**CS111 - Project 0: Warm-Up**

**INTRODUCTION:**

Every quarter a few students come into this course without basic C software development skills, and invest a great deal of time and frustrating effort in the first two projects before concluding they will fail the course and must drop. We have created this simple warm-up to determine whether or not students are prepared to work on C programming projects. Most students should find this project to be completely trivial (20 minutes of work). If you do not find this project to be trivial, you may want to reconsider whether or not you are ready to take this course.

**RELATION TO READING AND LECTURES:**

None. This project requires only skills that incoming students should already possess.

**PROJECT OBJECTIVES:**

* ensure students have a working Linux development environment.
* ensure students can access and understand Linux API documentation.
* ensure students can code, compile, test and debug simple C programs.
* introduce and demonstrate the ability to use basic POSIX file operations.
* introduce and demonstrate the ability to process command line arguments.
* introduce and demonstrate the ability to catch and handle run-time exceptions.
* introduce and demonstrate the ability to return informative exit status.
* demonstrate the ability to construct a standard Makefile.
* demonstrate the ability to write software that conforms to a Command Line Interface (CLI) specification.

**DELIVERABLES:**

A single tarball (.tar.gz) containing:

* a single C source module that compiles cleanly (with no errors or warnings).
* a Makefile to build the program and the tarball.
* two screen snapshot(s) from a *gdb(1)*: one showing a segfault and associated stack-trace, the second showing a breakpoint and variable inspection.
* a README file describing each of the included files and any other information about your submission that you would like to bring to our attention (e.g. limitations, features, testing methodology, use of slip days).

**PROJECT DESCRIPTION:**

1. (if you do not already have one) bring up a Linux development environment. If you do not already have one and do not know how to do this, the local Linux Users Group can help you. Your development environment should include (at least):
   * gcc
   * libc (e.g. glibc or libc6-dev)
   * make
   * gdb

If do not want a personal Linux development environment, you can also work directly on a [SEASNET server](http://www.seasnet.ucla.edu/lnxsrv).

1. (if you are not already familiar with them) study the following manual sections:
   * POSIX file operations ... *open(2), creat(2), close(2), dup(2), read(2), write(2), exit(2), signal(2)*, and this brief tutorial on [file descriptor manipulation](http://lever.cs.ucla.edu/classes/111_fall16/labs/file_descriptors.html).
   * *perror(3)* ... a function that interprets the error codes returned from failed system calls.
   * *getopt(3)* ... the framework we will use for argument handling in all projects for this course.

Feel free to seek out other examples/tutorials for these functions, but make sure you cite those sources in your README.

1. write a program that copies its standard input to its standard output by *read(2)*-ing from file descriptor 0 (until encountering an end of file) and *write(2)*-ing to file descriptor 1. If no errors (other than EOF) are encountered, your program should *exit(2)*with a return code of 0.

Your program executable should be called lab0, and accept the following command line arguments (in any combination):

* + **--input=***filename* ... use the specified file as standard input (making it the new fd0).   
    If you are unable to open the specified input file, report the failure (on stderr, file descriptor 2) using *fprintf(3)* and *perror(3)*, and *exit(2)* with a return code of 1.
  + **--output=***filename* ... create the specified file and use it as standard output (making it the new fd1).   
    If you are unable to create the specified output file, report the failure (on stderr, file descriptor 2) using *fprintf(3)* and *perror(3)*, and *exit(2)* with a return code of 2.
  + **--segfault** ... force a segmentation fault by setting a char \* pointer to NULL and then storing through it.
  + **--catch** ... use *signal(2)* to register a SIGSEGV handler that catches the segmentation fault, logs an error message (on stderr, file descriptor 2) and *exit(2)* with a return code of 3. (Note: catch should not trigger the segmentation fault. )

1. create a Makefile that supports the following targets:
   * (default) ... build the lab0 executable.
   * check ... runs a quick smoke-test on whether or not the program seems to work, supports the required arguments, and reports success or failure.   
     Please include a brief description (in your README) of what checks you chose to include in your smoke-test.
   * clean ... delete all makefile-created files to return the directory to its just installed state.
   * dist ... build the distribution tarball.
2. run your program (with the **--segfault** argument) under *gdb(1)*
   * take the fault
   * get a stack backtrace
   * take a screen snapshot (to be included with your submission)
3. run your program (with the **--segfault** argument) under *gdb(1)*
   * set a break-point at the bad assignment
   * run the program up to the breakpoint
   * inspect the pointer to confirm that it is indeed NULL
   * take a screen snapshot (to be included with your submission)

**Lab 1. Simpleton shell**

[[111 home](http://web.cs.ucla.edu/classes/winter17/cs111/index.html) > [syllabus](http://web.cs.ucla.edu/classes/winter17/cs111/syllabus.html)]

**Introduction**

You are a programmer for Big Data Systems, Inc., a company that specializes in large backend systems that analyze [big data](https://en.wikipedia.org/wiki/Big_data). Much of BDSI's computation occurs in a cloud or a grid. Computational nodes are cheap [SMP](https://en.wikipedia.org/wiki/Symmetric_multiprocessing) hosts with a relatively small number of processors. Nodes typically run simple shell scripts as part of the larger computation, and you've been assigned the job of improving the infrastructure for these scripts.

Many of the shell scripts have command sequences that look like the following (though the actual commands tend to be more proprietary):

(sort < a | cat b - | tr A-Z a-z > c) 2>> d

This command invokes three subcommands. The first runs the command sort with standard input being the file a and standard output being a unnamed pipe 1. The second runs the command cat b - with standard input being pipe 1 and standard output being pipe 2. The third runs the command tr A-Z a-z with standard input being pipe 2 and standard output being the file c. All three commands have standard error sent, via the same file descriptor, to file d in append-only mode.

BDSI's developers have several complaints about these shell scripts:

* The shell script syntax does not let them open files with special flags available at the system call level. For example, there is no way to open a file with the O\_DSYNC flag of the [open](http://pubs.opengroup.org/onlinepubs/9699919799/functions/open.html) system call.
* The developers want to be able to create arbitrary directed graphs of processes connected via pipes, but the shell syntax forces them into pipelines.
* The developers want a simpler way to invoke the shell, one that is more easily generated from their programs and scripts, one that does not require that they write a shell script. They do not mind if the simpler shell is harder for humans to use, because it's not intended to be used directly by programmers.

**Basic idea**

To address these issues, your boss proposes a new program simpsh, short for "SIMPleton SHell", a very simple, stripped down shell. simpsh does not use a scripting language at all, and you do not interact with it at a terminal or give it a script to run. Instead, developers invoke the simpsh command by passing it arguments telling it which files to access, which pipes to create, and which subcommands to invoke. It then creates or accesses all the files and creates all the pipes processes needed to run the subcommands, and reports the processes's exit statuses as they exit.

For example, the abovementioned command in the standard shell could be run using the following simpsh command. This invocation uses standard shell syntax, because it is invoking simpsh from the standard shell; the command itself, though, is just an array of strings and simpsh interprets this array and executes the same three subcommands that the abovementioned shell command does.

simpsh \

--rdonly a \

--pipe \

--pipe \

--creat --trunc --wronly c \

--creat --append --wronly d \

--command 3 5 6 tr A-Z a-z \

--command 0 2 6 sort \

--command 1 4 6 cat b - \

--wait

This example invocation creates seven file descriptors:

1. A read only descriptor for the file a, created by the --rdonly option.
2. The read end of the first pipe, created by the first --pipe option.
3. The write end of the first pipe, also created by the first --pipe option.
4. The read end of the second pipe, created by the second --pipe option.
5. The write end of the second pipe, also created by the second --pipe option.
6. A write only descriptor for the file c, created by the first --wronly option as modified by the preceding --creat and --trunc.
7. A write only, append only descriptor for the file d, created by the --wronly option as modified by the preceding --creat and --append options.

It then creates three subprocesses:

* A subprocess with standard input, output, and error being the file descriptors numbered 3, 5, and 6 above. This subprocess runs the command tr with the two arguments A-Z and a-z.
* A subprocess with standard input, standard output, and standard error being the file descriptors numbered 0, 2, and 6 above, respectively. This subprocess runs the command sort with no arguments
* A subprocess with standard input, output, and error being the file descriptors numbered 1, 4, and 6 above. This subprocess runs the command cat with the two arguments b and -.

It then waits for all three subprocesses to finish. As each finishes, it prints its exit status, followed by the command and arguments. The output might look like this:

0 sort

0 cat b -

0 tr A-Z a-z

although not necessarily in that order, depending on which order the subprocesses finished.

**simpsh options**

Here is a detailed list of the command-line options that simpsh should support. Each option should be executed in sequence, left to right.

First are the file flags. These flags affect the next file that is opened. They are ignored if no later file is opened. Each file flag corresponds to an *oflag* value of [open](http://pubs.opengroup.org/onlinepubs/9699919799/functions/open.html); the corresponding *oflag* value is listed after the option. Also see [Opening and Closing Files](http://www.gnu.org/software/libc/manual/html_node/Opening-and-Closing-Files.html) and [Open-time Flags](http://www.gnu.org/software/libc/manual/html_node/Open_002dtime-Flags.html).

--append

O\_APPEND

--cloexec

O\_CLOEXEC

--creat

O\_CREAT

--directory

O\_DIRECTORY

--dsync

O\_DSYNC

--excl

O\_EXCL

--nofollow

O\_NOFOLLOW

--nonblock

O\_NONBLOCK

--rsync

O\_RSYNC

--sync

O\_SYNC

--trunc

O\_TRUNC

Second are the file-opening options. These flags open files. Each file-opening option also corresponds to an *oflag* value, listed after the option. Each opened file is given a file number; file numbers start at 0 and increment after each file-opening option. Normally they increment by 1, but the --pipe option causes them to increment by 2.

--rdonly *f*

O\_RDONLY. Open the file *f* for reading only.

--rdwr *f*

O\_RDWR. Open the file *f* for reading and writing.

--wronly *f*

O\_WRONLY. Open the file *f* for writing only.

--pipe

Open a pipe. Unlike the other file options, this option does not take an argument. Also, it consumes two file numbers, not just one.

Third is the subcommand options:

--command *i o e cmd args*

Execute a command with standard input *i*, standard output *o* and standard error *e*; these values should correspond to earlier file or pipe options. The executable for the command is *cmd* and it has zero or more arguments *args*. None of the *cmd* and *args*operands begin with the two characters "--".

--wait

Wait for all commands to finish. As each finishes, output its exit status, and a copy of the command (with spaces separating arguments) to standard output.

Finally, there are some miscellaneous options:

--close *N*

Close the *N*th file that was opened by a file-opening option. For a pipe, this closes just one end of the pipe. Once file *N* is closed, it is an error to access it, just as it is an error to access any file number that has never been opened. File numbers are not reused by later file-opening options.

--verbose

Just before executing an option, output a line to standard output containing the option. If the option has operands, list them separated by spaces. Ensure that the line is actually output, and is not merely sitting in a buffer somewhere.

--profile

Just after executing an option, output a line to standard output containing the resources used. Use [getrusage](http://pubs.opengroup.org/onlinepubs/9699919799/functions/getrusage.html) and output a line containing as much useful information as you can glean from it.

--abort

Crash the shell. The shell itself should immediately dump core, via a segmentation violation.

--catch *N*

Catch signal *N*, where *N* is a decimal integer, with a handler that outputs the diagnostic *N* caught to stderr, and exits with status *N*. This exits the entire shell. *N* uses the same numbering as your system; for example, on GNU/Linux, a segmentation violation is signal 11.

--ignore *N*

Ignore signal *N*.

--default *N*

Use the default behavior for signal *N*.

--pause

Pause, waiting for a signal to arrive.

When there is a syntax error in an option (e.g., a missing operand), or where a file cannot be opened, or where is some other error in a system call, simpsh should report a diagnostic to standard error and should continue to the next option. However, simpshshould ignore any write errors to standard error, so that it does not get into an infinite loop outputting write-error  diagnostics.

When simpsh exits other than in response to a signal, it should exit with status equal to the maximum of all the exit statuses of all the subcommands that it ran and successfully waited for. However, if there are no such subcommands, or if the maximum is zero, simpsh should exit with status 0 if all options succeeded, and with status 1 one of them failed. For example, if a file could not be opened, simpsh must exit with nonzero status.

**Implementation**

Your implementation will take three phases:

* In Lab 1A, you'll warm up by implementing just the options --rdonly, --wronly, --command, and --verbose.
* In Lab 1B, you'll implement the rest of the options, except for --profile.
* In Lab 1C, you'll implement --profile and compare the performance of your implementation to that of bash and that of [dash](https://en.wikipedia.org/wiki/Almquist_shell). Design three nontrivial benchmarks of your own, and translate them from simpsh to POSIX shell form so that they can be run on bash and dash, and time all three benchmarks on all three implementations. Run each benchmark three times and take the average of the user + system CPU times, counting all overhead of starting and finishing the shell. Each bash and dash benchmark should use the times command at the end to output CPU times for the shell and its children, and you should compare that to the --profile output of the simpsh benchmark.

Before charging ahead and implementing, you should be familiar with the man pages for close, dup2, execvp, fork, getopt\_long, open, pipe, and sigaction.

Your program should come with a file named Makefile that supports the following actions.

* 'make' builds the simpsh program.
* 'make clean' removes the program and all other temporary files and object files that can be regenerated with 'make'.
* 'make check' tests the simpsh program on test cases that you design. You should have at least three test cases.
* 'make dist' makes a software distribution compressed tarball lab1-*yourname*.tar.gz and does some simple testing on it. This tarball is what you should submit via CCLE. All the files in the tarball should have names of the form lab1-*yourname*/... and one of the files should be lab1-*yourname*/Makefile.

Your solution should be written in the C programming language. Your code should be [robust](http://www.gnu.org/prep/standards/html_node/Semantics.html), for example, it should not impose an arbitrary limit like 216 bytes on the length of a string. You may use the features of [C11](https://en.wikipedia.org/wiki/C11_%28C_standard_revision%29) as implemented on the SEASnet GNU/Linux servers running RHEL 7. Please prepend the directory /usr/local/cs/bin to your PATH, to get the versions of the tools that we will use to test your solution. Your solution should stick to the standard [GNU C library](http://www.gnu.org/software/libc/) that is installed on SEASnet, and should not rely on other libraries.

You can test your program by running it directly. Eventually, you should put your own test cases into a file test.sh and run it automatically as part of 'make check'.

**Submit**

After you implement Lab 1A, submit via CCLE the .tar.gz file that is built by 'make dist'. Similarly for 1B and 1C. Your submission should contain a README file that briefly describes known limitations of your code and any extra features you'd like to call our attention to.

**CS111 - Project 2A: Races and Synchronization**

**INTRODUCTION:**

In this project, you will engage (at a low level) with a range of synchronization problems. Part A of the project (this part!) deals with conflicting read-modify-write operations on single variables and complex data structures (an ordered linked list).  It is broken up into multiple steps:

* Part 1 - updates to a shared variable:
* Write a multithreaded application (using pthreads) that performs parallel updates to a shared variable.
* Demonstrate the race condition in the provided “add” routine, and address it with different synchronization mechanisms.
* Do performance instrumentation and measurement.
* Analyze and explain the observed performance.
* Part 2 - updates to a shared complex data structure:
* Implement the four routines described in [SortedList.h](http://web.cs.ucla.edu/classes/winter17/cs111/labs/SortedList.h): SortedList\_insert, SortedList\_delete, SortedList\_lookup, and SortedList\_length.
* Write a multi-threaded application using pthread that performs parallel updates to a sorted doubly linked list data structure (using methods from the above step).
* Recognize and demonstrate the race conditions when performing linked list operations, and address them with different synchronization mechanisms.
* Do performance instrumentation and measurement.
* Analyze and explain the observed performance.

**RELATION TO READING AND LECTURES:**

The basic shared counter problem was introduced in section 28.1.

Mutexes, test-and-set, spin-locks, and compare-and-swap were described in (many sections of) chapter 28.

Synchronization of partitioned lists was introduced in section 29.2.

**PROJECT OBJECTIVES:**

* primary: demonstrate the ability to recognize critical sections and address them with a variety of different mechanisms.
* primary: demonstrate the existence of race conditions, and efficacy of the subsequent solutions
* secondary: demonstrate the ability to deliver code to meet CLI and API specifications.
* secondary: experience with basic performance measurement and instrumentation
* secondary: experience with application development, exploiting new library functions, creating command line options to control non-trivial behavior.

**DELIVERABLES:**

A single tarball (.tar.gz) containing:

* Four C source modules that compile cleanly (with no errors or warnings):
* **lab2\_add.c** - a C program that implements and tests a shared variable add function, implements the (below) specified command line options, and produces the (below) specified output statistics.
* **SortedList.h** - a header file (supplied by us) describing the interfaces for linked list operations.
* **SortedList.c***-*aC module that implements insert, delete, lookup, and length methods for a sorted doubly linked list (described in the provided header file, including correct placement of *yield* calls).
* **lab2\_list.c** - a C program that implements the (below) specified command line options and produces the (below) specified output statistics.
* A **Makefile** to build the deliverable programs, output, graphs, and tarball.  For your early testing you are free to run your program manually, but by the time you are done, all of the below-described test cases should be executed, the output captured, and the graphs produced automatically.  The higher level targets should be:
* **build** … compile all programs (default target)
* **tests** … run all (over 200) specified test cases to generate results in CSV files. Note that the lab2\_list program is expected to fail when running multiple threads without synchronization.  Make sure that your Makefile continues executing despite such failures (e.g. put a ‘-’ in front of commands that are expected to fail).
* **graphs** … use *gnuplot(1)* and the supplied data reduction scripts to generate the required graphs
* **tarball** … create the deliverable tarball
* **clean** … delete all generated programs and output
* **lab2\_add.csv** - containing all of your results for all of the Part-1 tests.
* **lab2\_list.csv** - containing all of your results for all of the Part-2 tests.
* graphs (**.png**files), created by *gnuplot(1)*on the above **csv** files with the supplied data reduction scripts:
* For lab2\_add
* **lab2\_add-1.png** ...threads and iterations required to generate a failure (with and without yields)
* **lab2\_add-2.png** … Average time per operation with and without yields.
* **lab2\_add-3.png** … Average time per (single threaded) operation vs. the number of iterations.
* **lab2\_add-4.png** threads and iterations that can run successfully with yields under each of the three synchronization methods.
* **lab2\_add-5.png** Average time per (multi-threaded) operation vs. the number of threads, for all four versions of the add function.
* For lab2\_list
* **lab2\_list-1.png** … average time per (single threaded) unprotected operation vs. number of iterations (illustrating the correction of the per-operation cost for the list length).
* **lab2\_list-2.png** … threads and iterations required to generate a failure (with and without yields).
* **lab2\_list-3.png** … iterations that can run (protected) without failure.
* **lab2\_list-4.png** … (corrected) average time per operation (for unprotected, mutex, and spin-lock) vs. number of threads.
* a **README.txt** file containing:
* descriptions of each of the included files and any other information about your submission that you would like to bring to our attention (e.g. limitation, features, testing methodology).
* brief (1-4 sentences per question) answers to each of the questions (below).

**PROJECT DESCRIPTION:**

To perform this assignment, you will need to learn a few things:

* pthread (<https://computing.llnl.gov/tutorials/pthreads/>)
* clock\_gettime(2) … high resolution timers
* GCC atomic builtins (<http://gcc.gnu.org/onlinedocs/gcc-4.4.3/gcc/Atomic-Builtins.html>)
* *gnuplot(1)* … is a general and powerful tool for producing a wide variety of graphs, and is commonly used for organizing and reporting performance data.  We are providing you with sample data reduction scripts for the first parts of this assignment:
* [lab2\_add.gp](http://web.cs.ucla.edu/classes/winter17/cs111/labs/lab2_add.gp)
* [lab2\_list.gp](https://www.google.com/url?q=https://drive.google.com/open?id%3D0B8f2cujmXkYXUVJGVWoxTm1YUTg&sa=D&ust=1476563305097000&usg=AFQjCNHNqWXYO89OGHd-ERzxPjgiQtENGw)

To use these scripts you will need a recent version of *gnuplot* installed on your system.

These scripts take no arguments, read *comma-separated value* (CSV) input files with standard names (lab2\_add.csv, lab2\_list.csv), and produce graphical output .png files with standard names.

In the next and final part of this assignment, you can use these as a basis for creating your own graphing scripts.

**PART 1: adds to a shared variable**

Start with a basic add routine:

        void add(long long \*pointer, long long value) {

                long long sum = \*pointer + value;

                \*pointer = sum;

        }

Write a test driver program called **lab2\_add** that:

* takes a parameter for the number of parallel threads (**--threads=**#, default 1)
* takes a parameter for the number of iterations (**--iterations=**#, default 1)
* initializes a (long long) counter to zero
* notes the (high resolution) starting time for the run (using *clock\_gettime(2)*)
* starts the specified number of threads, each of which will use the above add function to
* add 1 to the counter the specified number of times
* add -1 to the counter the specified number of times
* exit to re-join the parent thread
* wait for all threads to complete, and notes the (high resolution) ending time for the run
* prints to stdout a comma-separated-value (CSV) record including
* the name of the test (“add-none” for the most basic usage)
* the number of threads (from **--threads=#**)
* the number of iterations (from **--iterations=#**)
* the total number of operations performed (threads x iterations x 2, the “x 2” factor because you add 1 first and then add -1)
* the total run time (in nanoseconds)
* the average time per operation (in nanoseconds).
* the total at the end of the run (0 if there were no conflicting updates)
* If the run completes successfully, exit with a return code of zero.  If any errors (other than a non-zero final count) are encountered, exit with a non-zero exit code.

The supported command line options and expected output are illustrated below:

% ./lab2\_add --iterations=10000 --threads=10

add-none,10,10000,200000,6574000,32,374

%

Run your program for ranges of *threads* (2, 4, 8, 12) and *iterations* (100, 1000, 10000, 100000) values, capture the output, and note how many threads and iterations it takes to (fairly consistently) result in a failure (non-zero sum).

**QUESTION 2.1.1 - causing conflicts:**

**Why does it take many iterations before errors are seen?**

**Why does a significantly smaller number of iterations so seldom fail?**

There are ways to cause a thread to immediately yield (rather than waiting for a time slice end to preempt it).  Posix includes a sched\_yield operation, and Linux includes a pthread\_yield operation.  Extend the basic add routine to more easily cause the failures:

        int opt\_yield;

        void add(long long \*pointer, long long value) {

                long long sum = \*pointer + value;

                if (opt\_yield)

                        sched\_yield();

                \*pointer = sum;

        }

Add a new **--yield** option to your driver program that sets opt\_yield to 1.  If this option has been specified, each line of statistics output should begin with “**add-yield-none**”.  Re-run your tests, with yields, for ranges of threads (2,4,8,12) and iterations (10, 20, 40, 80, 100, 1000, 10000, 100000), capture the output, and see how many iterations and threads it takes to (fairly consistently) result in a failure.  It should now be much easier to cause the failures.

Compare the average execution time of the yield and non-yield versions a range threads (2, 8) and of iterations (100, 1000, 10000, 100000). You should note that the **--yield** runs are much slower than the non-yield runs.

**QUESTION 2.1.2 - cost of yielding:**

**Why are the --yield runs so much slower?  Where is the additional time going?  Is it possible to get valid per-operation timings if we are using the --yield option?  If so, explain how.  If not, explain why not.**

For a single thread, graph the average cost per operation (non-yield) as a function of the number of iterations.  You should note that the average cost per operation goes down as the number of iterations goes up.

If you install *gnuplot(1)* and append all of your test output to a file called **lab2\_add.csv**, you can use our sample data reduction scripts to produce this and all other required data plots.

**QUESTION 2.1.3 - measurement errors:**

**Why does the average cost per operation drop with increasing iterations?**

**If the cost per iteration is a function of the number of iterations, how do we know how many iterations to run (or what the “correct” cost is)?**

Implement three new versions of the add function:

* one protected by a pthread\_mutex, enabled by a new **--sync=m** option.  When running this test, the output statistics line should begin with “add-m” or “add-yield-m”.
* one protected by a spin-lock, enabled by a new **--sync=s** option.  You will have to implement your own spin-lock operation.  We suggest that you do this using the GCC atomic \_\_sync\_ builtin functions **\_\_sync\_lock\_test\_and\_set**and **\_\_sync\_lock\_release**.  When running this test, the output statistics line should begin with “add-s” or “add-yield-s”.
* one that performs the add using the GCC atomic \_\_sync\_ builtin function **\_\_sync\_val\_compare\_and\_swap** to ensure atomic updates to the shared counter, enabled by a new **--sync=c** option.  When running this test, the output statistics line should begin with “**add-c**” or “**add-yield-c**”.

Use your **--yield** option to confirm that, even for large numbers of threads (2, 4, 8, 12) and iterations (10,000 for mutexes and CAS, only 1,000 for spin locks) that reliably failed in the unprotected scenarios, all three of these serialization mechanisms eliminate the race conditions in the add critical section.  Capture the output from these confirmation runs.  [Note that we suggest a smaller number of threads/iterations when you test spin-lock synchronization]

Using a large enough number of iterations (e.g. 10,000) to overcome the issues raised in the question 2.1.3, test all four (no yield) versions (unprotected, mutex, spin-lock, compare-and-swap) for a range of number of threads (1,2,4,8,12) and capture the output.  Graph the average time per operation (non-yield), vs the number of threads.

**QUESTION 2.1.4 - costs of serialization:**

**Why do all of the options perform similarly for low numbers of threads?**

**Why do the three protected operations slow down as the number of threads rises?**

**Why are spin-locks so expensive for large numbers of threads?**

**PART 2: sorted, doubly-linked, list**

Review the interface specifications for a sorted doubly linked list package described in the header file [SortedList.h](https://www.google.com/url?q=https://drive.google.com/open?id%3D0B8f2cujmXkYXZE9CUVkyV3pLcjg&sa=D&ust=1476563305128000&usg=AFQjCNH8tK83CQ7qzWbv9WdYm0r9078K5A), and implement all four methods in a new module named **SortedList.c**.  Note that the interface includes three software-controlled yield options.  Identify the critical section in each of your four methods, and add calls to pthread\_yield or sched\_yield, controlled by the yield options:

* in SortedList\_insert if opt\_yield & INSERT\_YIELD
* in SortedList\_delete if opt\_yield & DELETE\_YIELD
* in SortedList\_lookup if opt\_yield & LOOKUP\_YIELD
* in SortedList\_length if opt\_yield & LOOKUP\_YIELD

to force a switch to another thread at the critical point in each method.

Write a test driver program called **lab2\_list** that:

* takes a parameter for the number of parallel threads (**--threads=**#, default 1).
* takes a parameter for the number of iterations (**--iterations=**#, default 1).
* takes a parameter to enable (any combination of) optional critical section yields (**--yield**=[**idl**], **i** for insert, **d** for delete, and **l** for lookups).
* initializes an empty list.
* creates and initializes (with random keys) the required number (threads x iterations) of list elements.  Note that we do this before creating the threads so that this time is not included in our start-to-finish measurement.
* notes the (high resolution) starting time for the run (using *clock\_gettime(2)*).
* starts the specified number of threads.
* each thread:
* starts with a set of pre-allocated and initialized elements **(--iterations=#)**
* inserts them all into a (single shared-by-all-threads) list
* gets the list length
* looks up and deletes each of the keys it had previously inserted
* exits to re-join the parent thread
* waits for all threads to complete, and notes the (high resolution) ending time for the run.
* checks the length of the list to confirm that it is zero.
* prints to stdout a comma-separated-value (CSV) record including:
* the name of the test, which is of the form: **list-***yieldopts***-***syncopts*
* where *yieldopts* = {**none**,**i,d**,**l,id,il,dl,idl**}
* Where *syncopts* = {**none**,**s**,**m**}
* the number of threads (from **--threads=#**)
* the number of iterations (from **--iterations=#**)
* the number of lists (always 1 in this project)
* the total number of operations performed (threads x iterations x (insert + lookup  + delete))
* the total run time (in nanoseconds) for all threads
* the average time per operation (in nanoseconds).
* exits with a status of zero if there were no errors, otherwise non-zero

In part one, a synchronization error merely resulted in the subtracts and adds not balancing out.  In this part, a synchronization error is likely to result in a corrupted list.  If, at any time, you find evidence of a corrupted list (e.g. you cannot find a key that you know you inserted, or the list length is not zero at the end of the test), you should log an error message (to stderr) and exit immediately without producing the above results record.  Note that in some cases your program may not detect an error, but may simply experience a segmentation fault.  When you look at your test results, you should consider any test that did not produce output to have failed.

The supported command line options and expected output are illustrated below:

% ./lab2\_list --threads=10 --iterations=1000 --yield=id

list-id-none,10,1000,1,30000,527103247,25355

%

Run your program with a single thread, and increasing numbers of iterations (10, 100, 1000, 10000, 20000), capture the output, and note the average time per operation. These results should be quite different from what you observed when testing your add function with increasing numbers of iterations.  Graph the time per operation vs the number of iterations (for **--threads=1**).

If you append all of your test output to a file called **lab2\_list.csv**, you can use the supplied data reduction script to produce this and all other required data plots.

You will observe that the time per iteration eventually becomes linear with the number of iterations!  This is because the time to insert into or search a sorted list is proportional to the list length.  This is to be expected … but we are primarily interested in the cost of serialization, and so we would like to separate the per operation costs from the per-element costs.  The easiest way to do this is to divide the cost per iteration (total / (threads \* iterations)) by the average search distance (iterations/4).  Why iterations/4?

* Inserts take list length from 0 to iterations, and then from iterations to 0.  Thus, the average list length is iterations/2.
* Each insert or search operation, on average, has to run through half the list, which gives us an average search distance of iterations/4.

With this correction, your program should (modulo startup time) report more stable per-operation costs.  Note that the provided data reduction script graphs both the raw time per operation and the time corrected for the list length.

Run your program and see how many parallel threads (2,4,8,12) and iterations (10,100,1000) it takes to fairly consistently demonstrate a problem.  Then run it again using various combinations of yield options and see how many threads (2,4,8,12) and iterations (2,4,8,16,32) it takes to fairly consistently demonstrate the problem.  Make sure that you can demonstrate:

* conflicts between inserts (**--yield=i**)
* conflicts between deletes (**--yield=d**)
* conflicts between inserts and lookups (**--yield=il**)
* conflicts between deletes and lookups (**--yield=dl**)

Add two new options to your program to call two new versions of these methods: one set of operations protected by pthread\_mutexes (**--sync=m**), and another protected by test-and-set spin locks (**--sync=s**).  Using your **--yield** options, demonstrate that either of these protections seems to eliminate all of the problems, even for large numbers of threads (12) and iterations (32).

Choose an appropriate number of iterations (e.g. 1000) to overcome start-up costs and rerun your program without the yields.  Note that you will only be able to run the unprotected method for a single thread, but you should be able to run the protected methods for a wide range of numbers of threads (1, 2, 4, 8, 12, 16, 24).  Graph the (corrected) per operation times (for each of the three synchronization options: unprotected, mutex, spin) vs the number of threads.

**QUESTION 2.2.1 - scalability of Mutex**

**Compare the variation in time per protected operation vs the number of threads (for mutex-protected operations) in Part-1 and Part-2, commenting on similarities/differences and offering explanations for them.**

**QUESTION 2.2.2 - scalability of spin locks**

**Compare the variation in time per protected operation vs the number of threads for Mutex vs Spin locks, commenting on similarities/differences and offering explanations for them.**

**SUBMISSION:**

Your tarball should have a name of the form lab2a-*studentID*.tar.gz and should be submitted via CCLE.

We will test it on a SEASnet GNU/Linux server running RHEL 7 (this is on lnxsrv09). You would be well advised to test your submission on that platform before submitting it.

**CS111 - Project 2B: Lock Granularity and Performance**

**INTRODUCTION:**

In Project-2A, Part-2, the mutex and spin-lock proved to be bottlenecks, preventing parallel access to the list. In this project, you will do additional performance instrumentation to confirm this, and extend your previous solution to deal with this problem. Project 2B (this part!) can be broken up into a few major steps:

* Do performance instrumentation and measurement to confirm the cause of the problem.
* Implement a new option to divide a list into sublists and support synchronization on sublists, thus allowing parallel access to the (original) list.
* Do new performance measurements to confirm that the problem has been solved.

**RELATION TO READING AND LECTURES:**

Partitioned lists and finer granularity locking are discussed in sections 29.2-4

**PROJECT OBJECTIVES:**

* demonstrate the ability to recognize bottlenecks on large data structures
* experience with partitioning a serialized resource to improve parallelism
* experience with basic performance measurement and instrumentation
* experience with execution profiling
* experience with finding, installing, and exploiting new development tools

**DELIVERABLES:**

A single tarball (.tar.gz) containing:

* SortedList.h - a header file containing interfaces for linked list operations.
* the source for a C source module (*SortedList.c*) that compiles cleanly (with no errors or warnings), and implements insert, delete, lookup, and length methods for a sorted doubly linked list (described in the provided header file, including correct placement of pthread\_yield calls).
* the source for a C program (*lab2\_list.c*) that  compiles cleanly (with no errors or warnings), and implements the specified command line options (--threads, --iterations, --yield, --sync, --lists), drives one or more parallel threads that do operations on a shared linked list, and reports on the final list and performance.
* A *Makefile* to build the deliverable programs, output, graphs, and tarball.  For your early testing you are free to run your program manually, but by the time you are done, all of the below-described test cases should be executed, the output captured, and the graphs produced automatically.  The higher level targets should be:
* **build** … compile all programs (default target)
* **tests** … run all specified test cases to generate CSV results
* **profile** … run tests with profiling tools to generate an execution profiling report
* **graphs** … use gnuplot to generate the required graphs
* **tarball** … create the deliverable tarball
* **clean** … delete all generated programs and output
* **lab\_2b\_list.csv**- containing your results for all of the Part-2 performance tests.
* execution profiling report showing where time was spent :
* In the un-partitioned mutex implementation, for 32 threads.
* In the un-partitioned spin-lock implementation, for 32 threads.
* graphs (.**png** files), created by *gnuplot(1)*on the above csv data showing:
* lab2b\_1.png … throughput vs number of threads for mutex and spin-lock synchronized adds and list operations.
* lab2b\_2.png … mean time per mutex wait and mean time per operation for mutex-synchronized list operations.
* lab2b\_3.png … number of successful iterations for each synchronization method.
* lab2b\_4.png … throughput vs number of threads for mutexes with partitioned lists.
* lab2b\_5.png … throughput vs number of threads for spin-locks with partitioned lists.
* a **README.txt** file containing:
* descriptions of each of the included files and any other information about your submission that you would like to bring to our attention (e.g., limitations, features, testing methodology, etc.).
* brief (a few sentences per question) answers to each of the questions (below).

**PREPARATION:**

To perform this assignment, you will need to research, choose, install and master a multi-threaded execution profiling tool.  Execution profiling is a combination of compile-time and run-time tools that analyze a program’s execution to determine how much time is being spent in which routines.  There are three standard Linux profiling tools:

* The standard Linux *gprof(1)* tool is quite simple to use, but its call-counting mechanism is not-multi-thread safe, and its execution sampling is not multi-thread aware.  As such, it is not usable for analyzing the performance of multi-threaded applications.  There are other tools that do solve this problem.  The two best-known are:
* valgrind … best known for its memory leak detector, which has an interpreted execution engine that can extract a great deal of information about where cycles are being spent, even estimating cache misses.  It does work for multi-threaded programs, but its interpreter does not provide much parallelism.  As such it is not useful for examining high contention situations.
* gperftools … a wonderful set of performance optimization tools from Google.  It includes a profiler that is quite similar to gprof (in that it samples real execution).  This is probably the best tool to use for this problem.

This project is about scalable parallelism, which is only possible on a processor with many cores.  You can do most of your development and testing on any Linux system, but if your personal computer does not have more than a few cores, you will not be able to do real multi-threaded performance testing on it.  Lab servers are available if you need access to a larger machine for your final testing.

**PROJECT DESCRIPTION:**

Review your results from Lab 2 Parts 1 (lab2\_add-5.png)  and 2 (lab2\_list-4.png).  In Part-1, for Compare-and-Swap and Mutex throughput (total operations per second), we saw that adding threads took us from tens of ns per operation to small hundreds of ns per operation. Looking at the analogous results in Part-2, we see the (un-adjusted for length) time per operation go from a few microseconds, to tens of microseconds.  For the adds, moderate contention added ~100ns to each synchronization operation.  For the list operations, moderate contention added ~10us to each synchronization operation.  This represents a 100x difference in the per operation price of lock contention.

Go back to your lab1 and lab2 data, and create a new plotting script that will graph the same data, but plotting the throughput (operations per second) rather than the time per operation:

* Mutex synchronized adds, 10,000 iterations, 1,2,4,8,12 threads
* Spin-lock synchronized adds, 10,000 iterations, 1,2,4,8,12 threads
* Mutex synchronized list operations, 1,000 iterations, 1,2,4,8,12,16,24 threads
* Spin-lock synchronized list operations, 1,000 iterations, 1,2,4,8,12,16,24 threads

In the previous lab, we gave you all of the necessary data reduction scripts.  In this lab, you will have to create your own … but you can use the scripts provided in the previous lab as a starter:

* To get the throughput, divide one Billion (number of nanoseconds in a second) by the time per operation (in nanoseconds).
* Remember that for the list operations there is only one lock/unlock for an entire list, so we should not divide the time per operation by the list length.
* Call this graph lab2b\_1.png

The most obvious differences, which we already knew:

* adds are much less expensive operations than inserts and searches in a long list.
* spin-locks waste increasingly more cycles as the probability of contention increases.

But these throughput graphs show us something that was not as obvious in the cost per operation graphs:

* The add throughput quickly levels off … we have saturated the CPU and the overhead of synchronization seems to increase only very slowly.
* The list operation throughput continues to drop, as the overhead of synchronization increases with the number of threads … and much worse for spin-locks.

The obvious conclusions (from both the cost-per-operation graphs you produced last week, and the throughput graphs you have just produced) are:

* The throughput of parallel synchronized linked list operations does not scale as well as the throughput of parallel synchronized add operations.
* The reduction in throughput with increasing parallelism is due to an increasing time per operation.

Since the code inside the critical section does not change with the number of threads, it seems reasonable to assume that the added execution time is being spent getting the locks.

**QUESTION 2.3.1 - Cycles in the basic implementation:**

**Where do you believe most of the cycles are spent in the 1 and 2-thread tests (for both add and list)?  Why do you believe these to be the most expensive parts of the code?**

**Where do you believe most of the time/cycles are being spent in the high-thread spin-lock tests?**

**Where do you believe most of the time/cycles are being spent in the high-thread mutex tests?**

It should be clear why the spin-lock implementation performs so badly with a large number of threads.  But the mutex implementation should not have this problem.  Now you have some theories about why these algorithms scale so poorly.  But theories are only theories.  We need some data to confirm our theories.

**Execution Profiling**

Build your program with debug symbols, choose your execution profiling package, install it, run the spin-lock list test (1,000 iterations 12 threads) under the profiler, and analyze the results to determine where the cycles are being spent.

The default output from google-pprof will show you which routine is consuming most of the cycles.  If you then re-run google-pprof with the --list option (specifying that routine), it will give you a source-level breakdown of how much time is being spent on each instruction.  You should get a very clear answer to the question of where the program is spending its time.  Update your Makefile to run this test and generate the results automatically (make profile), include your profiling report in your submitted tarball, and identify it in your README file.

To simplify the grading, in your final submission, your code should assume all the pprof libraries and headers are under the same directory where your tarball gets untarred. Please specify the pprof libraries and headers using relative path instead of absolute path.

**QUESTION 2.3.2 - Execution Profiling:**

**Where (what lines of code) are consuming most of the cycles when the spin-lock version of the list exerciser is run with a large number of threads?**

**Why does this operation become so expensive with large numbers of threads?**

**Timing Mutex Waits**

In the mutex case, we are not spinning.  A thread that cannot get the lock is blocked, and not consuming any cycles.  How could we confirm that, in the mutex case, most threads are spending most of their time waiting for a lock?

Update your mutex implementation to:

* Note the high-resolution time before and after getting the mutex, compute the elapsed difference, and add that to a (per-thread) total.
* When the program completes, add up the total lock acquisition time (for all threads) and divide it by the number of lock operations to compute an average wait-for-lock, and add this number, as a new last column, to the output statistics for the run.

Run the list mutex test again for 1,000 iterations and 1, 2, 4, 8, 16, 24 threads, and plot the wait-for-lock time, and the average time per operation against the number of competing threads.  Call this graph lab2b\_2.png.

**QUESTION 2.3.3 - Mutex Wait Time:**

**Look at the average time per operation (vs # threads) and the average wait-for-mutex time (vs #threads).**

**Why does the average lock-wait time rise so dramatically with the number of contending threads?**

**Why does the completion time per operation rise (less dramatically) with the**

**number of contending threads?**

**How is it possible for the wait time per operation to go up faster (or higher) than the completion time per operation?**

**Addressing the Underlying Problem**

While the details of how contention degrades throughput are different for these two mechanisms, all of the degradation is the result of increased contention.  This is the fundamental problem with coarse-grained synchronization.  The classic solution to this problem is to partition the single resource (in this case a linked list) into multiple independent resources, and divide the requests among those sub-resources.

Add a new **--lists=**# option to your lab2\_list program:

* break the single (huge) sorted list into the specified number of sub-lists (each with its own list header and synchronization object).
* change your lab2\_list program to select which sub-list a particular key should be in based on a simple hash of the key, modulo the number of lists.
* figure out how to (safely and correctly) obtain the length of the list, which now involves enumerating all of the sub-lists.
* each thread:
* starts with a set of pre-allocated and initialized elements **(--iterations=#)**
* inserts them all into the multi-list (which sublist the key should go into determined by a hash of the key)
* gets the list length
* looks up and deletes each of the keys it inserted
* exits to re-join the parent thread
* Include the number of lists as the fourth number (always previously 1) in the output statistics report.

The supported command line options and expected output are illustrated below:

% ./lab2\_list --threads=10 --iterations=1000 --lists=5 --yield=id --sync=m

List-id-m,10,1000,5,30000,23155406,1157

%

Confirm the correctness of your new implementation:

* Run your program with --yield=id, 4 lists, 1,4,8,12,16 threads, and 1, 2, 4, 8, 16 iterations (and no synchronization) to see how many iterations it takes to reliably fail (and make sure your Makefile expects some of these tests to fail).
* Run your program with --yield=id, 4 lists, 1,4,8,12,16 threads, and 10, 20, 40, 80 iterations, --sync=s and --sync=m to confirm that updates are now properly protected.
* Graph these results (as you did last week) and include the results as lab2b\_3.png.

Now that we believe the partitioned lists implementation works, we can measure its performance.  Rerun both synchronized versions without yields for 1000 iterations, 1,2,4,8,12 threads, and 1,4,8,16 lists.  For each synchronization mechanism, graph the aggregated throughput (total operations per second, as you did for lab2a\_1.png) vs the number of threads, with a separate line for each number of lists.  Call these graphs lab2b\_4.png and lab2b\_5.png

**QUESTION 2.3.4 - Performance of Partitioned Lists**

**Explain the change in performance of the synchronized methods as a function of the number of lists.**

**Should the throughput continue increasing as the number of lists is further increased?  If not, explain why not.**

**It seems reasonable to suggest the throughput of an N-way partitioned list should be equivalent to the throughput of a single list with fewer (1/N) threads.  Does this appear to be true in the above curves?  If not, explain why not.**

**SUBMISSION:**

Your tarball should have a name of the form lab2b-*studentID*.tar.gz and should be submitted via CCLE.

We will test it on a SEASnet GNU/Linux server running RHEL 7 (this is on lnxsrv09). You would be well advised to test your submission on that platform before submitting it.

**CS111 - Project 3A: File System Dump**

**INTRODUCTION:**

Project 3 is expected to involve much more programming than any of the other projects.  You will design and implement programs to analyze file systems and diagnose corruption.  In part A, you will write a program to read the image of a file system, analyze it, and summarize its contents in several csv files. These csv files will be used in part B, where we will analyze the consistency of file system data structures and diagnose problems.

Part A can be broken into two major steps:

1. Mount the provided image file on your own Linux and explore it with *debugfs(8)*.
2. Write a program to analyze the image file and output a summary to six csv files (describing the super block, cylinder groups, free-lists, inodes, indirect blocks, and directories).

**RELATION TO READING AND LECTURES:**

This project more deeply explores the file and directory concepts described in Arpaci chapter 39.

This project is based on the same EXT2 file system that is discussed in sections 40.2-40.5.  Part B of this project goes much deeper than the introductory discussion of integrity presented in sections 42.1-2.

**PROJECT OBJECTIVES:**

* reinforce the basic file system concepts of directory objects, file objects, and free space.
* reinforce the implementation descriptions provided in the text and lecture.
* gain experience with the examining, interpreting, and processing information in binary data structures.
* gain practical experience with complex data structures in general, and on-disk data formats in particular.
* (in 3B) reinforce the notions of consistency/integrity provided in the text and lecture.

**DELIVERABLES:**

A single tarball (.tar.gz) containing:

* a single C source module that compiles cleanly (with no errors or warnings).
* a Makefile to build the program and the tarball.
* a README file describing each of the included files, your UID, and any other information about your submission that you would like to bring to our attention (e.g. limitations, features, testing methodology, etc.).

**PROJECT DESCRIPTION:**

Historically, file systems were almost always been implemented as part of the operating system, running in kernel mode.  Kernel code is expensive to develop, difficult to test, and prone to catastrophic failures.  Within the past 15 years or so, new developments have made it possible to implement file systems in user mode, improving maintainability … and in some cases delivering even better performance than could be achieved with kernel code.  All of this project will be done as user-mode software.

To ensure data security and integrity, disks are generally protected from access by ordinary applications.  Linux supports the creation, mounting, checking, and debugging of file systems stored in ordinary files.  In this project, we will provide EXT2 file system images in ordinary files.  Because they are in ordinary files (rather than protected disks) you can access/operate on those file system images with ordinary user mode code.

To complete this assignment, you will need to learn a few things:

* *debugfs(8)* ([http://man7.org/linux/man-pages/man8/debugfs.8.html](https://www.google.com/url?q=http://man7.org/linux/man-pages/man8/debugfs.8.html&sa=D&ust=1479226705209000&usg=AFQjCNFZ1jhBlNEy3IrPKH4ONR-qWXKIQw))
* *pread(2)*
* csv format
* EXT2 file system ([http://www.nongnu.org/ext2-doc/ext2.html](https://www.google.com/url?q=http://www.nongnu.org/ext2-doc/ext2.html&sa=D&ust=1479226705211000&usg=AFQjCNG_SE-FfIkOhcsZSJA_IIJuA5pPjA))

**Step 1 - study a provided file system image**

Your program will be tested on multiple (broken) file systems. Here we provide one as an example. SEASnet servers do not support commands like*sudo(8), mount(8), or debugfs(8)*, so to play with our provided file system image, you will have to install a Linux distribution (if you do not have one) on your own computer. For simplicity and convenience, you may choose to install Linux inside a virtual machine, for example VirtualBox (it’s free!).

Then download [this image file](https://www.google.com/url?q=http://www.lasr.cs.ucla.edu/classes/111_fall16/labs/lab3A/disk-image&sa=D&ust=1479226705214000&usg=AFQjCNFXZAyYVX0yfeJ2JfSkuvXHmpBGKA), and mount it onto your **own** Linux, with the following commands:

mkdir fs  
sudo mount -o loop disk-image fs  
sudo chown -R *$USER* fs        … where $USER is your user-name

Note:  this file is 200 Mbytes in size.

Now, you can navigate the file system, just like an ordinary directory, with commands like *ls(1)*, *cat(1)*, and *cd(1)*. After you are done with it, you can unmount with the following command:

sudo umount fs

Before you start writing your C program to interpret the “disk-image” file, you can explore it further using *debugfs(8)* (in your own Linux). You may find many useful commands in its man page. Some particularly helpful ones are: *stats*, *stat*, *bd*, *testi*, and *testb*.  If you have problems interpreting parts of the file system, you can use the *debugfs* program to help you understand the correct contents.

To ensure you are correctly interpreting the file system image, we have included many unusual things, which would not likely be caught by an incorrect implementation:

* sparse files
* large files
* allocated data blocks full of zeroes
* unallocated blocks containing valid data
* files with data beyond their length
* files with long names
* files with syntactically strange names
* directories that span multiple blocks and have obsolete entries for deleted files.

**Step 2 - write a program to summarize the file system’s contents**

In this step, you will write a program called **lab3a** that:

* Reads (only) one file system image according to the provided file name. For example, we may run your program with the above file system image using the following command: ./lab3a disk-image
* Analyzes the provided file system image and outputs six csv files to the current directory. The contents of these csv files are described below. Your program **must** output these files with **exactly the same formats** as shown below. We will use sort and diff to compare your csv files with ours, so a different format will make your program fail the test and the content in the output csv file will be treated as error.

Please note that, although you cannot mount the provided image file on departmental servers, your lab3a program should be able to run on departmental servers, just like all previous assignments.

There are to be six csv-format files, each summarizing a different part of the file system.  Remember, you can always check your program’s output against *debugfs*’s output. All the information required for the summary can be manually found and checked by using debugfs.

The file system images your program examines will, in many cases be damaged (as might result from missing or mis-directed writes).  This means that you will have to do some *Defensive Programming* in your analysis program: validating parameters and pointers before you use them.  In addition to correctly analyzing the file system and group parameters, inodes, block pointers, and directory entries, your program will also be expected to report (to stderr) any invalid parameters or addresses.

1. **super block**

A basic set of file system parameters.

|  |  |
| --- | --- |
| **fields(9)** | **format** |
| magic number | hex |
| total number of inodes | dec |
| total number of blocks | dec |
| block size | dec |
| fragment size | dec |
| blocks per group | dec |
| inodes per group | dec |
| fragments per group | dec |
| first data block | dec |

Sample correct csv output

Ef53,51200,204800,1024,1024,8192,2048,8192,1

Sanity checking

* Magic number must be correct
* Block size must be reasonable (e.g. power of two between 512-64K)
* Total blocks and first data block must be consistent with the file size
* Blocks per group must evenly divide into total blocks
* Inodes per group must evenly divide into total inodes

Sample error messages (to stderr):

Superblock - invalid magic: 0xdead

Superblock - invalid block size: 666

Superblock - invalid block count 200000 > image size 50000

Superblock - invalid first block 100000 > image size 50000

Superblock - 20000 blocks, 1050 blocks/group

Superblock - 1000 Inodes, 66 Inodes/group

It is important to validate superblock parameters, because they are used to determine the addresses of inodes and to validate block pointers.  If a file system fails any of these tests, log an error and exit without any further processing.

1. **group descriptor**

Information about each cylinder group

|  |  |
| --- | --- |
| **fields(7)** | **format** |
| number of contained blocks | dec |
| number of free blocks | dec |
| number of free inodes | dec |
| number of directories | dec |
| (free) inode bitmap block | hex |
| (free) block bitmap block | hex |
| inode table (start) block | hex |

Sample output (just for the first cylinder group)

8192,7913,2024,2,4,3,5

Sanity checking

* Number of contained blocks must be consistent with superblock
* Bitmap and inode starting blocks must be within the group

Sample error messages (to stderr):

Group 7: 100000 blocks, superblock says 50000

Group 7: blocks 100000-150000, free Inode map starts at 165000

Group 7: blocks 100000-150000, free block map starts at 165000

Group 7: blocks 100000-150000, Inode table starts at 165000

Group descriptor validation is also critical because this table tells us where the inodes and free bit-maps begin.  If a group descriptor fails any of these tests, log an error, and do not attempt to dump out bit-maps or inodes that are not within the group.

1. **free bitmap entry:**

A list of free inodes and free blocks.

One line of output for each free inode or block in the bitmap.  No output should be produced for allocated inodes or blocks.

|  |  |
| --- | --- |
| **fields(2)** | **format** |
| block number of the map | hex |
| Free block/inode number | dec |

Sample output (just first five entries)

3,275

3,276

3,277

3,278

3,279

Sanity checking

* none … since there are no block numbers within the bit-map, there is no trivial way to tell if the map has been corrupted.

1. **Inode**

Key information for each allocated inode.

An allocated inode has a file type.

|  |  |
| --- | --- |
| **fields(11+15)** | **format** |
| inode number | dec |
| file type | char[[1]](http://web.cs.ucla.edu/classes/winter17/cs111/assign/cs111_project3A.html#ftnt1) |
| mode | oct |
| owner | dec |
| group | dec |
| link count | dec |
| creation time | hex |
| modification time | hex |
| access time | hex |
| file size | dec |
| number of blocks | dec |
| block pointers \* 15 | hex |

Sample output (first four inodes only)

1,?,0,0,0,0,58201197,58201197,58201197,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

2,d,40755,1000,0,5,58201213,58201213,582011bd,3072,3,105,a602,a107,0,0,0,0,0,0,0,0,0,0,0,0

3,?,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

4,?,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

Sanity checking

* All block numbers should be between the first data block number and the file system size (as indicated in the super block)

It is important to validate all block pointers before we use them:

* If the file type is directory, we will fetch each data block and interpret it as directory entries.
* If the file spills over into indirect blocks, we will fetch the indirect blocks and list the pointers.

Sample error message (to stderr):

Inode 407 - invalid block pointer[9]: 3ab403

If any block address falls outside the reasonable range, include it in the output, log an error message.  But if an out-of-range block is an indirect block or contains directory entries, do not attempt to further interpret its contents.

1. **directory entry**

The valid/allocated entries in each directory.

An allocated directory entry can be recognized by a non-zero inode number.  Any directory entry that has a zero inode number should be assumed to be empty and not included in the output.

The term *parent inode number*, below, refers to the inode number of the directory that contains all of the entries that are being listed.

The term *entry number*, below, is a number that starts at 0 for the first directory entry, and is incremented for each entry that is encountered (whether allocated or not).  If the first ten entries are allocated, the next is free, and the last is allocated, the output should describe entries: 0-9, and 11, with no output for entry 10.

|  |  |
| --- | --- |
| **fields(6)** | **format** |
| parent inode number | dec |
| entry number | dec |
| entry length | dec |
| name length | dec |
| inode number of the file entry | dec |
| name | string[[2]](http://web.cs.ucla.edu/classes/winter17/cs111/assign/cs111_project3A.html#ftnt2) |

Sample output (first three directories)

2,0,12,1,2,”.”

2,1,1012,2,2,”..”

2,2,20,10,11,”lost+found”

Sanity checking

* entry length should be reasonable (e.g. 8-1024 bytes) and fit within the file length
* name length should fit within the entry length
* file entry inode number should be within the super-block specified range (number of inodes)

If a group descriptor fails any of these tests, log an error.  If either the name or entry length is unreasonable, stop interpreting this directory:

Sample error message (to stderr):

Inode 204, block 3c05 - bad dirent: Inode = 107538

Inode 204, block 3c05 - bad dirent: len = 3040, namelen = 5240

1. **indirect block entry**

These are all the **non-zero** block pointers in an indirect block. The blocks that contain indirect block pointers are included.

|  |  |
| --- | --- |
| **fields(3)** | **format** |
| block number of the containing block | hex |
| entry number within that block | dec |
| block pointer value | hex |

Sample output (first three entries)

1e10f,0,1e110

1e10f,1,1e111

1e10f,2,1e112

Sanity checking

* all block numbers should be legal (within super-block specified range)

It is important to validate all block numbers in indirect blocks for the same reason we had to validate block numbers in inodes: because we will follow those pointers to further interpret the file system.

Sample error message to stderr:

Indirect block 3c4 - invalid entry[103] = 304c8df

If any block number is out of range log an error message.  If a out-of-range block is an indirect block, do not attempt to dump its contents.

**Output Files**

For each different kind of entry, output its summary in a separate file. The names of your files should be:

* super.csv                ...for super block;
* group.csv                ...for group descriptors;
* bitmap.csv                 ...for free inodes and blocks;
* inode.csv                ...for inodes;
* directory.csv                ...for directory entries; and
* indirect.csv                ...for indirect block entries.

In these csv files, each line represents one entry, and should be ended with a single new-line character ( ‘\n’, 0x0a).  The field orders in those csv files should be the same as listed above, and your program should **use and only use** a comma to separate each field.

For your convenience, here are the exact six csv files generated by our solution: [super.csv](https://www.google.com/url?q=http://www.lasr.cs.ucla.edu/classes/111_fall16/labs/lab3A/super.csv&sa=D&ust=1479226705314000&usg=AFQjCNGIfSrdMB--6YigM6WSUCQKgKOEIw), [group.csv](https://www.google.com/url?q=http://www.lasr.cs.ucla.edu/classes/111_fall16/labs/lab3A/group.csv&sa=D&ust=1479226705315000&usg=AFQjCNEnBReiBr7Yni4r314cN_shdc2VVw), [bitmap.csv](https://www.google.com/url?q=http://www.lasr.cs.ucla.edu/classes/111_fall16/labs/lab3A/bitmap.csv&sa=D&ust=1479226705315000&usg=AFQjCNGXpDinuD_1g9eXtKT0IZWGlM7NdQ),[inode.csv](https://www.google.com/url?q=http://www.lasr.cs.ucla.edu/classes/111_fall16/labs/lab3A/inode.csv&sa=D&ust=1479226705316000&usg=AFQjCNH4BMklofpXPoJl5wbpuO0fyFfMOg), [directory.csv](https://www.google.com/url?q=http://www.lasr.cs.ucla.edu/classes/111_fall16/labs/lab3A/directory.csv&sa=D&ust=1479226705317000&usg=AFQjCNH75U-S13JSA60QDFxWYSLKefx7Dg), and [indirect.csv](https://www.google.com/url?q=http://www.lasr.cs.ucla.edu/classes/111_fall16/labs/lab3A/indirect.csv&sa=D&ust=1479226705317000&usg=AFQjCNGbzX0C0uMpgA1BQ9lgsOblM68eMg).

**SUBMISSION:**

Please carefully check the Rubric section (especially the **Code review** subsection) and **all (latest) footnotes** before submitting your code.

Your tarball should have a name of the form lab3a-*studentID*.tar.gz and should be submitted via CCLE.

We will test your work on a SEASnet GNU/Linux server running RHEL 7 (this is on lnxsrv09). You would be well advised to test your submission on that platform before submitting it.

**CS111 - Project 3B: File System Analysis**

**INTRODUCTION:**

Project 3 is expected to be the most difficult project, where you will develop programs to analyze file systems and diagnose corruption. In part A, you produced a program to read in a file system image, analyze it, and summarize its contents in several csv files. In part B, you will write a program to analyze these csv files to diagnose the problems in the provided file system image.

Unlike previous assignments, you have the freedom to use whatever programming language you’d like, as long as that programming language is supported by SEASnet servers.

We will use some specialized software to detect cheating for Project 3.

**RELATION TO READING AND LECTURES:**

This project more deeply explores the file and directory concepts described in chapter 39.

This project is based on the same EXT2 file system that is discussed in sections 40.2-40.5.

This project goes much deeper than the introductory discussion of integrity in sections 42.1-2.

**PROJECT OBJECTIVES:**

* reinforce the basic file system concepts of directory objects, file objects, and free space.
* reinforce the implementation descriptions provided in the text and lecture.
* gain experience with the examining, interpreting, and processing information in binary data structures.
* gain practical experience with complex data structures in general, and on-disk data formats in particular.
* reinforce the notions of consistency/integrity provided in the text and lecture.

**DELIVERABLES:**

A single tarball (.tar.gz) containing:

* the source code for Project 3B.
* a Makefile to build the tarball (and compile the code).
* a README file describing each of the included files, your UID, the shell command to (compile and) execute your program, and any other information about your submission that you would like to bring to our attention (e.g. limitations, features, testing methodology, etc.).

**PROJECT DESCRIPTION:**

In Project 3B, you will write a program called **lab3b** that:

* Reads in the six csv files your lab3a program produced in Part A from the current directory, checks for certain file system corruptions listed in detail below, and outputs the error reports to a file called “lab3b\_check.txt” to current directory.

You can use whatever programming language you prefer, as long as that programming language is supported by SEASnet server lnxsrv09. Please note that we will not install any programming language or libraries for you; you can only use the available programming languages and libraries on lnxsrv09. The shell command to (compile and) execute your program should be provided in both your README file and in Makefile (see the paragraph before Submission section for more detail). For example, if you use Python[[1]](http://web.cs.ucla.edu/classes/winter17/cs111/assign/cs111_project3B.html#ftnt1), your filename should be “lab3b.py”, and you should provide a shell command like “python lab3b.py”.

We will test your lab3b program on our own csv files, so bugs/errors in your lab3a program would not affect your Part B score. When testing your code, we will put the six csv files into the same directory as your program, and after executing the shell command provided by you, there should be a file called “lab3b\_check.txt” in the same directory. We will use *sort* and *diff* to compare this file with ours. Failing to automatically read in the six csv files or generate the error report .txt file will automatically result in a zero for this project.

Here is the list of errors your lab3b program should check and the corresponding error report format, noting that the particular numbers shown below are merely examples, and do not correspond to numbers in the actual assigned file system:

1. **unallocated block**: blocks that are in use but also listed on the free bitmap. Here the INODEs should be listed in increasing order of the inode\_num.

UNALLOCATED BLOCK < block\_num > REFERENCED BY (INODE < inode\_num > (INDIRECT BLOCK < block\_num>) ENTRY < entry\_num >) \* n

For example,

UNALLOCATED BLOCK < 1035 > REFERENCED BY INODE < 16 > ENTRY < 0 > INODE < 17 > INDIRECT BLOCK < 10 > ENTRY < 0 >

1. **duplicately allocated block**: blocks that are used by more than one inodes. Here the INODEs should be listed in increasing order of the inode\_num.

MULTIPLY REFERENCED BLOCK < block\_num > BY (INODE < inode\_num > (INDIRECT BLOCK < block\_num>) ENTRY < entry\_num >) \* n

For example,

MULTIPLY REFERENCED BLOCK < 613 > BY INODE < 24 > ENTRY < 0 > INODE < 25 > ENTRY < 0 > INODE < 26 > ENTRY < 0 >

1. **unallocated inode**: inodes that are in use by directory entries (the inode number of the file entry field) but not shown up in inode.csv. Here the DIRECTORYs should be listed in increasing order of the inode\_num.

UNALLOCATED INODE < inode\_num > REFERENCED BY (DIRECTORY < inode\_num > ENTRY < entry\_num >) \* n

For example,

UNALLOCATED INODE < 21 > REFERENCED BY DIRECTORY < 2 > ENTRY < 12 >

1. **missing inode**: inodes that are not in use, and not listed on the free bitmap.

MISSING INODE < inode\_num > SHOULD BE IN FREE LIST < block\_num >

For example,

MISSING INODE < 34 > SHOULD BE IN FREE LIST < 4 >

1. **incorrect link count**: inodes whose link counts do not reflect the number of directory entries that point to them.

LINKCOUNT < inode\_num > IS < link\_count > SHOULD BE < link\_count >

For example,

LINKCOUNT < 1714 > IS < 3 > SHOULD BE < 2 >

1. **incorrect directory entry**: the '.' and '..' entries that don't link to correct inodes.

INCORRECT ENTRY IN < inode\_num > NAME < entry\_name > LINK TO < inode\_num > SHOULD BE < inode\_num >

For example,

INCORRECT ENTRY IN < 1714 > NAME < . > LINK TO < 1713 > SHOULD BE < 1714 >

1. **invalid block pointer**: block pointer that has an invalid block number.

INVALID BLOCK < block\_num > IN INODE < inode\_num > (INDIRECT BLOCK < block\_num >) ENTRY < entry\_num >

For example,

INVALID BLOCK < 1 > IN INODE < 2 > INDIRECT BLOCK < 3 > ENTRY < 4 >

or

INVALID BLOCK < 1 > IN INODE < 2 > ENTRY < 4 >

In “lab3b\_check.txt”, each line represents one error, and should be ended with a single new-line character ( ‘\n’, 0x0a). Please pay attention to the spaces between words, numbers, and symbols. Incorrect formatting in your “lab3b\_check.txt” will be treated as error content.

For your convenience, here is the “lab3b\_check.txt” generated by our solution: [lab3b\_check.txt](https://www.google.com/url?q=https://lasr.cs.ucla.edu/classes/111_fall16/labs/lab3B/lab3b_check.txt&sa=D&ust=1479487806327000&usg=AFQjCNGPXaO6muKXOKuPCzS3lGENsfoK5g). Here are the six csv files we used in Project 3A: [super.csv](https://www.google.com/url?q=https://lasr.cs.ucla.edu/classes/111_fall16/labs/lab3A/super.csv&sa=D&ust=1479487806328000&usg=AFQjCNEfrQMeeI1kj91QwtpiHr2zQA3npg), [group.csv](https://www.google.com/url?q=https://lasr.cs.ucla.edu/classes/111_fall16/labs/lab3A/group.csv&sa=D&ust=1479487806328000&usg=AFQjCNHqt7Ylc6c3IzjXgwO_BVfkLlAoEQ), [bitmap.csv](https://www.google.com/url?q=https://lasr.cs.ucla.edu/classes/111_fall16/labs/lab3A/bitmap.csv&sa=D&ust=1479487806329000&usg=AFQjCNGWZs_tjpZLN3rYNf4h7N7oVZewPQ), [inode.csv](https://www.google.com/url?q=https://lasr.cs.ucla.edu/classes/111_fall16/labs/lab3A/inode.csv&sa=D&ust=1479487806329000&usg=AFQjCNHID-LA3x_QmdyyF7j6mV-45WBBuQ), [directory.csv](https://www.google.com/url?q=https://lasr.cs.ucla.edu/classes/111_fall16/labs/lab3A/directory.csv&sa=D&ust=1479487806330000&usg=AFQjCNGIFPHRdhsZdsd-_SFxsZO-ONNUVw), and [indirect.csv](https://www.google.com/url?q=https://lasr.cs.ucla.edu/classes/111_fall16/labs/lab3A/indirect.csv&sa=D&ust=1479487806330000&usg=AFQjCNHpixlNy3N1u2ZCcZDXNhZ8RKUg-Q).

In your Makefile, the default action should compile your code (if you have to do so). Also, we will use “**make run**” to execute your program, so please make sure your Makefile does support this.

**SUBMISSION:**

Project 3B is due before midnight on **Monday, November 28**, 2016.

Your tarball should have a name of the form lab3b-*studentID*.tar.gz and should be submitted via CCLE. If you did this project with another student, then either one’s UID is acceptable. A team may **only** submit one tarball. We will deduct penalty points for double submission.

We will test your work on a SEASnet GNU/Linux server running RHEL 7 (this is on lnxsrv09). You would be well advised to test your submission on that platform before submitting it.

**Project 4: Embedded Systems**

PROJECT GOALS

This project will familiarize you with some simple distributed systems characteristics and tools.  It will also give you some experience in working with hardware for embedded systems and provide some exposure to basic tools for securing distributed systems.  The goals are:

* primary: build a simple client end of a client/server distributed system.
* primary: learn about working within the constraints of devices designed to support embedded systems.
* primary: learn how to use standard tools to provide secure encrypted communications between a client and server
* secondary: build a program to a specified protocol interface.
* secondary: obtain experience working with simple peripheral devices.
* secondary: obtain experience with simple networking debugging.

ASSIGNMENT OVERVIEW

The assignment is divided into three general parts.

1. Building an application that supports the use of a sensor to gather data on an embedded device.
2. Convert the application that interacts with the sensor to become a client using a predefined network protocol to interact with a remote server program.
3. Change the basic client application to make use of SSL/TLS to communicate securely to a remote server that requires cryptographic protection of communications.

In this assignment, you will:

* Learn how to perform basic operations on the Intel Edison.
* Learn how to connect simple sensor devices to the Edison and access them from a local application.
* Implement and demonstrate an application that uses sockets to communicate with a shared server application on a remote machine.
* Learn how to convert socket communications to use SSL/TLS to provide cryptographic protection of communications.

Your deliverables for this assignment will include:

* A program that integrates a sensor into the basic Edison platform and gathers readings from that sensor.
* A test run in which your program interacts with the remote server and performs all supported operations specified in the design for the application.  This test run will store a log file on the remote server machine, which will be used in grading your assignment.  You will also provide a log file from your Edison of the run.
* A test run in which the SSL/TLS version of the program interacts with the secure remote server and performs all supported operations specified in the design for the application.  This test run will store a log file on the remote server machine, which will be used in grading your assignment.  You will also provide a log file from your Edison of the run.

To perform this assignment, you will need to learn about the Edison platform.  There are several useful tutorials on working with the Edison that you might find helpful available on line:

[Edison tutorials](https://www.google.com/url?q=https://lasr.cs.ucla.edu/classes/edison_tutorials/&sa=D&ust=1488328238413000&usg=AFQjCNEpqaCnnp1rE6xpDGuC35ckPgDUnw)

Also, you will need to use a temperature sensor in the Grove sensor kit for this project.  Here’s a link that provides some information on using this sensor:

[Temperature sensor information](https://www.google.com/url?q=http://wiki.seeed.cc/Grove-Temperature_Sensor_V1.2/&sa=D&ust=1488328238415000&usg=AFQjCNE-IJb1HpxnoCOYmxVyYKTgGCvL4w)

PART 1 - Building a sample Edison embedded device

Summary of Deliverables

* the source for a C program and *Makefile* that cleanly (no warnings) builds using *gcc* on an Edison Linux system, implementing the functionality specified below.
* The contents of a log file showing the program operating on your Edison for at least 60 seconds.

Detailed Instructions

Write a program that uses the Edison to access the temperature sensor included in the Grove sensor kit.  The program should read the sensor once per second and output its reading (in Fahrenheit) to a shell attached to the serial port, as the tutorials indicate.  Also output these readings to a log file, in the format:

Timestamp        Temperature

(Use a space between the timestamp and temperature, not a tab or multiple spaces.)  The timestamp should be obtained by running the time() system call on the Edison, and should be converted to an HH:MIN:SEC format.  The temperature should be in the format ##.#.  (For example, 98.6.)  Take measures to ensure that you measure several different temperatures.  (Hint: holding your finger on the sensor is likely to produce a hotter reading than the air temperature.)

PART 2 - Integrate your Edison sensor device into a client/server system

Summary of Deliverables

* the source for a C module and *Makefile* that cleanly (no warnings) builds using *gcc* on an Edison system and implements the functionality specified below.
* A log file showing the temperatures your device sent to the server and all commands received from the server.

Detailed Instructions

The behavior of the basic Edison client and its interactions with the server can be found in a document on the class web page.

PART 3 - Convert your Edison sensor client program to use SSL/TLS to protect communications

Summary of Deliverables

* the source for a C module and *Makefile* that cleanly (no warnings) builds using *gcc* on an Edison system and implements the functionality specified below.
* A log file showing the temperatures your device sent to the server and all commands received from the server.

Detailed Instructions

The behavior of the SSL/TLS Edison client and its interactions with the server can be found in a document on the class web page.