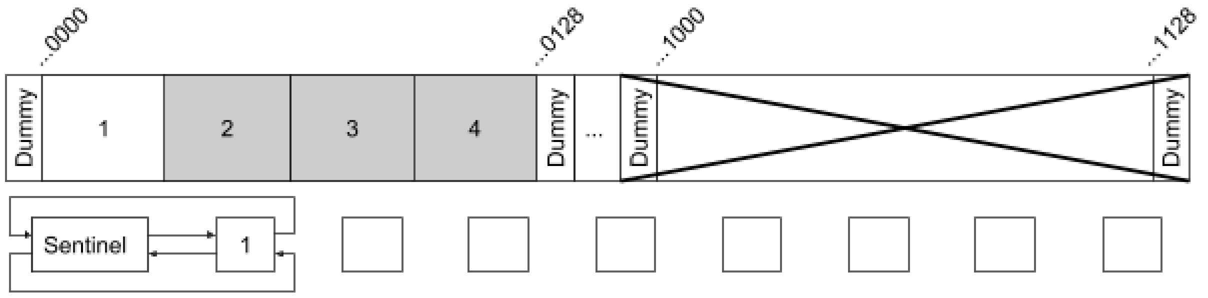
Block	Offset	Size	Left Size	Allocated
1	...0000	32	16	1
2	...0032	32	32	1
3	...0064	32	32	1
4	...0096	32	32	1

```
free(...0000 + sizeof(BoundaryTag)) // free block 1
```



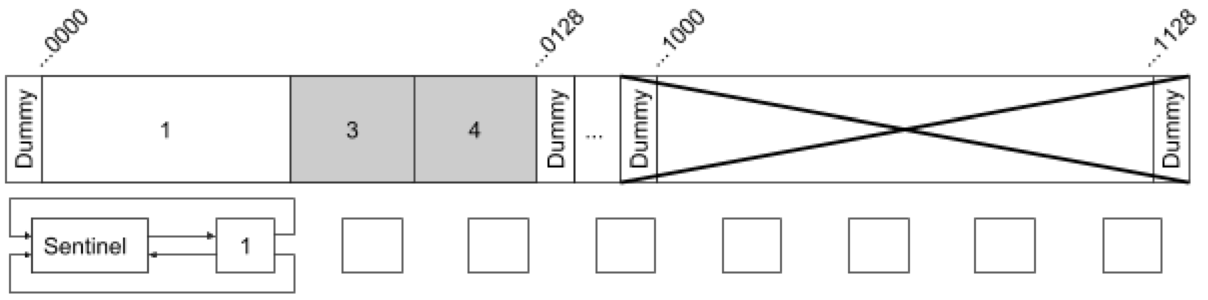
Block	Offset	Size	Left Size	Allocated
1	**...0000**	**32**	**16**	**0**
2	**...0032**	**32**	**32**	**1**
3	**...0064**	**32**	**32**	**1**
4	**...0096**	**32**	**32**	**1**

Question 11 (5 points)



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 + sizeof(BoundaryTag)) // free block 2
```



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

Shell Scripting

12 (5 points). Write a shell command to read the current user's lab2 grade from a file organized as follows:
/homes/cs252/grades.txt

user	lab1	lab2	lab3	midterm	final
hays1	75	100	90	85	75
...	...	...	...	...	...

```
function get_grade() {

  cat /homes/cs252/grades.txt | grep $USER | awk '{print $3}'

}
```

13 (5 points). Given two versions of a file, write a command to send an email containing the differences to a specified user.

```
# Usage: changelog <file1> <file2> <send_to>
function changelog() {

  diff $1 $2 | mail -s "differences" $3

}
```

14 (10 points). Given the name of a program, kill an instance of the program the current user has running

```
# Usage: killByName <name>
function killByName() {

    kill $(ps -u $USER | grep $1 | awk '{print $3}') OR
    pkill $1

}
```

15 (5 points). A program that prints "Hello?" followed by "Can you hear me now?" to standard error, each from a different process.

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?

```
// hello.c
#include <stdio.h>
#include <unistd.h>
#define MSG_LEN 128
int main(int argc, char ** argv) {
   char msg[MSG_LEN];

   int * pipeFd = malloc(2 * 4);
   pipe(pipeFd);
   dup2(pipeFd[0], 0);
   dup2(pipeFd[1], 1);

   if (fork() == 0) {
       puts("Hello?");
        fflush(stdout);
       fgets(msg, MSG_LEN, stdin);
   } else {
       fgets(msg, MSG_LEN, stdin);
       puts("Can you hear me now?");
        fflush(stdout);
   }
   fprintf(stderr, "%s\n", msg);
}
```

Answer:

a) **No**
b) **The pipe is not being closed properly**
OR
There’s a race condition between the two processes
c) **To fix this we can add another pipe to allow communication in the reverse direction and have specific dup code for each process.**

16 (10 points). A program that prints “M|Ms are the best”.

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?

```
// candy.c
#include <stdio.h>
#include <unistd.h>
int main(int argc, char ** argv) {
   const char * str = "M|Ms";
   if (fork() != 0) {
        str = "Skittlz";
   } else {
       sleep(1);
       printf("%s are the best\n", str);
   }
}
```

ANSWER:

a) **Yes: It behaves as described all of the time as the processes have their own memory and thus the mutation of str in the child does not affect the parent.**
b) **N/a**
c) **N/a**

Source: man 2 fork : DESCRIPTION:
"The child process and the parent process run in separate memory spaces. At the time of fork() both memory spaces have the same content. Memory writes, file mappings (mmap(2)), and unmappings (munmap(2)) performed by one of the processes do not affect the other."

Common Mistakes:
- "Assigning to a const char * is invalid"
  - The constness is applied to the char, not the char *. Therefore the reassignment of str in line 7 is completely valid.
- "There's a race condition"
  - The two processes have separate memory, so they don't affect each other, thus there is no race condition
  - In addition, many people also said that removing/adding sleeps would remove a race condition. That's not how race conditions work!!
- "The parent process always runs first"
  - This is not how forking works. Neither process is guaranteed to run first
- "Flip the if condition / Swap the contents of the if-else blocks"
  - This won't change the output of the process

17 (10 points). A program that prints "Something!" 100 times followed by "Empty!"
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?

```
// flow.c
#include<stdio.h>
#include<unistd.h>
#define BUF_SIZE 128
int main(int argc, char ** argv) {
   char buf[BUF_SIZE];
   int * pipeFd = malloc(2 * 4);
   pipe(pipeFd);
   if (!fork()) {
       dup2(pipeFd[1], 1);
       for (int i = 0; i < 100; i++) {
           printf("Something!\n");
       }
   } else {
       dup2(pipeFd[0], 0);
       while(fgets(buf, BUF_SIZE, stdin) != NULL) {
           printf("Pipeline: %s", buf);
       }
       printf("Pipeline: Empty!\n");
   }
}
```

ANSWER:
a) **No**
b) **The program prints Something! 100 times, but hangs due to the open end of the pipe in the sink process, which prevents the EOF from being sent to the fgets call in the sink process. "Nothing!" is never printed.**
c) **By properly closing the open end of the pipe in the sink process, the desired behavior can be achieved**

Source: man 7 pipe : I/O on pipes and FIFOs:
If a process attempts to read from an empty pipe, then read(2) will block until data is available.

...
If all file descriptors referring to the write end of a pipe have been closed, then an attempt
to read(2) from the pipe will see end-of-file (read(2) will return 0).
…
An application that uses pipe(2) and fork(2) should use suitable close(2) calls to close unnecessary duplicate
 file descriptors; this ensures that end-of-file and SIGPIPE/EPIPE are delivered when appropriate.