CS29206 Systems Programming Laboratory, Spring 2023–2024

Lab Test (Quiz)

08–April–2024 06:15pm–07:45pm Maximum marks: 60

Marks obtained (***To be filled in by the examiners***)

Q1	Q2	Q3	Q4	Q5	Q6	Total

Roll no: ____________________ **Name:** ____

[*Write in the respective spaces provided. Write syntactically correct codes (no credits for pseudocodes).*]

1. Prof. Artim and Prof. Sad propose a new cryptographic cipher called SPLAS. It is based on a math library that consists of three source files `splnumarithmetic.c`, `splnumalgebra.c`, and `splnumalgorithms.c`. They have to be clubbed together in the form of a static library `libsplnum.a`. Based upon this math library, the cipher library `libsplas.so` is to be built from `splas.c`. All the source files of the math library depend on the basic data type `splnum` defined in `splnum.h`. Moreover, every source file has a header file with the same name (but with the extension `.h`). These files are organized as follows (the files are listed alphabetically).

Before make

```
splas/
    include/
        splas.h
        splnum.h
        splnumalgebra.h
        splnumalgorithms.h
        splnumarithmetic.h
    lib/
    src/
        cipher/
            makefile
            splas.c
        makefile
        numlib/
            makefile
            splnumalgebra.c
            splnumalgorithms.c
            splnumarithmetic.c
```

After make

```
splas/
    include/
        splas.h
        splnum.h
        splnumalgebra.h
        splnumalgorithms.h
        splnumarithmetic.h
    lib/
        libsplas.so
        libsplnum.a
    src/
        cipher/
            makefile
            splas.c
            splas.o
        makefile
        numlib/
            makefile
            splnumalgebra.c
            splnumalgebra.o
            splnumalgorithms.c
            splnumalgorithms.o
            splnumarithmetic.c
            splnumarithmetic.o
```

After the user calls `make` at the directory `splas/src/`, the object files and the libraries are created as shown above on the right. Your task in this exercise is to write the three makefiles (in the directories `splas/src/`, `splas/src/cipher/`, and `splas/src/numlib/`). Assume that the C source files use the format `#include <...>` for including all header files. The compilation of the source files and the creation of the libraries are handled by the makefiles in the directories `splas/src/numlib/` and `splas/src/cipher/` only. The makefile in `splas/src/` do recursive makes in the two subdirectories. The creation of `libsplas.so` requires the other library `libsplnum.a`.

Also write clean targets in all the three makefiles. Like above, the cleaning is done only in the subdirectories `splas/src/numlib/` and `splas/src/cipher/`. The makefile in `splas/src/` initiates recursive makes. When the user enters `make clean` in the directory `splas/src/`, all the object files are removed, but the library files created in the directory `splas/lib/` are not deleted.

You do not have to write install and distclean targets. You also do not need to write any `.c` or `.h` file.

(a) Write the makefile in `splas/src/`. **(5)**

Solution

```
all:
        cd numlib; make
        cd cipher; make

clean:
        cd numlib; make clean
        cd cipher; make clean
```

(b) Write the makefile in `splas/src/numlib/`. **(5)**

Solution

```
SHELL := /bin/bash
CC := gcc
CFLAGS := -I../../include -Wall
LIBNAME := ../../lib/libsplnum.a
OBJS := splnumarithmetic.o splnumalgebra.o splnumalgorithms.o

LIBNAME: OBJS
       ar rcs ${LIBNAME} ${OBJS}

OBJS: ../../include/splnum.h
splnumarithmetic.o: ../../include/splnumarithmetic.h
splnumalgebra.o: ../../include/splnumalgebra.h
splnumalgorithms.o: ../../include/splnumalgorithms.h

clean:
        -rm -f ${OBJS}
```

(c) Write the makefile in `splas/src/cipher/`. **(5)**

Solution

```
SHELL := /bin/bash
CC := gcc
LIBNAME := ../../lib/libsplas.so
OBJ := splas.o

LIBNAME: OBJ
        ${CC} -Wall -fPIC -I../../include -L../../lib/ -c -o ${OBJ} splas.c -lsplnum
        ${CC} -shared -o ${LIBNAME} ${OBJ}

OBJ: ../../include/splas.h

clean:
        -rm -f ${OBJ}
```

2. In a company database, the following employee details are stored sequentially in a text file `empl.txt`.

```
Age,First_Name Last_Name,Gender,Mobile_Number,PIN_code
```

The company does not store the information of joining date or year, but the company maintains a steady growth by recruiting 20 employees each year. When a new employee joins, a line storing his/her records in the format mentioned above is appended to the database file. Assume that as of now, none of the employees has left the company for any reason.

For a performance evaluation, the board of directors of the company wishes to call all the female employees of age 55, who joined the company from the 10-th to the 15-th year of the company's recruitment and are from IIT Kharagpur (PIN_code 721302). Use a single line of Unix/Linux commands to print the records of all such employees. Assume that the file `empl.txt` is available in the current directory. Do not use awk. **(5)**

Solution

```
head -300 empl.txt | tail -120 | grep '^55' | grep '721302$' | grep ',F,'
```

3. The Foo Operating System supplies a game **barit**. The game uses a secret array A with 10^6 **int** entries. The array stores exactly 10^6 of the $10^6 + 1$ integers $0, 1, 2, \ldots, 10^6$ without repetitions, and is sorted in the ascending order. That is, exactly one integer s in the range $[0, 10^6]$ is missing in the array. The goal of the player is to guess s. The array A and the secret s are stored in the kernel memory, and are not accessible to the users. Only three system calls are provided as interfaces. The call **initsecret(key)** takes a string **key** as an argument, and creates A and s based upon **key**. Subsequently, the user can call **getelement(i)** that returns $A[i]$ after a delay of one second. Finally, the user can verify the correctness of his/her guess t of s by calling **checksecret(t)**. The OS keeps track of how many times **checksecret(t)** is called against every **key** supplied in the call **initsecret(key)**. If exactly one call of **checksecret(t)** returns *true* within ten minutes of calling **initsecret(key)**, a mail is sent to the FooOS headquarters, and you can claim a reward (a free six-month subscription to the popular game **fooit**). You cannot compile your C codes containing the above system calls. Instead, FooOS supplies a gdb-enabled binary file **barit** along with its source code **barit.c**. The kernel code that implements the above three system calls is not gdb-enabled, so you cannot step into these functions. The code **barit.c** is given below with line numbers.

```
 1:   #include <stdio.h>
 2:   #include <stdlib.h>
 3:
 4:   int main ()
 5:   {
 6:      char key[32];
 7:      int t;
 8:      printf("Enter key: ");
 9:      scanf("%s", key);
10:      initsecret(key);
11:      printf("Enter your guess: ");
12:      scanf("%d", &t);
13:      if (checksecret(t)) printf("You cracked it. Claim your reward.\n");
14:      else printf("Nope. Try with a different key to claim your reward.\n");
15:      exit(0);
16:  }
```

Explain how you can use gdb to claim your reward. Explain, in detail, what gdb commands you use, and when (the sequence of gdb commands that you enter). If you write a script, mention that clearly. Telling only an algorithm or idea or pseudocode will deserve no credits. Notice that the program **barit.c** does not use the call **getelement()**, but you can use gdb to call it as many times as you want—at the cost of one second per call, so in the ten-minute period given to you, you can make at most 600 such calls. **(10)**

Solution Run `barit` under gdb: `gdb ./barit`

Set a breakpoint at Line 11 (after `initsecret()` creates the secret): `b 11`

`run`

Enter your key

gdb stops at Line 11. Define the following command:

```
define guess
   if ((int)getelement(0) == 1)
      print 0
   else
      if ((int)getelement(999999) == 999999)
         print 1000000
      else
         set var $L = 1
         set var $R = 1000000
         while ($L < $R)
            set var $M = (int)(($L + $R) / 2)
            if ((int)getelement($M) == $M)
               set var $L = $M + 1
            else
               set var $R = $M
            end
         end
         print $L
      end
   end
end
```

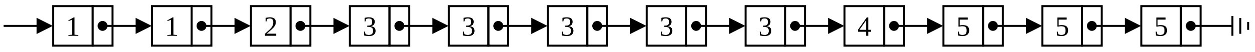
When the gdb prompt returns, enter the command `guess`.

After about 22 seconds, the value of s is returned by the command.

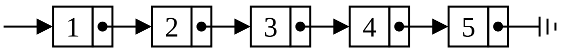
Continue the run, enter this value as t, and get your reward.

4. You create a sorted linked list, where duplicates are allowed. You use a standard insertion function that does not cause memory leaks. Later, you call a function `rmdup()` to remove all the duplicate entries from the sorted list. You do not make any effort to free any node anywhere in your code. Suppose that the list shown at the top of the following figure is created by using sorted insertion (no dummy node is used at the beginning of the list). Then, you call `rmdup()` and `exit()`. At the time of exiting, the list is as shown at the bottom of the figure below.

Initial list



After duplicate removal

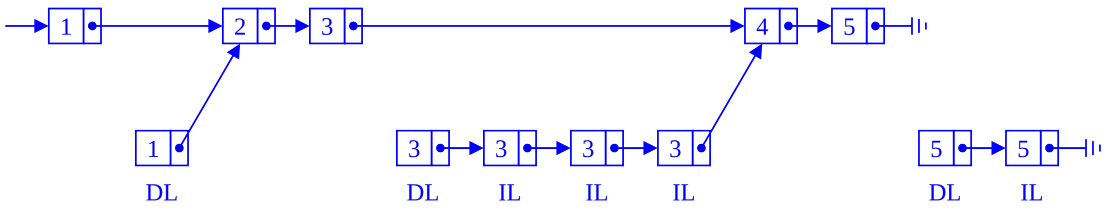


What types of memory leaks and losses will be reported by valgrind on your program? For each type, mention the amount of memory leak/loss (in bytes) and how many blocks are involved in that leak/loss (Example: x bytes still reachable in y blocks). Also, explain which blocks are responsible for these leaks and losses. Assume that each pointer is of size 8 bytes, and each node in the linked list is of size 16 bytes. The code of `rmdup()` that your program uses is given below.

```
void rmdup ( node *L )
{
    if (L == NULL) return;
    while (L -> next != NULL) {
        if (L -> data == L -> next -> data)
            L -> next = L -> next -> next;
        else
            L = L -> next;
    }
}
```

(10)

Solution The blocks after the call of `rmdup()` are shown below. The blocks that are directly (or definitely) lost are marked DL, and the indirectly lost blocks are marked IL. The five nodes after duplicate removal are reachable by the list header pointer.



Since each node is of size 16 bytes, we have the following memory losses. Note that there are no internal pointers.

Still reachable: 80 bytes in 5 blocks
Definitely lost: 48 bytes in 3 blocks
Indirectly lost: 64 bytes in 4 blocks
Possibly lost: 0 bytes in 0 blocks

5. For each of the two programs given below, a contiguous portion of the call graph as reported by gprof is shown. In each case, determine the values entered by the user. Show all your calculations and justifications.

(a) Find x, y, z from the program and the gprof output given below. **(5)**

Program

```
#include <stdio.h>

int sqr (int n) { return n * n; }
int cub (int n) { return sqr(n) * n; }
int sum (int n) { return sqr(n) + cub(n); }

int main ()
{
   int x, y, z, i;
   scanf("%d%d%d", &x, &y, &z);
   for (i=0; i<x; ++i) sqr(i);
   for (i=0; i<y; ++i) cub(i);
   for (i=0; i<z; ++i) sum(i);
}
```

gprof output

```
-----------------------------------------------
                0.00    0.00      6/223         main [9]
                0.00    0.00    217/223         sum [3]
[2]      0.0    0.00    0.00    223         cub [2]
                0.00    0.00    223/564         sqr [1]
-----------------------------------------------
```

Solution $x = 124$, $y = 6$, and $z = 217$.

The above lines correspond to the function **cub** (see the primary line). The first line says that **main** makes 6 calls of **cub**, therefore

$$y = 6.$$

The remaining calls of **cub** are made by **sum** (see the second line). Also, **sum** is called only by **main**. These observations imply that

$$z = 217.$$

In order to determine x, first note that each call of **cub** makes a single call of **sqr**. The remaining $564 - 223 = 341$ calls of **sqr** are made by other functions. Now, 217 calls of **sum** make 217 calls of **sqr**. The remaining $341 - 217 = 124$ calls of **sqr** are made by **main**, indicating that

$$x = 124.$$

(b) Find m, n from the program and the gprof output given below. **(5)**

Program

```
#include <stdio.h>

int Fib ( int n )
{
   if (n < 0) return -1;
   if (n == 0) return 0;
   if (n == 1) return 1;
   return Fib(n-1) + Fib(n-2);
}

int main ()
{
   int m, n, i;

   scanf("%d%d", &m, &n);
   for (i=0; i<m; ++i) Fib(n);
}
```

gprof output

```
index % time    self  children    called     name
                                   518           Fib [1]
                0.00    0.00      37/37          main [7]
[1]      0.0    0.00    0.00      37+518     Fib [1]
                                   518           Fib [1]
-----------------------------------------------
```

Solution $m = 37$, $n = 5$.

Since `main()` calls `Fib()` 37 times (see the second line), we have

$$m = 37.$$

Each call of `Fib()` from `main()` uses a fixed argument n, that is, all such calls make equal numbers of recursive calls. In this example, this number of recursive calls is $518/37 = 14$. We can now determine n by trial and error.

n	Total number of recursive calls
0	0
1	0
2	$2+0+0=2$
3	$2+2+0=4$
4	$2+4+2=8$
5	$2+8+4=14$

It therefore follows that

$$n = 5.$$

6. In the systems programming class, there are 155 students. Each student has a login in a linux server with the roll number of the student, starting with 22CS, as the login ID. For example if your roll number is 22CS30099, then your home directory is `/home/22CS30099`.

Throughout the semester, all the students solve a number of programming assignments, where assignments of each student are stored in separate directories somewhere in the student's home area. Unfortunately, most of the students do not follow any naming conventions, so it is difficult to identify the directory where the makefile assignment is stored. The only information is that the makefile assignment directory contains a `Makefile`, some C/C++ program file(s), and some header file(s) (with the extension `.h`). No directory of any other assignment contains a `Makefile`.

Write a bash shell script that first cleans and then compiles the student's make assignment using his/her `Makefile` and stores the student's roll number in `Success.txt` (roll numbers of the students whose make assignments compile successfully to generates the executable file `a.out`). If either the `Makefile` does not exist or there is a compilation error, the student's roll number will be written in `Failure.txt` (roll numbers of the students whose assignments fail to generate `a.out` using the `Makefile`). `Success.txt` and `Failure.txt` will be created in the current directory. **(10)**

Solution

```
#!/bin/bash
pwd=`pwd`
listfile=$pwd/makelist.txt
success=$pwd/success.txt
fail=$pwd/failure.txt
`rm -f $listfile $success $fail`

function getUser()
{
        awk -v "T=$line" 'BEGIN{split(T,a,"/");print a[3] }'
}

function getFolder()
{
        awk -v "T=$line" 'BEGIN {
                n=split(T,a,"Makefile");
                for (i=1; i<=n; ++i) { print a[i] }
        }'
}

find /home/ | grep "^/home/22CS*" | grep "Makefile$" > $listfile
n=`wc $listfile`
echo "Located $n Makefiles"
echo "=================================================="
if [ -r $listfile ];
then
        IFS=$'\n'
        for line in $(cat "$listfile")
        do
                user=`getUser $line`
                path=`getFolder $line`
                echo "--------------------------------------------------"
                echo "Working on user [$user] at $path"
                echo "--------------------------------------------------"
                if [ -s $line ]; then
                        cd $path;make clean -f $line;make -f $line
                fi
                exefile="$path/a.out"
                if [ -s $exefile ];
                then
                        echo -e $user >> $success
                else
                        echo -e $user >> $fail
                fi
        done
else
        echo "None of the students!!!!"
fi
```

Note: If you use **`find`**, then there are many alternative ways to get the directory name. First, because **`Makefile`** is a string of fixed length, you can pick a substring from a **`find`** entry. Second, you can use the command **`dirname`**.

The command **`ls -R`** does not add the directory name as the prefix of a file, and cannot be straightaway used in this exercise.

You can also use your customized directory browser (like **`dirtree`** in the slides).

ROUGH WORK

ROUGH WORK