A Comparison of Processes and Threads Creation

Martin Sysel[1[0000-0001-7177-0203]]

[1] Tomas Bata University in Zlín, FAI, Nad Stráněmi 4511,760 05 Zlín, Czech Republic
`Sysel@utb.cz`

Abstract. This article discusses the differences between processes and threads in the Linux operating system. Both are represented by the same data structures and similarly scheduled, and both are called tasks. It tries to answer whether the names Heavy-weight Process and Light Weight Process (LWP) are appropriate in Linux. It focuses on the time required to create a process or thread. First, the mechanism of creating a new process and thread is explained, and then several measurements are performed under different conditions. A fragments of source code which is used in the measurement are attached. Finally, the advantages and disadvantages of processes and threads are summarized with a recommendation when it is better to use processes or threads in the Linux operating system.

Keywords: Linux, Process, Thread

1 Introduction

The Traditional concept of processes can be divided into two parts: processes which dealing resource ownership and threads which are called a unit of execution (a stream of instructions) [1]. Normally when a program starts up and becomes a process, it starts with a default thread, e.g. `main` function in the C language. A process can create extra threads, so a single process may contain multiple threads or just only one. Linux uses a 1-1 threading model (Kernel Level Threads), and do not make distinction between processes and threads, everything is a runnable task. Every process in Linux is created by a parent process using a library function called `fork` (except INIT or system process), which trigger a proper system call [2]. There is a traditionally UNIX system call with the name `fork()` but this system call has been replaced by `clone()` which allows the child process to share parts of its execution context with the calling process. On Linux, both processes and threads are created with `clone()` system call, even a thread is created by different library functions – process by the function `fork` which means least sharing and thread by the function `pthread_create` which means most sharing [2].

2 Calling Function fork

A new process is created by the library function fork which trigger system call clone(). The child process consists of a copy of the address space of the original process, so it is a duplicate of the parent process. It uses Copy On Write (COW) method. This is an important optimization because the child's memory pages are initially mapped to the pages shared by the parent, and only if process tries to modify them then kernel copies them. Both processes (the parent and the child) continue execution at the same instruction after the calling function fork, with one difference: the return code for the fork is zero for the child process, whereas the actual nonzero PID of the child is returned to the parent [3]. The new child process has of course a unique PID.

Because executing the same code is useful just only occasionally, typical child process uses the exec() system call to replace the process's memory space with a new code after calling fork function. The new execution starts when exec() system call loads another binary code into address space and destroy the memory image of the previous process [3].

The parent waits with the wait() system call for the child process invoke exit() to complete work.

3 Calling Function phread_create

How has been mentioned, pthread_create uses system call clone() too, but it passes more arguments to share the virtual memory, file system, open files, shared memory and signal handlers with the parent process or thread [4].

While function fork does not have any arguments, pthread_create has following synopsis [5]:

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
  const pthread_attr_t *attr,
  void *(*start_routine) (void *), void *arg);
```

Once the function is called successfully, the variable whose address is passed as first argument will hold the thread ID of the newly created thread. The second argument may contain attributes. When an attribute object is not specified, it is NULL, and the thread is created with the default attributes (unbound, nondetached, default stack and stack size, inherited priority of parent process). The third argument is a function pointer. Each thread starts with a function and that functions address is passed here. The function may accept argument in form of a pointer to a void type. If a function accepts more than one argument, then this pointer could be a pointer to a structure [5]. NULL may be used if no argument is to be passed. A lot of beginners pass argument incorrectly because they pass the address of passing variable.

The fragment of incorrect code is here:

```
for(i=0; i<5; i++)
{
  rc = pthread_create(&th[i], NULL, PrintHello, &i);
  // or (void *) &i  // correct is (void *) i
}
```

This variable `i` is stored in shared memory space and it is visible to all threads. As the loop iterates, the value of this memory address changes, very often before the created thread can access it. For example, it should be right like this `(void *)i` or use a structure.

4 A Time measurement

It exists many methods but there are often not enough accurate or portable to other platforms because they depend on operating system or compiler version. The measurement in this contribution do not focus on CPU Time because it considers only time when CPU processing the program's instructions and e.g. I/O operations time is not included there. The end user is mainly interested in wall time, it is simply the total time elapsed during the measurement.

4.1 A Function clock_gettime()

It is available only in Linux systems. It can measure CPU time too, simply by replacing the constant `CLOCK_REALTIME` with `CLOCK_PROCESS_CPUTIME_ID`.

```
#include <stdio.h>
#include <time.h>
int main ()
{
    struct timespec begin, end;
    clock_gettime(CLOCK_REALTIME, &begin);
    //measured code
    clock_gettime(CLOCK_REALTIME, &end);
    long seconds = end.tv_sec - begin.tv_sec;
    long nanoseconds = end.tv_nsec - begin.tv_nsec;
    double elapsed = seconds + nanoseconds*1e-9;
    printf("Time: %.9f seconds.\n", elapsed);
}
```

Time measured by previous code shows overhead of `clock_gettime` function, this time is about 94 nanoseconds at CPU frequency 3,4 GHz (see chapter 5).

4.2 Using <chrono> library

It is available in Linux and Windows systems. It can measure only wall time, but it has access to several clocks in the machine, each of them with different purposes and characteristics. The clock with the highest resolution available is `high_resolution_clock` and provide nanoseconds resolution too.

```
#include <iostream>
#include <chrono>

int main()
{
    auto begin=std::chrono::high_resolution_clock::now();
    //measured code
    auto end = std::chrono::high_resolution_clock::now();
    auto elapsed = std::chrono::duration_cast<std::
      chrono::nanoseconds>(end - begin);
    std::cout << "Time: " << elapsed.count() * 1e-9;
}
```

Time measured by chrono library has slightly greater overhead, this time is about 110 nanoseconds at high speed CPU frequency.

5 Time Measurement

All time measurement has been processed on the notebook with CPU Intel Core i7-1065G7 (1.3 GHz, Intel Turbo Boost 3.9 GHz, HyperThreading) [7]; 16 GB RAM LPDDR4X (3733 GHz); SSD in M.2 PCIe/NVMe slot. Operating system Ubuntu 20.04 LTS, kernel 5.4.0-40-generic version #44-Ubuntu SMP.

As a processor can change a frequency because Intel Turbo Boost Technology is used, all measurements will be presented together with used CPU frequency.

5.1 Function fork

The following source code fragment presents time measurement of function fork using `clock_gettime` function.

```
clock_gettime(CLOCK_REALTIME, &begin);
if ((child_pid = fork()) == 0)    //Child process return 0
{
  clock_gettime(CLOCK_REALTIME, &child);
  …
  exit(0);
}
```

```
//Parent process
clock_gettime(CLOCK_REALTIME, &parent);
…
```

At a CPU base frequency of 1.3 GHz, the execution time is around 94 microseconds, but with Intel Turbo Boost on a 3.4 GHz CPU frequency, the `fork` function lasts 34 microseconds. This is the execution time of the `fork` function and return to the parent process. Executing the `fork` function and continuing in the child process takes more time, from 60 to 170 microseconds, depending on the CPU frequency. Very rarely, a process migrates to another CPU core, but this has no effect on performance in this simple case.

The execution speed of the `fork` function also depends on the size of the address space [4]. The `fork` function lasts longer, while allocated address space grow up, e.g. the execution time is about 20 milliseconds with 4 GB of allocated address space by `malloc(1e9 * sizeof(int))`. As has been mentioned before, not the entire address space is copied, but only the parts that do not use the COW method (e.g. page table).

A similar situation occurs when another program is started using the `exec()` system call in a child process. The calling process is overwritten by the program whose filename is passed as the first argument. This is the most used use case; the `fork()` system call is followed by an `exec` call in the newly created task (child process). Even with a simple HelloWorld program that sends text to the standard output, the time to create a new child process is several times longer and grow up as the program becomes more complex (see Tab. 1). Returning to the parent process after executing the `fork` function takes as long as in the first simple example, it is in the range of 34 - 94 microseconds.

5.2 Function pthread_create

The following source code fragment presents time measurement of function `pthread_create` using `clock_gettime` function.

```
void *mythread(void *ID)
{
  clock_gettime(CLOCK_REALTIME, &thread);
  …
}
int main(int argc, char *argv[])
{
  …
  clock_gettime(CLOCK_REALTIME, &begin);
  pthread_create(&p1, NULL, mythread, NULL);
  clock_gettime(CLOCK_REALTIME, &parent);
  …
}
```

At a CPU base frequency of 1.3 GHz, the new child thread is created in time around 260 microseconds, but with Intel Turbo Boost on a 3.4 GHz CPU frequency, it lasts 94 microseconds. It is slower than creating a new task (child process) by `fork` function, but this is because the `clone()` system call is called with the following flags:

```
const int clone_flags = (CLONE_VM | CLONE_FS
    | CLONE_FILES | CLONE_SYSVSEM | CLONE_SIGHAND
    | CLONE_THREAD | CLONE_SETTLS | CLONE_PARENT_SETTID
    | CLONE_CHILD_CLEARTID | 0);
```

This is defined in function named `create_thread` in `sysdeps/unix/sysv/linux/createthread.c` file. It shares the virtual memory, file system, open files, shared memory and signal handlers with the parent thread/process [4]. To be POSIX compliant, it passes additional flags to implement proper identification because all threads created from a single process have share its process ID as a Thread group ID (tgid). In Linux, thread ID number is indicated by LWP (Light Weight Process).

The execution time of the `pthread_create` function and returning to the parent task (thread) is in the range 49 - 143 microseconds. So, this corresponds with previous results of simple example.

However, a different situation occurs when a new thread is created from a process that has a large address space allocated, e.g. creating a new thread in a process with a 4 GB address space in the range of 170 - 290 microseconds, so creating a new thread is not affected by the parent address space size unlike the fork function, which depends on the size of the address space. In addition, this address space can be shared between threads. The major difference between the multiprocessing and multithreading is the ability for threads to share the resources of the process in which they exist. Threads share such process resources as global variables and file descriptors. Independent processes do not share anything [6].

5.3 Time measurement summary

Table 1 shows measured times with descriptions. The Range of measured time is caused mainly by different CPU frequencies.

Table 1. Time measurement summary.

Measured Time [µs]	Description
60 – 170	Execution time of function fork and return to the child task from system call clone().
34 – 94	Execution time of function fork and return to the parent task from system call clone().
about 20 000	Fork of a task with a large address space (4 GB). Return to the parent takes nearly the same time as return to the child task.

about 700 and more	Fork and exec of HelloWorld program in the child task. Time depends on program complexity.
34 – 94	Execution time of function fork and return to the parent task.
94 – 260	Create new task (thread) and return to the child task (thread) from system call clone().
49 - 143	Execution time of function pthread_create and return to the parent task from system call clone().
170 – 290	Create new task (thread) in a task with a large address space (4 GB) and return to the child task (thread).
60 – 147	Execution time of function pthread_create and return to the parent.

6 Threads versus processes

A question is whether processes or threads are better to use. The answer is: it depends. Task creating time is not the most important reason. The only one advantage of a process is its own address space. If process crashes or has a buffer overrun, it does not affect any other processes, whereas if a thread crashes, it influences other threads in the process. An unshared address space could be beneficial for performance on a multiprocessor system because synchronizing shared memory between different processors is expensive.

Threads share memory of the process with other threads of the same process, inter-thread communication is faster, context switching can be less expensive because it exists chance to use cache memory content [6]. Synchronization is faster, but it depends on application programmer, not on the kernel, so there is a much greater risk of race condition.

7 Conclusions

There is no difference between processes (the result of `fork`) and threads (the result of `pthread_create`) for the Linux kernel. Both are represented by the same data structures and scheduled similarly, and both are called tasks, the differences are mostly about what is shared between parent and child task [4].

It would be expected process creation to be more expensive than thread creation, but since `fork` and `pthread_create` route to the same system call `clone()` in Linux, the difference come from the different flags they pass in [4].

Processes are safer and more secure than threads, because each process runs in its own virtual address space, but processes have a higher overhead; it depends on the size of address space (unlike threads). Inter-process communication (IPC) is harder and slower than inter-thread communication. So, if the tasks use shared data, it is better to use threads.

References

1. Stallings W.: Operating Systems: Internals and Design Principles. 8th ed., Pearson Education Limited (2014).
2. Pillai Sarath: Difference Between Process and Thread in Linux. Slashroot.in, https://www.slashroot.in/difference-between-process-and-thread-linux, last accessed 2020/06/07.
3. Silberschatz A., Galvin P. B. & Gagne G.: Operating system concepts. 9th ed. Hoboken, NJ: Wiley (2013).
4. Bendersky E.: Launching Linux Threads and Processes with clone. Eli Bendersky's website, https://eli.thegreenplace.net/2018/launching-linux-threads-and-processes-with-clone/, last accessed 2020/04/16.
5. Linux manual page, https://man7.org/linux/man-pages/man3/pthread_create.3.html, last accessed 2020/06/30.
6. Farrell J., Buttlar D., Nichols B.: PThreads Programming. O'Reilly online learning, https://www.oreilly.com/library/view/pthreads-programming/9781449364724/ch01.html, accessed 2020/06/25.
7. Intel Product Specifications, Intel® Core™ i7-1065G7 Processor https://ark.intel.com/content/www/us/en/ark/products/196597/intel-core-i7-1065g7-processor-8m-cache-up-to-3-90-ghz.html, accessed 2020/06/30.