Design and Implementation of Full Fork Functionality in the Linux Kernel

Bharambe Soham Amit sohamb21
Divyansh divyansh21
Divyansh Chhabria divyanshc21

Rajeev Kumar rajeevks21
Sandeep Nitharwal nsandeep21

*@iitk.ac.in
Department of Computer Science and Engineering
Indian Institute of Technology Kanpur

Introduction

In computing, particularly in the context of the Unix operating system and its workalikes, `fork` is a functionality whereby a process creates a copy of itself. Fork is the primary method of process creation on Unix-like operating systems. In multitasking operating systems, processes (running programs) need a way to create new processes, e.g. to run other programs. Fork and its variants are typically the only way of doing so in Unix-like systems. [1]

For multi-threaded processes, `fork` doesn't handle all the threads and we extend this functionality in our work to `full_fork`. We also extend the functionality to fork an entire process tree including all the threads of each process. This could be useful for specific use cases like checkpointing, debugging, or creating parallel instances of the same process. Nonetheless, the implementation of full fork presents its own set of challenges, including synchronization issues, memory and resource overheads, all of which we aim to address in this project.

Following this introduction, the *scope* section outlines the detailed objectives of our project, *milestone planning* elucidates the progression of our work alongside deadlines.

Scope

Although multi-threaded programming has demonstrated its effectiveness in reducing computation time, it has also introduced challenges for programmers. One notable issue is the potential for subtle data races due to specific thread interleavings. It could be therefore beneficial to capture snapshots of the program's execution state at intermediate states.
This project seeks to address this need efficiently by introducing a new system call called `full_fork`. The `full_fork` system call involves replicating the entirety of a process sub-tree, including all of its threads, as opposed to the standard fork operation which duplicates only the calling thread. This means that not only the main process but also its children, their own children, etc, along with the auxiliary threads spawned by all of them are duplicated, resulting in an exact replica of the original process sub-tree with all its associated threads. This approach ensures that the new process maintains the same state and execution context as the original process tree, providing a complete copy of the entire process environment.

Fullfork

Forking a Process and All its Threads

To implement this we initiated a new syscall that runs our module of `fullfork` and caller process gets fullforked after the complete execution of our module. The `Fullfork` consist of three major steps which were:

1. `Stopping the thread processes`
2. `Creating the copy of the stopped threads and the parent process`
3. `Releasing the wait_queue to resume the newly created processes`

With the current process in our hands, we checked if there are any threads spawned by this process and if there are, we send a `SIGSTOP` to one of these threads using `kill_pid`, as this signal will then be propagated to the entire thread group. Now we want the current to wait for the threads to stop and this is accomplished using `fullfork_wait` which we have implemented inside kernel.

We needed to use multitude of hooks in various places in this part so that

- The signal which we sent won't affect current and would only affect the rest of its' siblings.
- current will wait for the threads which are its' siblings and not for children to stop.
- the threads will send the signal to thread group leader and not their actual parent which is also the parent of current.
- When the signal is received by current it will check if the signal is sent by a sibling and not by a children.

Now that all the processes are stopped we call the `leader_clone` function.

Inside `leader_clone`, we clone the process which called syscall 548 and then wait for its' threads to get cloned and get attached to this newly cloned process.
The newly cloned leader now is again caught by a hook in `schedule_tail` where all the newly forked processes come at the beginning.
We create a thread of this cloned leader process by calling `my_kernel_clone` which is implemented inside the kernel right beside the original `kernel_clone`.

```
    void my_syscall_handler(struct pt_regs *regs, int sysnr)
{
        if (sysnr == 548)
        {
                /*Stop all threads using signal*/
        // skip sending signal if there an=re no threads
                if (temp == current)
                        goto clone;

                ff_parent_changer = &parent_changer;
                ff_group_leader = &group_leader;
                ff_eligible_pid = &is_eligible_pid;

                err = kill_pid(temp->thread_pid, SIGSTOP, 1);

                err = fullfork_wait(temp->pid, &status);

                ff_eligible_pid = NULL;
                ff_parent_changer = NULL;
                ff_group_leader = NULL;


        clone:
                err = leader_clone(regs);
                if(temp != current) err = kill_pid((temp->
                   thread_pid), SIGCONT, 0);
        }
}
```

Then we get the `task_struct` of this newly formed thread using the `child_pid` which was returned from the function, and set the registers using the stopped thread of the original process.
This procedure is done multiple times by looping over the threads of original process , which we stopped previously, using the macro `next_thread`. We have currently concluded all the handling and so we wake up all the processes.

Now we go onto testing and benchmarking.

Testing and Benchmarking of Fullfork

In this part, we tested the current implementation using a variety of unit test cases consisting of cases where we are calling full fork during the cases when the processes are sleeping, rescheduled, spinning, forking, etc. As we tested a lot while implementing earlier, a few minor bugs were found, which were solved easily. We now moved on to the benchmarking phase, where we tried to identify the efficient bottleneck of our implementation in comparison to a normal fork. The results are as follows:

*Observations Our initial assumption was that our major amount of time would be taken in waiting for the processes to stop, but we were proven wrong by these observations as it seems cloning so many times was actually the bottleneck.

Reproducing Fullfork

In order to reproduce the fullfork functionality on any generic kernel we will need to modify certain functions of the kernel, add few hooks at required place. Here is a detailed list down of all the modifications needed:

1. **ThreadStop: ParentChanger**: Thread sends, by default, a signal to their parent on changing state. Send this to the thread group leader instead. Make the following changes in `do_notify_parent_cldstop` in `kernel/signal.c`

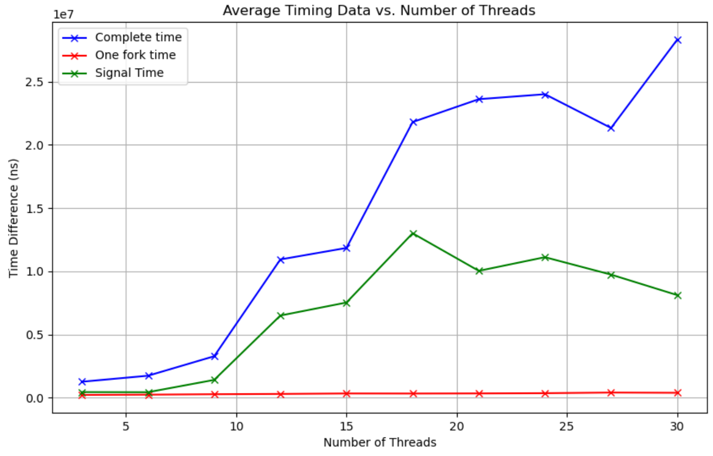

Figure 1: Fullfork time vs number of threads

```
        // hook above the function:
           do_notify_parent_cldstop

struct task_struct* (*ff_parent_changer)(struct
   task_struct *tsk) = NULL;
EXPORT_SYMBOL(ff_parent_changer);

// change inside: do_notify_parent_cldstop

    if (for_ptracer) {
        parent = tsk->parent;
    }
    else {
        tsk = tsk->group_leader;
        parent = tsk->real_parent;

        /* add this */
        if(ff_parent_changer)
        {
            parent = ff_parent_changer(tsk);
        }
        /* change ends */
    }
```

2. **ThreadStop: FullFork Wait** : The main wait function, to be called, to wait for the stopping of all threads. Add this in kernel/exit.c (preferably after kernel_wait4)

```
    int fullfork_wait(pid_t pid, int *stat)
{
        struct wait_opts wo = {
                .wo_type        = PIDTYPE_PID,
                .wo_pid         = find_get_pid(pid),
                .wo_flags       = WSTOPPED|WEXITED|
                   WUNTRACED|__WCLONE,
        };
        int ret;
        ret = do_wait(&wo);
        if (ret > 0 && wo.wo_stat)
```

```
                    *stat = wo.wo_stat;
        put_pid(wo.wo_pid);
        return ret;
}
EXPORT_SYMBOL(fullfork_wait);
```

3. **ThreadStop: Eligible PID** : Change the definition of what it means as an eligible pid, when waiting. In general, it is only a child that you can wait for. But extend this definitions for your siblings. Make changes in the following functions in kernel/exit.c
   (a) eligible_pid
   (b) is_effectively_child
   (c) do_wait_pid

```
    // Register the hook
bool (*ff_eligible_pid)(struct task_struct *p) = NULL;
EXPORT_SYMBOL(ff_eligible_pid);

/* change 1 : eligible_pid */
static int eligible_pid(struct wait_opts *wo, struct
   task_struct *p)
{
        /* change starts here */
        if(ff_eligible_pid)
        {
                if(ff_eligible_pid(p))
                {
                        return 1;
                }
        }
        /* change ends here */

        return  wo->wo_type == PIDTYPE_MAX ||
                task_pid_type(p, wo->wo_type) == wo->
                   wo_pid;
}
```

```
    /* change 2 */

static bool is_effectively_child(struct wait_opts *wo,
   bool ptrace,
                                 struct task_struct *
                                    target)
{
        struct task_struct *parent =
                !ptrace ? target->real_parent : target->
                   parent;

        /* change starts here */

        if(ff_eligible_pid)
                if(ff_eligible_pid(target))
                        return true;

        /* change ends here */

        return current == parent || (!(wo->wo_flags &
           __WNOTHREAD) &&
                                            same_thread_group(
                                               current, parent))
                                               ;
}
```

```
    static int do_wait_pid(struct wait_opts *wo)
{
        bool ptrace;
        struct task_struct *target;
        int retval;

        ptrace = false;
        target = pid_task(wo->wo_pid, PIDTYPE_TGID);

        /* change starts here */
        if(ff_eligible_pid)
                if(ff_eligible_pid(current))
                        target = pid_task(wo->wo_pid,
                           PIDTYPE_PID);
        /* change ends here */
        if (target && is_effectively_child(wo, ptrace,
           target)) {
                retval = wait_consider_task(wo, ptrace,
                   target);
                if (retval)
                        return retval;
        }
        ptrace = true;
        target = pid_task(wo->wo_pid, PIDTYPE_PID);
        if (target && target->ptrace &&
            is_effectively_child(wo, ptrace, target)) {
                retval = wait_consider_task(wo, ptrace,
                   target);
                if (retval)
                        return retval;
        }
        return 0;
}
```

4. **Syscall Hooks** : One hook is generic and the other is made for testing purposes. Make changes in arch/x86/entry/common.c

```
    void (*syscall_hook)(struct pt_regs *regs, int nr) =
       NULL;
bool (*syscall_hook_2)(struct pt_regs *regs, int nr) =
   NULL;
EXPORT_SYMBOL(syscall_hook);
EXPORT_SYMBOL(syscall_hook_2);

__visible noinstr void do_syscall_64(struct pt_regs *regs
   , int nr)
{
        add_random_kstack_offset();
        nr = syscall_enter_from_user_mode(regs, nr);

    /* change starts here */

        if(syscall_hook)
                syscall_hook(regs, nr);
        if(syscall_hook_2)
                if(syscall_hook_2(regs, nr)) return;

    /* change ends here */

        instrumentation_begin();

        if (!do_syscall_x64(regs, nr) && !do_syscall_x32(
           regs, nr) && nr != -1) {
                /* Invalid system call, but still a
                   system call. */
                regs->ax = __x64_sys_ni_syscall(regs);
        }

        instrumentation_end();
        syscall_exit_to_user_mode(regs);
}
```

5. **ThreadStop: No SIGSTOP to GroupLeader** : Do not send stop signal to the group leader. If sent, there is a pending signal to the leader, due to which it wont wait. Make changes in do_signal_stop in kernel/signal.c

```
// Register the hook

bool (*ff_group_leader)(struct task_struct* task) = NULL;
EXPORT_SYMBOL(ff_group_leader);

                // Make the change in do_signal_stop

                while_each_thread(current, t) {
                /*
                * Setting state to TASK_STOPPED for a
                   group
                * stop is always done with the siglock
                   held,
                * so this check has no races.
                */
                /* change starts here*/

                if (!task_is_stopped(t) &&
                task_set_jobctl_pending(t, signr | gstop)
                   ) {

                if(ff_group_leader){
                if(!ff_group_leader(t))
                {
                sig->group_stop_count++;
                if (likely(!(t->ptrace & PT_SEIZED)))
                      signal_wake_up(t, 0);
                else
                        ptrace_trap_notify(t);
                        }

                }
                else{
                        sig->group_stop_count++;
                        if (likely(!(t->ptrace &
                           PT_SEIZED)))
                                signal_wake_up(t, 0);
                        else
                                ptrace_trap_notify(t);
                        }
                        /* change ends here */
                }
        }
```

6. **PullBack: Do not let them exit**: Do not let the children exit; pull back them, and use them to your advantage. Make changes in schedule_tail of kernel/sched/core.c

```
    /* Hook registration */
void (*pull_back)(void) = NULL;
EXPORT_SYMBOL(pull_back);

asmlinkage __visible void schedule_tail(struct
   task_struct *prev)
        __releases(rq->lock)
{
        /*
         * New tasks start with FORK_PREEMPT_COUNT, see
            there and
         * finish_task_switch() for details.
         *
         * finish_task_switch() will drop rq->lock() and
            lower preempt_count
         * and the preempt_enable() will end up enabling
            preemption (on
         * PREEMPT_COUNT kernels).
         */

        finish_task_switch(prev);
        preempt_enable();
        if (current->set_child_tid)
                put_user(task_pid_vnr(current), current->
                   set_child_tid);
        calculate_sigpending();

        /* change starts here */
        if(pull_back) pull_back();
        /* change ends here */

}
```

7. **my_kernel_clone**:
This function will be used for cloning the threads. Just make a copy of kernel_clone and remove the wake_up_new_task line from it. copy the below function just above kernel_clone in include/linux/kernel/fork.c

```
    pid_t my_kernel_clone(struct kernel_clone_args *args)
{
        u64 clone_flags = args->flags;
        struct completion vfork;
        struct pid *pid;
        struct task_struct *p;
        int trace = 0;
        pid_t nr;

        if ((args->flags & CLONE_PIDFD) &&
            (args->flags & CLONE_PARENT_SETTID) &&
            (args->pidfd == args->parent_tid)){
                return -EINVAL;
        }

        if (!(clone_flags & CLONE_UNTRACED)) {
                if (clone_flags & CLONE_VFORK)
                        trace = PTRACE_EVENT_VFORK;
                else if (args->exit_signal != SIGCHLD)
                        trace = PTRACE_EVENT_CLONE;
                else
                        trace = PTRACE_EVENT_FORK;

                if (likely(!ptrace_event_enabled(current,
                    trace)))
                        trace = 0;
        }


        p = copy_process(NULL, trace, NUMA_NO_NODE, args)
            ;
        add_latent_entropy();

        if (IS_ERR(p)) {
                return PTR_ERR(p);
        }

        trace_sched_process_fork(current, p);

        pid = get_task_pid(p, PIDTYPE_PID);
        nr = pid_vnr(pid);

        if (clone_flags & CLONE_PARENT_SETTID)
                put_user(nr, args->parent_tid);

        if (clone_flags & CLONE_VFORK) {
                p->vfork_done = &vfork;
                init_completion(&vfork);
                get_task_struct(p);
        }

        if (IS_ENABLED(CONFIG_LRU_GEN) && !(clone_flags &
             CLONE_VM)) {
                /* lock the task to synchronize with
                   memcg migration */
                task_lock(p);
                lru_gen_add_mm(p->mm);
                task_unlock(p);
        }

        /* forking complete and child started to run,
           tell ptracer */
        if (unlikely(trace))
                ptrace_event_pid(trace, pid);

        if (clone_flags & CLONE_VFORK) {
                if (!wait_for_vfork_done(p, &vfork))
                        ptrace_event_pid(
                           PTRACE_EVENT_VFORK_DONE, pid);
        }
        put_pid(pid);
        return nr;
}

EXPORT_SYMBOL(my_kernel_clone);
```

8. **Globals: The exported symbols**: The following contains the list of symbols required to be exported.

```
    EXPORT_SYMBOL(kernel_clone); // in kernel/fork.c
    EXPORT_SYMBOL(task_clear_jobctl_pending); // in
       kernel/signal.c
    EXPORT_SYMBOL(wake_up_new_task)
```

CRIU

Checkpoint-Restore in User space, open source process Experimentation

CRIU is a project that implements checkpoint/restore functionality by freezing the state of the process and its sub-tasks.

CRIU makes use of ptrace to stop the process by attaching PTRACE_SEIZE request.

Then it injects parasite code to dump the process's memory pages into the image files to create recoverable checkpoints.

```
    /*Checkpointing */
    $ ./a.out
    $ sudo criu dump -t 4567 -D /temp/checkpoint --leave-
       running

    /*Restoring*/
    $ sudo criu restore  D  /temp/checkpoint
```

Docker currently has checkpointing as an experimental feature, and at the back, the functionality is being supported by CRIU only. We have modified the source to time the checkpointing process since we plan to use this as our benchmark.

Process Tree Clone

In this milestone, we again started by creating a new syscall numbered 549, called `ptree_clone`.

When a processes does syscall 549 it gets sent to `do_syscall64` where we catch it again in our hook.
We start by doing a dfs on the sub-tree and wait on a leaf node till that leaf node gets cloned.
`SIGSTOP` is sent to this leaf node by the manager who later waits for the threads in the process to stop.
We needed to change the hooks so that the signal which signifies that the group is stopped is sent to the manager and not the group leader.
We also needed to change the hook in `eligible_pid` so that the manager recognizes the signal as the one it was waiting for.
After the wait, the group leader is made to reschedule using `kick_process` with reschedule flag on.
We catch these group leaders when they get descheduled in a kprobe which is put before `schedule`.
After catching them as the current, we clone them as well as their sub-threads using the implementation in milestone 1 and put the newly cloned process' pid into a list.

This process now wakes up the manager which was waiting in the dfs which goes on to make the clone of the sub-tree.

While returning bottom up, the manager gets the task struct of the new clone and reparents it to the actual parent where we needed it to be.

Conclusion and Future work

While we originally planned to implement the checkpoint functionality for multi-threaded system our current progress is limited to process tree cloning of normal processes, with more time we could implement the required scenario.

We can also explore following tasks:

1. Extending current functionality to multi-threaded processes.
2. Extensive comparison with CRIU checkpointing and further optimization.
3. Extending the final functionality to Docker,sqlite, etc other applications