بسمه تعالی

بررسی Process Manager **در سیستم عامل** OpenBSD

متین مرادی – 98104488

فهرست کلی موضوعات

مقدمه 2

تولد پردازه 4

الف) تابع sys_fork 4

ب) تابع fork1 5

1. **تابع** thread_new 8
2. **تابع** process_new 9

مرگ پردازه 14

مقدمه

به طورکلی روند یک پردازه یا به اصطلاح چرخه حیات یک پردازه در این سیستم عامل به صورت زیر است:

fork(2) → sys_fork() → fork1() → sys_execve() → sys_exit() → exit1()

یعنی در ابتدا باید پردازه جدید fork شود. برای این منظور سیستم کال fork(2) فراخوانی میشود و تابع sys_fork() اجرا میشود. این تابع خودش fork1() را با یک سری پارامترها صدا میزند و طی مراحلی، پردازه جدید ساخته میشود. سپس با sys_execve() پردازه اجرا میشود. Sys_exit() و exit1() هم در انتها برای اتمام پردازه استفاده میشوند.

سایت راهنمای openBSD یک قسمت manual page دارد که انواع کامندها، سیستم کال ها، توابع کتابخانه ای و ... را توضیح داده است. در اینجا چند نمونه از مواردی که در بالا ذکر شد را مشاهده میکنید:

FORK(2) System Calls Manual FORK(2)

NAME

fork — create a new process

SYNOPSIS

#include <unistd.h>

pid_t
fork(*void*);

DESCRIPTION

fork() causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process) except for the following:

- The child process has a unique process ID, which also does not match any existing process group ID.
- The child process has a different parent process ID (i.e., the process ID of the parent process).
- The child process has a single thread.
- The child process has its own copy of the parent's descriptors. These descriptors reference the same underlying objects, so that, for instance, file pointers in file objects are shared between the child and the parent, so that an lseek(2) on a descriptor in the child process can affect a subsequent read(2) or write(2) by the parent. This descriptor copying is also used by the shell to establish standard input and output for newly created processes as well as to set up pipes.
- The child process has no fcntl(2)-style file locks.
- The child process' resource utilizations are set to 0; see getrusage(2).
- All interval timers are cleared; see setitimer(2).
- The child process' semaphore undo values are set to 0; see semop(2).
- The child process' pending signals set is empty.
- The child process has no memory locks; see mlock(2) and mlockall(2).

FORK1(9) Kernel Developer's Manual FORK1(9)

NAME

fork1 — create a new process

SYNOPSIS

#include <sys/types.h>
#include <sys/proc.h>

int
fork1(*struct proc *p1, int flags, void (*func)(void *), void *arg, register_t *retval, struct proc **rnewprocp*);

DESCRIPTION

fork1() creates a new process out of *p1*, which should be the current thread. This function is used primarily to implement the fork(2) and vfork(2) system calls, as well as the kthread_create(9) function.

The *flags* argument is used to control the behavior of the fork and is created by a bitwise-OR of the following values:

FORK_FORK The call is done by the fork(2) system call. Used only for statistics.

FORK_VFORK The call is done by the vfork(2) system call. Used only for statistics.

FORK_PPWAIT Suspend the parent process until the child is terminated (by calling _exit(2) or abnormally), or makes a call to execve(2).

FORK_SHAREFILES
Let the child share the file descriptor table with the parent through **fdshare**(). The default behavior is to copy the table through **fdcopy**().

FORK_IDLE The new thread will be left in the SIDL state. The default behavior is to make it runnable and add it to the run queue.

FORK_NOZOMBIE
The child will be dissociated from the parent and will not leave a status for the parent to collect. See wait(2).

FORK_SHAREVM
The child will share the parent's address space. The default behavior is that the child gets a copy-on-write copy of the address space.

FORK_SIGHAND
The child will share the parent's signal actions, including the handler, mask, and flags, with **sigactsshare**(). The default behavior is to copy the signal actions from the parent with **sigactsinit**(). FORK_SHAREVM must also be set.

NAME

execve — execute a file

SYNOPSIS

#include <unistd.h>

int
execve(*const char *path*, *char *const argv[]*, *char *const envp[]*);

DESCRIPTION

execve() transforms the calling process into a new process. The new process is constructed from an ordinary file, whose name is pointed to by *path*, called the *new process file*. This file is either an executable object file, or a file of data for an interpreter. An executable object file consists of an identifying header, followed by pages of data representing the initial program (text) and initialized data pages. Additional pages may be specified by the header to be initialized with zero data; see elf(5).

An interpreter file begins with a line of the form:

#! *interpreter* [*arg*]

When an interpreter file is passed to **execve**() the system instead calls **execve**() with the specified *interpreter*. If the optional *arg* is specified, it becomes the first argument to the *interpreter*, and the original *path* becomes the second argument; otherwise, *path* becomes the first argument. The original arguments are shifted over to become the subsequent arguments. The zeroth argument, normally the name of the file being executed, is left unchanged.

The argument *argv* is a pointer to a null-terminated array of character pointers to NUL-terminated character strings. These strings construct the argument list to be made available to the new process. At least one non-null argument must be present in the array; by custom, the first element should be the name of the executed program (for example, the last component of *path*).

The argument *envp* is also a pointer to a null-terminated array of character pointers to NUL-terminated strings. A pointer to this array is normally stored in the global variable *environ*. These strings pass information to the new process that is not directly an argument to the command (see environ(7)).

NAME

exit — perform normal program termination

SYNOPSIS

#include <stdlib.h>

void
exit(*int status*);

DESCRIPTION

The **exit**() function terminates a process.

Before termination it performs the following functions in the order listed:

1. Call the functions registered with the atexit(3) function, in the reverse order of their registration.
2. Flush all open output streams.
3. Close all open streams.
4. Unlink all files created with the tmpfile(3) function.

Following this, **exit**() calls _exit(2). Note that typically _exit(2) only passes the lower 8 bits of *status* on to the parent, thus negative values have less meaning.

RETURN VALUES

The **exit**() function never returns.

SEE ALSO

_exit(2), atexit(3), intro(3), sysexits(3), tmpfile(3)

STANDARDS

The **exit**() function conforms to ISO/IEC 9899:1999 (“ISO C99”).

تولد پردازه

ایجاد پردازه توسط دستور fork و توسط پردازه پدر انجام میشود. برای مشاهده چگونگی انجام این دستور، ابتدا کد منبع openBSD را از این لینک دریافت و مشاهده میکنیم. سپس به فایلی با این آدرس میرویم:

src/sys/kern/kern_fork.c

در این فایل c، توابع مربوط به ایجاد و fork پردازه جدید وجود دارند که در ادامه آنها را بررسی میکنیم.

الف) تابع sys_fork

```
int
sys_fork(struct proc *p, void *v, register_t *retval)
{
        void (*func)(void *) = child_return;
        int flags;

        flags = FORK_FORK;
        if (p->p_p->ps_ptmask & PTRACE_FORK) {
                flags |= FORK_PTRACE;
                func = fork_return;
        }
        return fork1(p, flags, func, NULL, retval, NULL);
}
```

این تابع در اوایل فایل مذکور وجود دارد و وقتی کاربر دستور fork را اجرا میکند، این تابع هم فراخوانی میشود. تابع sys_fork یک متغیر flags دارد که در ابتدا برابر با FORK_FORK قرار داده میشود. FORK_FORK یک macro هست که توسط کلیدواژه definen در جایی دیگر تعریف شده است و هرجا از ماکرو با این اسم استفاده شده باشد، موقع اجرا با قطعه کدی که برای آن ماکرو تعریف کرده بودیم جایگزین خواهد شد. FORK_FORK در فایل src/sys/sys/proc.h برابر با 1 قرار داده شده است به این معنی که فراخوانی توسط سیستم کال fork(2) انجام شده است:

```
/*
 * Flags to fork1().
 */
#define FORK_FORK        0x00000001
#define FORK_VFORK       0x00000002
#define FORK_IDLE        0x00000004
#define FORK_PPWAIT      0x00000008
#define FORK_SHAREFILES  0x00000010
#define FORK_SYSTEM      0x00000020
#define FORK_NOZOMBIE    0x00000040
#define FORK_SHAREVM     0x00000080
#define FORK_PTRACE      0x00000400
```

پس متغیر flags برابر با 1 قرار داده میشود و درادامه شرایط ptrace که مربوط به دیباگ و کنترل یک پردازه توسط یک پردازه دیگر هست، چک میشود و در صورت نیاز flags به روز رسانی میشود. درنهایت تابع fork1 فراخوانی میشود.

ب) تابع fork1

```
int
fork1(struct proc *curp, int flags, void (*func)(void *), void *arg,
    register_t *retval, struct proc **rnewprocp)
{
        struct process *curpr = curp->p_p;
        struct process *pr;
        struct proc *p;
        uid_t uid = curp->p_ucred->cr_ruid;
        struct vmspace *vm;
        int count;
        vaddr_t uaddr;
        int error;
        struct  ptrace_state *newptstat = NULL;

        KASSERT((flags & ~(FORK_FORK | FORK_VFORK | FORK_PPWAIT | FORK_PTRACE
            | FORK_IDLE | FORK_SHAREVM | FORK_SHAREFILES | FORK_NOZOMBIE
            | FORK_SYSTEM)) == 0);
        KASSERT(func != NULL);

        if ((error = fork_check_maxthread(uid)))
                return error;

        if ((nprocesses >= maxprocess - 5 && uid != 0) ||
            nprocesses >= maxprocess) {
                static struct timeval lasttfm;

                if (ratecheck(&lasttfm, &fork_tfmrate))
                        tablefull("process");
                nthreads--;
                return EAGAIN;
        }
        nprocesses++;
```

در ابتدای این تابع، current process که درواقع پردازه پدر هست را در curpr قرار میدهیم. همچنین شناسه کاربر موجود را در uid قرار میدهیم. متغیرهای بعدی هم مربوط به فضای آدرس پردازه میباشد. در ادامه تعداد ریسه ها و حداکثر تعداد آنها را بررسی میکنیم و در صورت مشکل ارور میدهیم. به صورت مشابه تعداد پردازه ها را هم بررسی میکنیم و اگر شرایط مناسب بود، nprocesses که تعداد کنونی پردازه ها هست را یکی زیاد میکنیم. در اینجا nprocess و maxprocess به ترتیب تعداد پردازه های کنونی و حداکثر تعداد پردازه هایی که میتوانیم داشته باشیم است. حال چک میکنیم که کاربر تعداد پردازه ها را بیشتر از سقف تعداد پردازه ها منهای 5، نکند. درواقع 5تا برای root رزرو کرده ایم. در ادامه تابع fork_check_maxthread را مشاهده میکنید که عملکرد مشابهی با آنچه توضیح دادیم دارد:

```

int
fork_check_maxthread(uid_t uid)
{
        /*
         * Although process entries are dynamically created, we still keep
         * a global limit on the maximum number we will create. We reserve
         * the last 5 processes to root. The variable nprocesses is the
         * current number of processes, maxprocess is the limit.  Similar
         * rules for threads (struct proc): we reserve the last 5 to root;
         * the variable nthreads is the current number of procs, maxthread is
         * the limit.
         */
        if ((nthreads >= maxthread - 5 && uid != 0) || nthreads >= maxthread) {
                static struct timeval lasttfm;

                if (ratecheck(&lasttfm, &fork_tfmrate))
                        tablefull("thread");
                return EAGAIN;
        }
        nthreads++;

        return 0;
}

```

درادامه ی تابع fork1 از تابع دیگری با عنوان chgproccnt استفاده میشود. پس بهتر است ابتدا نگاهی به آن بیاندازیم. این تابع در آدرس src/sys/kern/kern_proc.c تعریف شده است:

```
/*
 * Change the count associated with number of threads
 * a given user is using.
 */
int
chgproccnt(uid_t uid, int diff)
{
        struct uidinfo *uip;
        long count;

        uip = uid_find(uid);
        count = (uip->ui_proccnt += diff);
        uid_release(uip);
        if (count < 0)
                panic("chgproccnt: procs < 0");
        return count;
}
```

این تابع برای تغییر تعداد ریسه ها برای یک کاربر مشخص است. به این صورت که uidinfo شامل اطلاعات کاربر است و با استفاده از uid_find و شناسه کاربر مورد نظرمان که uid هست، اطلاعات کاربر را پیدا کرده و در uip قرار میدهیم. سپس تعداد پردازه هایی که مربوط به این کاربر است را با uip -> ui_proccnt به دست آورده و با مقدار دلخواهی که به تابع پاس داده ایم، جمع میکنیم. پس از بررسی شرایط، این مقدار را در قالب متغیر count برمیگردانیم.

به ادامه تابع forkبرمیگردیم. عملکرد chgproccnt را دیدیم و حال برای توضیح خطوط بعدی از کامنت موجود در کد استفاده میکنیم:

Don't allow a nonprivileged user to exceed their current limit.

پس در صورتی که شرط if درست بود باید -1 را در پارامتر diff به chgproccnt پاس دهیم. همچنین تعداد پردازه ها و ریسه ها را یکی کم کنیم.

در ادامه با استفاده از تابع uvm_uarea_alloc، حافظه ای را اختصاص میدهیم برای PCB، استک و ... و در uaddr میریزیم. سپس چک میکنیم که اگر این متغیر مساوی صفر شد، همان روند قبلی را طی میکنیم. باید -1 را در پارامتر diff به chgproccnt پاس دهیم. همچنین تعداد پردازه ها و ریسه ها را یکی کم کنیم.

1. تابع thread_new

در تابع fork1 پس از این، تابع thread_new را داریم که ابتدا پردازه ی فرزند در فضای کاربر را از استخر پردازه ها با تابع pool_get میگیرد و در p قرار میدهد. سپس فرایند initialize با صفر کردن section مربوط به این پردازه انجام میشود، سپس بخش مربوطه از پدر کپی میشود، timeout ست میشود و درنهایت ریسه جدید بازگردانده میشود. در اینجا پیاده سازی تابع مذکور را مشاهده میکنید:

```

/*
 * Allocate and initialize a thread (proc) structure, given the parent thread.
 */
struct proc *
thread_new(struct proc *parent, vaddr_t uaddr)
{
        struct proc *p;

        p = pool_get(&proc_pool, PR_WAITOK);
        p->p_stat = SIDL;                       /* protect against others */
        p->p_runpri = 0;
        p->p_flag = 0;

        /*
         * Make a proc table entry for the new process.
         * Start by zeroing the section of proc that is zero-initialized,
         * then copy the section that is copied directly from the parent.
         */
        memset(&p->p_startzero, 0,
            (caddr_t)&p->p_endzero - (caddr_t)&p->p_startzero);
        memcpy(&p->p_startcopy, &parent->p_startcopy,
            (caddr_t)&p->p_endcopy - (caddr_t)&p->p_startcopy);
        crhold(p->p_ucred);
        p->p_addr = (struct user *)uaddr;

        /*
         * Initialize the timeouts.
         */
        timeout_set(&p->p_sleep_to, endtsleep, p);

        return p;
}

```

2. تابع process_new

بعد از فراخوانی تابع thread_new، در خط بعدی، تابع دیگری با نام process_new صدا زده میشود که درادامه بخشی از آن را مشاهده میکنید:

```
210  /*
211   * Allocate and initialize a new process.
212   */
213  struct process *
214  process_new(struct proc *p, struct process *parent, int flags)
215  {
216          struct process *pr;
217
218          pr = pool_get(&process_pool, PR_WAITOK);
219
220          /*
221           * Make a process structure for the new process.
222           * Start by zeroing the section of proc that is zero-initialized,
223           * then copy the section that is copied directly from the parent.
224           */
225          memset(&pr->ps_startzero, 0,
226              (caddr_t)&pr->ps_endzero - (caddr_t)&pr->ps_startzero);
227          memcpy(&pr->ps_startcopy, &parent->ps_startcopy,
228              (caddr_t)&pr->ps_endcopy - (caddr_t)&pr->ps_startcopy);
229
230          process_initialize(pr, p);
231          pr->ps_pid = allocpid();
232          lim_fork(parent, pr);
233
234          /* post-copy fixups */
235          pr->ps_pptr = parent;
236          pr->ps_ppid = parent->ps_pid;
237
238          /* bump references to the text vnode (for sysctl) */
239          pr->ps_textvp = parent->ps_textvp;
240          if (pr->ps_textvp)
241                  vref(pr->ps_textvp);
242
243          /* copy unveil if unveil is active */
244          unveil_copy(parent, pr);
245
```

اوایل این تابع، مشابه با thread_new است که توضیحات آن را دادیم. بعد از این قسمت، تابعی به نام process_initialize صدا زده میشود که پیاده سازی آن چنین است:

```
/*
 * Initialize common bits of a process structure, given the initial thread.
 */
void
process_initialize(struct process *pr, struct proc *p)
{
        /* initialize the thread links */
        pr->ps_mainproc = p;
        TAILQ_INIT(&pr->ps_threads);
        TAILQ_INSERT_TAIL(&pr->ps_threads, p, p_thr_link);
        pr->ps_refcnt = 1;
        p->p_p = pr;

        /* give the process the same creds as the initial thread */
        pr->ps_ucred = p->p_ucred;
        crhold(pr->ps_ucred);
        KASSERT(p->p_ucred->cr_ref >= 2);       /* new thread and new process */

        LIST_INIT(&pr->ps_children);
        LIST_INIT(&pr->ps_orphans);
        LIST_INIT(&pr->ps_ftlist);
        LIST_INIT(&pr->ps_sigiolst);
        TAILQ_INIT(&pr->ps_tslpqueue);

        rw_init(&pr->ps_lock, "pslock");
        mtx_init(&pr->ps_mtx, IPL_MPFLOOR);

        timeout_set_kclock(&pr->ps_realit_to, realitexpire, pr,
            KCLOCK_UPTIME, 0);
        timeout_set(&pr->ps_rucheck_to, rucheck, pr);
}
```

به این تابع، دو پارامتر پاس داده میشود. یکی pr و دومی p که به ترتیب پردازه فرزند و پردازه پدر هستند. هدف این تابع، initialize کردن پردازه مان است به این صورت که در ابتدا p را در pr->ps_mainproc ریخته و بعد یک صف تشکیل میدهد که head آن را با استفاده از TAILQ_INIT برابر با pr->ps_threads قرار میدهد و در tail این صف با استفاده از TAILQ_INSERT_TAIL، p را قرار میدهد و درادامه فرایند initializing انجام میشود.

حال به ادامه تابع process_new بازمیگردیم جایی که باید یک pid به پردازه جدید اختصاص دهیم که با استفاده از تابع allocpid انجام میشود که پیاده سازی آن را مشاهده میکنید:

```
/* Find an unused pid */
pid_t
allocpid(void)
{
        static pid_t lastpid;
        pid_t pid;

        if (!randompid) {
                /* only used early on for system processes */
                pid = ++lastpid;
        } else {
                /* Find an unused pid satisfying lastpid < pid <= PID_MAX */
                do {
                        pid = arc4random_uniform(PID_MAX - lastpid) + 1 +
                            lastpid;
                } while (ispidtaken(pid));
        }

        return pid;
}
```

در واقع کار اصلی که انجام میشود این است که در یک حلقه مدام با استفاده از arc4random_uniform اعداد تصادفی تولید میشوند و چک میشود که آیا قبلا این عدد به عنوان شناسه به پردازه ای اختصاص داده شده است یا نه و اگر اشغال نشده بود، به عنوان شناسه پردازه جدید قرار میگیرد. این چک کردن توسط تابع ispidtaken انجام میشود که بررسی میکند که عدد کاندیدی که به آن پاس داده میشود، نه به processای نسبت داده شده باشد، نه به یک process group و نه به یک zombie process که به ترتیب توسط توابع prfind، pgfind و zombiefind چک میشوند.

```
/*
 * Checks for current use of a pid, either as a pid or pgid.
 */
pid_t oldpids[128];
int
ispidtaken(pid_t pid)
{
        uint32_t i;

        for (i = 0; i < nitems(oldpids); i++)
                if (pid == oldpids[i])
                        return (1);

        if (prfind(pid) != NULL)
                return (1);
        if (pgfind(pid) != NULL)
                return (1);
        if (zombiefind(pid) != NULL)
                return (1);
        return (0);
}
```

در ادامه ی تابع process_new، تنظیمات دیگری انجام میشوند از جمله ست کردن اشاره گری در پردازه فرزند برای اشاره به پردازه پدر، اضافه کردن پردازه جدید به لیست allprocess و ... و در نهایت پردازه جدید (pr) از تابع return میشود.

به ادامه ی تابع fork1 بازمیگردیم یعنی بعد از اینکه thread_new و process_new را فراخوانی کرده بودیم. کارهایی انجام میشود مثل اینکه یک سری متغیرهای پدر مانند pr->ps_fd و pr->ps_vmspace در پردازه فرزند ریخته میشود و بعد از این اقدامات، تابع cpu_fork صدا زده میشود که PCB را کپی و آپدیت کرده و پردازه فرزند را برای اجرا آماده میکند. سپس بسته به اینکه فراخوانی توسط fork انجام شده یا vfork، متغیرهای شمارشی که برای کنترل تعداد fork ها و vfork ها طراحی شده اند، به روز رسانی میشوند.

FORK(2) System Calls Manual

NAME

fork — create a new process

SYNOPSIS

#include <unistd.h>

pid_t
fork(*void*);

VFORK(2) System Calls Manual

NAME

vfork — spawn new process and block parent

SYNOPSIS

#include <unistd.h>

pid_t
vfork(*void*);

سپس با استفاده از تابع ()alloctid که مانند همان ()allocpid هست که توضیح دادیم، یک شناسه به ریسه مان اختصاص میدهیم. در خطوط بعدی هم پردازه پدرو فرزند را به یک سری لیست مانند curpr و ... اضافه میکنیم. درادامه اقدامات نهایی انجام میشوند از جمله

- ست کردن وضعیت ptrace
- چک کردن اینکه اگر پردازه فرزند در حالت IDLE بود با استفاده از تابع fork_thread_start به صف اجرا اضافه شود
- متغیرهای کنترلی و شمارشی را پس از اطمینان از موفقیت آمیز بودن fork آپدیت میکنیم
- اگر سیستم کال ما vfork بود طبق آنچه در manual page بود و اسکرین شات آن را در بالا دیدیم، باید استفاده کنیم از tsleep برای اینکه پدر تا اتمام کار فرزند منتظر بماند.
- برگرداندن pid فرزند به پردازه پدر

مرگ پردازه

حال که عملیات ایجاد پردازه را به صورت مفصل توضیح دادیم، به سراغ terminate کردن پردازه ها میرویم و توضیحی اجمالی در آن باب میدهیم.

پس در ابتدا به فایل در آدرس src/sys/kern/kern_exit.c میرویم. خارج شدن از پردازه توسط تابع sys_exit انجام میشود.

```
/*
 * exit --
 *      Death of process.
 */
int
sys_exit(struct proc *p, void *v, register_t *retval)
{
        struct sys_exit_args /* {
                syscallarg(int) rval;
        } */ *uap = v;

        exit1(p, SCARG(uap, rval), 0, EXIT_NORMAL);
        /* NOTREACHED */
        return (0);
}

```

درون این تابع، یک تابع دیگر به نام exit1 صدا زده میشود. همانطور که میبینید، فرایند ها بسیار شبیه به آنچیزی بود که در fork داشتیم.

```c
/*
 * Exit: deallocate address space and other resources, change proc state
 * to zombie, and unlink proc from allproc and parent's lists.  Save exit
 * status and rusage for wait().  Check for child processes and orphan them.
 */
void
exit1(struct proc *p, int xexit, int xsig, int flags)
{
        struct process *pr, *qr, *nqr;
        struct rusage *rup;
        int s;

        atomic_setbits_int(&p->p_flag, P_WEXIT);

        pr = p->p_p;

        /* single-threaded? */
        if (!P_HASSIBLING(p)) {
                flags = EXIT_NORMAL;
        } else {
                /* nope, multi-threaded */
                if (flags == EXIT_NORMAL)
                        single_thread_set(p, SINGLE_EXIT, 1);
                else if (flags == EXIT_THREAD)
                        single_thread_check(p, 0);
        }

        if (flags == EXIT_NORMAL && !(pr->ps_flags & PS_EXITING)) {
                if (pr->ps_pid == 1)
                        panic("init died (signal %d, exit %d)", xsig, xexit);

                atomic_setbits_int(&pr->ps_flags, PS_EXITING);
                pr->ps_xexit = xexit;
                pr->ps_xsig  = xsig;

```

در این تابع اقداماتی انجام میشود از جمله آزاد کردن فضای آدرسی که اشغال شده بود، آزاد کردن بقیه منابع در اختیار پردازه موردنظر، تغییر دادن وضعیت پردازه به زامبی، آپدیت کردن یک سری لیست ها مانند allproc، مدیریت فرزندان پردازه ای که دارد میمیرد و ...

و در انتها برای schedule کردن منابع آزاد شده مانند vmspace و stack پردازه مورد نظر تابع exit2 صدا زده میشود:

```
388  /*
389   * We are called from cpu_exit() once it is safe to schedule the
390   * dead process's resources to be freed.
391   *
392   * NOTE: One must be careful with locking in this routine.  It's
393   * called from a critical section in machine-dependent code, so
394   * we should refrain from changing any interrupt state.
395   *
396   * We lock the deadproc list, place the proc on that list (using
397   * the p_hash member), and wake up the reaper.
398   */
399  void
400  exit2(struct proc *p)
401  {
402          mtx_enter(&deadproc_mutex);
403          LIST_INSERT_HEAD(&deadproc, p, p_hash);
404          mtx_leave(&deadproc_mutex);
405
406          wakeup(&deadproc);
407  }
```

و در آخر با صدا زدن cpu_exit اجرای پردازه کاملا پایان میابد:

```
362
363          /*
364           * Finally, call machine-dependent code to switch to a new
365           * context (possibly the idle context).  Once we are no longer
366           * using the dead process's vmspace and stack, exit2() will be
367           * called to schedule those resources to be released by the
368           * reaper thread.
369           *
370           * Note that cpu_exit() will end with a call equivalent to
371           * cpu_switch(), finishing our execution (pun intended).
372           */
373          uvmexp.swtch++;
374          cpu_exit(p);
375          panic("cpu_exit returned");
376  }
```