Process Monitor

REVIEW | CODE REVIEW 24 | HISTORY

▾ src/linux_parser.cpp 14

```
#include "linux_parser.h"

#include <dirent.h>
#include <unistd.h>

#include <iostream>
#include <sstream>
#include <string>
#include <vector>

using std::stof;
using std::string;
using std::to_string;
using std::vector;

```

SUGGESTION

YOU CAN ALSO DEFINE SOME GENERIC FUNCTIONS AS WELL AS UTILITY FUNCTIONS WHICH WILL HELP YOU MINIMISE REPETITIVE CODE !

Here is an example:

```
template <typename T>
T findValueByKey(std::string const &keyFilter, std::string const &filename) {
  std::string line, key;
  T value;

  std::ifstream stream(kProcDirectory + filename);
  if (stream.is_open()) {
    while (std::getline(stream, line)) {
      std::istringstream linestream(line);
      while (linestream >> key >> value) {
        if (key == keyFilter) {
          return value;
        }
      }
    }
  }
  return value;
};

template <typename T>
T getValueOfFile(std::string const &filename) {
  std::string line;
  T value;

  std::ifstream stream(kProcDirectory + filename);
  if (stream.is_open()) {
    std::getline(stream, line);
    std::istringstream linestream(line);
    linestream >> value;
  }
  return value;
};
```

Usage of the above template to get the value:

Use of findValueByKey

```
// Read and return the system memory utilization
float LinuxParser::MemoryUtilization() {
  string memTotal = "MemTotal:";
  string memFree = "MemFree:";
  float Total = findValueByKey<float>(memTotal, kMeminfoFilename);// "/proc/memInfo"
  float Free = findValueByKey<float>(memFree, kMeminfoFilename);
  return (Total - Free) / Total;
}
```

Use of getValueOfFile

```
string LinuxParser::Command(int pid) {
  return std::string(getValueOfFile<std::string>(std::to_string(pid) + kCmdlineFilename));
```

```
}
```

```
// DONE: An example of how to read data from the filesystem
string LinuxParser::OperatingSystem() {
  string line;
  string key;
  string value;
  std::ifstream filestream(kOSPath);
  if (filestream.is_open()) {
    while (std::getline(filestream, line)) {
      std::replace(line.begin(), line.end(), ' ', '_');
      std::replace(line.begin(), line.end(), '=', ' ');
      std::replace(line.begin(), line.end(), '"', ' ');
      std::istringstream linestream(line);
      while (linestream >> key >> value) {
        if (key == "PRETTY_NAME") {
```

SUGGE STION

I think you should know this:

> PRETTY_NAME=
>
> A pretty operating system name in a format suitable for presentation to the user. May or may not contain a release code name or OS version of some kind, as suitable. If not set, defaults to "PRETTY_NAME="Linux"". Example: "PRETTY_NAME="Fedora 17 (Beefy Miracle)"".

Source : man pages

```
          std::replace(value.begin(), value.end(), '_', ' ');
          return value;
        }
      }
    }
  }
  return value;
}

// DONE: An example of how to read data from the filesystem
string LinuxParser::Kernel() {
  string os, version, kernel;
  string line;
  std::ifstream stream(kProcDirectory + kVersionFilename);
  if (stream.is_open()) {
```

SUGGE STION

Most of the students do not close the stream once defined in the function,It does not create any problem until and unless you are
using the same stream object to open some other file, If you wish to do the same then you can do two things

1. Either open the file with some other ifstream object
2. Close the ifstream object first and then use the same stream to open the next file again by statement `streamName.close()` after operation

It is a good practice to close the stream once you are done with the file opening.

```
    std::getline(stream, line);
    std::istringstream linestream(line);
    linestream >> os >> version >> kernel;
```

AWE SOME

Normally learners do a mistake here!
It is so nice that you gave extra attention here!

```
  }
  return kernel;
}

// BONUS: Update this to use std::filesystem
vector<int> LinuxParser::Pids() {
  vector<int> pids;
  DIR* directory = opendir(kProcDirectory.c_str());
  struct dirent* file;
  while ((file = readdir(directory)) != nullptr) {
    // Is this a directory?
    if (file->d_type == DT_DIR) {
      // Is every character of the name a digit?
      string filename(file->d_name);
      if (std::all_of(filename.begin(), filename.end(), isdigit)) {
        int pid = stoi(filename);
        pids.push_back(pid);
      }
    }
  }
  closedir(directory);
  return pids;
}

// DONE: Read and return the system memory utilization
float LinuxParser::MemoryUtilization() {
  float Memtotal = 0;
```

SUGGESTION

Please follow **some convention** throughout the code while naming variables!

Hey!I use snake_case style of naming variables.

snake_case: In which you separate the words by an underscore
camelCase: the first letter starts with the lower alphabet and then the second word starts with a capital letter
ProperCase/PascalCase: capitalising first letter of every word

You must go through this link

```
75  float Memfree = 0;
76
77  string line;
78  string key;
```

AWESOME

Awesome!
Very few people remain consistent with the C++ Core Guideline ES 10:
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-name-one
by defining only one variable per line!

```
79  string value;
80  std::ifstream filestream(kProcDirectory + kMeminfoFilename);
81  if (filestream.is_open()) {
82    while (std::getline(filestream, line)) {
83      std::istringstream linestream(line);
84      while (linestream >> key >> value) {
85        if (key == "MemTotal:") {
86          Memtotal = stof(value);
```

SUGGESTION

Warning

For some processes, there are very fewer data and so for them, there might be the case that they do not have some specific data and in that case, if you are doing something like string to int(or float, long) conversion it will throw an error invalid_argument which will stop the execution of the program which is not something you would want!

When can the error be thrown:

When the string is having other characters than digits (whether in conversion stoi,stol or stof).

IF THE STRING IS EMPTY AND YOU WISH TO CONVERT IT TO A LONG VALUE OR ANY INT VALUE OR ANY FLOAT VALUE IT WILL THROW UP AN ERROR *INVALID ARGUMENT*

Example Code

If you want to see an example then you can run this example :

```cpp
// CPP code for illustration of stoi()
// function when invalid_argument
// exception is thrown.
#include <bits/stdc++.h>
using namespace std;

int main() {

  // An invalid input string that has no
  // integer part.
  string invalid_num = "";

  // stoi() throws invalid_argument exception
  // when conversion process fails.
  try{
    cout << stof(invalid_num) << "\n";
  }

  // catch invalid_argument exception.
  catch(const std::invalid_argument){
    cerr << "Invalid argument" << "\n";
  }
  return 0;
}
```

When you run the above program then it will print an invalid argument. So you now you know that the program throws an error when the string is empty which means you need to take care of this case and then proceed further.

How to ensure execution does not get interrupted:

Before passing the argument to `stoi`, `stol`, `stof`, you can make sure that the argument is not empty (it is sure that it will not contain characters other than the digits) by using an if statement.
If it is empty then you can put the corresponding value to be unavailable or zero.

Questions and Answers

Question: Is there any process which has indifferent file content than the usual processes?

Answer: Yes there are processes that do not use up system resources but they do have an existence. Those processes are called Zombie Processes.

Zombie Process:
A process in Unix or Unix-like operating systems becomes a zombie process when it has completed execution but one or some of its entries are still in the process table. If a process is ended by an "exit" call, all memory associated with it is reallocated to a new process; in this way, the system saves memory. But the process' entry in the process table remains until the parent process acknowledges its execution, after which it is removed. The time between the execution and the acknowledgment of the process is the period when the process is in a zombie state.

When a process dies on Linux, it isn't all removed from memory immediately — its process descriptor stays in memory (the process descriptor only takes a tiny amount of memory - That is why you can see the memory column is showing almost nothing at all).

A zombie process is also known as a defunct process.

PS - Do not worry if you do not understand some of the terms and it is perfectly normal because these terms are taught in a course Operating Systems a which too is a vast course in itself. I just gave you a glimpse so that you know that there is nothing wrong with your implementation and let you know why it is happening.

```
87
88          } else if (key == "MemFree:") {
89            Memfree = stof(value);
90            break;
91          }
92        }
93      }
94    }
95    float perc = (Memtotal - Memfree) / Memtotal;
96    return perc;
97  }
98
99  // DONE: Read and return the system uptime
100 long LinuxParser::UpTime() {
101   string uptime1, uptime2;
102   string line;
103   std::ifstream stream(kProcDirectory + kUptimeFilename);
104   if (stream.is_open()) {
105     std::getline(stream, line);
106     std::istringstream linestream(line);
107     linestream >> uptime1 >> uptime2;
108   }
109
110   long times = stol(uptime1);
111
112   return times;
113 }
```

SUGGE STION

Add To Knowledge

You can also verify the system uptime by the command w on the terminall as shown

```
consentsam@sattu:~$w
 02:57:10 up 2 days, 22:53,  1 user,  load average: 4.42, 3.14, 2.34
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
consents tty7     :0               Thu04    2days  2:29m  6.40s /sbin/upstart -
```

```
114
115 // DONE: Read and return the number of jiffies for the system
116 long LinuxParser::Jiffies() {
117   return LinuxParser::ActiveJiffies() + LinuxParser::IdleJiffies();
118 }
119
120 // DONE: Read and return the number of active jiffies for a PID
121 // REMOVE: [[maybe_unused]] once you define the function
122 long LinuxParser::ActiveJiffies(int pid) {
123   string line;
124   string utime, stime, cutime, cstime;
125   long jiffies;
126
127   std::ifstream stream{kProcDirectory + to_string(pid) + kStatFilename};
128
129   if (stream.is_open()) {
130     std::getline(stream, line);
131     std::istringstream linestream{line};
132
133     for (int i = 0; i < 14; i++) linestream >> utime;
134
135     linestream >> stime >> cutime >> cstime;
136
137     jiffies = std::stol(utime) + std::stol(stime) + std::stol(cutime) +
138               std::stol(cstime);
139   }
140
141   return jiffies;
142 }
143
144 // DONE: Read and return the number of active jiffies for the system
145 long LinuxParser::ActiveJiffies() {
146   vector<string> active{CpuUtilization()};
```

```
  return (stol(active[CPUStates::kUser_]) + stol(active[CPUStates::kNice_]) +
          stol(active[CPUStates::kSystem_]) + stol(active[CPUStates::kIRQ_]) +
          stol(active[CPUStates::kSoftIRQ_]) +
          stol(active[CPUStates::kSteal_]) + stol(active[CPUStates::kGuest_]) +
          stol(active[CPUStates::kGuestNice_]));
}

// DONE: Read and return the number of idle jiffies for the system
long LinuxParser::IdleJiffies() {
  vector<string> idle_{CpuUtilization()};

  long idle_value =
      stol(idle_[CPUStates::kIdle_]) + stol(idle_[CPUStates::kIOwait_]);

  return idle_value;
}

// DONE: Read and return CPU utilization
vector<string> LinuxParser::CpuUtilization() {
  string line;
  string key;
  string value;

  vector<string> cpu_val;

  std::ifstream filestream(kProcDirectory + kStatFilename);

  if (filestream.is_open()) {
    std::getline(filestream, line);

    std::istringstream linestream(line);

    while (linestream >> value) {
      if (value != "cpu") {
        cpu_val.push_back(value);
      }
    }
  }

  return cpu_val;
}

// DONE: Read and return the total number of processes
int LinuxParser::TotalProcesses() {
  string line;
  string key;
  string value;
  int value_i{0};

  std::ifstream filestream(kProcDirectory + kStatFilename);
  if (filestream.is_open()) {
    while (std::getline(filestream, line)) {
      std::istringstream linestream(line);
      while (linestream >> key >> value) {
        if (key == "processes") {
          value_i = stof(value);
        }
      }
    }
  }
  return value_i;
}

// DONE: Read and return the number of running processes
int LinuxParser::RunningProcesses() {
  string line;
  string key;
  string value;
  int value_i{0};
  std::ifstream filestream(kProcDirectory + kStatFilename);
  if (filestream.is_open()) {
    while (std::getline(filestream, line)) {
      std::istringstream linestream(line);
      while (linestream >> key >> value) {
        if (key == "procs_running") {
          value_i = stof(value);
        }
      }
    }
  }
  return value_i;
}

// DONE: Read and return the command associated with a process
// REMOVE: [[maybe_unused]] once you define the function
string LinuxParser::Command(int pid) {
  string line;
  string key;
  string value;

  string Pid = to_string(pid);

  string path = "/proc/" + Pid + "/cmdline";

  std::ifstream stream(path);

  if (stream.is_open()) {
    std::getline(stream, line);
  }
  return line;
}

// TODO: Read and return the memory used by a process
```

SUGGESTION

After solving the TODO you can remove it from the source code.

A lot of development teams use comments in the code starting with TODO in order to identify codes that are incomplete, or that need to be fixed or new features that should be implemented. After solving the problem or implementing the feature, it's interesting to remove the TODO to identify that the code does not need attention anymore.

Just a curiosity: there are some IDEs that even highlight the comments of this type to help developers find the code that needs to be done.

Note: This is valid for other parts of your code as well.

```
251 // REMOVE: [[maybe_unused]] once you define the function
252 string LinuxParser::Ram(int pid) {
253   string line;
254   string key;
255   string value;
256
257   string Pid = to_string(pid);
258
259   string path = "/proc/" + Pid + "/status";
260
261   std::ifstream filestream(path);
262
263   if (filestream.is_open()) {
264     while (std::getline(filestream, line)) {
265       std::istringstream linestream(line);
266       while (linestream >> key >> value) {
267         if (key == "Vmsize:") {
```

SUGGE STION

I understand that you are following the Udacity guidelines and that is why you have extracted value corresponding to the keyword `VmSize`

But i should tell you that this will give you memory usage more than your Physical RAM size!

Because VmSize is the sum of all the virtual memory as you can see on the manpages also.
Search for `VmSize` and you will get the following line

```
 * VmSize: Virtual memory size.
```

Whereas when you use VmData then it gives the exact physical memory being used as a part of Physical RAM. So it is recommended to replace the string `VmSize` with `VmData` as people who will be looking at your GitHub might not have any idea of Virtual memory and so they will think you have done something wrong!

PS - Moreover when you replace then please put a comment stating that you have used `VmData` instead of `VmSize` because it might happen that another reviewer is following the Udacity guideline and so he/she might make it a required change but once you put the comment with the link to the resources then he will surely understand that!

```
268           break;
```

RE QUIRE D

You should divide by 1024 as the value you have extracted is in KB and to convert it in MB. You have to divide by 1024
1 MB = 1024 KB

```
269         }
270       }
271     }
272   }
273   return value;
274 }
275
276 // TODO: Read and return the user ID associated with a process
277 // REMOVE: [[maybe_unused]] once you define the function
278
279 string LinuxParser::Uid(int pid) {
280   string line;
281   string key;
282   string value;
283
284   string Pid = to_string(pid);
285
286   string path = "/proc/" + Pid + "/status";
287
288   std::ifstream filestream(path);
289
290   if (filestream.is_open()) {
291     while (std::getline(filestream, line)) {
292       std::istringstream linestream(line);
293       while (linestream >> key >> value) {
294         if (key == "Uid:") {
295           break;
296         }
297       }
298     }
299   }
300   return value;
301 }
```

```
302
303 // TODO: Read and return the user associated with a process
304 // REMOVE: [[maybe_unused]] once you define the function
305 string LinuxParser::User(int pid) {
306   string line;
307   string key;
308   string val;
309   string val1;
310   string check = LinuxParser::Uid(pid);
311
312   std::ifstream filestream(kPasswordPath);
313
314   if (filestream.is_open()) {
315     while (std::getline(filestream, line)) {
316       std::replace(line.begin(), line.end(), ':', ' ');
```

AWE SOME

Nice use of regex to extract values from the lines.
❤️

```
317
318       std::istringstream linestream(line);
319       while (linestream >> key >> val >> val1) {
320         if (val1 == check) {
321           break;
```

RE QUIRE D

Please note that when you will put a break it will just break the first while loop but still the outer loop will be executing which should not have happened because we got what we were looking for!

Please make the required changes and also confirm the username of a process by running the following command with the PID for which you want to confirm your username!

```
top -p PID
```

For example:
If you want to run for PID = 1200 you should run

```
top -p 1200
```

```
322         }
323       }
324     }
325   }
326   return key;
327 }
328
329 // TODO: Read and return the uptime of a process
330 // REMOVE: [[maybe_unused]] once you define the function
331 long LinuxParser::UpTime(int pid) {
332   string line;
333   string key;
334   string value;
335   long tim{0};
336
337   string Pid = to_string(pid);
338
339   string path = "/proc/" + Pid + "/stat";
340
341   std::ifstream filestream(path);
342
343   if (filestream.is_open()) {
344     string v1;
345
346     std::getline(filestream, line);
347     std::istringstream linestream(line);
348
349     for (int i = 0; i < 22; i++) {
350       linestream >> v1;
351     }
352     tim = (stol(v1) / sysconf(_SC_CLK_TCK));
```

RE QUIRE D

By going through the Linux documentation/manual.
So when you go through the same then you will see the following:

```
(22) starttime  %llu
                      The time the process started after system boot.  In
                      kernels before Linux 2.6, this value was expressed
                      in jiffies.  Since Linux 2.6, the value is expressed
                      in clock ticks (divide by sysconf(_SC_CLK_TCK)).

                      The format for this field was %lu before Linux 2.6.
```

```
(22) starttime  %llu
          The time the process started after system boot.  In
          kernels before Linux 2.6, this value was expressed
          in jiffies.  Since Linux 2.6, the value is expressed
          in clock ticks (divide by sysconf(_SC_CLK_TCK)).

          The format for this field was %lu before Linux 2.6.
```

Please find the reference of the page here

The 22nd value that you are grabbing is the

time the process started after system boot

That means in order to get the unit of time it has been running since start you need to subtract it from the `UpTime()` of the system and so you need to do as follows:

```
int upTimePid = UpTime() - stol(var)/sysconf(_SC_CLK_TCK);
return upTimePid;
```

Room for Improvement

As you can see this is dependent on the Kernel version so you need to set it according to the kernel version (meaning you need to put an if-else according to the if-else condition.
OR
Simply you can write the code for versions greater than Linux 2.6 because very few people would have been using the Linux versions lower than 2.6. (But obviously it won't be a good practice!

For the version greater than 2.6 (Since for implementing according to the Linux version you just need an if-else condition based on the version value - which i suppose you can do that easily)

```
353   }
354
355   return tim;
356 }
357
```

▸ **src/system.cpp** 2

▸ **src/processor.cpp** 2

▸ **src/process.cpp** 2

▸ **src/format.cpp** 1

▸ **Makefile** 1

▸ **include/ncurses_display.h** 1

▸ **include/linux_parser.h** 1

▸ **src/ncurses_display.cpp**

▸ **src/main.cpp**

▸ **README.md**

▸ **include/system.h**

▸ **include/processor.h**

▸ **include/process.h**

▸ **include/format.h**

▸ **CMakeLists.txt**

Learn the best practices for revising and resubmitting your project.

RETURN TO PATH