Bahir Dar University

Faculty of Computing

Department of Software Engineering

Operating System and System Programming Assignment

Topic: System call

Submitted by: Nathnael Theodros

ID number: BDU1311646

Submitted to: Instructor Wendimu Baye

Before going Straight to the question, I want to say something about what system call is.

What is system call?

- A system call is also known as "syscall".
- is the programmatic method by which a computer software asks the operating system's kernel for a service while it is being run. This could involve contact with hardware-related services, such as process scheduling, as well as the creation and execution of new processes. Examples of hardware-related services include accessing the device's camera or hard drive. A crucial interface between a process and the operating system is provided via system calls.
- There are two modes in using a computer. It is the user mode and the kernel mode. User mode have no direct accesses to the resources. The resources can be memory. Whereas the kernel mode has direct access to memory. For this reason, the kernel mode is also referred as the privileged mode. User mode is considered to be safer than kernel mode because if something happens while using the user mode, there will not be a big problem because the user mode has no direct access to direct resources of the computer thus it will not bring a problem to the memories and processors. But if something happens while using the kernel mode, it will bring a big problem to the entire system. Sometimes, the user mode may want to have an access of the resources of the computer. That is when the system call come in handy. The user mode will request the kernel mode to have an access to the resources of the computer. System call is simply a means of sending a request from the user mode to the kernel mode.

We have said enough about the meaning of system call. The type of system call given to me is:

int timer_create(clockid_t clockid, struct sigevent *sevp, timer_t *timerid)

1)What, Why, How this system call?

The timers created by **timer_create**() are commonly known as "POSIX (interval) timers"

Using timer create(), a fresh per-process interval timer is created. The ID of the new timer is returned in the buffer pointed to by timerid, which must be a non-null pointer. Up until the timer is removed, this ID is the only one used in the operation. At first, the new timer is disarmed.

The clock that the new timer uses to keep time is specified by the clockid argument. A single one of the following values may be supplied for it:

CLOCK REALTIME-a programmable, global real-time clock.

CLOCK MONOTONIC-a non-settable monotonically growing clock that doesn't alter after system startup and measures time beginning at some unknown point in the past.

CLOCK PROCESS CPUTIME ID (since Linux 2.6.12)-a clock that counts the amount of user and system CPU time the caller process uses for each thread.

CLOCK THREAD CPUTIME ID (since Linux 2.6.12)-a clock that counts the amount of (user and system) CPU time the calling thread uses.

CLOCK BOOTTIME (Since Linux 2.6.39)-This clock increases monotonically, similar to CLOCK MONOTONIC. The CLOCK BOOTTIME clock, in contrast to the CLOCK MONOTONIC clock, does include the time that a system is suspended; the CLOCK MONOTONIC clock does not. Applications that require suspend awareness can benefit from this. Since that clock is impacted by sporadic changes to the system clock, CLOCK REALTIME is not appropriate for such applications.

CLOCK REALTIME ALARM (since Linux 3.0)-Similar to CLOCK REALTIME, but capable of awaking a suspended system to set a timer against this clock, the caller must possess the CAP WAKE ALARM capability.

CLOCK BOOTTIME ALARM (since Linux 3.0)-Similar to CLOCK BOOTTIME, this clock will wake a suspended system. The CAP WAKE ALARM capability is required in order to set a timer against this clock.

2) Briefly describe the list of parameters and flags.

The parameters and flags have the following meaning:

- clockid- it refers to the type of clock to use.
- sevp- it is pointer to the sigevent structure, which describes how the caller will be informed when the timer expires.
- timerid- it is a pointer to buffer that will receive the timer id.

3) List the flags, their purpose with code implementation (give example source code with output).

The whole code can be implemented in the following manner. This can allow us to see the effect of flags in the syste call.

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
#include <time.h>

#define CLOCKID CLOCK_REALTIME
#define SIG SIGRTMIN

#define errExit(msg)
```

```
do
{
        perror(msg); exit(EXIT_FAILURE); \
 } while (0)

static void
print_siginfo(siginfo_t *si)
{
   timer_t *tidp;
   int or;

   tidp = si->si_value.sival_ptr;

   printf("    sival_ptr = %p; ", si->si_value.sival_ptr);
   printf("    *sival_ptr = 0x%lx\n", (long) *tidp);

   or = timer_getoverrun(*tidp);
   if (or == -1)
      errExit("timer_getoverrun");
   else
      printf("    overrun count = %d\n", or);
}

static void
handler(int sig, siginfo_t *si, void *uc)
{
   printf("Caught signal %d\n", sig);
   print_siginfo(si);
   signal(sig, SIG_IGN);
}

int
main(int argc, char *argv[])
{
   timer_t timerid;
   struct sigevent sev;
   struct itimerspec its;
   long long freq_nanosecs;
   sigset_t mask;
   struct sigaction sa;

   if (argc != 3) {
      fprintf(stderr, "Usage: %s <sleep-secs> <freq-nanosecs>\n",
              argv[0]);
      exit(EXIT_FAILURE);
   }
```

```
    printf("Establishing handler for signal %d\n", SIG);
    sa.sa_flags = SA_SIGINFO;
    sa.sa_sigaction = handler;
    sigemptyset(&sa.sa_mask);
    if (sigaction(SIG, &sa, NULL) == -1)
        errExit("sigaction");

    printf("Blocking signal %d\n", SIG);
    sigemptyset(&mask);
    sigaddset(&mask, SIG);
    if (sigprocmask(SIG_SETMASK, &mask, NULL) == -1)
        errExit("sigprocmask");

    sev.sigev_notify = SIGEV_SIGNAL;
    sev.sigev_signo = SIG;
    sev.sigev_value.sival_ptr = &timerid;
    if (timer_create(CLOCKID, &sev, &timerid) == -1)
        errExit("timer_create");

    printf("timer ID is 0x%lx\n", (long) timerid);

    freq_nanosecs = atoll(argv[2]);
    its.it_value.tv_sec = freq_nanosecs / 1000000000;
    its.it_value.tv_nsec = freq_nanosecs % 1000000000;
    its.it_interval.tv_sec = its.it_value.tv_sec;
    its.it_interval.tv_nsec = its.it_value.tv_nsec;

    if (timer_settime(timerid, 0, &its, NULL) == -1)
         errExit("timer_settime");

    printf("Sleeping for %d seconds\n", atoi(argv[1]));
    sleep(atoi(argv[1]));

    printf("Unblocking signal %d\n", SIG);
    if (sigprocmask(SIG_UNBLOCK, &mask, NULL) == -1)
        errExit("sigprocmask");

    exit(EXIT_SUCCESS);
}
```

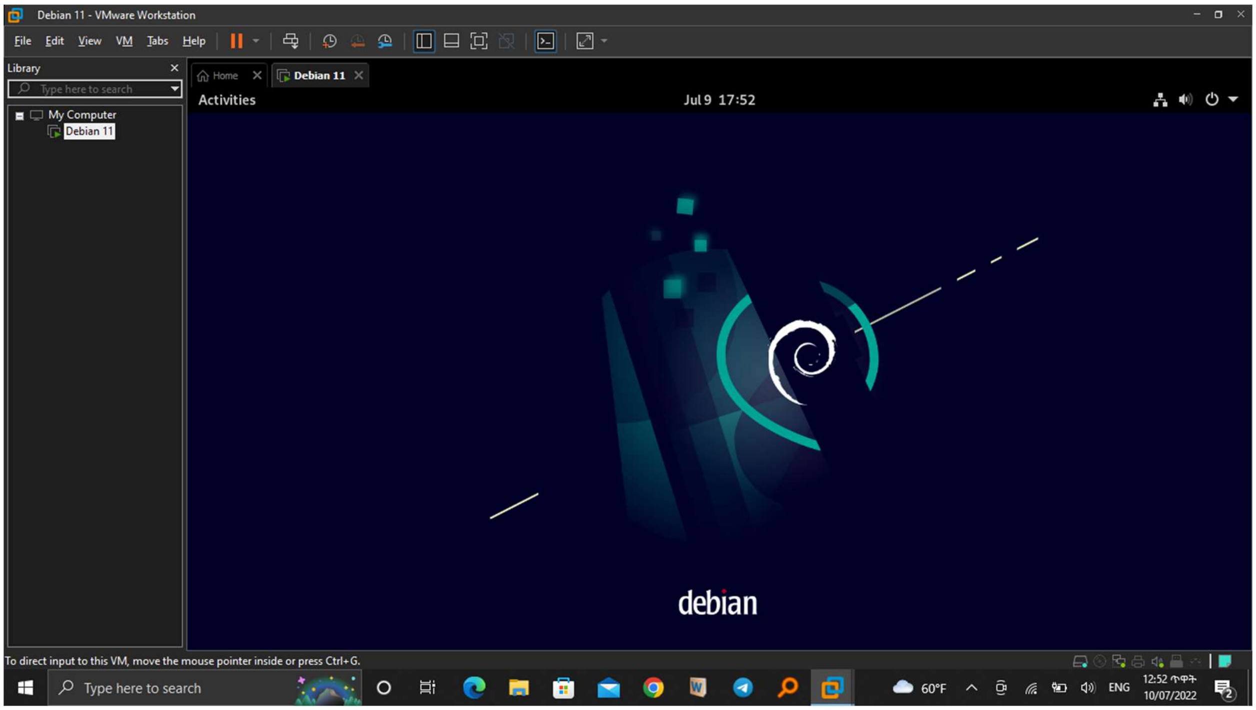
By using this code. We can go to the direct implementation

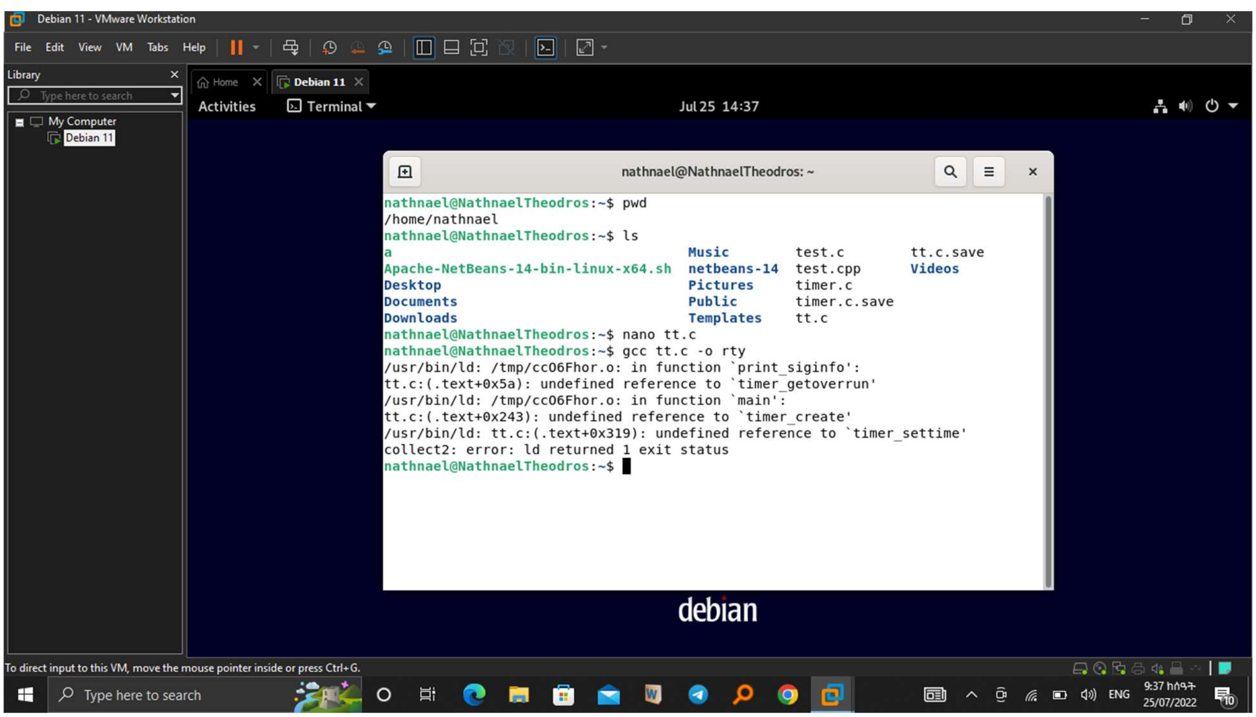
To implement, we should follow the following steps.

1. Open terminal

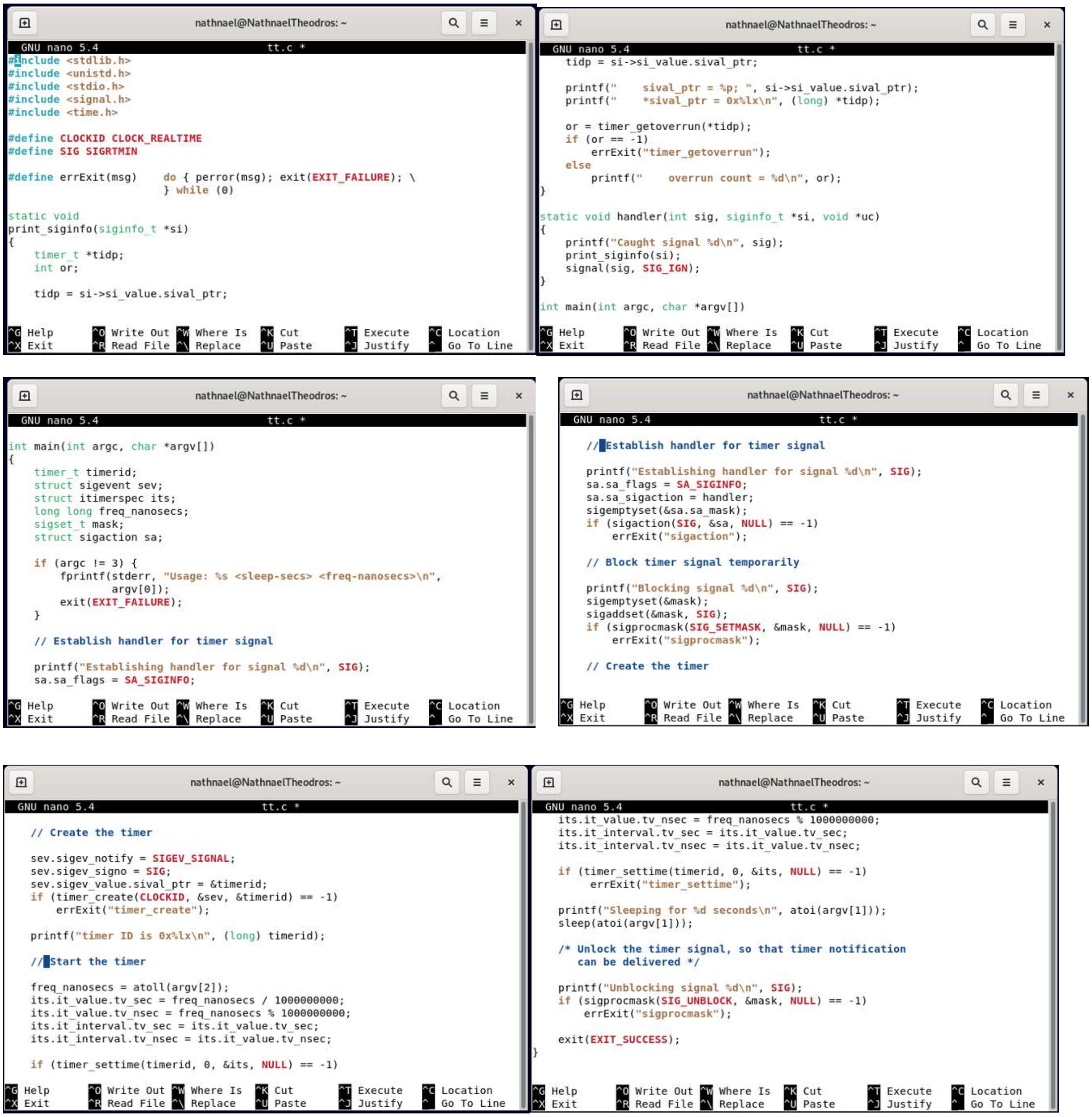
Here, what I did is I used the vmware virtual machine to open my virtual debian 11 operating system

And then I opened the terminal. This simple action is concluded by the following picture.





Step 2: I created a c file called “tt.c” by using the “nano tt.c” command. That allowed me to write the program above. This step implementation is shown in the picture below.



Step 3: After I wrote and save the file, I tried to execute the code by using “gcc tt.c –o n” command. The compiler was not able to compile it. This is because I need to add “-lpthread –lrt” command after the “gcc tt.c –o n”. This made the program to be properly executed. And then when we run a normal program, we have to use “./n” but, since the main function of the c program needs arguments. Finally the system call will work like this.

```
nathnael@NathnaelTheodros:~$ gcc tt.c -o n -lpthread -lrt
```

```
nathnael@NathnaelTheodros:~$ ./n 3 3
Establishing handler for signal 34
Blocking signal 34
timer ID is 0x55c25370d6b0
Sleeping for 3 seconds
Unblocking signal 34
Caught signal 34
    sival_ptr = 0x7fff3b5932a0;     *sival_ptr = 0x55c25370d6b0
    overrun count = 1002179064
nathnael@NathnaelTheodros:~$
```