Performance Debugging Tools For Linux Based Operating Systems

Madhav Jivrajani
Department of Computer Science and Engineering
PES University
Bangalore, India
madhav.jiv@gmail.com

Sparsh Temani
Department of Computer Science and Engineering
PES University
Bangalore, India
sparshtemani31415@gmail.com

***Abstract*—In the modern era, the majority of the systems consist of hardware which uses multicore processors which support multithreading operations on a given task. The multithreading of tasks helps in near-parallel execution of instructions where multiple instructions appear to be executed simultaneously by using the power of multiple. The OS virtualizes the memory by implementing paging. It is important to know how frequently the memory needs to be referred for translations as this is a computationally expensive task. It is possible to debug the program performance-wise from the analysis of threads running on cores at various instances and by analysing the frequency of core, page migrations.**

I. INTRODUCTION

Process

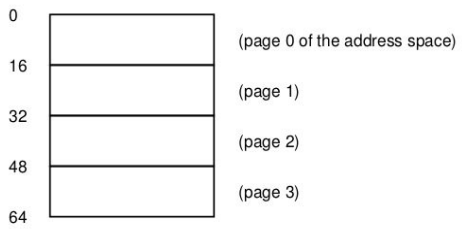
A program/ command when executed, an instance is provided and this starts a new process which has a unique identifier called the PID (Process ID). Example: When a command named pwd is run to find the current directory location, a process starts. The PID of the task is used by the Linux scheduler to track the process.

Types of processes :

- Parent and Child process: Every process has a parent process associated with it identified with a PPID.
- Zombie and Orphan process: When a parent process is terminated before the termination of a child process, the child is left without a parent and the parent PID is assigned to init by default.
- Daemon process: These are the system-related process which requires root permissions and mostly run in the background.

Threads

A thread is a unit of processing with a set of instructions which is the smallest executable a scheduler can work on. A process consists of multiple instructions and these instructions are grouped into several threads which are executed asynchronously. This way it brings in the capability of independent handling of a particular service.



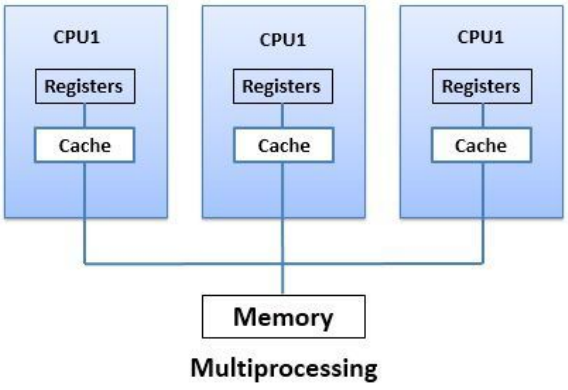
source[1]

Difference between threads and processes:

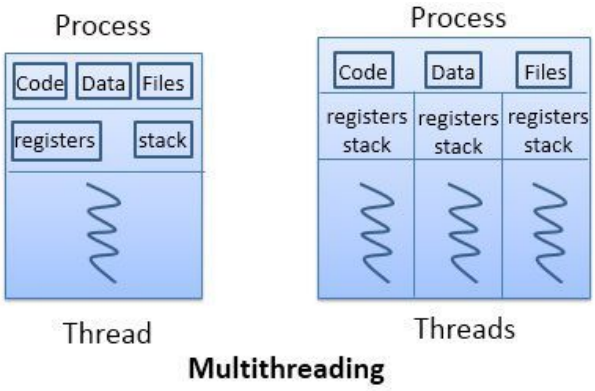
- Threads share the same address space while execution while processes don't share them.
- The process execution is synchronous in nature and independent in nature whereas thread synchronization needs to be taken care of by the parent process itself since the thread instructions can depend on one another.
- The communication between processes is achieved through standard inter-process communication while threads with the same parent process can communicate since they share most of the resources like memory, text segment, etc.

Memory Pages

In modern operating systems and architectures, the address space is split into contiguous blocks of memory of fixed length called pages. For example, a 64-byte address space could have four 16 byte pages as shown. This method of managing memory is known as paging. Due to paging, the physical memory is split into pages as well which sometimes are referred to as page frames. Paging strongly enables the operating systems to carry out its goal of virtualization. This method provides flexible and simple memory management as free-space management becomes relatively very simple as compared to segmentation, as free memory is treated as nothing but a set of unused memory pages and the operating system can easily maintain a list of free pages. It provides flexibility in the sense that with the implementation of paging, we have a layer of abstraction due to which assumptions such as the way in which the heap and stack grow for a particular process do not need to be made.



source[7]



source[7]

II. LITERATURE SURVEY

Threads and Processes

We have discussed processes as a combination of the following two characteristics :

- Ownership of resources: A process includes a virtual address space that is used to hold in the process image which is a collection of program data, stack and other attributes. In regular time intervals, a process is allocated ownership of resources like I/O channels and devices. The OS makes sure to avoid conflict in the allocation of resources for processes.
- Scheduling: A process is used to execute one or more programs with several threads each sharing the resources of the process while the process takes care of the scheduling so as to avoid deadlock between the threads.
- Deadlock is avoided by locking up space in memory when a thread had access to space and another thread can only access the space once the previous thread execution is completed. The process also takes care of assigning priority to the threads.
- To distinguish between these 2 characteristics of the thread and to make them independent, the resource ownership part is referred to as a task or a process while the scheduling/dispatching part is referred to like the lightweight process (A Thread).

Multithreading:

Multithreading in multi-core processors refers to the ability of the OS to execute multiple instructions within a single process at the same time to make it appear as if all of them were running at the same time.

To understand how multithreading works, we should have a basic knowledge of multiprocessing.

OS manages multiprocessing by giving a finite small amount of execution time before moving on to the next process so as to give the appearance as if all the processes were running at the same time on the processor. This way, none of the processes have to wait before the previous process is completely executed.

In a multithreaded environment, a process is defined as the unit of resource allocation. A process has a process image in a virtual address space associated with it and protected access to the resources of the processor and file, I/O resources.

A single process has multiple threads. The processor executes these threads on separate cores of the processor while having access to their process's set of resources and page tables.

A thread has an execution state (Running or Sleep), saved thread context, execution stack, thread-based local storage for local variables, access to the resources of the processes.

Execution state: Depending on the scheduling done by the process, a thread could be in running state or in a sleep state (Other threads belonging to the same process are executed because of higher priority assigned). The thread will also be in a sleep state when the process is not running.

Thread context, once the thread has had its share of time and resources, the processor has to move on to the execution of the next thread in the instruction buffer, before it does so, it has to save the context and the data generated during the current cycle of execution. To do so it saves this data as the thread context.

During the execution, the thread needs access to local variables to store intermediate results and temporary variables. For this purpose, it has access to the local storage also.

For a process model with a single thread, it includes the control block and the user address space. It also includes kernel stacks to manage behaviour during the execution of the process. The registers are controlled by the process while it is running and the content is saved in the register when the process is not running. In a multithreaded process, there is still a single model for the process but there are different instances of the stack for each of the threads along with a separate instance of control block to store the register data for each of the blocks along with their priority for the instruction cycle of the process' execution. All threads still share the state of the same process. If one of the threads change an address location, the same will be reflected in the rest of the threads also.

If one of the threads open a file with reading privileges, the same shall be reflected in the rest of the threads also

implying that the threads share the same privileges to the storage space of the user storage.
The advantages of threads in performance applications are the following:

- Creation of a thread takes way less time compared to the creation of a brand new process. This implies that this is much more efficient to create a new thread if it needs access to the same storage space instead of creating a new process.
- Termination of a thread also takes lesser time compared to the termination of a process.
- During the execution of a process, there are several thread switches to enable multithreading. This means that to speed up the execution of the program, the execution time can be minimized by minimizing the time it takes to switch between the instructions. Since threads share a significant amount of process resources with the other threads, it's faster and more memory efficient at the same time to start a new thread instead of a new process.
- When threads are dependent to receive data from one another, it is much easier to receive data from a sibling thread than from another process because that way they have to wait for an entire instruction cycle to receive the data till that process runs. If a thread is used in the same case, the threads can just write to the same address space to communicate with one another.

Multiprocessing systems have a lot of use cases, a few of them are:

- Running Background tasks in desktop applications where multiple independent instructions are passed simultaneously.
- Example: In a spreadsheet, one of the threads could be used to update the storage with the spreadsheet data and the other to display the spreadsheet data at the same time.
- Asynchronous processing: used to process async functions without having one of them to finish before moving on to the next function. This can be used autosave applications to prevent loss of unsaved work.
- Speed of execution: The resultant speed of execution is much faster due to multithreading.
- Processes can be divided into modules and run on separate threads giving rise to a modular code structure for which the maintenance is easier.

As mentioned in the previous section, parent processes can have child processes. This can be done using the fork() system call. A fork() call creates a new child process for the currently running process and the current process becomes the parent process with respect to the child process.
The fork() call returns a value based on whether the process of observation is a child process or a parent process and returns value less than 0 if an error occurred while forking the process.
Example call : int rc = fork();
There are three possible ranges for the value of rc:

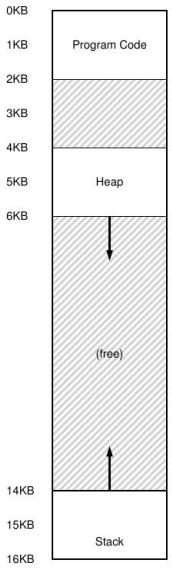
- rc < 0 implies that an error occurred while forking processes.
- rc = 0 implies that the instance of the program being executed is in the child process.
- rc > 0 implies that the instance of the program being executed is in the parent process.

We can use getpid() method to find the pid of the process that is being executed to determine whether it is a child or a parent process also. getpid() returns a void value which can then be typecasted implicitly to int.

When a child process is forked, the new process does not start running the entire program from the start, it starts running from the line instance where it was forked, so if there was a printf("Hello World"); right before the line where it was forked, it would be printed only once which would be part of the parent process which can be confirmed using the parent process. Even though the child process appears to be an exact copy of the parent process, it does not share the same address space and has its own space, registers and so forth. In the above example, there is no relation between the termination of the parent and child process but it is quite useful for a parent process to finish what a child process is doing and then terminate its execution. For this purpose, we can use the wait() system call where a parent process waits before executing the next line of code until the child process has terminated. If the child process has to wait till it's sibling process has been terminated, waitpid() method can be used instead. The above-mentioned piece of code works only when you want to execute the same piece of code in the child process also. If you want to execute a separate piece of code, use the exec() system call.
There are six variants to exec(): execl, execlp(), execle(), execv(), execvp(), and execvpe().

Virtual Memory and Paging

Virtualizing memory
The goal of the operating system is to provide the illusion that every process that wants to run has the right irrespective of limited processing power and memory. This process is known as virtualizing the CPU and virtualizing the memory respectively. If we run the same process simultaneously twice, we notice that the memory addresses allocated to each of the two processes is the same but both these processes update their values independently. The operating system achieves this illusion by providing each running process with its own virtual address space. Each process has its own virtual address space. These addresses are mapped onto addresses of the physical memory using certain mechanisms. That's why a process may have access to the same virtual addresses of their own virtual address space but will map onto distinct addresses in the physical memory.



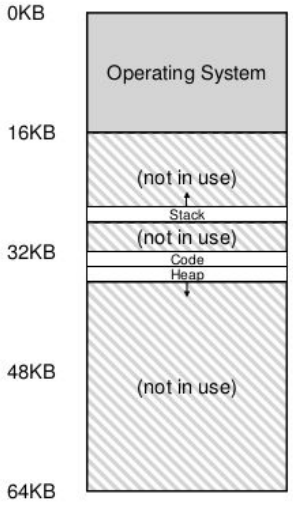
source[1]

Memory management mechanisms have evolved over time. One such mechanism is segmentation. Earlier, the virtual memory was divided into three logical segments: code, heap and stack. The memory between the stack and heap is left unused but is still mapped to the physical memory leading to wastage of space. This mechanism would not work for large address spaces as most of the virtual to physical mappings would be used by the unused chunk of memory in between the stack and the heap.

To overcome this, a mechanism known as segmentation was introduced. The idea is to have a base bound pair of register not for the entire memory management unit, but for each logical segment of the memory management unit. This means that each of the three segments, i.e. stack, heap and code can independently be placed into the physical memory. This means that the unused portion of the virtual address space is not mapped to the physical memory.
Segmentation can be looked as dynamic relocation of memory, but this has its own drawbacks in managing free-space as memory tends to get fragmented over time and the flexibility we desire may not be achieved from segmentation.

Memory Pages

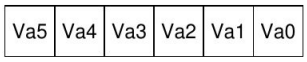
To overcome the above-mentioned drawback of segmentation, another mechanism called paging was introduced. In paging, instead of splitting the address space into three logical segments of variable size, it is split into contiguous blocks of memory which are of fixed size called pages. Due to this splitting of the address space, the physical memory is also split up into pages and we refer to each page of the physical memory as a page frame. In segmentation, the drawback was the difficulty in the management of free-space. Paging provides a simple and flexible way to manage free-space. With paging, the free memory can be looked at as nothing but pages which are not in use and the operating system maintains a free list of all the free pages and the operating system can at any time fetch the top n pages from this list to utilize the free space available.



source[1]

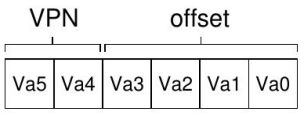
Each page in the virtual memory is mapped to a page frame in the physical memory. To keep a record of where each virtual page is placed in the physical memory a per-process data structure is maintained called the page table. The primary role of the page table is to hold the virtual to physical memory mappings of the pages. An important characteristic of the page table is that it is a per-process data structure which means the OS will have to maintain a separate page table per running process.

For the sake of understanding, consider a process with an address space of 64 bytes. To translate the virtual address that the process generated, we divide this virtual address into two parts: the Virtual Page Number (VPN) and the offset within the page. Since our address space is 64 bytes long, we need 6 bits in total to represent all addresses of the address space. Therefore, our virtual address would look like this:



source[1]

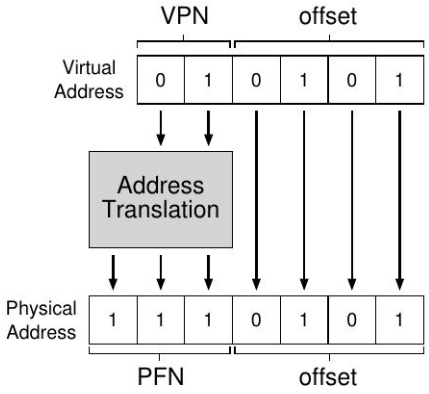
Here, Va5 and Va0 are the MSB and LSB of the virtual address. Each page is of 16 bytes, therefore, we can divide it as follows:



source[1]

If the page size is 16 bytes and the size of the address space is 64 bytes then we would require 4 pages. 2 bits are allocated to the VPN and therefore we have 4 virtual page numbers. The remaining 4 bits are called the offset and this tells us which byte of the page is of interest to the process
.

When a virtual address is generated, the OS and the hardware together translate this to the physical address. For example, if there was an instruction to virtual address “21”. Converting “21” to its binary equivalent we get “010101”. The first two bits as seen above are the VPN (01) and the remaining four bits (0101) is the offset. Or in other words, the virtual address “21” is available on the 5th byte (0101) of the virtual page “01”. Therefore with a VPN we can now index our page table and retrieve the address of the page frame that the virtual page resides within. The translation of the virtual address to the physical address takes place with the help of the hardware and the VPN is translated to a Physical Frame Number (PFN) and offset remains as it is since the job of the offset is to tell us were inside the page we want to access data from. Therefore, the PFN in this example is 7 (111).



source[1]

Note that this is, in fact, a small example to demonstrate the process of translation whereas in actuality the address spaces are significantly much larger.

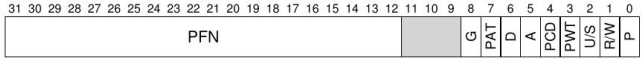
Page tables as mentioned earlier are a per-process data structure and for large address spaces, page tables can grow significantly in size and there is a separate page table for each process. For a 32-bit address space, the page table would need to maintain around a million translations just of a single process. Each entry in the page table is called a Page Table Entry (PTE). Since the page tables can get huge in size, on-chip hardware in the memory management unit is not kept for the current running process, instead, we store the page table for each process somewhere in the memory itself. Doing this again comes with a disadvantage, since they are stored in the memory, each reference to the page table is computationally very expensive since it would have to go to the memory each time.

A page table is nothing but a data structure which is used to map virtual addresses to physical addresses and as it was seen in the above example, the page table can be indexed with the VPN and a translation to the PFN can be achieved. Therefore, the simplest data structure that could work is a linear one and is called the linear page table and this is nothing but an array. The VPN acts as the index and the OS accesses the page table and looks up that index to get the page frame number.

Each PTE has a number of bits which signify something such as:

- Valid bit: This bit signifies whether a given translation is valid or not. For example, the memory segment of a running program will have code and heap at one end and the opposite end will have the stack. The space in between is marked as invalid. Therefore, all these memory pages which are marked invalid will not have physical page frames allotted to them thus saving memory.
- Present bit: This bit signifies whether a page is in the physical memory or not. This bit is important for page swapping where pages are swapped in and out of disk to handle large address spaces.
- Protection bits: These bits denote whether the page could be read from, executed or written to. If an operation that is against the protection bits is performed on the contents of these pages, the operating system executes a system trap by invoking the system trap handler.
- Dirty bit: This bit signifies if the page has been modified since it was brought into memory.
- Reference bit: This bit is used to keep track of whether a page has been accessed or not. This bit is also called the accessed bit. The reference bit tells us how popular a page might be. This is particularly useful to identify pages that were accessed frequently and these pages must not be swapped out of memory to reduce access overheads.

Example of an x86 PTE:



source[1]

TLB Cache

Address translation is a computationally expensive process because for every address translation we need to make a memory reference. To speed up the address translation process we add an additional cache known as the translation-lookaside buffer or a TLB. A TLB is part of the MMU chip and is a cache which contains address translations for pages that were frequently accesses or “popular” pages. Whenever there is a virtual memory reference, the TLB is first checked for the translation required, if the translation is found then the translation is performed quickly without having to refer to the actual page table in the physical memory which holds the entire set of translations. If a translation is found in the TLB, it is known as a TLB hit else it is known as a TLB miss.

TLB, like all other caches, is built to optimise the common case, i.e. it is used to perform fast translations of commonly accessed pages. When a TLB miss occurs, an overhead of an extra memory reference to the page table is incurred. Therefore, the goal is to minimise TLB misses. With hardware caches, the idea is to have two types of localities: spatial locality and temporal locality. The ideology of temporal locality is that some data or instruction that was accessed recently will be accessed again. Therefore with this

principle, the TLB keeps popular pages. TLB misses can be handled in two distinct ways: handled by the hardware and handled by the software. Most modern architectures adopt the latter for doing so. There are some very significant advantages in doing so. When TLB misses are handled by the software, hardware's only job is to raise an exception that a miss has occurred. Once this is done the current execution is halted, privilege level is raised to kernel mode and control is transferred to a piece of code known as the trap handler. When executed, the code looks up the page table for the translation, update the TLB and then return from the trap handler to this point (unlike a system call which returns to the next instruction) where the hardware would then retry the instruction resulting in a hit. On the other hand, when TLB misses are handled by the hardware, the hardware not only has to raise an exception, it also has to know where exactly in the physical memory the page resides and what the exact format of the PTE is, since, on a miss, the hardware would traverse or "walk" through the page table and retrieve the translation, update the TLB and then retry the instruction.

Context Switches and TLB

Each entry of the TLB contains the VPN, PFN and a few other bits such as valid bit, dirty bit, protection bits and address-space identifiers.

The TLB is a hardware cache and is maintained while a process is running. When a context switch occurs, the TLB has to be reset to hold the translations of the pages of the new process since translations of the previous process are of no use to it. To reset the TLB, a simple solution would be to mark the entries of the TLB invalid through the valid bit but this would mean the TLB would have to repopulate every time even though the context switches back to the original process. To overcome this, some additional hardware support is provided to support TLB sharing. This is done by adding an additional field known as address space identifier (ASID) to differentiate identical translations of different processes.

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

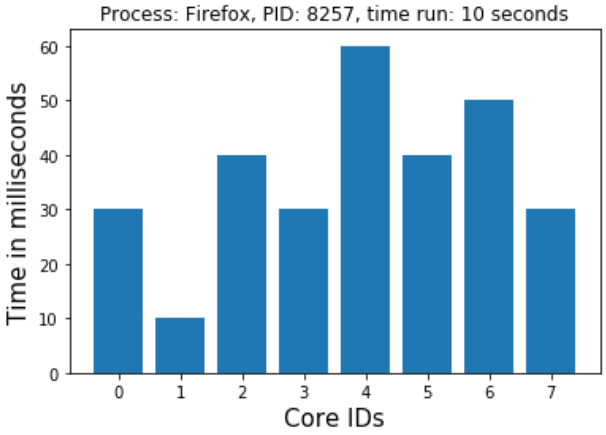
source[1]

To repopulate the TLB for the new process, a TLB miss would occur for every instruction initially as there is no context built for the new process. Once the TLB has been repopulated, the number of TLB misses drop.

III. IMPLEMENTATION

Extracting core migrations and plotting the results

As explained in the previous sections, threads usually migrate from one core to another when there is a change in the instruction cycle. It is possible to observe this migration



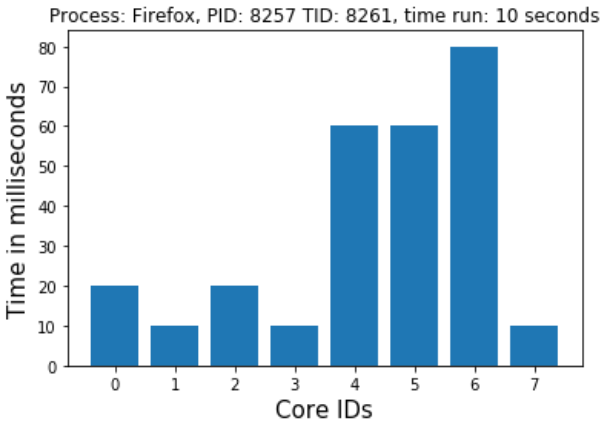
by extracting the core ID on which the thread is running at regular intervals and analysing the output. The results obtained were from running this tool on an 8 core processor.

It is possible to extract the data for core ID and time spent on a core using the top command in Linux. **top -H** can be used to extract thread-level info at regular intervals. The data extracted is appended to a file named **<PID>.txt** from which the second thread reads data. This extracted info can be dumped to a file and saved for later analysis.

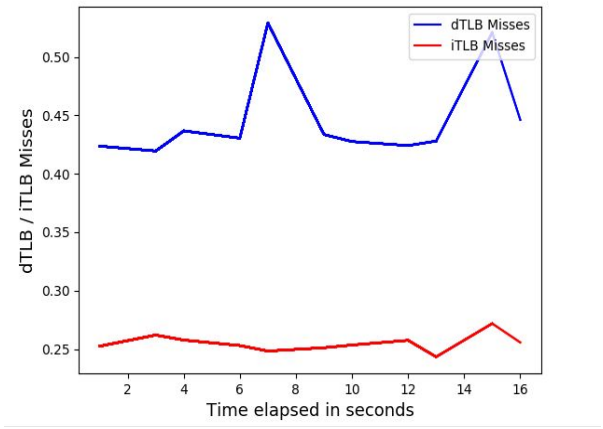
Once the extracted data is saved, using python visualisation libraries like matplotlib.pyplot, it is possible to visualize the results obtained from the data in top command.

As the activity of the process changes, the thread switches and the change in the core activity can be observed.

To run the 2 tasks at the same time in python, threads can be implemented here such that one of the threads work on extracting the data while the other thread works on updating the graph as the data is updated. This is another great use of threads. The plotted graph is updated using plt.draw() function in the matplotlib library with a sufficient delay.



Extracting TLB Misses along with additional info

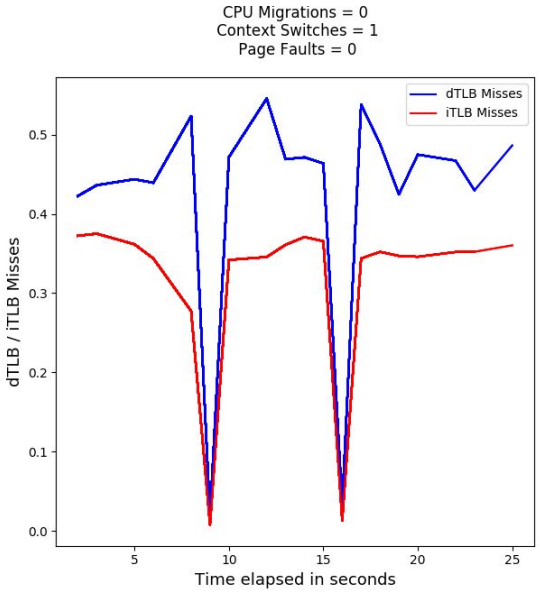


The tool we used to extract information for building this tool is called *perf*. This tool gets its information from /proc, which is a virtual file system holding the kernel's data structures. This file system does not exist in memory, it is a virtual file system created by the kernel every time we want to read the contents of the kernel's data structures. The entire page table for a process can be accessed from the path ***/proc/PID/pagemap***

TLB Misses and their causes have been explained in the previous sections. It is possible to extract the count of these misses and visualise them using matplotlib. There are 2 kinds of TLB misses, iTLB(data TLB) and dTLB (data TLB). To extract this info, use the perf command. The "perf stat dTLB-load-misses" gives the number of dTLB load misses in that duration. Similarly, iTLB can also be extracted using the same command. Perf stat also extracts cpu-migrations, context-switches and page-faults which can be used to determine the activity of the process. These 5 processes give insights about a process to the user and help the user better understand the flow of instructions as it happens in a dynamic manner so as to enable the user to enhance the program using this tool. The above graphs are for a process running in the terminal called htop.

Observations from the graph:

Initially, when there is no activity in the process, the plot is more or less idle and the variation is small. The rise and fall in the misses occur when the user begins to interact with the process. This behaviour can be explained in the following way: Initially the instructions running are only the basic tasks that the process needs to run, when there is activity, the TLB Tries to gain context due to which the rate of miss increases but once it gains the context in some time, it is noticed that the number of TLB Misses drop down significantly.



IV. CONCLUSION AND ACKNOWLEDGMENT

This paper explains the concept for multiprocessing and multithreading along with memory virtualization and paging followed by the concept of TLB caches and TLB misses. This knowledge can be used to build tools to extract important information from the processes and visualize them to help the programmer better understand the instruction cycle as it happens and enables for efficient performance debugging of a certain process with the help of these visualisations.

We would like to thank **Dr K V Subramaniam**, Centre for Cloud Computing and Big Data, PES University for guiding us through the duration of this project and helping us thoroughly understand the concepts.

V. REFERENCES

[1] Remzi H. Arpaci-Dusseau_ Andrea C Arpaci-Dusseau - Operating Systems Three Easy Pieces

[2] William Stallings - Operating Systems Internals and Design

[3] POSIX Thread Programming Documentation

[4] Matplotlib Documentation

[5] linux/tools/perf Documentation

[6] An Introduction to Programming with Threads by Andrew D. Birrell

[7] Tech Differences - Difference Between Multiprocessing and Multithreading