

**Bahir dar university**

**Faculty of computing**

**Operating system system programing assignment2**

By JOHN ASSEFA

Id-1308160

Submitted to webdimu.B

Submission date 18/11/2014 E.C

**The Clone system calls**

The clone system call(\_clone2,clone3,-create child process) is a system call for creating a “child” process. the method is similar to that of the fork system call but varies in that the clone system calls provide more precise control over what pieces of execution context are shared between the calling process and the child process.

In the clone system call, we specify a function, which is simply a memory address the OS will set the program counter to when it runs the thread. Then, we specify a stack. This is a generic memory address that will become the stack for the child thread.

The flags portion of the clone system call is where the power is. This is where we can specify which resources are shared and what type of thread we want–**heavyweight thread** or **lightweight thread**. Here it might be important to mention the difference between lightweight and heavyweight. Processes come in two basic flavours, heavy-weight and light-weight. Heavy-weight processes are those that you normally think of when you run a program. They contain their own address space and program execution context on the other hand Light-weight processes are normally thought of as threads. These light-weight processes share the address space of the parent, and only contain some subset of the context elements. This is the one area that the clone system call differs from that of the fork system call.the clone system call can accommodate for both lightweight and heavyweight programs while the fork call only applies for heavyweight programs.

The clone system call has the following general structure

**int \_\_clone(int (\*fn) (void \*arg), void \*child\_stack, int flag, void \*args)**

as can be seen the system call has several parameters to it. All the parameters refered here correspond to one of the features mentioned above.he first parameter (“int (\*fn) (void \*arg)”)refers to the fuction on which the child procees is ment to run on.this part must refer to a function that returns an intiger variable.

The second part of the parameter deals with setting up the stack for the “child ” process.here the programmer sets its ideal stack storage for the child process.one good thing to remember while setting stack storage is that the stack set should be as fitting to the task given to the child process,to much of stack strage is waste of computer resource while to less of stack storage makes difficulity o the child process to run.

The third part of parameter(“int flag”) deals with the flags of the system call. for a short description, flags are small additives on system calls that enables a modified application of the call.for example the command “ls” is a single command to display all possible files and directories in the given directory, but with addition of the flag “-l” it can display more information about the files and directories in that given directory.like wisethe clone system call has several flags that can be applied to it and the place to specify which flag to be used is this third region**.**

The last parameter(“**void \*args**”) is a pointer to the arguments that will pass to the function that the child process will execute.in case the function doesn’t need to accept parameter(s) ,it is possible to set this parameter to null.

**Flags of the clone call**

As most system calls the clone call also has its flags.The flags available to the clone call are:

* **CLONE\_VM - share memory**

The code implementation of this flag is

**int \_\_clone(int (\*fn) (void \*arg), void \*child\_stack,** CLONE\_VM, **void \*args)**

Example

*#define \_GNU\_SOURCE*

*#include<unistd.h>*

*#include<stdio.h>*

*#include<sys/syscall.h>*

*#include<sys/utsname.h>*

*#include<sched.h>*

*#include<stdlib.h>*

*#include<string.h>*

*#include<signal.h>*

*#include<linux/sched.h>*

*#include<sys/wait.h>*

*int child\_function(void\* arg) {*

*char\* ch = (char\*)arg;*

*printf("Child sees ch = \"%s\"\n", ch);*

*strcpy(ch, " operating system assignment2,done by son of assefa,MESSGE FROM CHILD");*

*return 0;*

*}*

*int main(int argc, char\*\* argv) {*

*const int STACK\_SIZE = 65536;*

*char\* stack = malloc(STACK\_SIZE);*

*if (!stack) {*

*perror("malloc");*

*exit(1);*

*}*

*char ch[70];*

*strcpy(ch, "operating system assignment2,done by john ,MESSAGE FROM PARENT");*

*clone(child\_function, stack + STACK\_SIZE, CLONE\_VM, ch);*

*if (clone(child\_function, stack + STACK\_SIZE, CLONE\_VM, ch) == -1)*

*{*

*perror("clone");*

*int status;*

*if (wait(&status) == -1) {*

*perror("wait");*

*exit(1);*

*}*

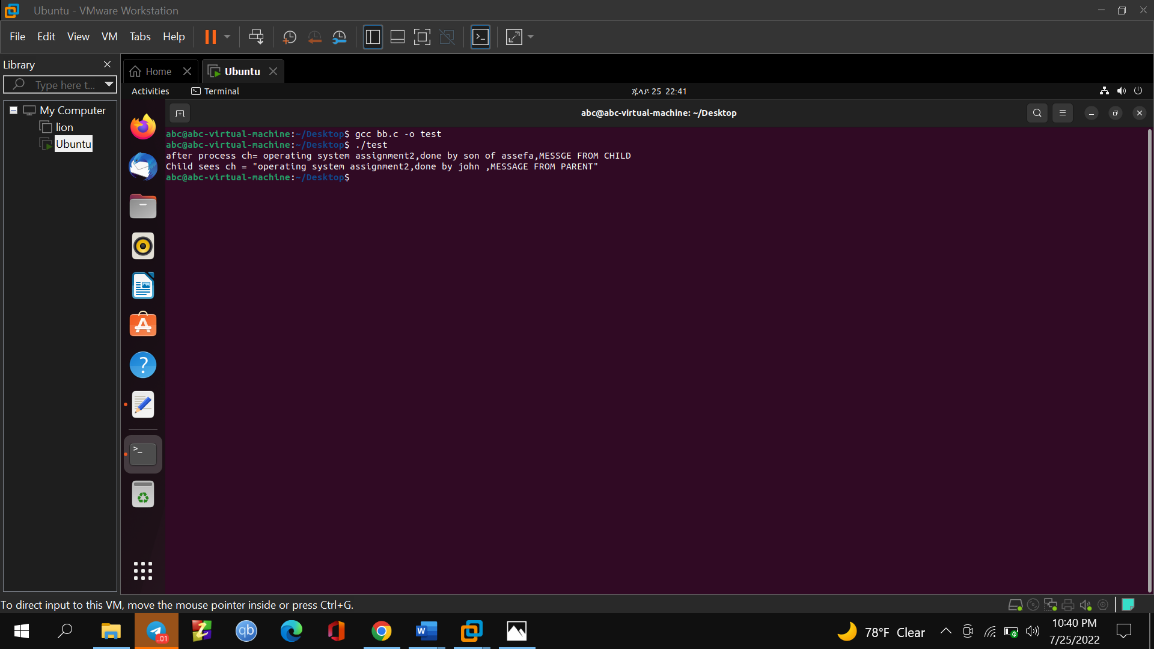
*printf("after process %d. ch = \"%s⁶0 status, ch);*

*return 0;*

*}*

when the vm argument is passed, CLONE\_VM is set and the child task shares the parent's memory. Its writing into ch will now be observable from the parent:

the output is



* **CLONE\_FILES - share file descriptors**

If CLONE\_FILES is set, the calling process and the child process share the same file descriptor table. Any file descriptor created by the calling process or by the child process is also valid in the other process. Similarly, if one of the processes closes a file descriptor, or changes its associated flags the other process is also affected.

If CLONE\_FILES is not set, the child process inherits a copy of all file descriptors opened in the calling process at the time of the clone call. Subsequent operations that open or close file descriptors, or change file descriptor flags, performed by either the calling process or the child process do not affect the other process. Note, however, that the duplicated file descriptors in the child refer to the same open file descriptions as the corresponding file descriptors in the calling process, and thus share file offsets and file status flags.

The code implementation of this flag is

**int \_\_clone(int (\*fn) (void \*arg), void \*child\_stack,** CLONE\_FILES, **void \*args)**

**Example**

*#define \_GNU\_SOURCE*

*#include<unistd.h>*

*#include<stdio.h>*

*#include<sys/syscall.h>*

*#include<sys/utsname.h>*

*#include<sched.h>*

*#include<stdlib.h>*

*#include<string.h>*

*#include<signal.h>*

*#include<linux/sched.h>*

*#include<sys/wait.h>*

*int child\_function(void\* arg) {*

*char\* ch = (char\*)arg;*

*printf("Child sees ch = \"%s\"\n", ch);*

*strcpy(ch, " operating system assignment2,done by son of assefa,MESSGE FROM CHILD");*

*return 0;*

*}*

*int main(int argc, char\*\* argv) {*

*const int STACK\_SIZE = 65536;*

*char\* stack = malloc(STACK\_SIZE);*

*if (!stack) {*

*perror("malloc");*

*exit(1);*

*}*

*char ch[70];*

*strcpy(ch, "operating system assignment2,done by john ,MESSAGE FROM PARENT");*

*clone(child\_function, stack + STACK\_SIZE, CLONE\_VM, ch);*

*if (clone(child\_function, stack + STACK\_SIZE, CLONE\_FILES, ch) == -1)*

*{*

*perror("clone");*

*exit(1);*

*}*

*int status;*

*if (wait(&status) == -1) {*

*perror("wait");*

*exit(1);*

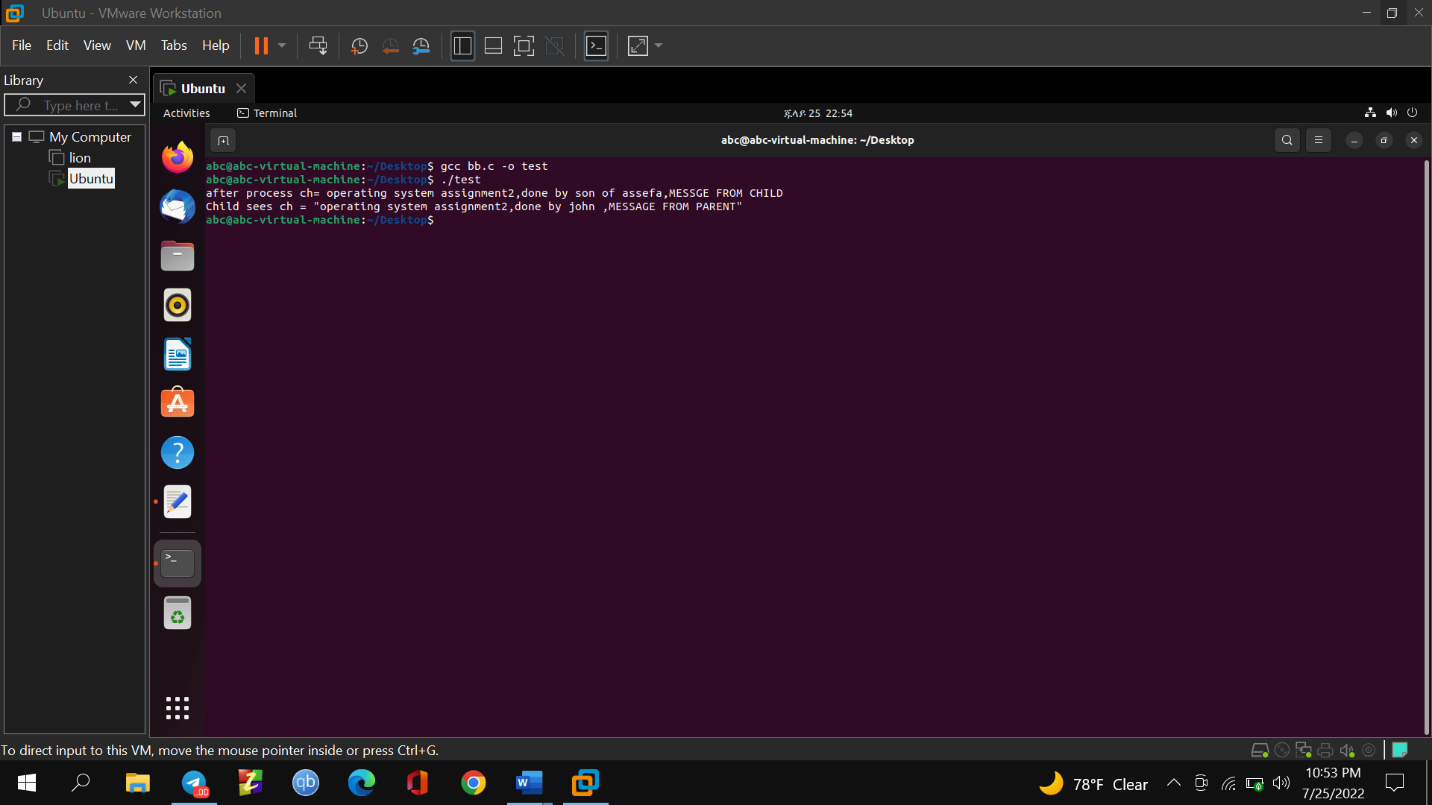
*}*

*printf("after process %d. ch = \"%s\"", status, ch);*

*return 0;*

*}*

Baised on the descripion above the output of the program is



* **CLONE\_SIGHAND - share signal handlers**

The code implementation of this flag is

**int \_\_clone(int (\*fn) (void \*arg), void \*child\_stack,** CLONE\_SIGHAND, **void \*args)**

Example

*#define \_GNU\_SOURCE*

*#include<unistd.h>*

*#include<stdio.h>*

*#include<sys/syscall.h>*

*#include<sys/utsname.h>*

*#include<sched.h>*

*#include<stdlib.h>*

*#include<string.h>*

*#include<signal.h>*

*#include<linux/sched.h>*

*#include<sys/wait.h>*

*int child\_function(void\* arg) {*

*char\* ch = (char\*)arg;*

*printf("Child sees ch = \"%s\"\n", ch);*

*strcpy(ch, " operating system assignment2,done by son of assefa,MESSGE FROM CHILD");*

*return 0;*

*}*

*int main(int argc, char\*\* argv) {*

*const int STACK\_SIZE = 65536;*

*char\* stack = malloc(STACK\_SIZE);*

*if (!stack) {*

*perror("malloc");*

*exit(1);*

*}*

*char ch[70];*

*strcpy(ch, "operating system assignment2,done by john ,MESSAGE FROM PARENT");*

*if (clone(child\_function, stack + STACK\_SIZE, CLONE\_SIGHAND, ch) == -1)*

*{*

*perror("clone");*

*exit(1);*

*}*

*int status;*

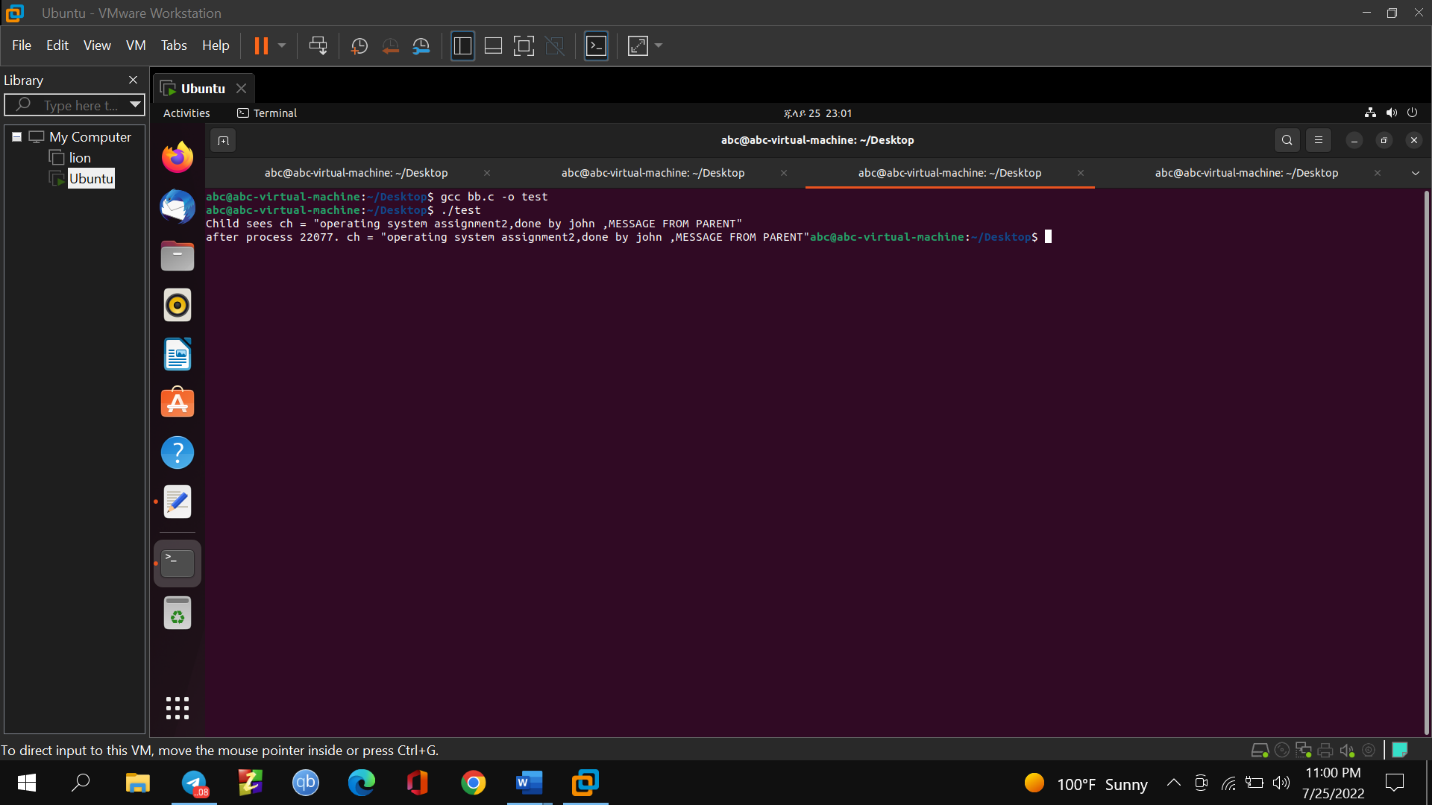
*printf("after process %d. ch = \"%s\"", status, ch);*

*return 0;*

*}*

Since this flag enables the sharing of only signal handlers the virtual memory is copied into the child. so when the child writes any other text on it the new text will not be accessible from the parent since it has the same address with the initial data set on it.

The output is

****

**SOME OTHER FLAGS INCLUDE**

**CLONE\_VFORK** - allow child to signal parent on exit

The code implementation of this flag is

**int \_\_clone(int (\*fn) (void \*arg), void \*child\_stack,** CLONE\_VFORK**, void \*args)**

**CLONE\_FS** - share filesystem

The code implementation of this flag is

**int \_\_clone(int (\*fn) (void \*arg), void \*child\_stack,** CLONE\_FS**, void \*args)**

**CLONE\_CHILD\_CLEARTID** (since Linux 2.5.49)

Clear (zero) the child thread ID at the location point to by *child\_tid* (**clone**()) or *cl\_args.child\_tid* (**clone3**()) in child memory when the child exits, and do a wakeup the futex at that address. The address involved may be changed by the [set\_tid\_address(2)](https://man7.org/linux/man-pages/man2/set_tid_address.2.html) system call. This is used by threading libraries.

**CLONE\_CHILD\_SETTID** (since Linux 2.5.49)

Store the child thread ID at the location pointed to by *child\_tid* (**clone**()) or *cl\_args.child\_tid* (**clone3**()) in the child's memory. The store operation completes before the clone call returns control to user space in the child process. (Note that the store operation may not have completed before the clone call returns in the parent process, which is relevant if the **CLONE\_VM** flag is also employed.)

**CLONE\_CLEAR\_SIGHAND** (since Linux 5.5)

By default, signal dispositions in the child thread are the same as in the parent. If this flag is specified, then all signals that are handled in the parent are reset to their default dispositions (**SIG\_DFL**) in the child. Specifying this flag together with **CLONE\_SIGHAND** is nonsensical and disallowed.

reference

* <https://eli.thegreenplace.net/2018/launching-linux-threads-and-processes-with-clone/>
* <https://www.linuxjournal.com/article/5211>
* <https://man7.org/linux/man-pages/man2/clone.2.html>
* Linux manual page