多进程编程(1)

麦当劳喜欢您来，喜欢您再来

扫描关注
Linux阅码场

多进程解决什么问题

- 解决生命周期问题(init, systemd)
- 解决进程程序背景的问题(exec)
- 解决进程之间的通信问题(IPC)
- 设计进程与进程之间的界限

多进程编程(1)

1.1 多进程模式fork、vfork、exec、wait

1.2 多进程模型里的subreaper

1.3 main函数进去前和出来后，做了些什么？

1.4 exit vs _exit

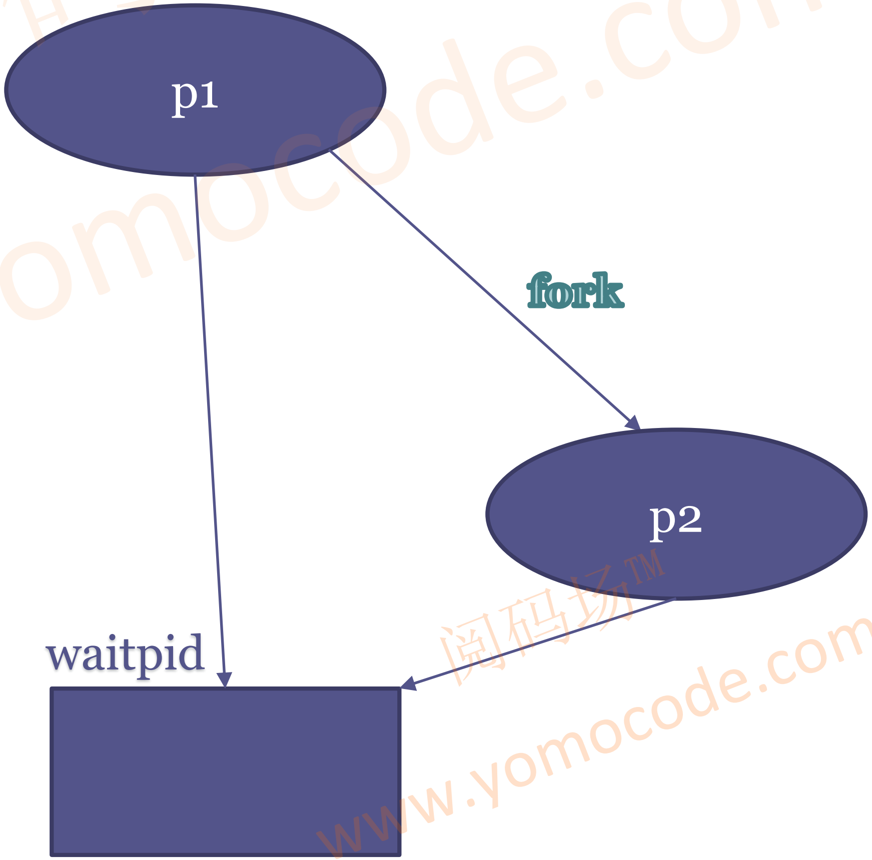
1.5 flush IO

1.6 atexit()钩子

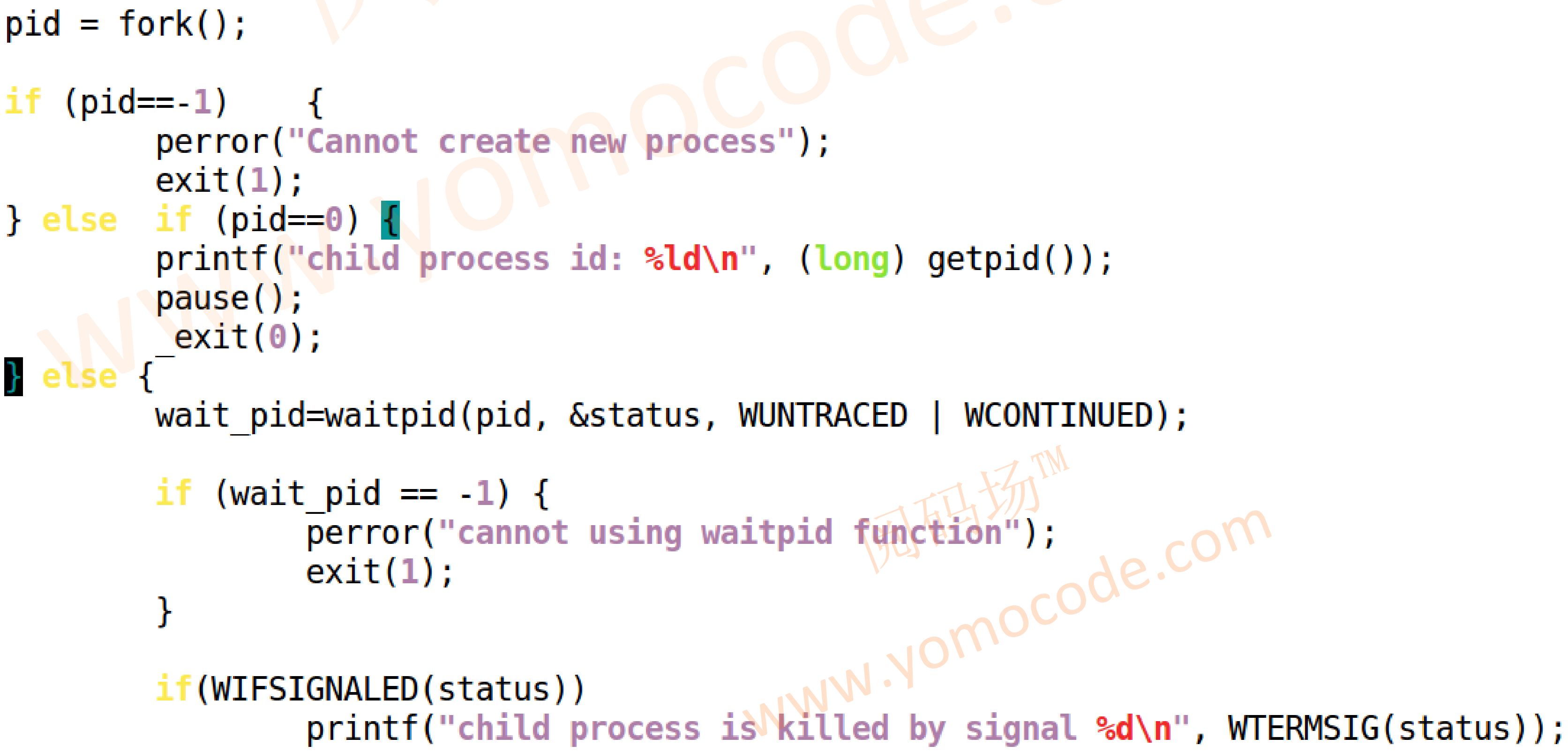
1.7 动态链接库的构造函数和析构函数

1.8 LD_PRELOAD + 构造函数完成leak sanitizer helper

进程生命周期

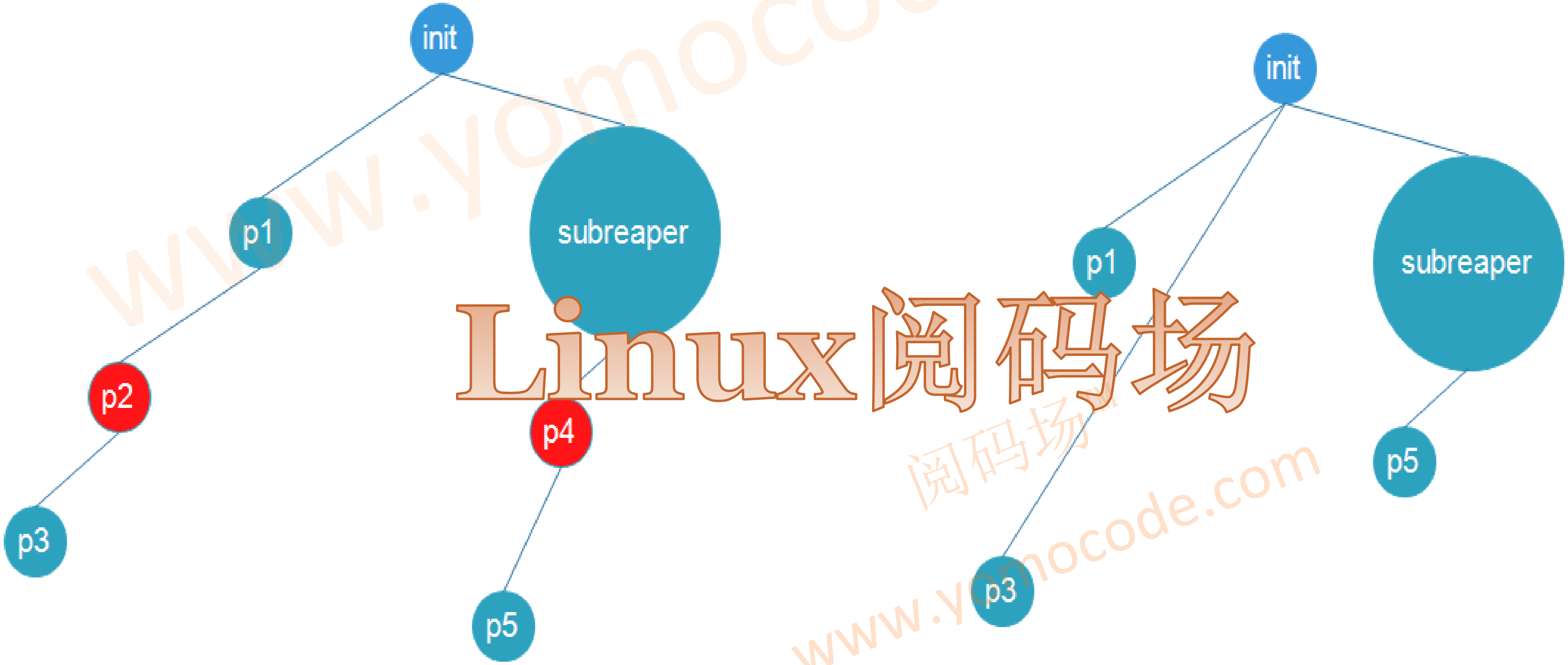


子死父清场



```
pid = fork();

if (pid==-1)    {
        perror("Cannot create new process");
        exit(1);
} else  if (pid==0) {
        printf("child process id: %ld\n", (long) getpid());
        pause();
        _exit(0);
} else {
        wait_pid=waitpid(pid, &status, WUNTRACED | WCONTINUED);

        if (wait_pid == -1) {
                perror("cannot using waitpid function");
                exit(1);
        }

        if(WIFSIGNALED(status))
                printf("child process is killed by signal %d\n", WTERMSIG(status));
```

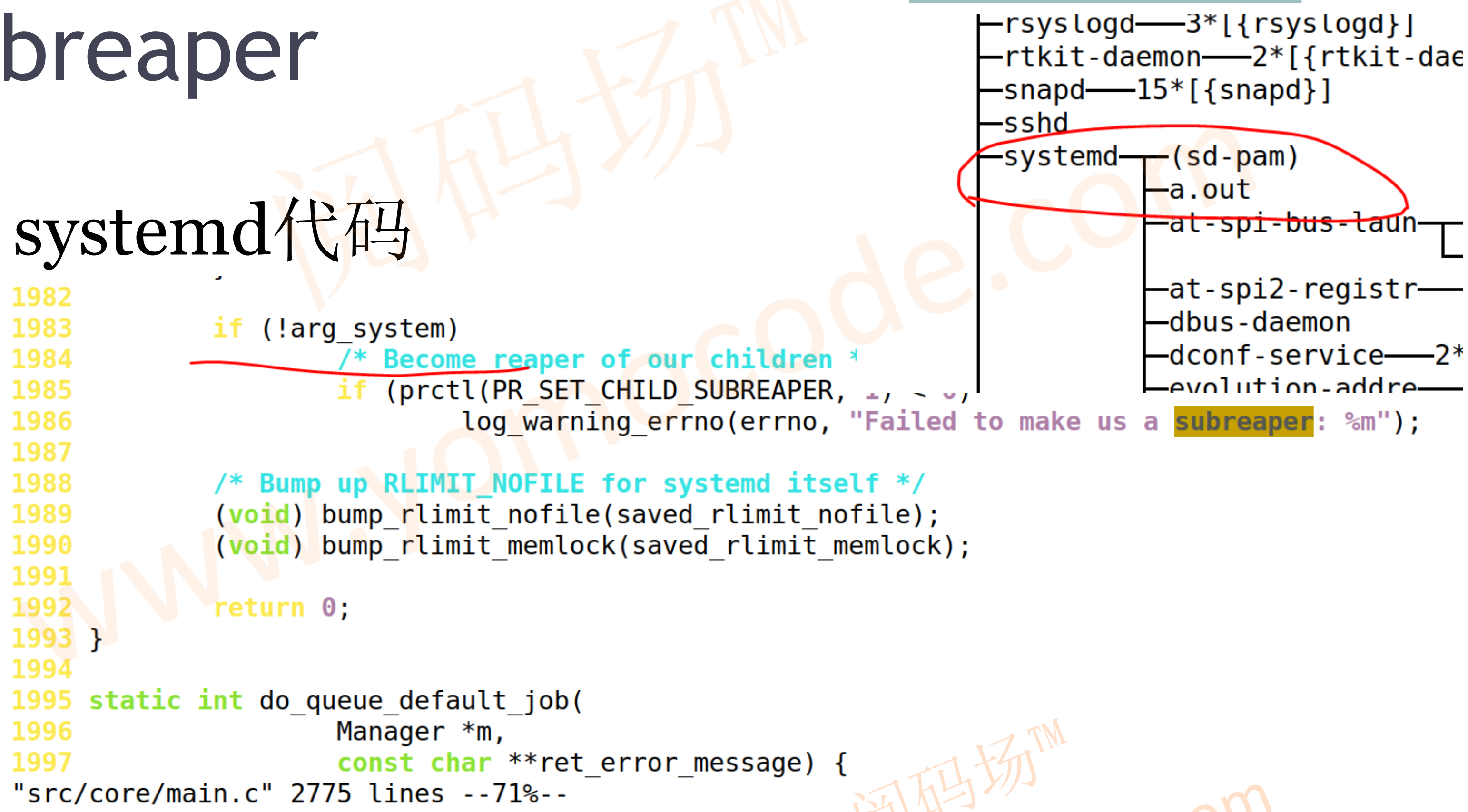
init vs. SUBREAPER



p2和p4死

subreaper

- systemd代码

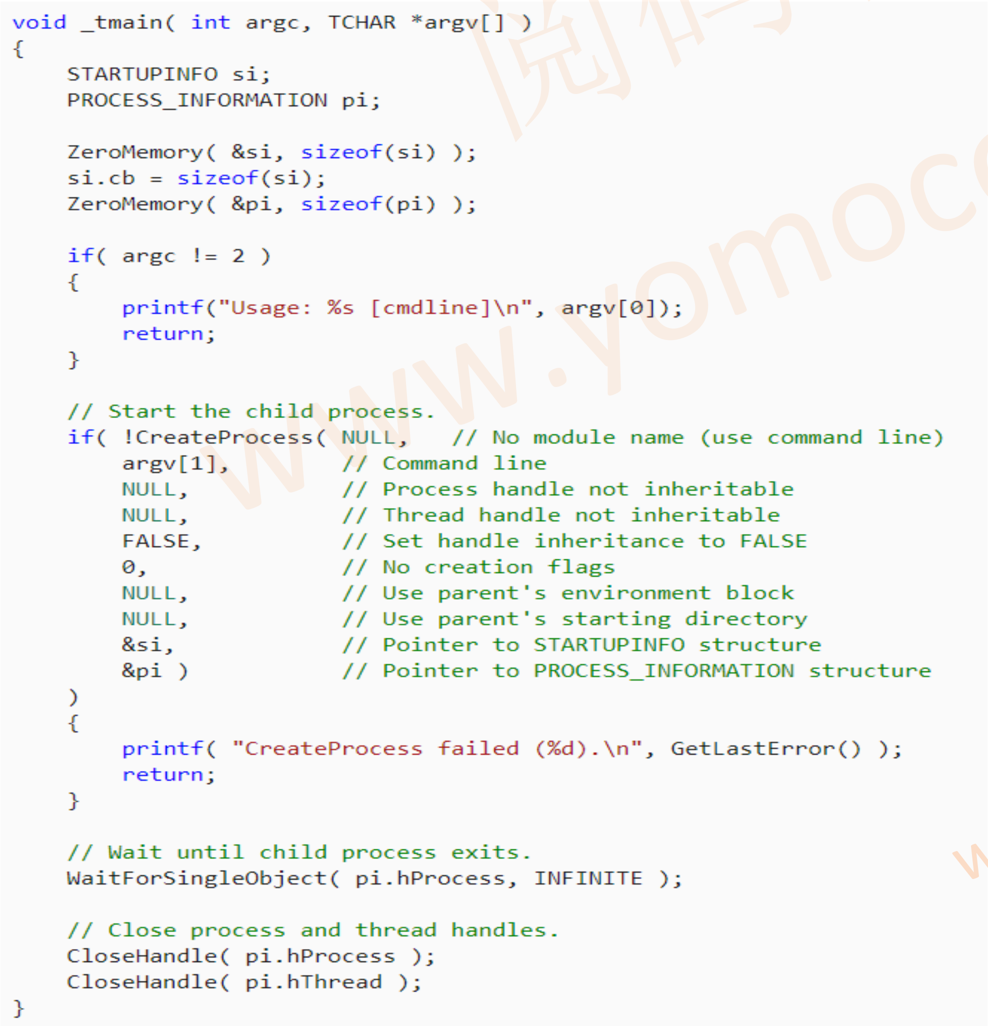


```
├─rsyslogd───3*[{rsyslogd}]
├─rtkit-daemon───2*[{rtkit-dae
├─snapd───15*[{snapd}]
├─sshd
├─systemd─┬─(sd-pam)
│         ├─a.out
│         ├─at-spi-bus-laun─┬
│         │                 └
│         ├─at-spi2-registr──
│         ├─dbus-daemon
│         ├─dconf-service───2*
│         ├─evolution-addre──
```

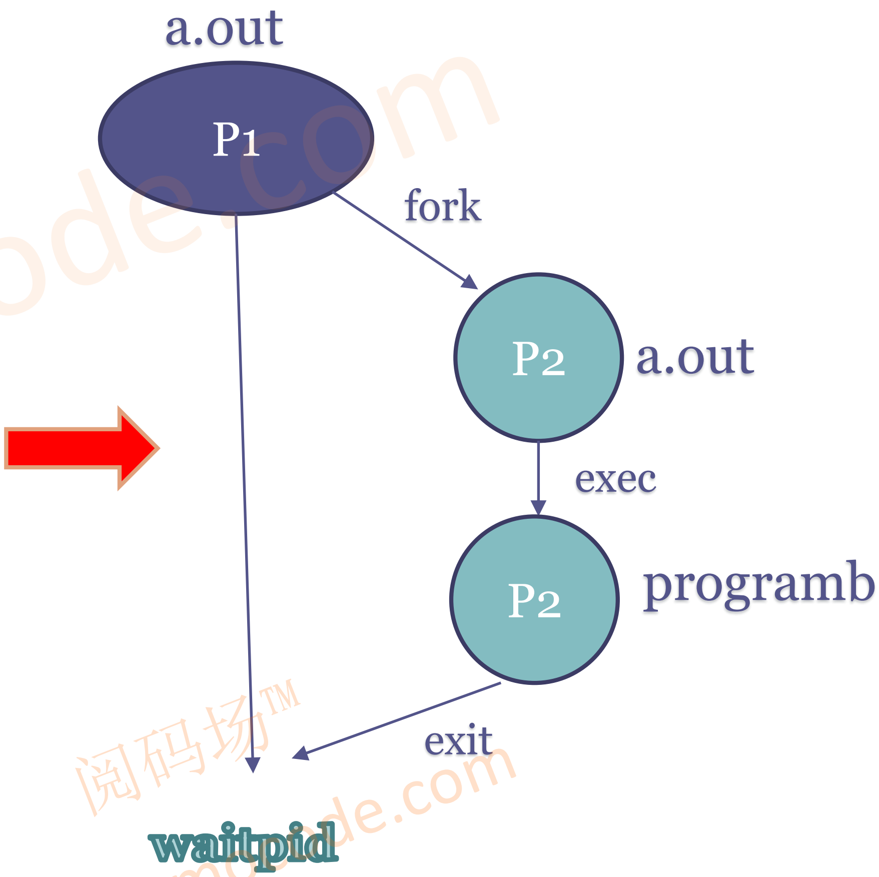
```

        if (!arg_system)
                /* Become reaper of our children *
                if (prctl(PR_SET_CHILD_SUBREAPER, 1, < 0,
                        log_warning_errno(errno, "Failed to make us a subreaper: %m");

        /* Bump up RLIMIT_NOFILE for systemd itself */
        (void) bump_rlimit_nofile(saved_rlimit_nofile);
        (void) bump_rlimit_memlock(saved_rlimit_memlock);

        return 0;
}

static int do_queue_default_job(
                Manager *m,
                const char **ret_error_message) {
"src/core/main.c" 2775 lines --71%--
```

exec

CreateProcess

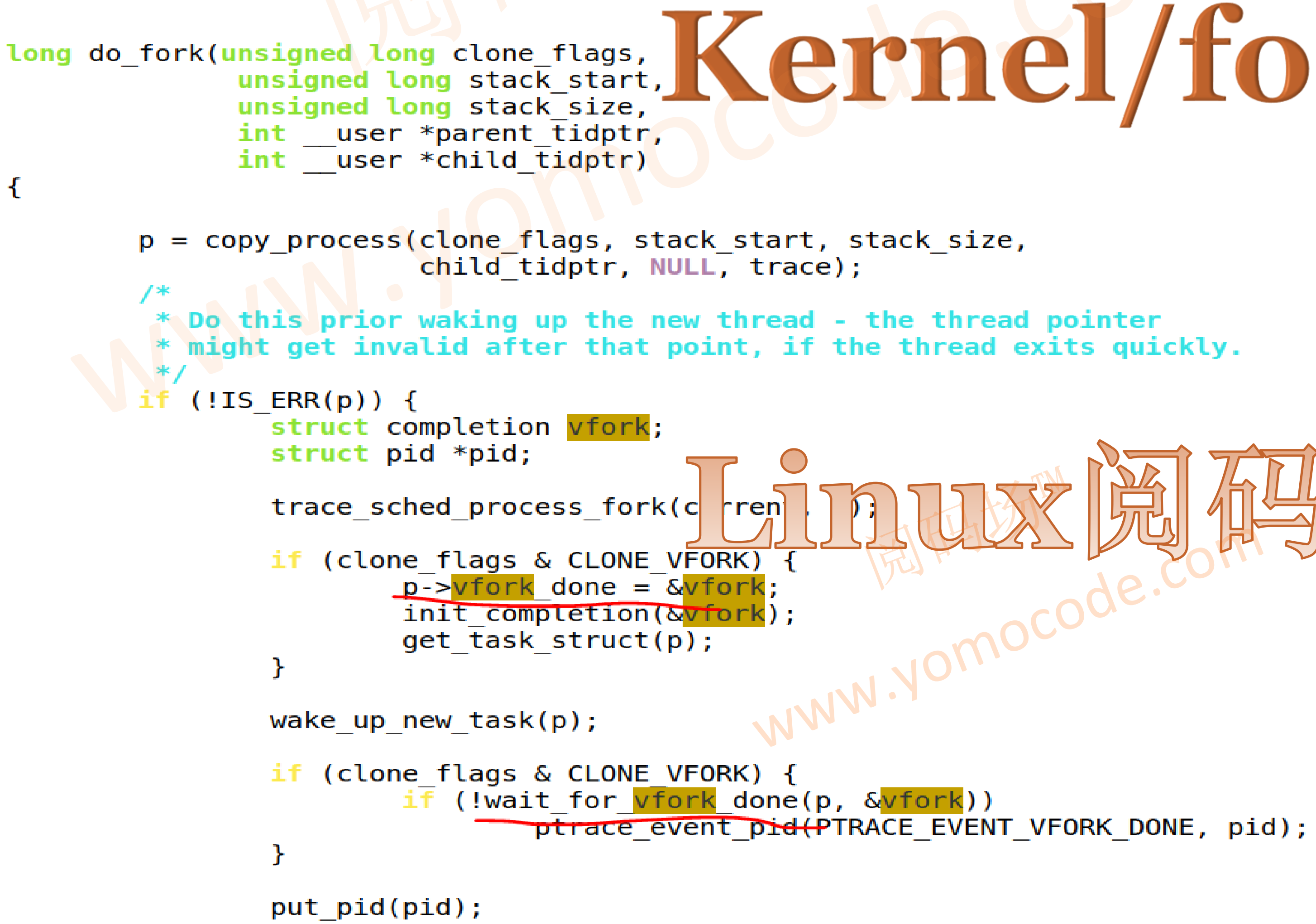


```
void _tmain( int argc, TCHAR *argv[] )
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    if( argc != 2 )
    {
        printf("Usage: %s [cmdline]\n", argv[0]);
        return;
    }

    // Start the child process.
    if( !CreateProcess( NULL,   // No module name (use command line)
        argv[1],        // Command line
        NULL,           // Process handle not inheritable
        NULL,           // Thread handle not inheritable
        FALSE,          // Set handle inheritance to FALSE
        0,              // No creation flags
        NULL,           // Use parent's environment block
        NULL,           // Use parent's starting directory
        &si,            // Pointer to STARTUPINFO structure
        &pi )           // Pointer to PROCESS_INFORMATION structure
    )
    {
        printf( "CreateProcess failed (%d).\n", GetLastError() );
        return;
    }

    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```



vfork:没有写时拷贝

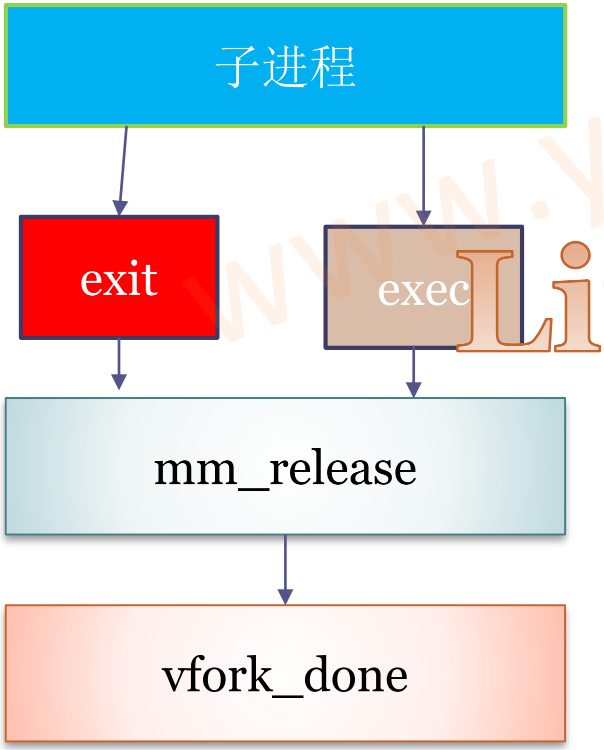
Kernel/fork.c

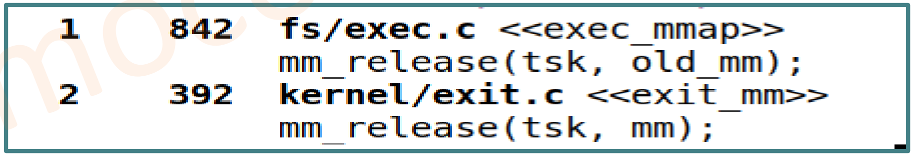


```
long do_fork(unsigned long clone_flags,
              unsigned long stack_start,
              unsigned long stack_size,
              int __user *parent_tidptr,
              int __user *child_tidptr)
{

        p = copy_process(clone_flags, stack_start, stack_size,
                         child_tidptr, NULL, trace);
        /*
         * Do this prior waking up the new thread - the thread pointer
         * might get invalid after that point, if the thread exits quickly.
         */
        if (!IS_ERR(p)) {
                struct completion vfork;
                struct pid *pid;

                trace_sched_process_fork(current, p);

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

                wake_up_new_task(p);

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

                put_pid(pid);
```

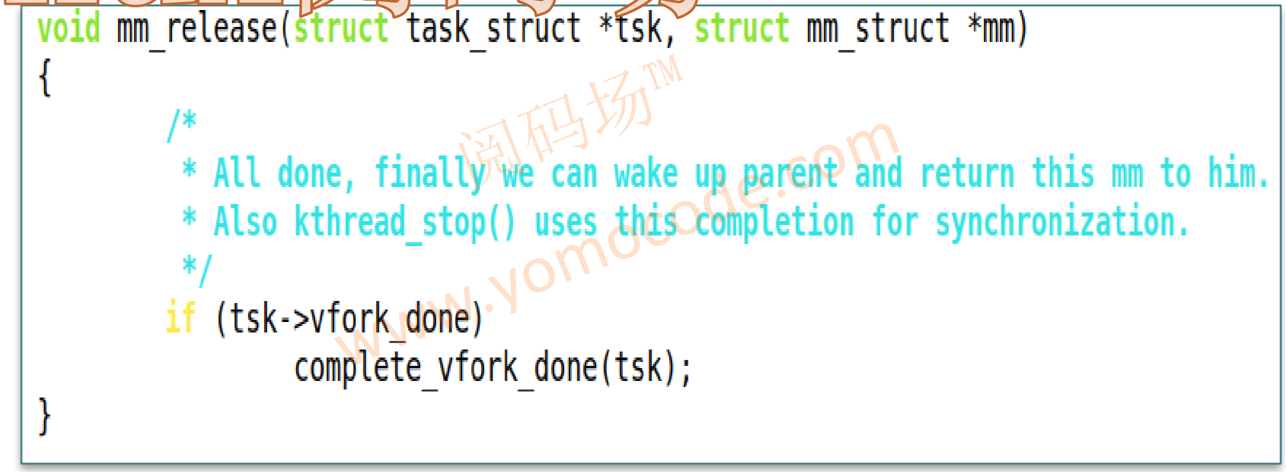
vfork: 什么时候父进程继续

- 当父进程的mm不再被子进程使用





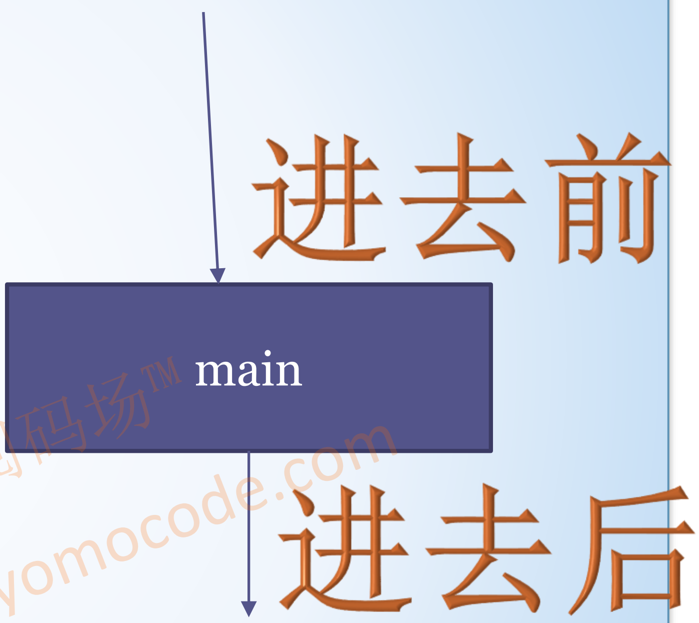
```
1    842  fs/exec.c <<exec_mmap>>
          mm_release(tsk, old_mm);
2    392  kernel/exit.c <<exit_mm>>
          mm_release(tsk, mm);
```



```
void mm_release(struct task_struct *tsk, struct mm_struct *mm)
{
        /*
         * All done, finally we can wake up parent and return this mm to him.
         * Also kthread_stop() uses this completion for synchronization.
         */
        if (tsk->vfork_done)
                complete_vfork_done(tsk);
}
```

main()函数进去之前和退出之后

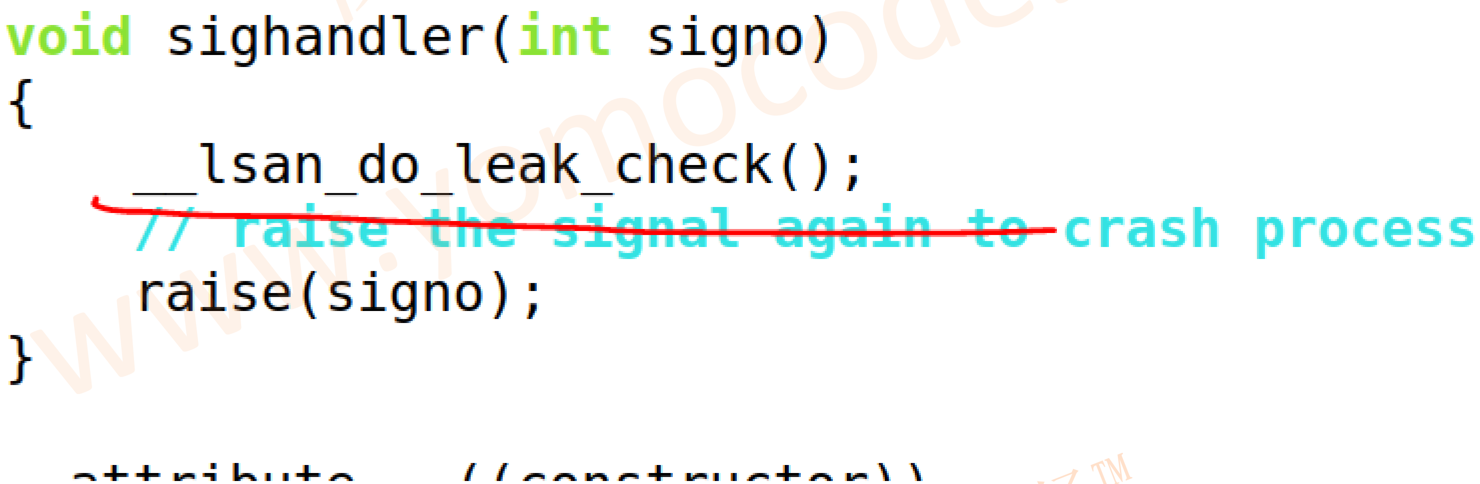
- main函数进去之前
  - ✓ 执行了动态库的构造函数
- main函数退出之后
  - ✓ 执行动态库的析构函数
  - ✓ Flush stdio
  - ✓ 执行atexit()上的函数



动态库构造和析构函数

```
__attribute__ ((constructor))
void ctor()
{
    int sigs[] = {
        SIGILL, SIGFPE, SIGABRT, SIGBUS,
        SIGSEGV, SIGHUP, SIGINT, SIGQUIT,
        SIGTERM
    };
    int i;
    struct sigaction sa;
    sa.sa_handler = sighandler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESETHAND;
    for(i = 0; i < sizeof(sigs)/sizeof(sigs[0]); ++i) {
        if (sigaction(sigs[i], &sa, NULL) == -1) {
            perror("Could not set signal handler");
        }
    }
}
```

动态库构造和析构函数:lsan-helper



```
void sighandler(int signo)
{
        __lsan_do_leak_check();
        ~~// raise the signal again to~~ crash process
        raise(signo);
}

    __attribute__((constructor))
```

谢谢！

阅码场
YOMOCODE

阅码场出品