CLONE SYSTEM CALL

GUMMA SRI SOUGANDHIKA - 121AD0020

What is Clone system call

In Unix-like operating systems, the *clone()* system call is used to create a new process or thread. It is similar to the *fork()* system call but provides more control over the characteristics of the new process or thread. *clone()* allows you to specify which resources are shared between the calling process and the newly created one, such as the memory space, file descriptors, signal handlers, and more.

A simple example of Clone

We can implement a simple code snippet as follows on ubuntu:

CODE-1: CLONE_VM | SIGCHLD

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <sched.h>
#include <unistd.h>
#include <sys/wait.h>

#define STACK_SIZE (1024 * 1024)

// Function to be executed by the new process
int child_function(void *arg) {
    printf("Child process created successfully! PID: %d\n", getpid());
    return 0;
}

int main() {
    // Allocate stack for the new process
    void *stack = malloc(STACK_SIZE);
    if (stack == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }
```

```
    // Call clone() to create a new process
    int flags = CLONE_VM | SIGCHLD;
    pid_t pid = clone(child_function, (char *)stack + STACK_SIZE, flags, NULL);
    if (pid == -1) {
        perror("clone");
        exit(EXIT_FAILURE);
    }

    // Wait for the child process to terminate
    if (waitpid(pid, NULL, 0) == -1) {
        perror("waitpid");
        exit(EXIT_FAILURE);
    }

    // Free the allocated stack
    free(stack);
    return 0;
}
```

At the top, a child_function is defined that will be executed by the new process. Inside the main() function, we have first allocated a stack as memory for clone() through malloc().We have fixed the desired flags to be: CLONE_VM | SIGCHLD.

CLONE_VM: this flag indicates that the new process should share memory space of the calling process. In other words, parent and child processes have access to the same memory.

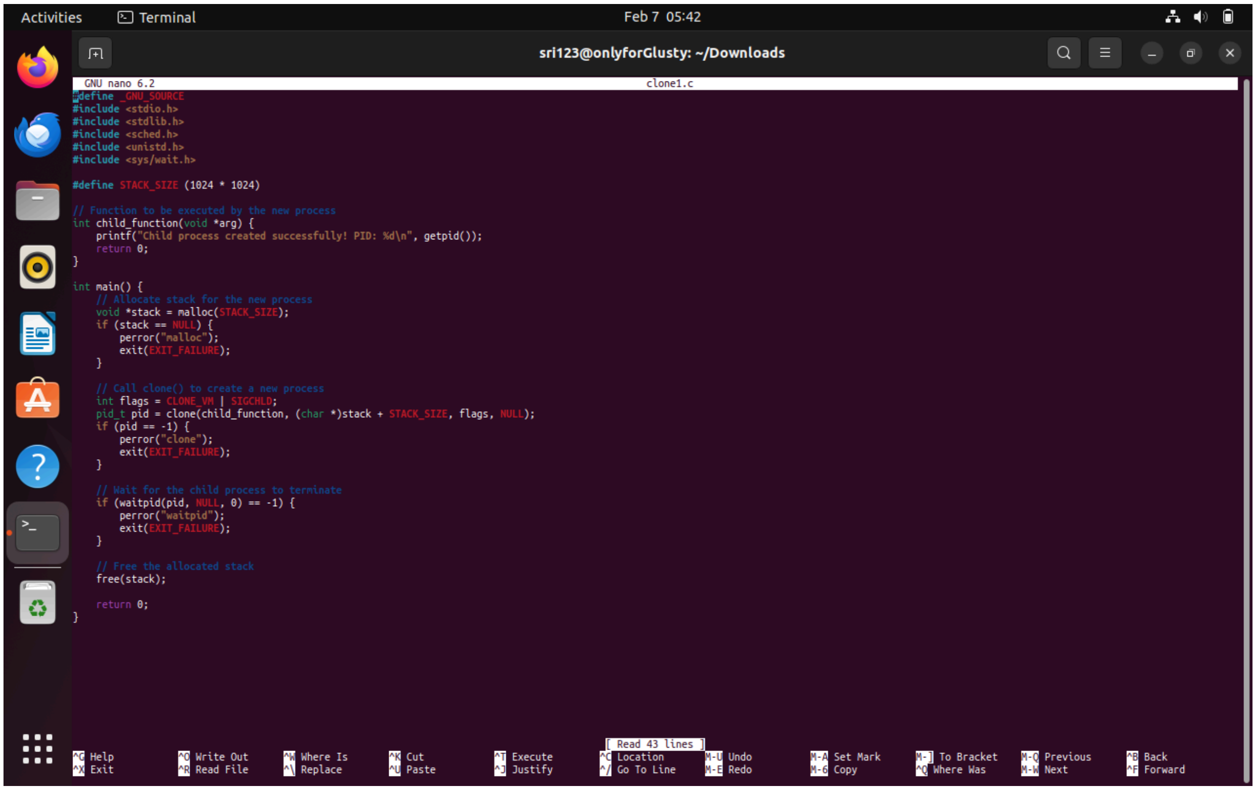
SIGCHLD: this flag is the signal number for the child process termination signal. This flag allows the parent to process information and perform cleanups and handling tasks.

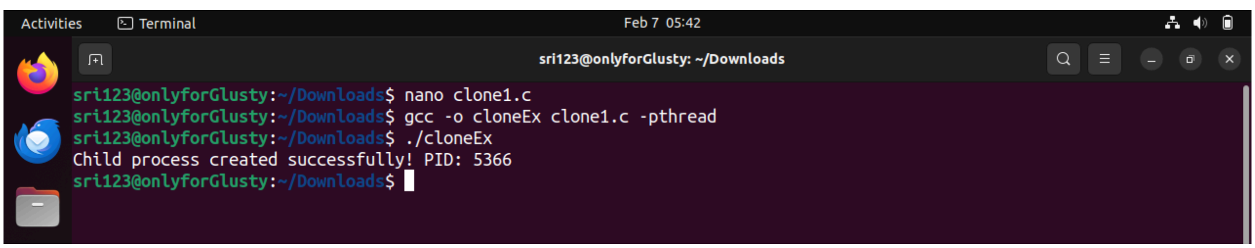
A bitwise operation **OR(|)** is done to combine both the flags to single integer value.

Next we call the function. It has the following arguments:

1. **child_function**: the function that has been defined above.
2. **(char*)stack + STACK_SIZE:** (char*)stack casts stack to a pointer, and stack_size is the size added to stack to point to the top of the stack.
3. **flags:** we defined them above, which will define the options for the clone process.
4. **NULL:** it is actually an argument pointer for the child_function. Since we are not having any arguments, we put NULL there.

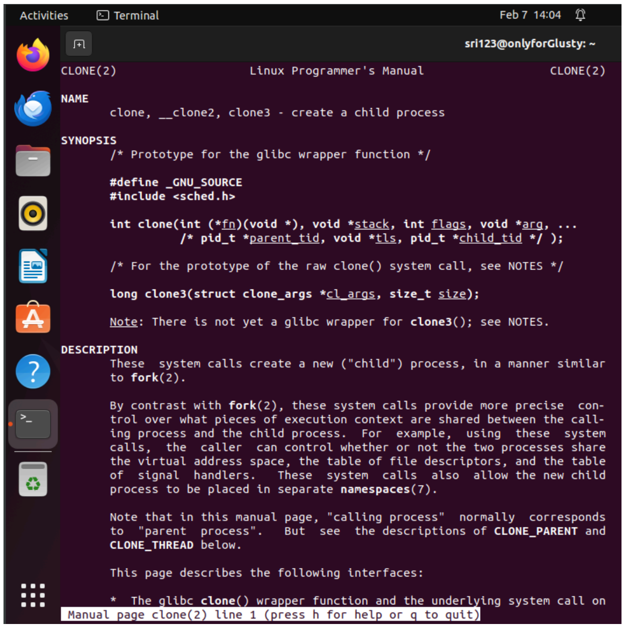
Next we have coded the error handling part, in case any errors occur. Finally we free the allocated memory stack. On running the program the results are as follows:

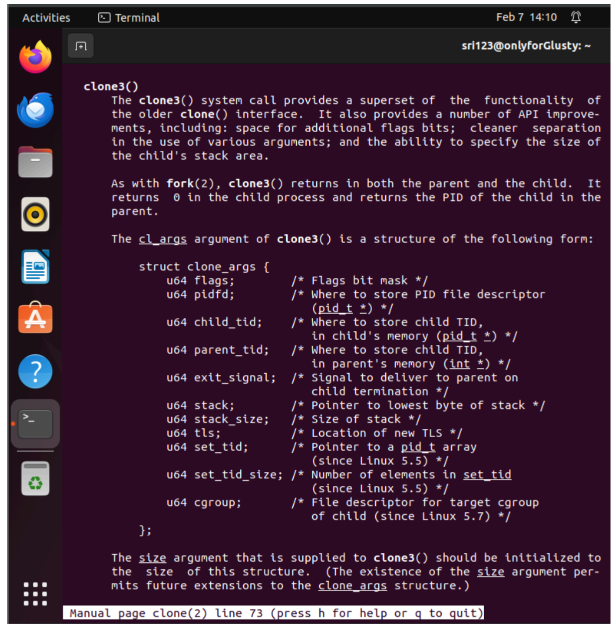




Clone documentation: man 2 clone

- In the “man 2 clone” command, we can see that the requirements are _GNU_SOURCE definition and the library <sched.h>.
- It mentions the newer *clone3() call.* It provides a superset of functionality of older clone(). It involves additional flag bits, cleaner separation in use of various arguments, and ability to specify a child's stack area as well.





- The child termination signal: this termination signal is specified in the low byte of <u>flags.</u> If this signaling is not done then the parent process does not know of the child process termination.
- Various flag details are also mentioned in the manual with their descriptions.
- Return value: On success the thread ID of the child is returned. Else, -1 is returned and no child process will be created.
- After that a brief description of errors is provided such as **EAGAIN**(when too many processes are already running), **EEXIST**(in clone3() only, PIDs specified are already existing in the corresponding namespace), **EBUSY**(in clone3() only, cgroup is currently managing resources and cannot accept new processes), **EINVAL** and more.
- This clone() system call is mainly used for implementing the threads, multiple flows of control in a program that run concurrently in shared address space.

Experimenting with the flags

There are many flags for the clone() call, some common ones are:

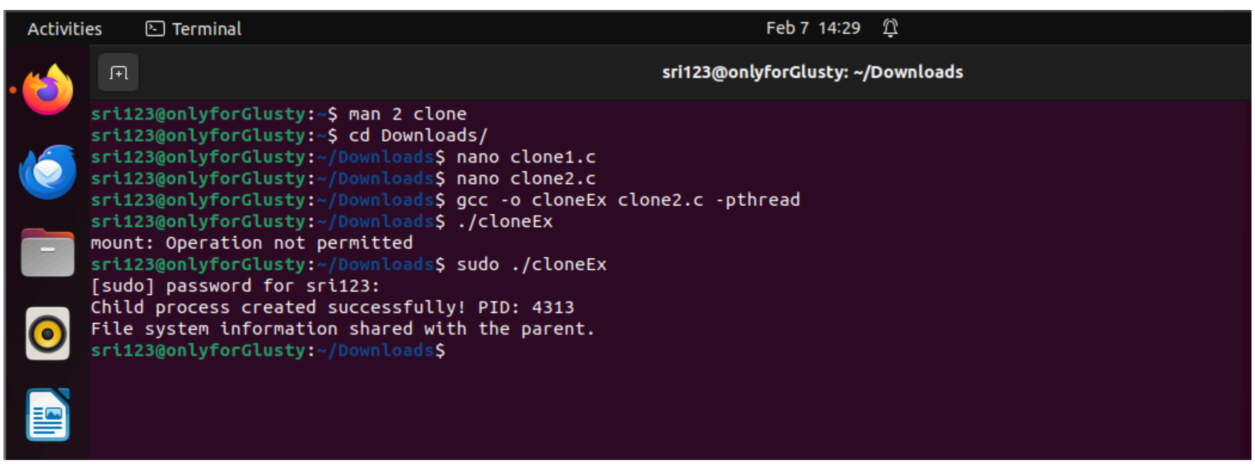
1. **CLONE_VM**: Shares the memory space with the calling process.
2. **CLONE_FS**: Shares the file system information.
3. **CLONE_FILES**: Shares file descriptors.
4. **CLONE_SIGHAND**: Shares signal handlers.
5. **CLONE_THREAD**: Indicates that the new process is a thread in the same thread group.
6. **CLONE_NEWPID**: Creates a new PID namespace for the new process.
7. **CLONE_NEWNET**: Creates a new network namespace.

In the above program we have seen that CLONE_VM shares the memory space. Now we will have a code for the CLONE_FS, or sharing file system information.

CODE-2: CLONE_VM | CLONE_FS | SIGCHLD

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>
#include <sys/wait.h>
#include <sys/mount.h>

#define STACK_SIZE (1024 * 1024)

// Function to be executed by the new process
int child_function(void *arg) {
    // Check if the filesystem information is shared
    if (mount(NULL, "/", NULL, MS_SHARED | MS_REC, NULL) == -1) {
        perror("mount");
        exit(EXIT_FAILURE);
    }

    printf("Child process created successfully! PID: %d\n", getpid());
    printf("File system information shared with the parent.\n");
    return 0;
}
```

```
int main() {
    // Allocate stack for the new process
    void *stack = malloc(STACK_SIZE);
    if (stack == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    // Call clone() to create a new process with CLONE_FS flag
    int flags = CLONE_VM | CLONE_FS | SIGCHLD;
    pid_t pid = clone(child_function, (char *)stack + STACK_SIZE, flags, NULL);
    if (pid == -1) {
        perror("clone");
        exit(EXIT_FAILURE);
    }

    // Wait for the child process to terminate
    if (waitpid(pid, NULL, 0) == -1) {
        perror("waitpid");
        exit(EXIT_FAILURE);
    }

    // Free the allocated stack
    free(stack);

    return 0;
}
```



As we can see above, we can see that some information is getting exchanged, by “mount” action and we are now sharing file system information along with the memory allocated between child and parent processes.

Another interesting flag can be CLONE_NEWPID, it assigns addresses in a new namespace for new processes.

CODE-3: CLONE_VM | CLONE_NEWPID | SIGCHLD

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>
#include <sys/wait.h>

#define STACK_SIZE (1024 * 1024)

// Function to be executed by the new process
int child_function(void *arg) {
    printf("Child process created successfully! PID: %d\n", getpid());

    // Execute 'ps' command in the child process to show its PID
    system("ps -o pid,ppid,cmd");

    return 0;
}

int main() {
    // Allocate stack for the new process
    void *stack = malloc(STACK_SIZE);
    if (stack == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    // Call clone() to create a new process with CLONE_NEWPID flag
    int flags = CLONE_VM | CLONE_NEWPID | SIGCHLD;
    pid_t pid = clone(child_function, (char *)stack + STACK_SIZE, flags, NULL);
    if (pid == -1) {
        perror("clone");
        exit(EXIT_FAILURE);
    }

    // Wait for the child process to terminate
    if (waitpid(pid, NULL, 0) == -1) {
        perror("waitpid");
        exit(EXIT_FAILURE);
    }

    // Free the allocated stack
    free(stack);
```

```
    return 0;
}
```

Output is as follows:

```
sri123@onlyforGlusty:~/Downloads$ nano clone1.c
sri123@onlyforGlusty:~/Downloads$ gcc -o cloneEx clone1.c -pthread
sri123@onlyforGlusty:~/Downloads$ sudo ./cloneEx
[sudo] password for sri123:
Child process created successfully! PID: 1
    PID    PPID CMD
   5051    5047 sudo ./cloneEx
   5052    5051 ./cloneEx
   5053    5052 ./cloneEx
   5054    5053 sh -c ps -o pid,ppid,cmd
   5055    5054 ps -o pid,ppid,cmd
sri123@onlyforGlusty:~/Downloads$
```

The parent process (./cloneEx) has PID 5051, and its parent process (PPID) is the process with PID 5047. The child process (./cloneEx) created with the clone() system call has PID 5052, and its parent process (PPID) is the parent process (PID 5051). This confirms that **5052 is a direct child of 5051.**

Subsequent processes (PID 5053, 5054, 5055) are descendants of PID 5052. They are created within the child process's namespace, and their PIDs are unique within that namespace.

Let us now look at another example of clone with flags CLONE_VM and CLONE_FS with some file handling to get a better picture of how the resources are shared and up to what extent.

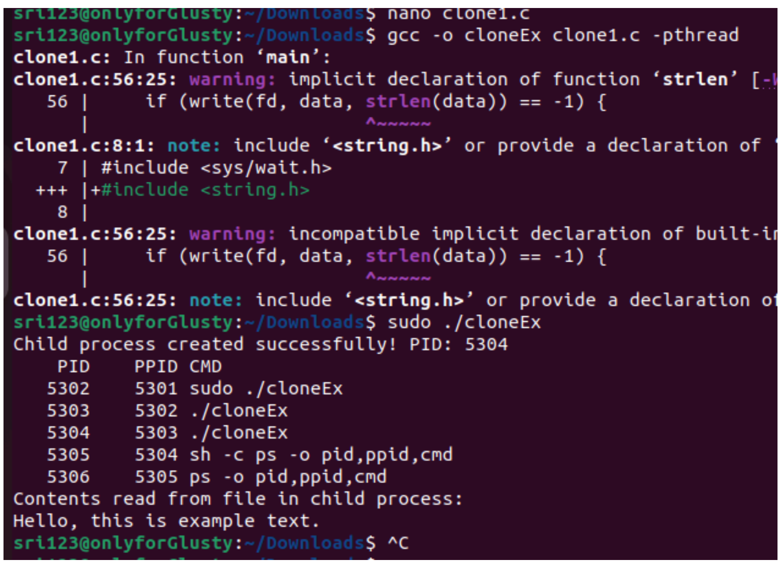
CODE-4: CLONE_VM | CLONE_FS | SIGCHLD (WITH FILE-HANDLING)

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sched.h>
#include <sys/wait.h>

#define STACK_SIZE (1024 * 1024)

// Function to be executed by the new process
int child_function(void *arg) {
```

```c
    printf("Child process created successfully! PID: %d\n", getpid());
    system("ps -o pid,ppid,cmd");
    // Open a file in the child process
    int fd = open("example.txt", O_RDONLY);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    // Read from the file
    char buffer[256];
    ssize_t bytes_read = read(fd, buffer, sizeof(buffer));
    if (bytes_read == -1) {
        perror("read");
        exit(EXIT_FAILURE);
    }

    // Print the contents read from the file
    printf("Contents read from file in child process:\n");
    write(STDOUT_FILENO, buffer, bytes_read);
    printf("\n");

    // Close the file descriptor
    close(fd);

    return 0;
}

int main() {
    // Allocate stack for the new process
    void *stack = malloc(STACK_SIZE);
    if (stack == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    // Create a file and write some content to it
    int fd = open("example.txt", O_CREAT | O_WRONLY | O_TRUNC, 0644);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    const char *data = "Hello, this is example text.";
    if (write(fd, data, strlen(data)) == -1) {
        perror("write");
        exit(EXIT_FAILURE);
    }
    close(fd);
```

```
    // Call clone() to create a new process with CLONE_FS flag
    int flags = CLONE_VM | CLONE_FS | SIGCHLD;
    pid_t pid = clone(child_function, (char *)stack + STACK_SIZE, flags, NULL);
    if (pid == -1) {
        perror("clone");
        exit(EXIT_FAILURE);
    }

    // Wait for the child process to terminate
    if (waitpid(pid, NULL, 0) == -1) {
        perror("waitpid");
        exit(EXIT_FAILURE);
    }

    // Free the allocated stack
    free(stack);

    return 0;
}
```

Output:



The parent process has PID 5302, and its parent process (PPID) is the process with PID 5301.The child process (./cloneEx) created by the parent process (5302) has PID 5303, and its parent process (PPID) is the parent process (5302). We can see the PID 5304 (child). The child process (./cloneEx) created by the child process (5303) has PID 5304, and its parent process (PPID) is the child process (5303). Hence, all processes (5302, 5303, 5304) are running in the same namespace, unlike the case where we used the CLONE_NEWPID flag.

A network namespace allows each of these processes to see an entirely different set of networking interfaces.

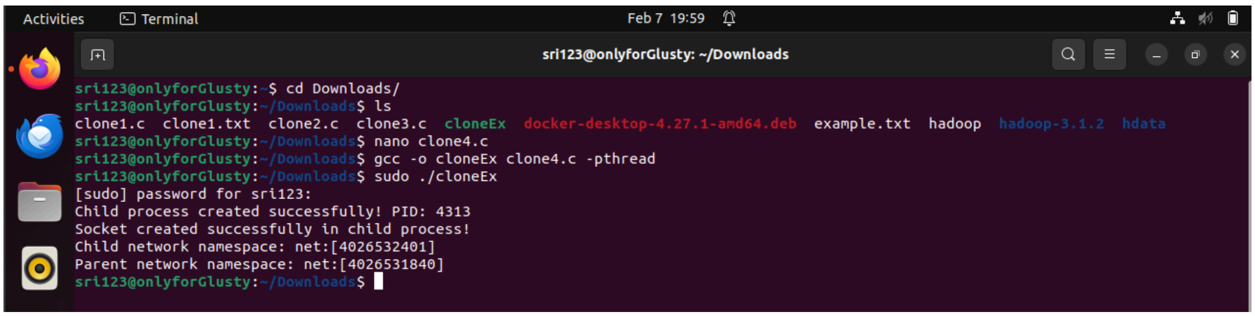
CODE-5: CLONE_VM | CLONE_NEWNET | SIGCHLD

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>
#include <sys/wait.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#include <string.h>

#define STACK_SIZE (1024 * 1024)

// Function to be executed by the new process
int child_function(void *arg) {
    printf("Child process created successfully! PID: %d\n", getpid());

    // Create a new network socket in the child process
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        perror("socket");
        exit(EXIT_FAILURE);
    }
    printf("Socket created successfully in child process!\n");

    // Close the socket
    close(sockfd);

    // Get the network namespace of the child process
    char child_ns[256];
    snprintf(child_ns, sizeof(child_ns), "/proc/%d/ns/net", getpid());

    // Read the symbolic link target
    char child_target[256];
    ssize_t child_read = readlink(child_ns, child_target, sizeof(child_target)
- 1);
    if (child_read == -1) {
        perror("readlink");
        exit(EXIT_FAILURE);
    }
    child_target[child_read] = '\0';
```

```c
    printf("Child network namespace: %s\n", child_target);

    return 0;
}

int main() {
    // Allocate stack for the new process
    void *stack = malloc(STACK_SIZE);
    if (stack == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    // Call clone() to create a new process with CLONE_NEWNET flag
    int flags = CLONE_VM | CLONE_NEWNET | SIGCHLD;
    pid_t pid = clone(child_function, (char *)stack + STACK_SIZE, flags, NULL);
    if (pid == -1) {
        perror("clone");
        exit(EXIT_FAILURE);
    }

    // Wait for the child process to terminate
    if (waitpid(pid, NULL, 0) == -1) {
        perror("waitpid");
        exit(EXIT_FAILURE);
    }

    // Free the allocated stack
    free(stack);

    // Get the network namespace of the parent process
    char parent_ns[256];
    snprintf(parent_ns, sizeof(parent_ns), "/proc/%d/ns/net", getpid());

    // Read the symbolic link target
    char parent_target[256];
    ssize_t parent_read = readlink(parent_ns, parent_target,
sizeof(parent_target) - 1);
    if (parent_read == -1) {
        perror("readlink");
        exit(EXIT_FAILURE);
    }
    parent_target[parent_read] = '\0';

    printf("Parent network namespace: %s\n", parent_target);

    return 0;
}
```

In this code, the child process has a new network socket using the socket() system call. This demonstrates that the child process has its own isolated network stack. The successful creation of socket itself tells that it is in an isolated new network namespace.

To further support this statement, we have printed the network namespace of the child process by constructing the path to the symbolic link /proc/<pid>/ns/net and reading its target using the readlink() function. The results quite clearly tell that a new network has been created.



Linux also maintains a data structure for all the mountpoints of the system. It includes information like what disk partitions are mounted, where they are mounted, whether they are read-only. Creating separate mount namespace has an effect similar to doing a chroot(), This allows you to have a different root for each isolated process, as well as other mount points that are specific to those processes.

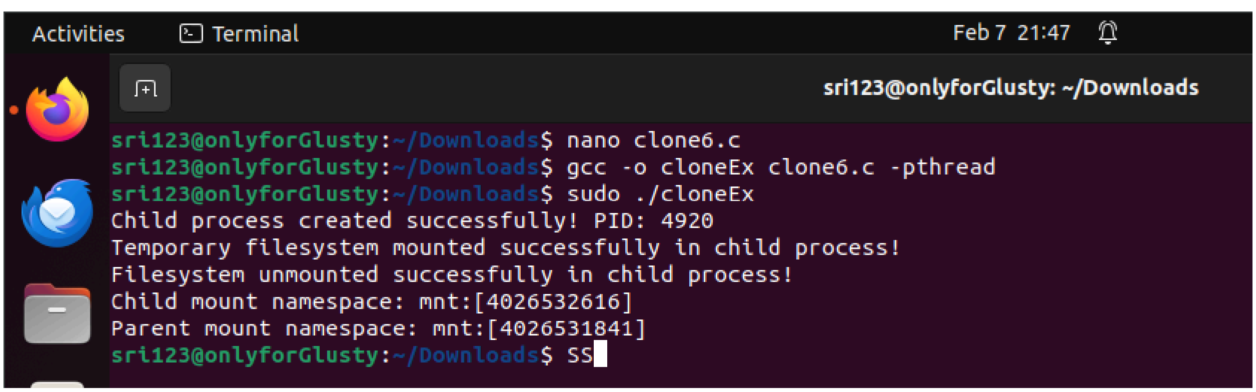
CODE-6: CLONE_VM | CLONE_NEWNS | SIGCHLD

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>
#include <sys/wait.h>
#include <sys/mount.h>
#include <errno.h>
#include <string.h>

#define STACK_SIZE (1024 * 1024)

// Function to be executed by the new process
int child_function(void *arg) {
```

```
    printf("Child process created successfully! PID: %d\n", getpid());

    // Mount a temporary filesystem in the child process
    if (mount("none", "/mnt", "tmpfs", 0, NULL) == -1) {
        perror("mount");
        exit(EXIT_FAILURE);
    }
    printf("Temporary filesystem mounted successfully in child process!\n");

    // Unmount the filesystem
    if (umount("/mnt") == -1) {
        perror("umount");
        exit(EXIT_FAILURE);
    }
    printf("Filesystem unmounted successfully in child process!\n");

    // Get the mount namespace of the child process
    char child_ns[256];
    snprintf(child_ns, sizeof(child_ns), "/proc/%d/ns/mnt", getpid());

    // Read the symbolic link target
    char child_target[256];
    ssize_t child_read = readlink(child_ns, child_target, sizeof(child_target)
- 1);
    if (child_read == -1) {
        perror("readlink");
        exit(EXIT_FAILURE);
    }
    child_target[child_read] = '\0';

    printf("Child mount namespace: %s\n", child_target);

    return 0;
}

int main() {
    // Allocate stack for the new process
    void *stack = malloc(STACK_SIZE);
    if (stack == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    // Call clone() to create a new process with CLONE_NEWNS flag
    int flags = CLONE_VM | CLONE_NEWNS | SIGCHLD;
    pid_t pid = clone(child_function, (char *)stack + STACK_SIZE, flags, NULL);
    if (pid == -1) {
        perror("clone");
        exit(EXIT_FAILURE);
```

```
    }

    // Wait for the child process to terminate
    if (waitpid(pid, NULL, 0) == -1) {
        perror("waitpid");
        exit(EXIT_FAILURE);
    }

    // Free the allocated stack
    free(stack);

    // Get the mount namespace of the parent process
    char parent_ns[256];
    snprintf(parent_ns, sizeof(parent_ns), "/proc/%d/ns/mnt", getpid());

    // Read the symbolic link target
    char parent_target[256];
    ssize_t parent_read = readlink(parent_ns, parent_target,
sizeof(parent_target) - 1);
    if (parent_read == -1) {
        perror("readlink");
        exit(EXIT_FAILURE);
    }
    parent_target[parent_read] = '\0';

    printf("Parent mount namespace: %s\n", parent_target);

    return 0;
}
```

In the child function above, we are using the “mount” and “unmount” calls, we mount and then unmount a temporary filesystem. This demonstrates that the child process's mount namespace is isolated from the parent process's mount namespace.



How clone() is different from pthread_create() and fork()

FORK(): The child process is an exact copy of the parent process, including memory, file descriptors, and other resources. Since the parent and child processes have different address spaces, any modifications made to one process will not reflect on the other.
PTHREAD_CREATE(): Threads share the same memory space and resources within the process. Changes made by one thread are visible to all other threads. Threads share the same address space and can communicate more easily but must be synchronized to avoid conflicts.
CLONE(): The level of resource sharing can be customized using flags. The clone() system call is an upgraded version of the fork call. It's powerful since it creates a child process and provides more precise control over the data shared between the parent and child processes.

Real world use-cases of Clone

One real-world use case of the clone() system call is in containerization technologies such as Docker. In Docker, each container runs in its own isolated environment, providing a lightweight, portable, and consistent way to package and deploy applications. The clone() system call is instrumental in creating these isolated environments. It is used in the following concepts:

1. **Namespace Isolation**: using flags like "**CLONE_NEWPID**" creates separate namespaces for various resources such as PID, network, mount, and more.
2. **File System Isolation**: clone() along with **chroot** and **mount** is helpful in creating a separate file system namespace for each container.
3. **Resource Control:** with appropriate flags we can enforce resource limits on the newly created container, ensuring that it doesn't consume more resources than allocated.
4. **Process Isolation:** clone() provides necessary isolation and encapsulation.

Conclusion

We have explored the clone() command in detail, where we covered the main arguments, working of clone(), flags in clone() and their speciality in controlling the extent of shared resources between the parent and child processes.

We also have gained insight on how useful these flags can be in creating environments where the isolated networks and isolated process ID namespaces are required. One such beautiful example, which we have already come across is Docker, which has easy and convenient features to manage and handle the resources.

Network isolation, hardware and software resource allocations, process ID allocations, everything play a huge role in containerizing technologies. All these are configured using the clone command.

Apart from that, we also have compared on how it is better than fork() or pthread_create() calls. With this statement, we conclude that the clone() call is significant for system administrators, developers, and kernel programmers.

We have learned about the importance of "namespaces" in the applications. Linux namespaces allow other aspects of the operating system to be independently modified as well. We can say that namespaces are important where isolation is required, for example, a server where multiple tasks are being run at once.

A virtual machine is heavy and emulates a hardware layer over the system OS and runs another OS over it. But, the namespaces, such as the ones used in Docker containers, ensure a similar level of isolation, but without the emulating hardware over the host OS. This is what makes them light-weight.

References

https://man7.org/linux/man-pages/man2/clone.2.html

https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces