

The Lifecycle of a Linux Process

Table of Contents

Table of Contents.....	2
Table of Figures.....	2
1 Introduction	3
2 Tracing the cat Process and Memory Images	3
3 Context Switching, CPU Modes and Traps.....	4
4 Physical and Virtual Memory	5
5 Paging.....	6
6 References	8

Table of Figures

Figure 2-1 - Executing the cat command with a test file	3
Figure 2-2 - The initial shell is cloned, then the cat process replaces the cloned shell	3
Figure 2-3 - The stack and heap growing towards each other	4
Figure 3-1 – The Execve C subroutine calling the cat binary	4
Figure 3-2 - The openat() system call opening the "test1" file	5
Figure 4-1 - Memory mapping the process.....	6
Figure 5-1 - Paging in logical and physical memory along with the page table and valid / invalid bits	7
Figure 5-2 - Translating a logical address using the TLB	8

1 Introduction


When a program is executed, an instance of it is provided by the operating system to the process. The instantiation of the process occurs through a series of system calls, for example `open()`, `mmap()`, and they can be traced through a utility tool called `strace`. The instance of a process is made up of any services or resources that may be needed by the process for execution (Negi, 2017).

The Linux Operating System keeps track of processes by giving them a five-digit number, called the process ID or PID. Each process is given a unique process ID and in Linux you can check the current active processes by running the `ps` command (Negi, 2017).

2 Tracing the cat Process and Memory Images

The beginning of a process starts with the command line. For this assignment, I will be using the `cat` command as an example as it achieves all the parts that I need to explain in this assignment. `Cat` is used to concatenate files and print them on the standard output. The `cat` command would place a file in memory, open it, read the contents, write the contents to the standard output, and then close the file (man7, 2020a).

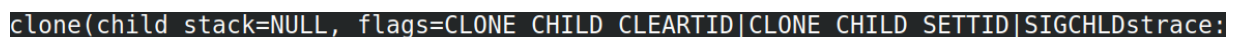
I created a simple file that contains the string “abc” in it and called it `test1`. Opening a shell and executing “`cat test1`” outputs just as expected, it concatenates the contents and prints them on the standard output.



```
[razvan@overlord ~]$ cat test1
abc
```

Figure 2-1 - Executing the `cat` command with a test file

To analyze this process, I will use a tool called `strace`. `Strace` is a “diagnostic utility tool (...) used to monitor and tamper with interactions between processes and the Linux kernel, which include system calls, signal deliveries, and changes of process state” (strace.io, 2022). Simply, it intercepts the system calls and nicely outputs it to the standard output for me to analyze. I first used just “`strace cat test1`”, however I wanted to analyze the process more in depth, so I decided to run “`sudo strace -fp <process ID of the bash shell that runs “cat test1”>`”. This shows me that the shell process is cloned to a child process. When a process is cloned, its stack, heap, and code are cloned. These make up the memory image of a process.



```
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLDstrace:
```

Figure 2-2 - The initial shell is cloned, then the `cat` process replaces the cloned shell

When a process is created, the operating system prepares a part of memory in the RAM for the memory image of the process. The memory image is split into four parts - the text section, data section, the heap, and the stack. The text section contains the compiled program code, read in from non-volatile storage, for example a hard drive. The data section contains the global and static variables which are initiated before executing the `main()` function. The heap is used for dynamic memory allocations, for example if the program is using the `malloc()` function or the `free()` function to allocate and to free memory,

respectively. The stack is used for function return values or to store local variables. So, when a function is called and the program branches off, this creates a new stack frame. Space on the stack memory is reserved for local variables when they are declared, for example at the function entrance, i.e., a function's parameters. The space is released when the variables go out of scope, such as when a function ends and the code returns to the main program. The stack and heap start at opposite ends of the memory allocated to the process and grow towards each other. If the stack and heap should meet, a stack overflow error might occur or a call to `new()` or `malloc()` may fail as there is no memory available. (University of Illinois Chicago, 2019b)

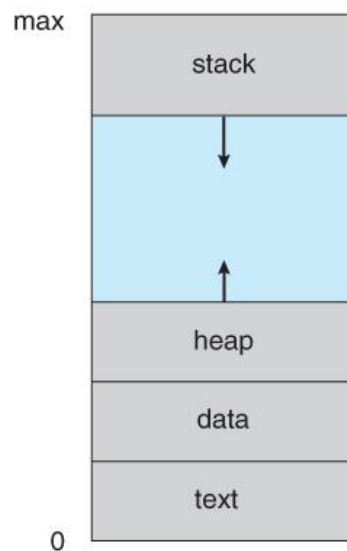


Figure 2-3 - The stack and heap growing towards each other

3 Context Switching, CPU Modes and Traps

```
execve("/usr/bin/cat", ["cat", "test1"], 0x7ffc36cf7888 /* 56 vars */) = 0
```

Figure 3-1 – The Execve C subroutine calling the cat binary

The first line where the `cat` binary is mentioned is the `execve()` C subroutine call. `Execve()` takes three arguments. The first of the arguments is the path to our executable binary file, which is the path to the `cat` binary. Secondly, it passes an array of pointers to strings which are the command-line arguments where the first of the arguments in the array must refer to the filename being executed. `Execve()` replaces the cloned shell process with the new process being referred to in the first argument, in this case `/usr/bin/cat`. When the new process is instantiated, it is given a new stack, heap, and data segments. As the bash child process' memory image is replaced by `cat`'s memory image, the process identifier remains the same because no new process is created. (man7, 2021)

When the `bash` process is replaced with the `cat` process, this is called context switching. Context switching is an essential part of any modern operating system, and it is used as it allows multiple processes to use a single CPU. Context switching occurs when the state of a process is stored so that it can resume execution later, for example after `cat test1` has finished and has displayed the output of `test1`, the bash shell returns. The process state includes any registers that the process is using,

especially the program counter, and it is all stored in a *process control block*, or PCB, a data structure created by the operating system when a process is initialized. The PCB specifies the state of the process i.e., new, ready, running, waiting, or terminated. It must be kept in an area of memory protected from user mode access, as a result the operating system stores the PCB at the beginning of the kernel stack, in the kernel memory. (David, Carlyle and Campbell, 2007; Onsmann, 2018; Tanenbaum and Bos, p. 94, 2015).

The CPU has multiple privilege levels, and it is crucial for it to have them for security. When the *cat* process is executing normal regular code that doesn't require a high privilege level, like `execve()`, the CPU is set to ring 3, which is the user mode. The user cannot run privileged instructions while in this mode. Any attempt to do so, like trying to execute system calls, is met with an error. This is so that, for example a malicious program wouldn't be able to write to memory that was not allocated to it or tried to access peripherals it is not meant to. To run high privilege code, the CPU must be set to ring 0, the kernel mode. If a user process needs to perform any high privilege code, the CPU must switch to kernel mode, perform the action, then switch back to user mode. User modes and kernel modes are implemented and separated by using operating system protection rings, which in turn are implemented using CPU modes (Biles, 2022).

To run the *cat* process, the operating system switches between user mode and kernel mode, usually to access input / output devices. For example, the process calls the `openat()` system call in order to access the file that we want to output its contents from, so the CPU goes into kernel mode. This is also called a *trap*. A trap is also defined as an exception in the user process that requires attention.

```
openat(AT_FDCWD, "test1", 0_RDONLY) = 3
```

Figure 3-2 - The `openat()` system call opening the "test1" file

The operating system needs to handle the trap when it is initialized. The trap instruction has a parameter, `int n`, that indicates the type of interrupt. A system call has a different value of `n` from a keyboard interrupt and from a disk interrupt. When `int n` is ran, the CPU mode changes to kernel mode. The CPU looks up the `n` value entry in the Interrupt Descriptor Table, or IDT, to get the address of the kernel code that handles this specific trap, it saves the stack pointer to an internal register, switches it to the kernel stack of the current process and it saves the old stack pointer and old instruction pointer where the execution stopped before the trap occurred so that it can be retrieved later, then it sets the instruction pointer to the value retrieved from the IDT, pointing to the kernel trap handler. After the trap has been handled, the kernel invokes the *iret* command to return to the user mode (Biles, 2022; Vutukuru, 2021).

4 Physical and Virtual Memory

The addresses stored in a process's memory image when the context is switched are logical memory addresses. A logical, or virtual, address is an address at which a process thinks a piece of data resides. They are generated by the CPU as a process is running and so they are used as a reference to access the physical memory space. The logical addresses might be the same or they might be different to the physical address, in which case the logical address must be mapped into the physical address. The mapping is done by the Memory Management Unit, or MMU. The instructions and data of a process's

memory image are stored in physical addresses therefore the MMU must translate the logical address that the CPU wants to access to the physical address to fetch the instruction or data. Modern operating systems manage the logical memory granularly with the paging technique. (Biles, 2022; Bisht, 2018; Worcester Polytechnic Institute, 2006).

The operating system calls for the *cat* process to be mapped into memory, along with the *test1* file.

```
mmap(NULL, 139264, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f1cc4c1b000
```

Figure 4-1 - Memory mapping the process

Mmap() is a system call that creates a new mapping in the logical address space of the calling process. The starting address for the new mapping is "NULL", which lets the kernel choose the page-aligned address at which to create the mapping. The length of the mapping is 139264, which is the length of the memory image. The operating system will then allocate it to pages with the page sizes it determines for maximum performance. The mapping has a read and write protection, where the pages may be read or written. The other flags don't allow other processes to view the mapping. (man7, 2021b).

5 Paging

Paging is a memory management scheme where the operating system retrieves data from secondary storage into the main memory in equal-size blocks. Paging allows the process's physical memory to be discontinuous which makes the use of memory more efficient, increasing capacity and performance. Paging divides physical memory into several equal-sized blocks called *frames*, additionally it also divides a process's logical memory space into same-sized blocks called *pages*, so the pages must be the same size as the frames. A logical address is composed of the process's page number, in which the address resides, and the page offset from the beginning of the page. The number of bits in the page number limits how many pages a single process can address and the number of bits in the offset determines the maximum page size. A physical address is the same as a logical address, however it refers to frames instead of pages. The number of bits in the frame number limits how many frames the system can address and the number of bits in the offset determines the size of each frame, which should correspond with the number of bits in the page offset. Any process's page can be placed into any available frame. To be able to access the instructions and data and to point at the locations where they are stored, each process has a page table. The page table of any one process contains the mapping of the process's logical pages onto the physical frames. The operating system builds the page table when allocating memory for a process and the MMU uses the page table to translate the logical addresses to physical addresses. (Biles, 2022; GeeksforGeeks, 2016; University of Illinois Chicago, 2019a; Worcester Polytechnic Institute, 2006)

The page table can help to protect from memory being accessed by processes that shouldn't be accessed, whether it's their own memory or not. A few bits can be added to a page table to make it read-write, read-only, read-execute, or any combination of these permissions. Memory references can be used to ensure that a process is accessing the memory in the correct mode. Valid or invalid bits can be added to entries in the page table to show entries that are not in use by the process. (University of Illinois Chicago, 2019a)

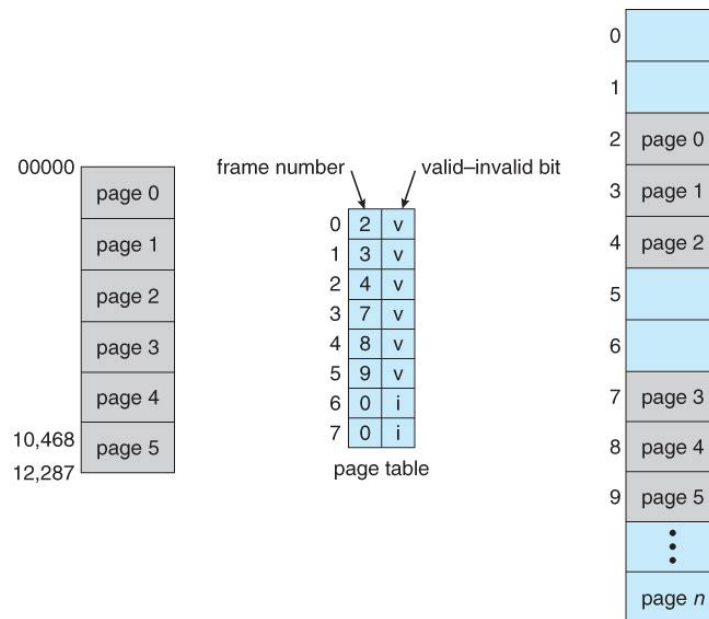


Figure 5-1 - Paging in logical and physical memory along with the page table and valid / invalid bits

To access the translations, the MMU must access the page table. The page table is stored in the main memory and the CPU knows where to find it as its memory address is stored in the CR3 register, which is used as the page-table base register, or PTBR. With paging, every access requires two memory accesses. One for the page table and one for the data or instruction. This can be slow and so to solve this we can use a very high-speed, associative lookup hardware cache called the Translation Look-aside Buffer, or TLB, located within the MMU. Each TLB entry consists of two parts, the key, and the value, where the key is the page number, and the value is the frame number. When the CPU looks for a page entry, the TLB is accessed. If there is a TLB *hit*, the corresponding lookup value is returned. If there is a TLB *miss*, the page table in slower memory is accessed and then the TLB is updated with the corresponding values. (Biles, 2022; Turnamian, 2011; University of Illinois Chicago, 2019a; Zare, 2020)

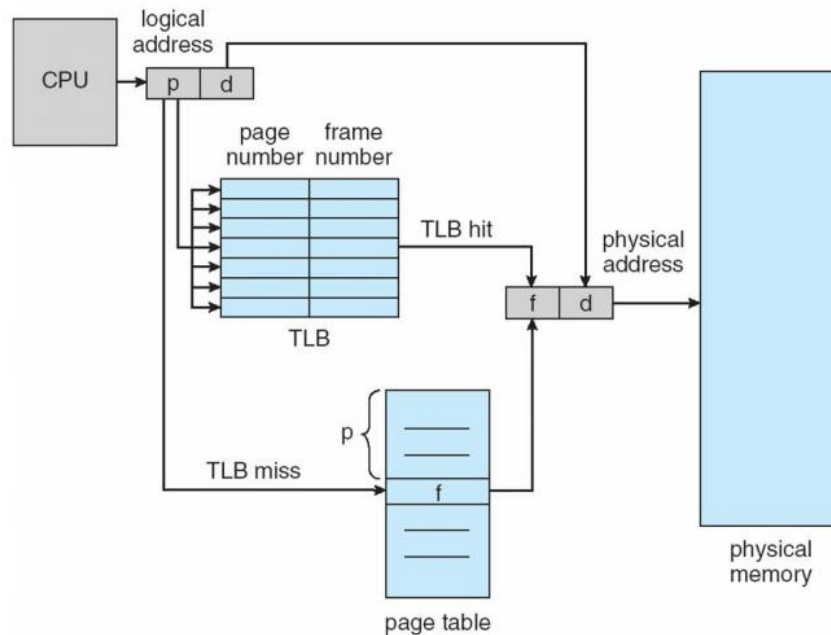


Figure 5-2 - Translating a logical address using the TLB

6 References

Biles, S. (2022). *Operating Systems in the Cyber Context - Week 12*.

Bisht, A. (2018). *Logical and Physical Address in Operating System*. [online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/logical-and-physical-address-in-operating-system/> [Accessed 1 May 2022].

David, F.M., Carlyle, J.C. and Campbell, R.H. (2007). Context switch overheads for Linux on ARM platforms. [online] San Diego, California: Association for Computing Machinery. Available at: <https://doi.org/10.1145/1281700.1281703>.

GeeksforGeeks. (2016). *Paging in Operating System*. [online] Available at: <https://www.geeksforgeeks.org/paging-in-operating-system/> [Accessed 1 May 2022].

man7. (2020a). *cat(1) - Linux manual page*. [online] Available at: <https://www.man7.org/linux/man-pages/man1/cat.1.html> [Accessed 27 Apr. 2022].

man7. (2020b). *strace(1) - Linux manual page*. [online] Available at: <https://www.man7.org/linux/man-pages/man1/strace.1.html> [Accessed 27 Apr. 2022].

man7. (2021a). *execve(2) - Linux manual page*. [online] Available at:

<https://www.man7.org/linux/man-pages/man2/execve.2.html> [Accessed 27 Apr. 2022].

man7. (2021b). *mmap(2) - Linux manual page*. [online] Available at:

<https://man7.org/linux/man-pages/man2/mmap.2.html> [Accessed 1 May 2022].

Negi, P. (2017). *Processes in Linux/Unix - GeeksforGeeks*. [online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/processes-in-linuxunix/> [Accessed 27 Apr. 2022].

Onsman, A. (2018). *What is Process Control Block (PCB)?* [online] Tutorialspoint.com.

Available at: <https://www.tutorialspoint.com/what-is-process-control-block-pcb>.

strace.io. (2022). *strace*. [online] Available at: <https://strace.io/> [Accessed 27 Apr. 2022].

Tanenbaum, A.S. and Bos, H.J. (2015). *Modern Operating Systems, 4th Edition*. Pearson Higher Education.

Turnamian, M. (2011). *Chapter 8: Memory Management*. [online] Available at:

http://www.cs.fsu.edu/~lacher/courses/COP4610/lectures_8e/ch08.pdf [Accessed 1 May 2022].

University of Illinois Chicago (2019a). *Operating Systems: Main Memory*. [online] Uic.edu. Available at:

https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/8_MainMemory.html [Accessed 1 May 2022].

University of Illinois Chicago (2019b). *Operating Systems: Processes*. [online] Uic.edu.

Available at: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/3_Processes.html [Accessed 29 Apr. 2022].

Vutukuru, M. (2021). *Lecture 24: Trap handling in xv6*. [online] Available at:

https://www.cse.iitb.ac.in/~mythili/os/anno_slides/lecture24.pdf [Accessed 28 Apr. 2022].

Worcester Polytechnic Institute (2006). *Virtual vs Physical Addresses*. [online] Available at:

<http://web.cs.wpi.edu/~cs3013/c06/week4-paging.pdf> [Accessed 1 May 2022].

Student ID: 2136685

Zare, N. (2020). *Caching Page Tables*. [online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/caching-page-tables/> [Accessed 1 May 2022].