CUT - cpu usage tracker

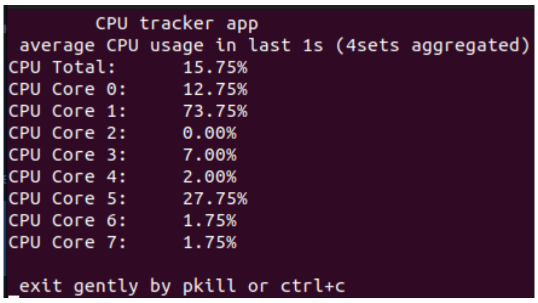
Project documentation
by Igor Elche

Project link - https://github.com/pontiaak/cpu_tracker_tieto.git

This application gathers statistics from /proc/stat, calculates and prints usage percentage for each CPU core alongside total.
To compile this application issue command $ make final
To launch it after compilation issue command $ ./cpu_tracker_app
To exit press Ctrl+C or issue $ pkill cpu_tracker_app
This way the application is compiled using gcc, to use clang instead change the following lines in makefile

```
CC = gcc                                    CC = clang
CFLAGS = -Wall -Wextra -lpthread     →      CFLAGS = -Weverything -lpthread
```

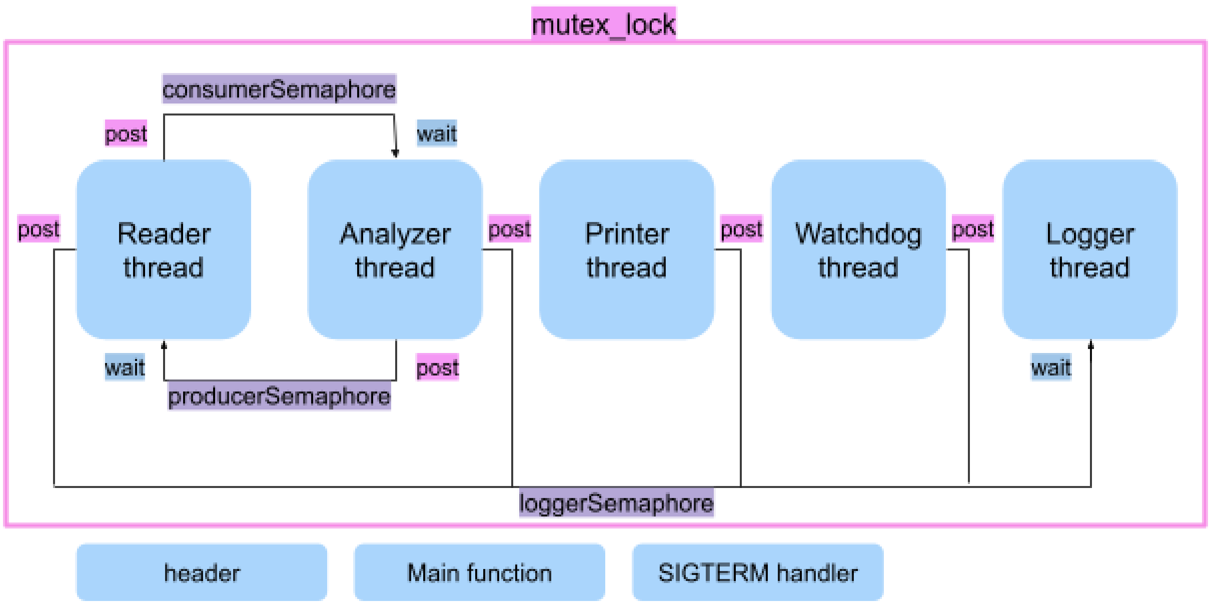
After launch you will be greeted with following interface:



```
        CPU tracker app
 average CPU usage in last 1s (4sets aggregated)
CPU Total:       15.75%
CPU Core 0:      12.75%
CPU Core 1:      73.75%
CPU Core 2:      0.00%
CPU Core 3:      7.00%
CPU Core 4:      2.00%
CPU Core 5:      27.75%
CPU Core 6:      1.75%
CPU Core 7:      1.75%

 exit gently by pkill or ctrl+c
```

Project Scheme



As shown in the project scheme above the application is a multithread one consisting of concurrently running 5 threads. Those threads are: Reader, Analyzer, Printer, Watchdog, and Logger. Those threads are initialised and created in main function alongside with mutex and semaphores and then joined and semaphores are destroyed to ensure no memory leaks. Threads and main() all function simultaneously until the terminationRequest flag is set to 1 by either signal being issued by the user to stop application or some kind problem with creation of threads or semaphores or opening files.

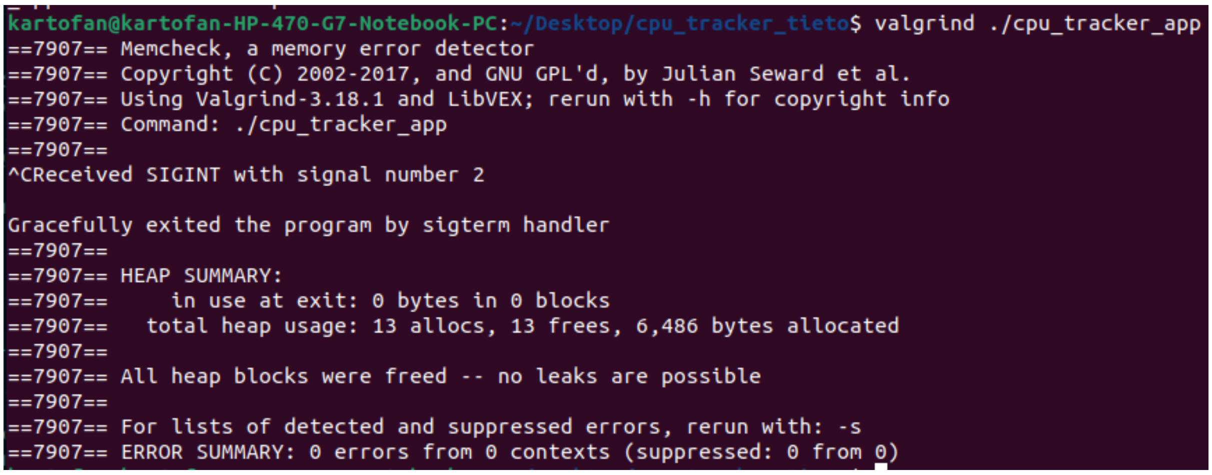
Producer - Consumer problem solution.
All thread's main chunks are encased in a mutex lock to ensure they don't interfere with each other or try to access one of multiple shared memory buffers at the same time because it can lead to issues or simply incorrect results. Reader and Analyzer threads are interlocked with a pair of single semaphores: producerSemaphore and consumerSemaphore. That means that they can only function in a strict one-after-another order. For example each time Reader thread's loop stops at sem_wait(&producerSemaphore); and waits until this semaphore is set to a positive value (1) which happens at the end of analyzer thread(but before the sleep thus not harming application's speed and productivity) meaning analyzer thread finished processing the shared array of structures cpuStatisticsStruct cpuStatistics[maximumCpuNumber]; in which all the cpu's statistics are stored.
The only other semaphore - loggerSemaphore is used to wake up a logger when we want it to write something into logs.txt, for example time the app started and was closed. Each thread that has some output that we want to log simply posts the semaphore, thus the Logger wait lets it go further and it writes the value corresponding to the index in shared memory buffer loggerMessage. This is the whole functionality of Logger thread.

The rough explanation of the project is as follows:
Reader thread continuously opens the /proc/stat file, then inside the mutex lock and after semaphore wait explained before it, inside a loop that iterates for the number of lines in that file choses only the lines with "cpu" in them (that means total cpu and individual cores) and writes into an array of structures cpuStatistics all the ten statistics mentioned there. Thus, for example with cpu cores 0-7 and total cpu 9 structures will be created. After that Reader posts the consumer semaphore, letting the analyzer do the further work.
Analyzer thread, after semaphore wait and inside mutex lock, iterates by the number of structures created and uses them inside this formula: https://stackoverflow.com/questions/23367857/accurate-calculation-of-cpu-usage-given-in-percentage-in-linux to calculate the cpu usage for each core. But the most interesting part is how it is stored in shared memory: Because we need the next thread, printer to print only each second, but the data from the reader comes roughly twice as fast, i needed to aggregate the sets of statistics. The simplest and fastest way to do so that i found is actually storing those statistics as SUMS of sets of calculated cpu percentage usages and alongside it, the number of sets of data aggregate, thus printer will only need to divide the sum of cpu usage percentages by the number of sets and it will get the AVERAGE that is needed. Also before it, because the formula needs to sets of data to calculate we need to skip the first set that we get and write it straight into previous statistics(prevIdle and prevTotal which are the sums of

actual statistics to be used later). This sum of cpu percentage usage is in the shared memory and then is accessed by Printer.
Printer, as described before, divides the sums of data by the number of sets of data and then cleared up both shared memory variables to be able to use them again after. Printer shows those statistics inside the interface shown in the first picture above.
Watchdog's functionality is to monitor is all the threads actually function continuously and are not deadlocked. To accomplish this it, every 2 seconds checks if all the main threads have set the shared memory flag watchdogFlags to 1, and they all do that during their functioning. If th do not, the watchdog sends the error message and closes the application graciously using terminationRequest flat that closes all threads and functions and cleans up memory.
Sigterm is a signal handler, when receiving either a SIGTERM or SIGINT, meaning user presses ctrl+c or enters pkill command, it closes the app using terminationRequest flag.

In this application i included one unit test responsible for testing the Reader thread for if it got correct data and CPU total statistics is greater or equal than first CPU core statistics (which it should because it is a total of all cores) which uses function assert(), that checks if the equation inside it is true. To activate this test you would need to compile the application in make test mode and then launch it as auto_test_cpu_app.

Valgrind output:



```
kartofan@kartofan-HP-470-G7-Notebook-PC:~/Desktop/cpu_tracker_tieto$ valgrind ./cpu_tracker_app
==7907== Memcheck, a memory error detector
==7907== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7907== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==7907== Command: ./cpu_tracker_app
==7907==
^CReceived SIGINT with signal number 2

Gracefully exited the program by sigterm handler
==7907==
==7907== HEAP SUMMARY:
==7907==     in use at exit: 0 bytes in 0 blocks
==7907==   total heap usage: 13 allocs, 13 frees, 6,486 bytes allocated
==7907==
==7907== All heap blocks were freed -- no leaks are possible
==7907==
==7907== For lists of detected and suppressed errors, rerun with: -s
==7907== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```