Topics for Exam #2

CSE-381: Systems 2

|  |  |
| --- | --- |
|  | As per the syllabus, Exam #2 for CSE-381: Systems 2 is scheduled    **Thu, Nov 14 (Time: 7 PM to 9 PM)**  **In Hug 141 (Hughes hall)** |

|  |
| --- |
| The topics enumerated below are a succinct list of major concepts that you are expected to know for Exam 2. It may not include all details covered in lectures and labs. Consequently, this is not an exhaustive list and you must use in-class exercises, lab exercises, and homework assignments in addition to these topics to fully prepare for the exam. Particularly this is the time to review the notes you have been diligently recording during lectures.  The exam will be closed notes and closed books. A one-sheet handout on common methods and Linux commands will be supplied along with the exam -- see CommonMethodsAndCommands.pdf. No other reference materials or discussions will be permitted. Use of electronic equipment (other than those required for life support and have suitable doctor certification along with them) other than calculators (calculator must not have a qwerty keyboard or a graphical stylus input pad) is strictly prohibited.  ***Type of questions***:  Same general format as most exams: Multiple-choice, fill in the blanks, short answer, “what does the following C++/assembly code do”, and “write the C++/assembly code to do the following" types of questions. Specifically, you will be expected to read, comprehend, analyze, troubleshoot, and develop programs involving concepts covered in the course. Programs will involve suitable object oriented concepts and standard library concepts covered in the course. |

**Text coverage:**

E-book titled “Operating System Concepts" -- Link in Syllabus page on Canvas (all students have free access to the electronic book):

* + Chapter 4: Threads
  + Chapter 6: Synchronization
  + Chapter 10: File system interface
  + Chapter 11: File system implementation

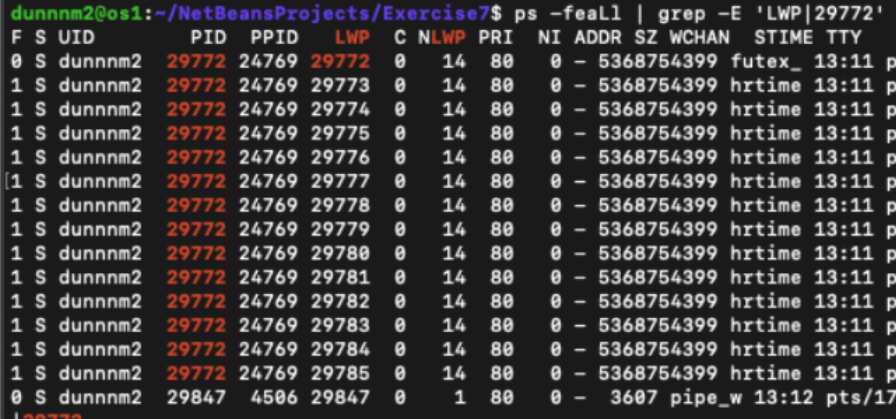
* + **Basics of Managing threads in Linux**

1. Basics of process & thread management and process hierarchies.
2. Viewing threads in Linux using ps -fealL
   1. Interpreting output from ps -fealL

Important notes: -f, -e, -a, take care of displaying

format, every process, and all processes respectively.

-L displays all thread processes.



1. Starting and killing threads via Linux terminal.

We create all our threads in C++ with the std::thread() function.

Killing threads is done by executing kill<tid> where tid equals the LWP column from above. Note, This will kill all threads for a given process.

1. Setting thread priorities and nice values.

Process priorties in Linux range from 0 (high priority) to 99 (low priority).

Niceness values are an additional modifier given to processes and threads to determine run order. They range -20(most favorable to run) to 19 (lowest priority).

In ps, the Niceness value is determined by the NI column, the PRI column displays priority.

Niceness can be changed by doing renice <niceValue> <tid/LWP>

1. Process IDs, Parent process IDs, and Monitoring processes using ps command.

Discussed ad nauseum previously

* + **Basics of Linux operations**

1. Creating and navigating directories via Linux terminal.

mkdir

1. Creating, copying, and deleting files using commands in a Linux terminal.

scp <from> <to>

1. Basics of compiling and running C/C++ program in Linux in a Linux terminal.
2. Foreground vs. background processes in Linux.
   * **Threads and multithreading (Chapter 4)**
3. Concurrency and multi-tasking.

Concurrency/Multi-tasking: We are doing multiple things at the same time to accomplish an overall process faster.

1. Concept of multithreading.

We create multiple “workers” in the form of threads, all of whom take portions of the work to do.

1. Concept of a thread. Single vs. multi-threaded processes.

A thread is a partition of one of the cores of our CPUs. A single threaded process is only capable of accomplishing a single task at a time, which is done exactly in the order it is stated to be accomplished.

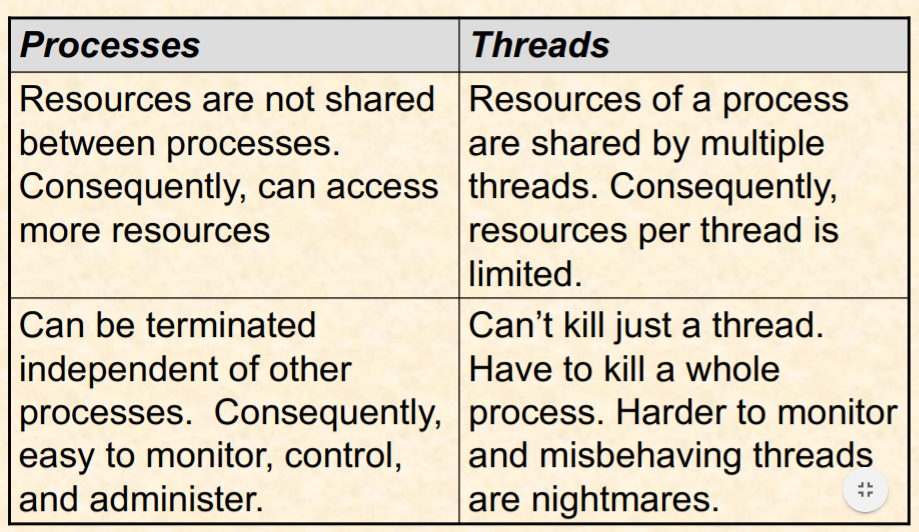
In Multi-threaded processes, we can utilize more of our CPU, enabling multiple tasks to be accomplished in a faster amount of time.

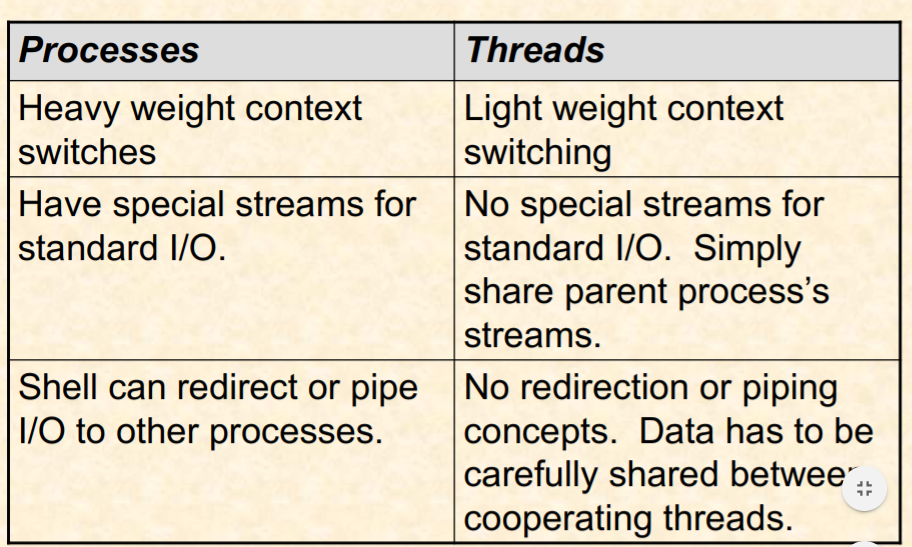
1. Resources shared between threads versus resources available only to a thread.

Resources that are created in the scope of a thread are called “thread-safe” and cannot be accessed by other threads. Shared resources, which are passed into several thread declarations can be manipulated by many threads, and are not thread safe.

1. Processes vs. Threads --advantages vs. disadvantages.

Some key advantages to Processes are that we can abort individual processes and have a great deal of administrative control over which processes can run at a given time. If we kill one thread, we kill them all. In general it is better to use processes when we want independent control over what is running, especially if there may be malicious activity.





1. Thread lifecycle (same as process life cycle):

New→Ready→Running→Waiting→Terminated

Most of these are self explanatory, threads often wait in join() call, and are often

terminated following a join() call.

1. Creating threads in C++

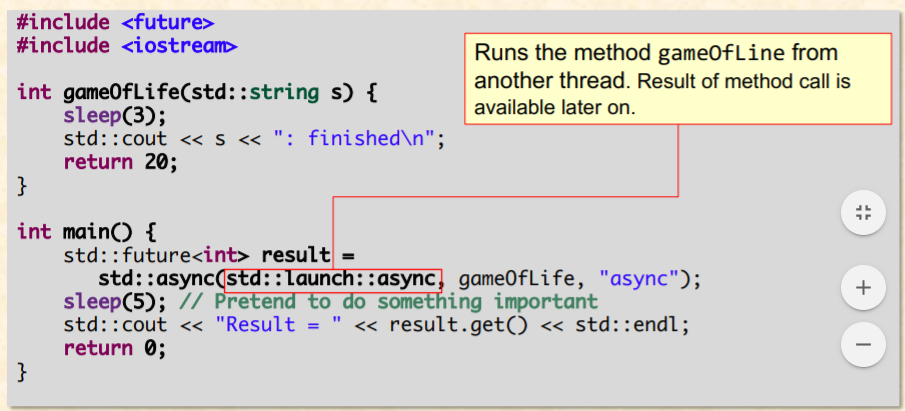
std::thread(threadMethodName, arg1, arg2, ..)

remember, don’t use parenthesis for the method name

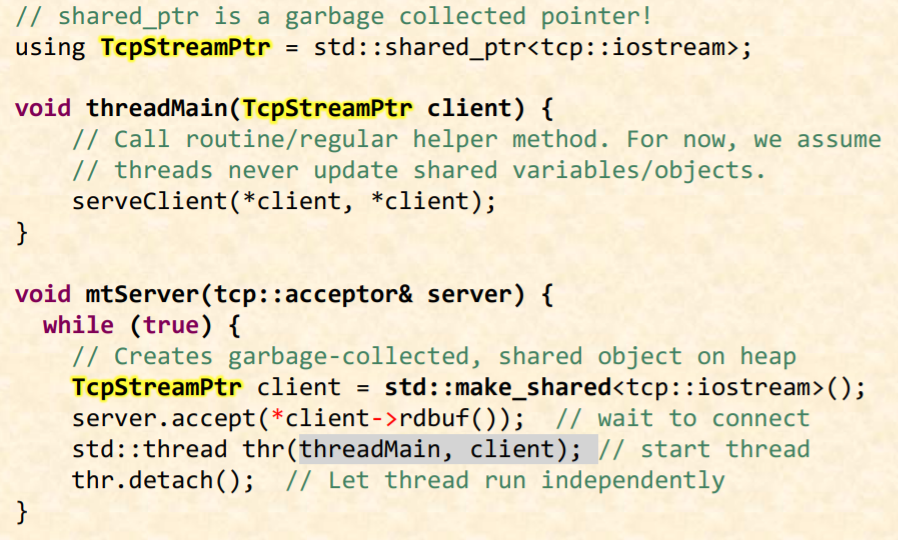
1. Foreground vs. background (detached) threads

We have more manipulation control over foreground threads in the form of std::thread.join(), unlike detached threads which we don’t care about grabbing anything from or manipulating once they are loose

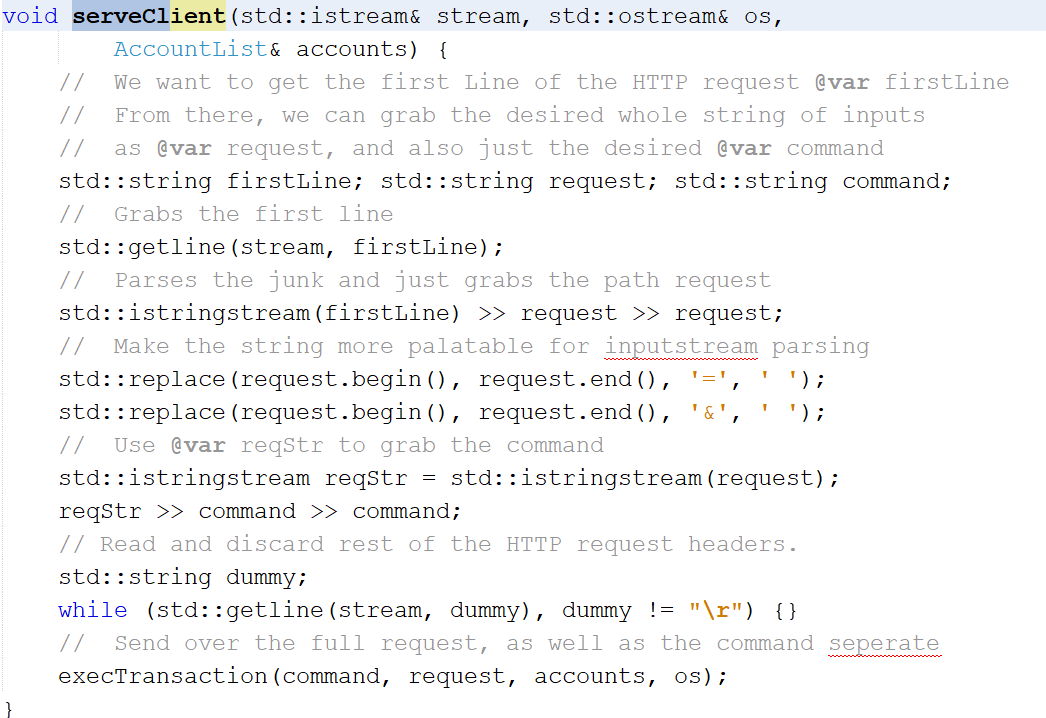
1. Developing synchronization-free multithreaded programs in C++
   1. One thread per task (using std::async)

4

* 1. Multithreaded web-server



* 1. Multithreaded client



* 1. Data parallel program -- several items per thread.

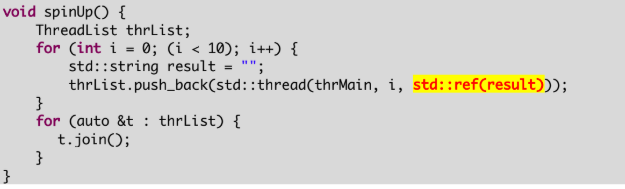
We can only make one initial call per thread, so we need to pass in enough information to the initial method to be able to handle several tasks per one thread.

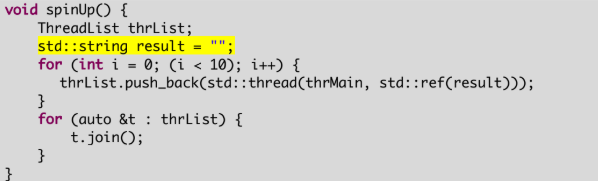
* + **Synchronization (Chapter 6)**

1. Race conditions -- identifying and demonstrating race conditions

Race conditions are any piece of data that we attempt to manipulate in more than one place at the same time. They are hard to find occasionally, and they often don’t produce the same results multiple times in a row.

1. Identification of code snippets with race conditions / incorrect multi-threading.





1. Symptoms of race conditions.

Address Sanitization errors, garbled input output, values that don’t make any sense.

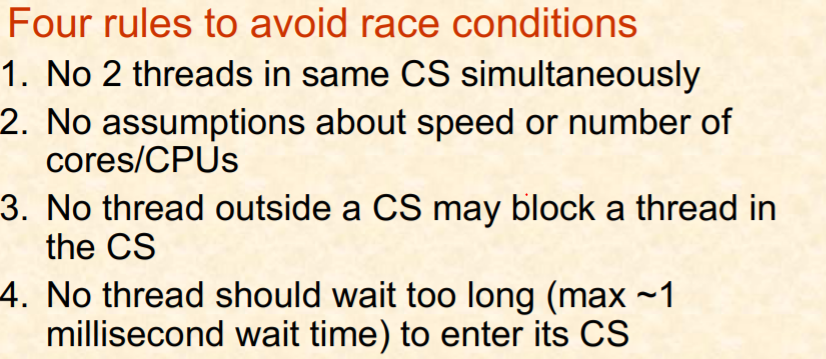
1. Need for synchronization.

We require synchronous behavior for ordered output, and also for any manipulation of shared resources.

1. Concept of a critical section.

These are sections of the code that we are required to not only enforce fully synchronous behavior, but we also need to lock an area of the code to only one thread

1. **4 Rules to create critical sections.**

****

1. Concept of a Semaphore and mutex

A semaphore is a shared counter managed by the OS, it ensures that changes to it's value are thread safe. Threads check the semaphore before operating.

1. Using std::mutex to create a critical section

A mutex is a binary semaphore, used with the std::lock\_guard() method in order to enable automatic locking + unlocking outside a scope.

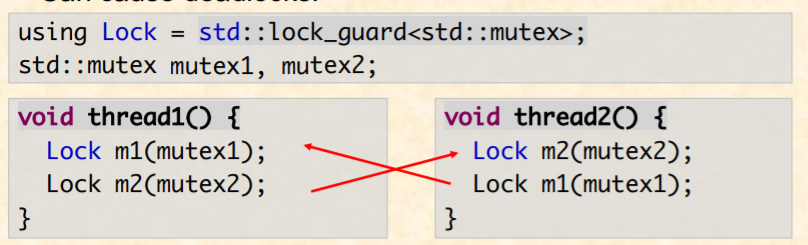
1. Need and use of a std::lock\_guard with std::mutex

Lock\_guard automatically locks upon creation and automatically unlocks at the conclusion of a scope.

1. Identification of critical sections in a code fragment.

Critical sections should involving access a shared piece of memory, should grab/set what is needed, and get out. NO I/O or data processing should be done if possible in a critical section

1. Concept of deadlocks
   1. Locking multiple mutexes using std::lock to avoid deadlocks

Don’t do this

1. Priority inversion

This comes up when threads have different priorities, higher priority thread is waiting on a critical section locked by a lower priority thread, OS keeps scheduling the higher priority thread, so the lower priority thread does not get a chance to unlock the critical section.

1. Producer-consumer multithreading model with fixed size shared queue.
   1. Busy-wait/spin-lock approach -- advantages vs. disadvantages

Producer-consumer model: We fix 1 or more threads to be the producer and create

requests for what we need processed. We then fix 1 or more threads to be the

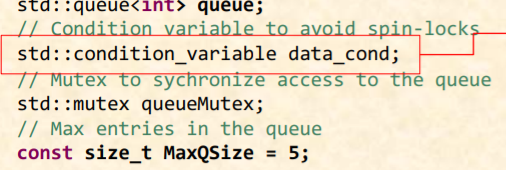
consumers and take those requests and do the processing.

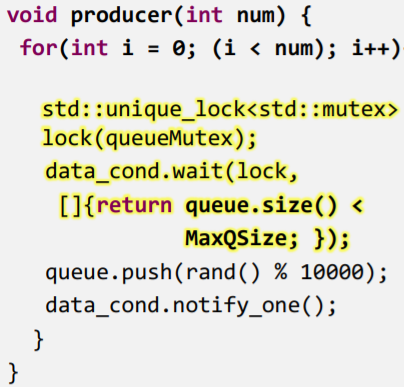
Spin-Locks are incredibly quick and dirty approaches to the locking problem, but

they hog the CPU’s resources for longer than is usually desired or needed.

1. Using Monitors (or condition variables) to avoid busy waiting
   1. Using std::condition\_variable

Monitor/Condition Variables keep a critical section unlocked from the CPU until a condition is met.





* 1. Understanding wait-notify

std::condition\_variable.notify\_one() will flag an immediate check to any resource using that variable, so the program knows to do a check.

* 1. Advantages vs. disadvantages over busy-wait

Adv: Much better for the CPU, puts more power in the programmer’s hands.

Dis: Requires significantly more overhead, requires knowledge of the architecture of all the necessary halting conditions.

1. Using std::async for multithreading

Use in cases where we need one shot, small data operations that don’t share information, code addressed above.

1. Using std::atomic for MT-Safe operations on primitive data types

Creating std::atomic<type> x(10); gives us thread safe variables

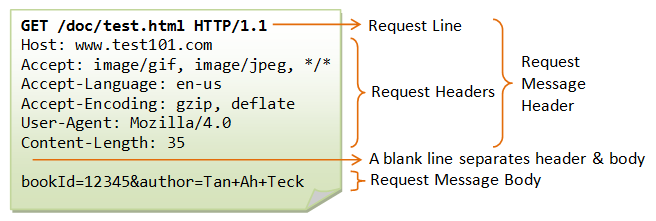
* + **Networking -- Concepts related to WWW and HTTP**

1. Terminology and acronyms
2. Concept of a protocol

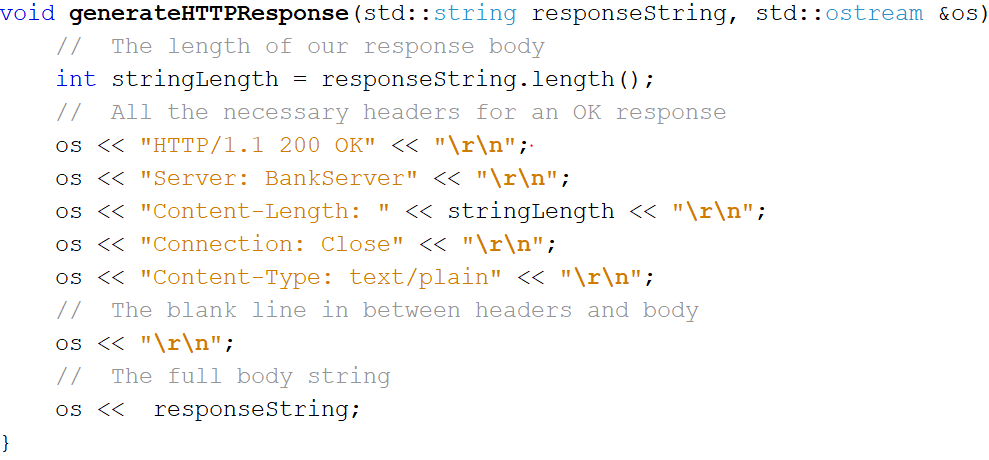
A set of systematic instructions between a client and a server program that is agreed upon by a particular board.

1. Basics of HTTP protocol and line endings "\r\n"

Standard HTTP Request:



Standard HTTP Response:



1. Basic structure of GET requests
   1. URL encoding & decoding

Only important line is the first line of the header

FORMAT:

GET /a/b/c.html HTTP/1.1

Occasionally

GET /a/b/c?param1=arg1&param2=arg2&param3=arg3 HTTP/1.1

1. HTTP headers
   1. HTTP response headers
   2. Basic content types in HTTP response

text/plain, text/html, text/json

* 1. Concept of MIME type in HTTP response

This enables the server to know the format of whatever we are sending them

1. Basics of system integration via fork & exec

We have to fork a process off before we execute it

1. Identifying parameters from an HTML form

Addressed above

* + **File systems (Chapter 10, 11)**

1. Need for a file system

-Ease typical user operations

-Store things efficiently

-Provide fault tolerance

-Infrastructure for security and privacy

1. Relative and absolute paths

Relative paths route from the current directory, never start with a /

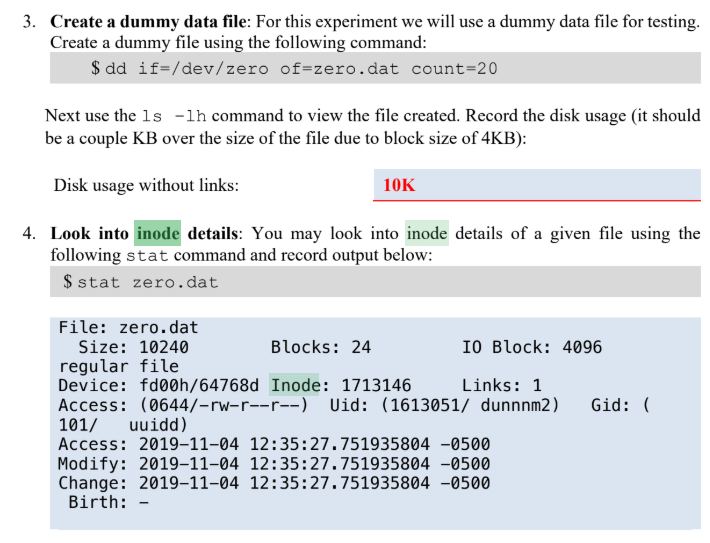
Absolute paths start from base directory and always begin with a /

1. Motivation and use of key data structures in a file system
   1. Root directory and use of inodes

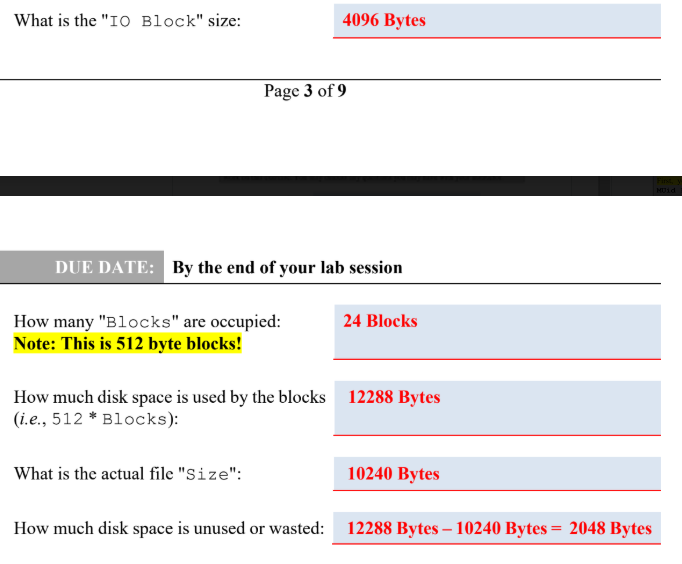
The root directory is a “table” of all other directory that handles data partitioning. The inodes are all the root blocks of data, in which all other files are stored.

* 1. Inspecting inode information and interpreting output of fstat command

Getting the data



Examining the data:



* 1. Tracing File chains -- example: File Allocation Table (FAT). See lab exercise for example of tracing file blocks

Look at Exercise10

1. Relative and absolute path
   * Conversion between relative ⇔ absolute path

Add a bunch of ../../ until we hit root

1. Links
   1. Links (or hard links) -- see ln command

Links or hard links associates a name with a file or a file system

Directories are a special kind of file that can be hard linked. It is used

in conjunction with file systems that allow multiple hard links to be created for the same file.

* 1. Symbolic or soft links -- see ln -s command

These are references to files or directories in the form of absolute/relative

paths

* 1. Influence on file sizes and use of disk storage

Inodes are clustered in large blocks of data, Hard Links + soft links do not significantly impact disk storage until we start throwing a large number in, taking up multiple INodes blocks

1. Security and Privacy:
   1. File permissions -- using chmod to assign users, group members, and other users different read, write, and execute permissions to manage privacy.

chmod u+rwx, g+rx-w, o+r-xw

**C/C++ programming concepts:**

* + **Basic program constructs**

1. Stages in compiling a C++ program.
2. Variables & expressions
3. Constant variables vs. literal constants
4. Signed vs. unsigned data types
5. if and if-else statements
6. switch statement
7. Looping constructs (for, while, do-while, range-for)
8. Basic mathematical problem solving concepts

* Deciding number is even/odd, positive/negative, factor/divisor/dividend/quotient
* Using division and modulo operations for basic number manipulation, e.g.: reverse a number with loops & math (without using string)
* Detecting if a number is prime.
* Identifying largest/smallest number in a set of inputs
* Finding average (i.e., mean) of a given set of numbers

1. Functions/methods
   * + - Pass by value versus pass by reference
         1. Preferred approach for primitive data types vs. objects
       - Memory/copy impact of pass-by-value
       - Using const keyword for parameters.
2. Default values for parameters
   * **Basics of objects**
3. Differences between primitive and object data types in C++
4. Calling methods on objects (e.g.: string::length)
5. Using std::string
   * + - Constructors for string.
       - String comparisons
       - Methods for operating and accessing strings
       - Conversion to-and-from numeric data types to std::string.
       - Formatting strings into HTML, given HTML tags to use (you don't need to know HTML)

* + **Arrays**

1. Basics of old-style arrays.
2. 1-D arrays
3. 2-D arrays
4. Command-line arguments

argc = number of command line args - 1

argv[0] = name of the executable, the remainder are the command line

args.

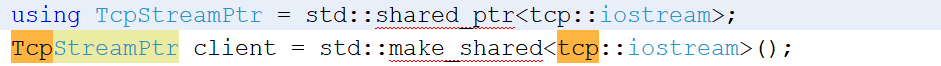
* + - * Designing programs that use command-line arguments
      * Figuring out what and how many command-line arguments a program ought to take.
      * When to prefer command-line arguments instead of reading data from files/console.
  + **Basics of Pointers**

1. Concept of memory and address
2. Basics of pointers to hold addresses
3. Basic pointer operators
   * + - Address of operator (&)
       - Indirections/dereferencing a pointer (\*)
       - Using object dereference operator (->)
4. Pointer arithmetic
5. Pointers ↔ array operation similarities and code conversion
6. Understanding command-line arguments

* Arrays of pointers

1. Using shared\_ptr in lieu of pointers

Shared Pointers do automatic garbage collection



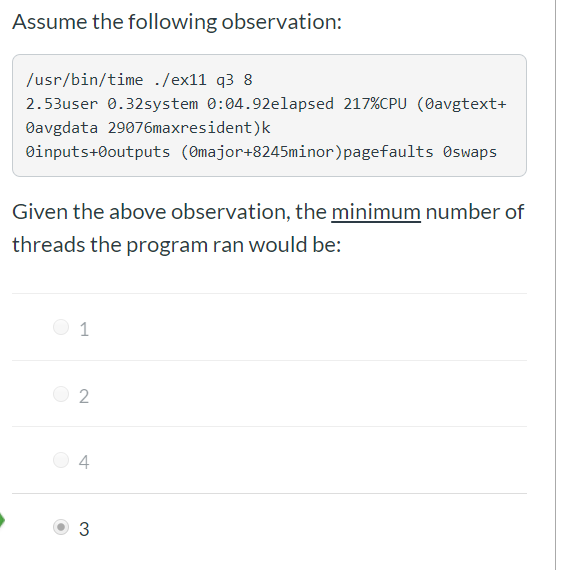
* + **Vectors**

1. Use of vectors instead of arrays for processing data
2. Differences between vectors and arrays
3. Defining and using vectors of different data types
4. Using vectors in method definitions and method calls
5. Create type aliases via the using clause in C++
   * + - Creating aliases given English description
       - Tracing aliases back to their original types.
6. Operations on a vector: adding elements, accessing elements, removing elements, etc.
7. Reading/printing/writing vectors to I/O streams
8. Vectors of user-defined classes
   * **Hash maps (unordered\_map)**
9. Concept of unordered\_map
10. Using unordered\_map as associative arrays
11. Defining and using unordered\_maps of different data types
12. Looking-up values in unordered\_maps
13. Iterating over all the entries in a map and processing them
14. Reading/printing/writing vectors to I/O streams
15. Maps of user-defined classes
    * **Basic text file I/O operations**
16. Reading and writing data to console using std::cin and std::cout.
17. Using stream-insertion (<<) and stream-extraction (>>) operators to read and write data.
    * + - Understanding these operators and how they handle whitespaces.
18. Using std::getline method to read a full line of text
19. Using std::ifstream and std::ofstream to read/write text files.
20. Using std::istringstream and std::ostringstream to perform I/O with strings.
    * **Other exercises**
21. Converting English statements to corresponding C++ statements
22. Describing C++ statements in English
23. Code walkthroughs to determine operation and output from a C++ program
24. Developing a C++ program given a functional description
25. Identifying performance or memory issues in C++ programs
26. Rewriting C++ program to address memory or performance issue
    * **Linux commands and shell**
27. Basic operations at the shell prompt
    * Navigating directory structures
    * Listing files
    * Copying files -- **including using scp**

**scp <from> <to>**

* + Troubleshooting common problems given error message(s)

1. Compiling and running programs
2. Using pipes to create ad hoc software pipelines
   * Redirection to create (>) or append (>>) to existing files
   * Redirection to supply input from a file (<)
   * Using pipe (|) to create software pipelines
3. Using /usr/bin/time to measure runtime characteristics of programs
   * **Elapsed time, %CPU**
4. Interpreting output of /usr/bin/time for single vs. multithreaded programs



**Preparation Suggestions:**

1. As a general note you should expect to repeat questions from lab exercises and homework.
2. You should know all the material in lecture slides.
3. Do read the E-book materials used in homework while paying attention to implementation/application details.
4. Redo lab exercises. Develop short programs to test/verify your understanding of concepts. Review developing classes and overloading operators. Review how to call overloaded operators. Review vectors, how to use vectors. Review unordered\_map and how to use it.
5. Review homework solutions on Canvas.
6. Review the functionality of pertinent methods and commands in the supplied method/command sheet.
7. Review the handouts material and videos on Canvas.