CITS 2002 Final Examination
TOP SECRET

1) Consider the function `subString()`, whose C99 prototype follows:

```
bool subString(char *haystack, char *needle);
```

The purpose of the function is to determine if the character string pointed to by `needle` is a substring of the character string pointed to by `haystack`.

The function returns `true` if `needle` is a substring of `haystack` (i.e. `needle` completely appears within `haystack`), and `false` otherwise. You may assume that both `haystack` and `needle` are valid pointers to strings.

Write the function `subString()` in C99.
Do not simply call the similar C99 function `strstr()`.

(10) 2018(1)

2) Consider the function dayOfWeek(), whose C99 prototype follows:

```
int dayOfWeek(int day, int month, int year, int
firstJan);
```

The function takes four integer parameters, each providing information about a date and its year. The goal of the function is to firstly ensure that the provided date is valid and, if so, to determine on which day of the week the date falls.

The first parameter, `day`, provides the day of interest – its values must range from 1 to 31 (inclusive). The second parameter, `month`, provides the month of interest – its values must range from 1 to 12 (inclusive). The third parameter, `year`, provides the year of interest – any integer value of 1970 or greater is permitted.

The final parameter, `firstJan`, indicates the day of the week on which the first of January falls in the provided year. Its permitted values are 0 (representing Sunday), 1 (representing Monday), and so on, up to 6 (representing Saturday).

If all provided parameters are valid, the function will return the day of the week on which the indicated date falls. For example, the call

```
dayOfWeek(13, 11, 2017, 0);
```

will return the integer 1 (representing Monday).
If the provided date is invalid, the function should return the integer -1. Write the function `dayOfWeek()` in C99.

(10) 2017(1)

3) Consider the C99 function `isSubset()`, whose prototype follows:

```
bool isSubset(int set1[], int len1, int set2[], int len2);
```

The function receives two arrays of integers, named `set1` and `set2`, whose lengths are provided by parameters `len1` and `len2`, respectively.
All elements of `set1` are guaranteed to be unique, but are not necessarily sorted. The same is also true of the elements of `set2`.
There is no requirement that `len1` and `len2` have the same value.
The goal of the function is to determine if `set2` is a non-empty subset of `set1`.
Write the function `isSubset()` in C99.
(10) 2018-S(1) 2017-S(1) 2016(1)

4) Consider the function `subString()`, whose C99 prototype follows:

```
bool subString (char *haystack, char *needle);
```

The purpose of the function is to determine if the character string pointed to by `needle` is a substring of the character string pointed to by `haystack`.

The function returns `true` if `needle` is a substring of `haystack` (i.e. `needle` completely appears within `haystack`), and `false` otherwise. You may assume that both `haystack` and `needle` are valid pointers to strings.

Write the function `subString()` in C99.
Do not simply call the similar C99 function `strstr()`.

(10) 2015(1) 2015-S(1)

5) A *concordance* is a list of words appearing in a body of text. The list of words includes only "true" words, consisting entirely of alphabetic characters, after all punctuation and whitespace characters have been discarded.

Write a C99 function, with the prototype:

```
char **concordance(char *textfilename, int *nwords);
```

to build a concordance from a named text file.

The first parameter, `textfilename`, provides the name of the text file containing the words. You may assume that the text file contains only alphabetic, punctuation, and whitespace characters.

The second parameter, `nwords`, provides a pointer to an integer. On successful return from the function, the integer pointed to by `nwords` will contain the number of distinct words found in the file.

On successful return, `concordance` will return a vector of strings, containing the distinct words found in the text file.

If any problems are detected during the execution of `concordance`, the function should return the `NULL` pointer.

(10) 2014(1) 2017(2)

QUESTION 2

6) Consider a 2-dimensional spreadsheet – a rectangular grid of cells, each of which holds an arbitrary string of characters, or an empty string if the value of the cell has not yet been defined. Each cell is identified by 2 non-negative integer values, identifying a specific row and column.

Define a C99 user-defined data type to represent an instance of the 2-dimensional spreadsheet.

Next, define and implement three C99 functions:

- • one to return a pointer to allocated memory suitable for storing and managing the spreadsheet,
- one to swap the contents of two indicated columns in a spreadsheet, and
- one to completely deallocate all memory allocated to a spreadsheet.

(10) 2018(2) 2017-S(2) 2015(2) 2015-S(2)

7a) Different applications for processing large text documents (such as a book's chapter) may sometimes prefer a chapter's text to be available as a single (long) string and, at other times, prefer the chapter to be available as a sequence of sentences. This question requires you to write two functions to perform the

conversion between chapters and sentences.
Write a function in C99, named `makeSentences()`, that converts a chapter into a vector of sentences. The function should match the following prototype:

```
char **makeSentences(char *chapter, int *nSentences);
```

The function receives a single string (`chapter`) that points to the chapter's text. Assume that all sentences end with a full-stop character ('.'), and that `chapter` is guaranteed to end with a complete sentence.
The function produces 2 values. The function returns all sentences as a dynamically allocated array of dynamically allocated strings. The number of sentences is returned via the function's second parameter (`nSentences`). If any call to dynamically allocate memory fails, the function should return NULL.

7b) Write a function in C99, named `makeChapter()`, that converts a vector of sentences into a single string that represents the chapter. The function should match the following prototype:

```
char *makeChapter(char **sentences, int nSentences);
```

The function receives a vector of sentences (as might be produced by `makeSentences()`, above) and converts the sentences into a single dynamically allocated string holding a whole chapter of sentences. When (re)constructing the chapter, ensure that you add a single space after each complete sentence. If any calls to dynamically allocate memory fail, `makeChapter()` should return NULL.
You should assume that `nSentences` is always strictly greater than zero, and that `sentences` contains `nSentences` elements. Assume also that each string in `sentences` represents a complete sentence.
(10) 2018-S(2) 2016(2)

8) 2014(2)

2a) Write a C99 function named `cloneArgv` that duplicates an array of strings, as received by a C program's `main` function. Your function should have the following prototype:

```
char **cloneArgv(int argc, char *argv[]);
```

where `argc` is the number of strings in `argv`. You should assume `argc` is always strictly greater than zero and that `argv` contains at least `argc` elements. You should assume all strings in `argv` are terminated with a NULL-byte.

Your function should make an exact duplicate of the array and its contents, allocating new memory as required.

(5)

2b) Write a C99 function named `freeArgv` that deallocates all memory associated with an array of strings. Your function should have the following prototype:

```
void freeArgv(int argc, char *argv[]);
```

where `argc` is the number of strings in `argv`. You should assume `argc` is always strictly greater than zero and that `argv` contains `argc` elements. You should assume all strings in `argv` are terminated with a NULL-byte.

(5)

QUESTION 3

9) Consider environment variables used in Unix-based operating systems. A frequently seen environment variable is named `PATH`, and is used by command interpreters, or shells, to identify the names of directories to be searched to find executable programs.

For example, a typical value of `PATH` may be: `"/Users/chris/bin:/usr/local/bin:/usr/bin:/bin:."`

which provides a colon-separated list of directories to search for a required program.

Write a C99 function, named `executeUsingPATH()`, which accepts the name of a program to execute and a NULL-pointer terminated vector of arguments to be passed to that program.

The requested program may be specified using just its name, or with either an absolute or a relative pathname.

`int executeUsingPATH(char *programName, char *arguments[]);` Function `executeUsingPATH()` should attempt to execute `programName` from each

directory provided via `PATH`, in order.

If `programName` is found and may be executed (passing to it the indicated program arguments), the function should wait for its execution to terminate, and then return the exit status of the terminated process. If the function cannot find and execute `programName` then it should simply return the integer -1.

Your function should not simply call the similar library function named `execvp()`.

(10) 2018(3) 2015(3) 2015-S(3)

QUESTION 6

10)Consider the following operating system dependent system-calls:

```
int unlink(char *filename);
int rmdir(char *directoryname);
```

The `unlink()` system-call may be used to remove a file from its parent directory, and the `rmdir()` system-call may be used to remove a directory from its parent directory provided that `directoryname` itself contains no other files or directories.

Assume that both system-calls return 0 on success, and 1 on failure.

Using the `unlink()` and `rmdir()` system-calls, write a C99 function named `removeDirectory()` that removes the indicated (potentially non-empty) directory and all of its contents. Your function should have the following prototype:

```
int removeDirectory(char *directoryname);
```

You should assume `directoryname` contains only files and directories.

Your function should attempt to remove as many files and sub-directories as possible, returning 0 on complete success and non-zero otherwise. If `directoryname` could not be opened as a directory, your function should return -1.

(10) 2018(6) 2017-S(6) 2014(6)

11)Consider the function `countEntries()`, whose prototype follows:

```
void countEntries(char *dirName, int *nFiles, int *nDirs);
```

The parameter named `dirName` points to a character string, providing the name of a directory to be recursively explored for files and subdirectories.
You may assume that `dirName`, and its subdirectories, contain only files and directories.
After `countEntries()` has explored everything below the named directory, the parameters `nFiles` and `nDirs` should provide, to the calling function, the number of files and directories found, respectively.
If `countEntries()` finds a directory that cannot be opened, it should report an error and continue its processing (i.e. the function should not exit on such an error).
Write the function `countEntries()` in C99.

(10) 2018-S(6) 2016(6) 2015-S(6)

12)Consider a hierarchical file-system consisting of directories, each of which possibly contains files and other sub-directories. There are no other types of file-system objects in this file- system.

When wishing to perform an incremental backup of all files in and below a named directory of the file-system, we require knowledge of how much space will be required to copy all files.

However, for an incremental backup, we don't wish to copy all files, but only the files that have been modified since a certain time.

Write a C99 function to return the number of bytes contained in all files below a named directory that have been modified since an indicated time.

Your function should have the following prototype:

```
int totalBytes(char *directoryname, time_t since);
```

Any errors resulting from conditions such as being unable to access a sub-directory or a file

should simply be ignored.
If `directoryname` can not be opened as a directory, your function should return -1.

(10) 2017(6) 2015(6)

QUESTION 3 4 5 (Non Code Questions)

13 PROCESS MANAGEMENT

a) Explain clearly the following state transitions for processes and the reasons for the transitions:

1. from Running to Blocked.
2. from Blocked to Blocked-Suspend.
2018(4) 2018-S(3) 2017-S(3) 2016(3)

b) Explain clearly the difference between a *program* and a *process*. What are the important pieces of information that a multi-processing operating system needs to save during process switching? 2018(4) 2018-S(3) 2016(3)

c) With reference to two distinct examples, explain *The Principle of Referential Locality*, and why it is important to operating-system design. 2017-S(3) 2017(4) 2014(4)

14 MEMORY MANAGEMENT

a) Explain clearly the differences between *paging* and *segmentation* of memory. What are the advantages and disadvantages of each scheme? 2017(3) 2015(4)

b) With the aid of diagrams, explain the differences between *internal* memory fragmentation and *external* memory fragmentation. 2017(3) 2018-S(4) 2015(4) 2015-S(4) 2017-S(5) 2016(5)

c) Explain the importance of logical to physical address translation.
Consider a 20-bit computer in which the logical address has a 12-bit offset. Explain clearly, with the aid of a diagram, how the logical to physical address translation is performed for this computer.
What is the maximum number of pages that a process can be allocated in this computer? 2018(5) 2017-S(4) 2016(4)

Consider a 20-bit computer in which the logical address has a 12-bit offset. 2018-S(4)

d) Explain clearly how virtual memory helps in improving processor utilization in a multiprogramming system. 2015-S(4)

e) Explain how an operating system's use of virtual memory can enable the operating system to appear to support the use of more memory than is physically installed in a computer. 2017(4b) 2014(4b)

15 SYSTEM CALL

a) What are *system calls*, and why are they important in the design and implementation of an operating system?
Explain why almost every process needs to execute at least one system call during its execution. 2018(5) 2017-S(4) 2016(4) 2015(5)

b) With reference to a diagram, explain the actions of a Unix-based operating system when a process invokes the *fork()* system-call. 2018-S(5) 2017(5) 2017-S(5) 2016(5) 2015-S(5) 2014(3)

c) Explain how an operating system's use of virtual memory can enable the operating system to appear to support the use of more memory than is physically installed in a computer. 2018-S(5) 2015-S(5)

d) Three common file allocation mechanisms, in order of increasing complexity, are *contiguous*, chaining, and indexed.
Briefly describe one advantage and one disadvantage of each mechanism. 2017(5) 2014(3)

16 DISK AND I/O

a) With reference to at least one example, explain the following disk scheduling policies:
• Shortest Seek Time First (SSTF), SCAN, and C-SCAN. 2015(5a)

5a) Why has one prominent textbook author described I/O management as the "*messiest aspect of operating system design*"?

(5)

5b) Using at least three distinct points, describe the evolution of I/O controllers, and describe how their support for I/O has increased in complexity over time.

(5)

2014(5)