Lab 04: Debugging with Valgrind and simplefs

Preliminaries

In this lab you will first perform exercises in debugging C programs with memory errors, such memory leaks and invalid reads. You will also write a simple file system under a single directory structure that can store arbitrary number of files. **You will likely to complete Task 1 and Task 3 in the lab period**, and you will be able to start Task 4, which will need to be completed **individually** outside of lab.

Lab Learning Goals

In this lab, you will learn the following topics and practice C programming skills.

1. Debugging Memory Leaks with Valgrind
2. Debugging Segfaults with `gdb`
3. Resizing arrays
4. Memory management
5. Implement a simple, single-directory filesystem
6. Timestamps and time formats
7. Compiler preprocessors, `#define` `#ifndef` `#endif`

Lab Setup

Run the following command

```
~aviv/bin/ic221-up
```

Change into the lab directory

```
cd ~/ic221/labs/04
```

All the material you need to complete the lab can be found in the lab directory. All material you will submit, you should place within the lab directory. **Throughout this lab, we refer to the lab directory**, which you should interpret as the above path.

Submission Folder

For this lab, all scripts for submission should be placed in the following folder:

```
~/ic221/labs/04/
```

This directory contains two sub-directories; `examples` and `src`. In the `examples` directory you will find any source code in this lab document. All lab work should be done in the `src` directory.

- **Only source files found in the folder will be graded**.
- **Do not change the names of any source files**

Finally, in the top level of the lab directory, you will find a `README` file. You must complete the `README` file, and include any additional details that might be needed to complete this lab.

Compiling your programs with `clang` and `make`

We have provided you with a Makefile to ease the compilation burden for this lab. To compile a given executable, simply type `make` and then the name of the executable. For example, to compile the test program, `testfs`, type:

```
make testfs
```

Before submitting, you should clean your `src` directory by typing:

```
make clean
```

which will remove any lingering executables and other undesirable files.

Part 1: Debugging Memory Errors with Valgrind

In this lab, you are required to dynamically allocate memory in multiple context, and you are also required to ensure that your program does not have memory leaks or memory violations. Fortunately for you, there exists a wonderful debugging program which can capture both, Valgrind.

Memory Leaks

A memory leak occurs when you have dynamically allocated memory, using `malloc()` or `calloc()` that you do not free properly. As a result, this memory is lost and can *never* be freed, and thus a **memory leak** occurs. It is vital that memory leaks are plugged because they can cause system wide performance issues as one program begins to hog all the memory, affecting access to the resources for other programs.

To understand a memory leak, let's look at perhaps the most offensive memory leaking program ever written:

```
#include <stdio.h>
#include <stdlib.h>

#include <unistd.h>
#include <sys/types.h>

int main(){

  while(1){
    malloc(1024); //memory leak!
    sleep(0.5); //slow down the leak
  }
```

```
}
```

At the `malloc()`, new memory is allocated, but it is never assigned to a pointer. Thus there is no way to keep track of the memory and no way to deallocate it, thus we have a memory leak. This program is even more terrible in that it **loops forever** leaking memory. If run, it will eventually slow down and cripple your computer. **DON'T RUN THIS PROGRAM WITHOUT CAREFUL SUPERVISION**.

Normally, memory leaks are less offensive. Let's look at a more common memory leak.

memleak_example.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char * argv[]){

  int * a = malloc(sizeof(int));

  *a = 10;

  printf("%d\n", *a);

  a = calloc(3, sizeof(int));
  a[0] = 10;
  a[1] = 20;
  a[2] = 30;

  printf("%d %d %d\n", a[0], a[1], a[2]);
}
```

This is a simple program that uses an integer pointer `a` in two allocations. First, it allocates a single integer and assigns the value `10` to the allocated memory. Next, it uses `a` to reference an array of integers of length 3. It prints out the values for both cases. Here is some program output and compilation. (The `-g` is to compile with debugging information, which will become important later.)

```
#> clang memleak_example.c -g -o memleak_example
#> ./memleak_example
10
10 20 30
```

On it's face, there doesn't seem to be anything wrong with this program in terms of its intended output. It compiles without errors, and it runs as intended. Yet, this program is wrong, and there is a memory leak in it.

Upon the second allocation and assignment to `a`, the previous allocation is not freed. The assignment of the second allocation from `calloc()` will overwrite the previous pointer value, which use to reference the initial allocation, the one by `malloc()`. As a result, the previous pointer value and the memory it referenced is lost and cannot be freed; a classic memory leak.

Ok, so we know what a memory leak is and how to recognize one by reading code, but that's hard. Why can't the compiler or something figure this out for us? Turns out that this is **not** something that a compiler can easily check for. The only foolproof way to determine if a program has a memory leak is to run it and see what happens.

The valgrind debugger is exactly the tool designed to that. It will run your program and track the memory allocations and checks at the end if all allocated memory has been freed. If not, some memory was lost, then it will generate a warning. Let's look at the valgrind output of running the above program.

```
#> valgrind ./memleak_example
==30134== Memcheck, a memory error detector
==30134== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==30134== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright in
==30134== Command: ./memleak_example
==30134==
10
10 20 30
==30134==
==30134== HEAP SUMMARY:
==30134==     in use at exit: 16 bytes in 2 blocks
==30134==   total heap usage: 2 allocs, 0 frees, 16 bytes allocated
==30134==
==30134== LEAK SUMMARY:
==30134==    definitely lost: 16 bytes in 2 blocks
==30134==    indirectly lost: 0 bytes in 0 blocks
==30134==      possibly lost: 0 bytes in 0 blocks
==30134==    still reachable: 0 bytes in 0 blocks
==30134==         suppressed: 0 bytes in 0 blocks
==30134== Rerun with --leak-check=full to see details of leaked memory
==30134==
==30134== For counts of detected and suppressed errors, rerun with: -v
==30134== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

Check out the LEAK SUMMARY section, and you find that 16 bytes were "definitely" lost. Let's rerun the valgrind with the --leak-check option set to "full" to see more details, which additonally prints the HEAP SUMMARY

```
#> valgrind --leak-check=full ./memleak_example
(...)

==30148== 4 bytes in 1 blocks are definitely lost in loss record 1 of 2
==30148==    at 0x4C2B6CD: malloc (in /usr/lib/valgrind/vgpreload_memchec
==30148==    by 0x4005F7: main (memleak_example.c:6)
==30148==
==30148== 12 bytes in 1 blocks are definitely lost in loss record 2 of 2
==30148==    at 0x4C29DB4: calloc (in /usr/lib/valgrind/vgpreload_memchec
==30148==    by 0x400636: main (memleak_example.c:12)
```

It lists the two allocatoins. The first call to malloc() allocated 4 bytes, the size of an integer. The second allocation, allocated 3 integers, or 12-bytes, with calloc(). With this information, the programmer can track down the memory leak and fix it, which is exactly what you'll do for this task.

Task 1

Change into the valgrind directory in your lab folder. Compile and execute memleak.c. Verify the output and try and understand the program.

Answer the following questions in your worksheet:

1. Run valgrind on the memleak program, how many bytes does it say have been definitely lost?
2. What line does valgrind indicate the memory leak has occurred?
3. Describe the memory leak.
4. Try and fix the memory leak and verify your fix with valgrind. Describe how you fixed the memory leak.

You will submit your fixed memleak.c program in your submission, and we will verify that you fixed the memory leak.

Memory Violations

Memory leaks are not just the only kind of memory errors that valgrind can detect, it can also detect memory violations. A **memory violation** is when you access memory that you shouldn't or access memory prior to it being initialized.

Let's look at really simple example of this, printing an uninitialized value.

```
int a;
printf("%d\n", a);
```

The problem with this program is clear; we're printing out the value of a without having previously assigned to it. This error can be detected by the compiler with the `-Wall` option:

```
#> clang -Wall memviolation_simple.c
memviolation_simple.c:7:18: warning: variable 'a' is uninitialized when u
  printf("%d\n", a);
```

But other memory violations are harder to recognize particular those involving arrays. Let's look at the program below. You should be able to spot the error.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char * argv[]){

  int i, *a;

  a = calloc(10, sizeof(int));

  for(i=0;i <= 10; i++){
    a[i] = i;
  }
  for(i=0;i <= 10; i++){
```

```
    printf("%d\n", a[i]);
  }

}
```

However, if we were to compile and just run this program, you may not recognize that anything is wrong:

```
#> ./memviolation_array
0
1
2
3
4
5
6
7
8
9
10
```

No errors are reported and the numbers up to 10 are printed, but we know that we are actually writing *out-of-bounds* in our array, and we shouldn't do that! Valgrind, fortunately, can detect such errors:

```
#> valgrind ./memviolation_array
==30588== Memcheck, a memory error detector
==30588== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==30588== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright in
==30588== Command: ./memviolation_array
==30588==
==30588== Invalid write of size 4
==30588==    at 0x4005D8: main (in /home/scs/aviv/git/ic221/current/lab/0
==30588==  Address 0x51f2068 is 0 bytes after a block of size 40 alloc'd
==30588==    at 0x4C29DB4: calloc (in /usr/lib/valgrind/vgpreload_memchec
==30588==    by 0x4005B4: main (in /home/scs/aviv/git/ic221/current/lab/0
==30588==
0
1
2
3
4
```

```
5
6
7
8
9
==30588== Invalid read of size 4
==30588==    at 0x40060F: main (in /home/scs/aviv/git/ic221/current/lab/0
==30588==  Address 0x51f2068 is 0 bytes after a block of size 40 alloc'd
==30588==    at 0x4C29DB4: calloc (in /usr/lib/valgrind/vgpreload_memchec
==30588==    by 0x4005B4: main (in /home/scs/aviv/git/ic221/current/lab/0
==30588==
10
==30588==
==30588== HEAP SUMMARY:
==30588==     in use at exit: 40 bytes in 1 blocks
==30588==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==30588==
==30588== LEAK SUMMARY:
==30588==    definitely lost: 40 bytes in 1 blocks
==30588==    indirectly lost: 0 bytes in 0 blocks
==30588==      possibly lost: 0 bytes in 0 blocks
==30588==    still reachable: 0 bytes in 0 blocks
==30588==         suppressed: 0 bytes in 0 blocks
==30588== Rerun with --leak-check=full to see details of leaked memory
==30588==
==30588== For counts of detected and suppressed errors, rerun with: -v
==30588== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 2 from 2)
```

If you notice in the execution output, there is an "Invalid read of size 4" occurring when `array[10]` is indexed and printed to the screen. This is a rather simple example, but invalid reads and writes and other kinds of memory violations can cause all sorts of problems in your program, and they should be investigated and fixed when possible.

Task 2

Change into the `valgrind` directory in your lab folder. Compile and execute the `memviolation.c` program. Complete the following tasks and answer the questions in your worksheet.

1. Describe the output and exeuction of the program. Does it seem to be consistent?
2. Run the program under valgrind, identify the line of code that is causing

the memory violation and its input.

3. Debug the memory violation and describe the programming bug.
4. Fix the memory violation and verify your fix with valgrind.

Your submission will include the fixed `memviolation.c` program.

Part 2: Debugging SEGFAULT's and Core Dump's with GDB

Among the many errors you will encounter while programming C, perhaps the most daunting is the dreaded `segfault`. A `segfault` occurs when you try and read/write from memory that is off limits, which is a common mistake for all programers. The most frustrating part of a `segfault` is that no detailed error information is provided. The program just halts with the standard, `segfault` message. Frustrating.

There are a couple different strategies for debugging `segfault`'s. A common one strategy is to just place a bunch of print statements in your code and see how far you get before the `segfault`, working backwards to identify the bug and squash it. While this is effective strategy, it can be quite tedious. There is a better way.

The Gnu Debugger, or gdb, is an amazingly powerful to for tracing and analyzing programs. One thing that gdb is really helpful with is *backtracing* a `segfault` which will report the exact line of code that caused the `segfault`. Gdb won't tell you exactly how to fix the `segfault` but it can give you really useful information.

Let's look at a simple example first of a buffer overflow:

```
void foo(){
  char str[10];
  printf("Enter String:\n");
  scanf("%s", str);
}
int main(int argc, char *argv[]){
  foo();
}
```

```
#> clang -g segfault_example.c -o
#> ./segfault_example
Enter String:
theraininspainfallsmainlyinthepla
Segmentation fault (core dumped)
```

We all know what happened here: The string only has enough space to store 10 characters, but we read in a lot more overwriting the return point, thus segfault on return. Let's suppose that this error was nested rather deep in our program, and it wasn't so obvious. Then we can use gdb to figure out where the segfault occurred.

```
#> gdb segfault_example
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.h
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/scs/aviv/git/ic221/current/lab/04/stu/examples
(gdb) r
Starting program: /home/scs/aviv/git/ic221/current/lab/04/stu/examples/se
Enter String:
theraininspainfallsmainlyintheplains

Program received signal SIGSEGV, Segmentation fault.
0x00000000004005d5 in foo () at segfault_example.c:8
8       }
```

First we run the program with gdb at the top (make sure you've compiled the program with the -g flag). Once gdb loads, type r to run the program. At this point the program is running like normal, so we can provide the offending input that causes a segfault. BUT! Notice the output. It gives us a line number where the error occured, line 8, which is **here**, at the return.

That's a simple example, let's look at a more complicated situation where a segfault might occurred, when you dereference NULL. Such an error is actually quite common. Here is a simple program with segfault caused by NULL dereference.

```
#+BEGIN_SRC c
typedef struct{
```

```
#> clang -g segfault_nullreferenc
#> ./segfault_nullreference
```

```
    int left;
    int right;
  }pair_t;

  int main(int argc, char *argv[]){
    int i;
    pair_t ** pairs = calloc(10, sizeof(pair_t *));

    for(i = 0 ; i < 10; i++){
      pairs[i] = malloc(sizeof(pair_t));
      pairs[i]->left = i;
      pairs[i]->right = i+1;
    }

    //...
    free(pairs[2]);
    pairs[2] = NULL;
    //...

    for(i=0 ; i < 10; i++){
      printf("%d: left: %d right: %d\n",
             i, pairs[i]->left, pairs[i]->right);
    }

    return 0;
  }
```

```
0: left: 0 right: 1
1: left: 1 right: 2
Segmentation fault (core dumped)
```

The program allocates an array of pointers to `pair_t`'s, allocates each of the `pair_t`, but then also free's and set's one to `NULL`. Unfortunately, there is no check prior to the `printf()` and in `pairs[i]->left` `NULL` is dereferenced, `segfault`.

Let's look at this output under gdb:

```
#> gdb segfault_nullreference
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.h
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
```

```
Reading symbols from /home/scs/aviv/git/ic221/current/lab/04/stu/examples
(gdb) r
Starting program: /home/scs/aviv/git/ic221/current/lab/04/stu/examples/se
0: left: 0 right: 1
1: left: 1 right: 2

Program received signal SIGSEGV, Segmentation fault.
0x00000000004006f8 in main (argc=1, argv=0x7fffffffeb28) at segfault_null
25          printf("%d: left: %d right: %d\n", i, pairs[i]->left, pairs[i
```

Again, it points right at the line the caused the segfault, but it is important to recongize that knowing where a segfault occurred is only the first step in fixing the problem. You still have trace backwards and debug your program to properly identify the error, but knowing where to start really helps.

Task 3

Change into the gdb directory and read the program segfault. The program tries to amange an array of pair_t pointers. You can add, remove, and delete the entire array, but there are a couple segfaults in the program you must correct. Complete the following tasks and answer the questions in your worksheet.

1. Reviewing the program segfault.c. Describe the expected output.
2. Compile and execute the program (don't forget the -g compilation flag). Use gdb to identify the segfault.
3. Fix the segfault, and continue to the debug the program until the desired output is reached:

```
0: left: 2 right: 10
1: left: 0 right: 3
2: left: 13 right: 7
------------
0: left: 2 right: 10
2: left: 13 right: 7
```

Your submission will include the fixed segfault.c program.

Part 3: Implementing simplefs

Task 4

In this part of the lab you will program a simple filesystem that consists of a single directory containing files. You will be provided a small shell-like program to test your file system, and your filesystem must be able to handle the following operations:

1. mkdir : create the single filesystem directory
2. rmdir : remove the single filesystem directory
3. touch : create a file or update it's timestamp
4. rm : remove a file
5. ls : list all the current files

Your filesystem implementation must manage memory properly — no memory leaks or memory violations — and must be able to handle an arbitrary number of files.

You will complete the functions listed in `filseystem.c` with descriptions found in `filesystem.h`. You do not need to edit any other files. You can test your filesystem with `testfs.c` program, which should catch common mistakes. When complete, you can interact with your filesystem with the `shell` program, source found in `shell.c`. **You do not need to, nor should you, edit shell.c**.

Compilation will be done using `make`. To compile the test program `testfs`, run

```
make testfs
```

To compile the shell enviroment for simplefs

```
make shell
```

To compile everything, type:

```
make
```

The Filesystem Structures

The filesystem consists of two main structures, defined in `filesystem.h`.

```
/**
 * Structure for a file
 **/
typedef struct{

  char * fname; //name of file
  time_t last; //last modified

} file_t;


/**
 * Structure for a directory
 **/
typedef struct{

  file_t ** flist; //list of file pointers
  int nfiles; //number of files
  int size; //size of the flist array


} dir_t;
```

The `file_t` structure represents a file. It has two data members, a pointer to a string and a timestamp `time_t` (explained in more detail in the next sections). The directory structure contains three members, an array of `file_t` pointers which is used to manage all the files, the number of files, and the size of array of `file_t` pointers.

Managing the array of file_t pointers

You should think of a directory as just a manager of an array of pointers to `file_t`'s. Some of the `file_t`'s have been allocated and some have not, the unallocated ones will reference `NULL` while the allocated ones will reference a `file_t` allocated on the heap.

Initially, the size of the array storing the `file_t's` will be set to `INIT_SIZE`, which is a constant for the program.

```
#define INIT_SIZE 5
```

The `#define` for the init size is a **compiler preprocessor** that allows us to define constants. Whenever the compiler sees `INIT_SIZE` it replaces it with the numeral 5.

When a new file is created, with `touch`, it will be allocated and referenced from the array from the first non- `NULL` index. If the array is full, **you must resize the array by allocating a larger array and copying the contents over**. Look into the `realloc()` function to help with this.

When a file is removed, you will deallocate the `file_t` using `free()` and set that spot in the array to `NULL`. That means that the slot in the array can be filled with a new file when the next touch occurs.

Timestamps and Time formats

One the new things in this lab is that you will be using timestamps. A timestamp in Unix is just a number, a `long`, that counts the number of seconds since the epoch, Jan 1st 1970. Your file system, through `touch` and `ls`, will need to handle timestamps.

When you first create a file via `touch`, you'll allocate a new `file_t`, set the name, and the timestamp of the time of creation. To retrieve a timestamp, you using the `time()`.

```
time_t ts = time(NULL); //get the current time
```

On subsequent touches of the file, you will just need to reset the timestamp to the new current time, just like the Unix `touch` command.

When listing all the files, printing the timestamp as a number is not very useful. Clearly, we are not computers, and cannot read timestamps, so we need to generate a more human readable format. The easiest way to do that is the `ctime()` function, which takes a pointer to a timestamp and returns a string to a normal date-like formatted string that you're familiar with from `ls`. For example:

```
time_t ts = time(NULL); //get the current time
printf("%s\n", ctime(&ts));
```

The shell

We have provide a shell environment where you can interact with your filesystem. It's a lot like the standard file interaction commands. You can even use the up-arrow and down-arrow to review old commands, like the familiar shell environment. From the shell, you will use the standard file system commands, below is a sample run of the shell.

```
#> ./shell
fs_shell (-) > mkdir
fs_shell (*) > ls
fs_shell (*) > touch a
fs_shell (*) > ls
  a     Mon Jan 27 18:46:44 2014
fs_shell (*) > touch b c
fs_shell (*) > ls
  a     Mon Jan 27 18:46:44 2014
  b     Mon Jan 27 18:46:49 2014
  c     Mon Jan 27 18:46:49 2014
fs_shell (*) > touch b
fs_shell (*) > ls
  a     Mon Jan 27 18:46:44 2014
  b     Mon Jan 27 18:46:52 2014
  c     Mon Jan 27 18:46:49 2014
fs_shell (*) > rm b
fs_shell (*) > ls
  a     Mon Jan 27 18:46:44 2014
  c     Mon Jan 27 18:46:49 2014
fs_shell (*) > touch g
fs_shell (*) > ls
  a     Mon Jan 27 18:46:44 2014
  g     Mon Jan 27 18:46:56 2014
  c     Mon Jan 27 18:46:49 2014
```

```
fs_shell (*) > rmdir
fs_shell (-) > ls
ERROR: curdir not set: call mkdir
fs_shell (-) >
```

Briefly, the shell requires the creation of the file system's current directory via `mkdir`. You cannot create multiple file system directory, just the base directory. Similarly, the current directory can be remove with `rmdir`, but unlike the standard `rmdir`, our version does not require the directory to be empty first. The code for the shell is in `shell.c`, but you do not need to edit this file. It is provided for you, as is.

Compilation Process Explained

You may notice that this is the first lab where code is separated between source files, those ending in `.c` and header files, those ending in `.h`. This is because you are writing a library for your file system that will need to be compiled with the shell and the test file. The file system code, `filesystem.c` and `filesystem.h` do not have `main()` functions, but the test program and shell do. What the Makefile is doing is compiling the filesystem code with the programs that have `main()` functions to generate binary executables.

To do this, we have to first compile the `filesystem.c` into an object file. An object file is compiled program that hasn't been assembled yet for execution. It's like half-way compiled. To do this, you use the `-c` option with the compiler:

```
#> clang -g -c filesystem.c -o filesystem.o
```

You can think of the object file as the library, and to use the library we have to compile that with another source file that has a `main()` function.

```
#> clang -g -c tesfs.c -o testfs.o
#> clang -g tesfs.o filesystem.o -o testfs
```

The result of compiling the two object files together is finally assembled into the executable `testfs`, which we can now run:

```
#> ./tesfs
```

This is a cumbersome process, and to help, we've provided you with a Makefile which will do the compilation for you. To compile your program, just type:

```
#> make testfs
#> make shell
```

And you're done. In the future, we'll review the compilation process further, and you'll eventually need to be able to generate your own Makefiles.

Hints and Tips

- Work in small parts and build up to bigger parts. Get `mkdir()` and `rmdir()` working before moving on to work on something else, like `touch()` and `ls()`, then get that working, and then do `rm()`.
- Test as you go. Don't assume anything is just going to work.
- Remember that anything you allocated with `malloc()` or `calloc()` must be deallocated with `free()`.
- Modularize your code. Use functions you've already written. For example, isn't `rmdir()` a lot like calling `rm()` on all the files?
- Use valgrind and gdb liberally! You will make mistakes, but you have the tools to recognize those mistakes and fix them.
- Dealing with strings can be annoying, as you learned in the last lab. Check out `strcpy()` and `strdup()` which are super useful, the former copies strings and the later will duplicate a string, allocating the right about memory in the process.

Sample output from testfs

```
 #> ./testfs
----TEST SIMPLE----
  Hello Mon Jan 27 18:45:45 2014
  Goodbye       Mon Jan 27 18:45:45 2014
```

```
  what  Mon Jan 27 18:45:45 2014
----
  Hello Mon Jan 27 18:45:45 2014
  what  Mon Jan 27 18:45:45 2014
----



--- TEST EXPAND---
  file0.txt     Mon Jan 27 18:45:45 2014
  file1.txt     Mon Jan 27 18:45:45 2014
  file2.txt     Mon Jan 27 18:45:45 2014
  file3.txt     Mon Jan 27 18:45:45 2014
  file4.txt     Mon Jan 27 18:45:45 2014
  file5.txt     Mon Jan 27 18:45:45 2014
  file6.txt     Mon Jan 27 18:45:45 2014
  file7.txt     Mon Jan 27 18:45:45 2014
  file8.txt     Mon Jan 27 18:45:45 2014
  file9.txt     Mon Jan 27 18:45:45 2014
  file10.txt    Mon Jan 27 18:45:45 2014
  file11.txt    Mon Jan 27 18:45:45 2014
  file12.txt    Mon Jan 27 18:45:45 2014
  file13.txt    Mon Jan 27 18:45:45 2014
  file14.txt    Mon Jan 27 18:45:45 2014
  file15.txt    Mon Jan 27 18:45:45 2014
  file16.txt    Mon Jan 27 18:45:45 2014
  file17.txt    Mon Jan 27 18:45:45 2014
  file18.txt    Mon Jan 27 18:45:45 2014
  file19.txt    Mon Jan 27 18:45:45 2014
  file20.txt    Mon Jan 27 18:45:45 2014
  file21.txt    Mon Jan 27 18:45:45 2014
  file22.txt    Mon Jan 27 18:45:45 2014
  file23.txt    Mon Jan 27 18:45:45 2014
  file24.txt    Mon Jan 27 18:45:45 2014
  file25.txt    Mon Jan 27 18:45:45 2014
  file26.txt    Mon Jan 27 18:45:45 2014
  file27.txt    Mon Jan 27 18:45:45 2014
  file28.txt    Mon Jan 27 18:45:45 2014
  file29.txt    Mon Jan 27 18:45:45 2014
----
  file0.txt     Mon Jan 27 18:45:45 2014
  file1.txt     Mon Jan 27 18:45:45 2014
  file2.txt     Mon Jan 27 18:45:45 2014
  file3.txt     Mon Jan 27 18:45:45 2014
  file4.txt     Mon Jan 27 18:45:45 2014
  file5.txt     Mon Jan 27 18:45:45 2014
  file6.txt     Mon Jan 27 18:45:45 2014
```

```
  file7.txt     Mon Jan 27 18:45:45 2014
  file8.txt     Mon Jan 27 18:45:45 2014
  file9.txt     Mon Jan 27 18:45:45 2014
  file10.txt    Mon Jan 27 18:45:45 2014
  file11.txt    Mon Jan 27 18:45:45 2014
  file12.txt    Mon Jan 27 18:45:45 2014
  file13.txt    Mon Jan 27 18:45:45 2014
  file14.txt    Mon Jan 27 18:45:45 2014
----
  file0.txt     Mon Jan 27 18:45:45 2014
  file1.txt     Mon Jan 27 18:45:45 2014
  file2.txt     Mon Jan 27 18:45:45 2014
  file3.txt     Mon Jan 27 18:45:45 2014
  file4.txt     Mon Jan 27 18:45:45 2014
  file5.txt     Mon Jan 27 18:45:45 2014
  file6.txt     Mon Jan 27 18:45:45 2014
  file7.txt     Mon Jan 27 18:45:45 2014
  file8.txt     Mon Jan 27 18:45:45 2014
  file9.txt     Mon Jan 27 18:45:45 2014
  file10.txt    Mon Jan 27 18:45:45 2014
  file11.txt    Mon Jan 27 18:45:45 2014
  file12.txt    Mon Jan 27 18:45:45 2014
  file13.txt    Mon Jan 27 18:45:45 2014
  file14.txt    Mon Jan 27 18:45:45 2014
  new0.txt      Mon Jan 27 18:45:45 2014
  new1.txt      Mon Jan 27 18:45:45 2014
  new2.txt      Mon Jan 27 18:45:45 2014
  new3.txt      Mon Jan 27 18:45:45 2014
  new4.txt      Mon Jan 27 18:45:45 2014
  new5.txt      Mon Jan 27 18:45:45 2014
  new6.txt      Mon Jan 27 18:45:45 2014
  new7.txt      Mon Jan 27 18:45:45 2014
  new8.txt      Mon Jan 27 18:45:45 2014
  new9.txt      Mon Jan 27 18:45:45 2014
  new10.txt     Mon Jan 27 18:45:45 2014
  new11.txt     Mon Jan 27 18:45:45 2014
  new12.txt     Mon Jan 27 18:45:45 2014
  new13.txt     Mon Jan 27 18:45:45 2014
  new14.txt     Mon Jan 27 18:45:45 2014
  new15.txt     Mon Jan 27 18:45:45 2014
  new16.txt     Mon Jan 27 18:45:45 2014
  new17.txt     Mon Jan 27 18:45:45 2014
  new18.txt     Mon Jan 27 18:45:45 2014
  new19.txt     Mon Jan 27 18:45:45 2014
  new20.txt     Mon Jan 27 18:45:45 2014
  new21.txt     Mon Jan 27 18:45:45 2014
  new22.txt     Mon Jan 27 18:45:45 2014
```

```
  new23.txt     Mon Jan 27 18:45:45 2014
  new24.txt     Mon Jan 27 18:45:45 2014
  new25.txt     Mon Jan 27 18:45:45 2014
  new26.txt     Mon Jan 27 18:45:45 2014
  new27.txt     Mon Jan 27 18:45:45 2014
  new28.txt     Mon Jan 27 18:45:45 2014
  new29.txt     Mon Jan 27 18:45:45 2014



--- TEST TOUCH ---
  Hello Mon Jan 27 18:45:45 2014
  Goodbye       Mon Jan 27 18:45:45 2014
----
  Hello Mon Jan 27 18:45:47 2014
  Goodbye       Mon Jan 27 18:45:45 2014
----
  Goodbye       Mon Jan 27 18:45:47 2014
----
  Hello Mon Jan 27 18:45:47 2014
  Goodbye       Mon Jan 27 18:45:47 2014
```