Process Management

Dongyoon Lee

Summary of last lectures

- Getting, building, and exploring the Linux kernel
- System call: interface between applications and kernel
- Kernel data structures
- Kernel modules
- Kernel debugging techniques

Today's agenda

- Process management in Linux kernel
  - Process
  - The process descriptor: `task_struct`
  - Process creation
  - Threads
  - Process termination

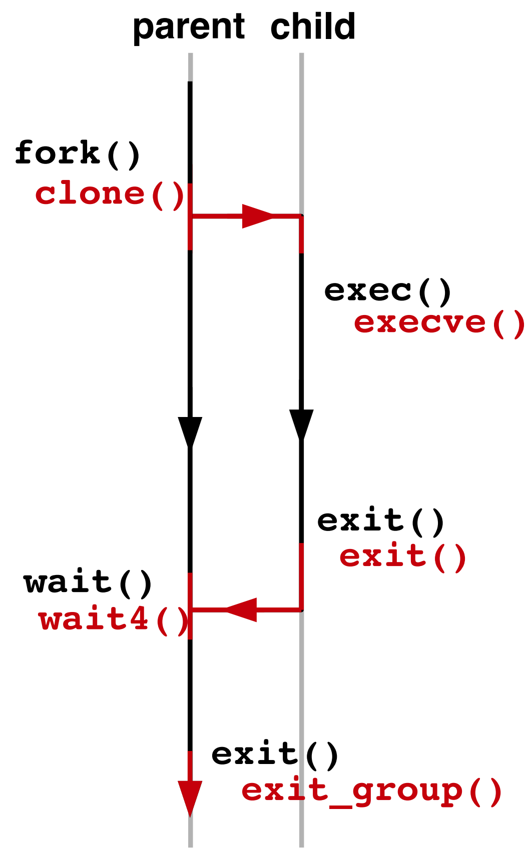
Process

- A program currently executing in the system
- A process is composed of
  - CPU registers
  - program code (i.e., *text section*)
  - state of memory segments (data, stack, etc)
  - kernel resources (open files, pending signals, etc)
  - threads
- Virtualization of processor and memory

Process from an user-space view

- `pid_t fork(void)`
  - creates a new process by duplicating the calling process
- `int execv(const char *path, const char *arg, ...)`
  - replaces the current process image with a new process image
- `pid_t wait(int *wstatus)`
  - wait for state changes in a child of the calling process
  - the child terminated; the child was stopped or resumed by a signal

Process from an user-space view



`fork()` example

```
int main(void)
{
    pid_t pid;
    int wstatus, ret;
    pid = fork(); /* create a child process */
    switch(pid) {
        case -1:  /* fork error */
            perror("fork");
            return EXIT_FAILURE;
        case 0:   /* pid = 0: new born child process */
            sleep(1);
            printf("Noooooooo!\n");
            exit(99);
        default:  /* pid = pid of child: parent process */
            printf("I am your father!: your pid is %d\n", pid);
            break;
    }
    /* A parent wait until the child terminates */
    ret = waitpid(pid, &wstatus, 0);
    if(ret == -1)
        return EXIT_FAILURE;
    printf("Child exit status: %d\n", WEXITSTATUS(wstatus));
    return EXIT_SUCCESS;
}
```

Let’s check this example using `strace`

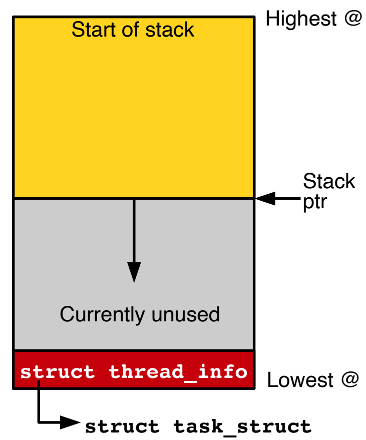
```
$ strace -f ./process
execve("./process", ["./process"], 0x7ffffb44f068 /* 64 vars */) = 0
...
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x
strace: Process 16888 attached
[pid 16887] fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 3), ...}) = 0
[pid 16888] nanosleep({tv_sec=1, tv_nsec=0},  <unfinished ...>
[pid 16887] brk(NULL)                         = 0x164e000
[pid 16887] brk(0x166f000)                    = 0x166f000
[pid 16887] brk(NULL)                         = 0x166f000
[pid 16887] write(1, "I am your father!\n", 18I am your father!) = 18
[pid 16887] wait4(16888,  <unfinished ...>
[pid 16888] <... nanosleep resumed> 0x7ffefe1b4500) = 0
[pid 16888] fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 3), ...}) = 0
[pid 16888] brk(NULL)                         = 0x164e000
[pid 16888] brk(0x166f000)                    = 0x166f000
[pid 16888] brk(NULL)                         = 0x166f000
[pid 16888] write(1, "Noooooooo!\n", 11Noooooooo!) = 11
[pid 16888] exit_group(0)                     = ?
[pid 16888] +++ exited with 0 +++
<... wait4 resumed> [{WIFEXITED(s) && WEXITSTATUS(s) == 0}], 0, NULL) = 16888
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=16888, si_uid=1000, si_status=0, si_
write(1, "Child exit status: 0\n", 21Child exit status: 0)  = 21
```

Processor descriptor: `task_struct`

```
/* linux/include/linux/sched.h */
struct task_struct {
    struct thread_info   thread_info;  /* thread information */
    volatile long        state;        /* task status: TASK_RUNNING, etc */
    void                 *stack;       /* stack of this task */

    int                  prio;         /* task priority */
    struct sched_entity  se;           /* information for processor scheduler */
    cpumask_t            cpus_allowed; /* bitmask of CPUs allowed to execute */

    struct list_head     tasks;        /* a global task list */
    struct mm_struct     *mm;          /* memory mapping of this task */

    struct task_struct   *parent;      /* parent task */
    struct list_head     children;     /* a list of child tasks */
    struct list_head     sibling;      /* siblings of the same parent */

    struct files_struct  *files;       /* open file information */
    struct signal_struct *signal;      /* signal handlers */
    /* ... */

    /* NOTE: In Linux kernel, process and task are interchangably used. */
};
```

Processor descriptor: `task_struct`

- In old kernels, `task_struct` (or `thread_info` until v4.9) is allocated at the bottom of the kernel stack of each process
    - Getting current `task_struct` is just masking out the 13 least-significant bits the stack pointer



Processor descriptor: `task_struct`

- Since v4.9, `task_struct` is dynamically allocated at heap because of potential exploit when overflowing the kernel stack
- For efficient access of current `task_struct`, kernel maintains per-CPU variable, named `current_task`
  - Use `current` to get `current_task`

```
/* linux/arch/x86/include/asm/current.h */

DECLARE_PER_CPU(struct task_struct *, current_task);
static __always_inline struct task_struct *get_current(void)
{
    return this_cpu_read_stable(current_task);
}

#define current get_current()
```

Process Identifier (PID): `pid_t`

- Maximum is 32768 (`int`)
- Can be increased to 4 millions
- Wraps around when maximum reached

Process status: `task->state`

- `TASK_RUNNING`
  - A task is runnable (running or in a per-CPU scheduler run queue)
  - A task could be in user- or kernel-space

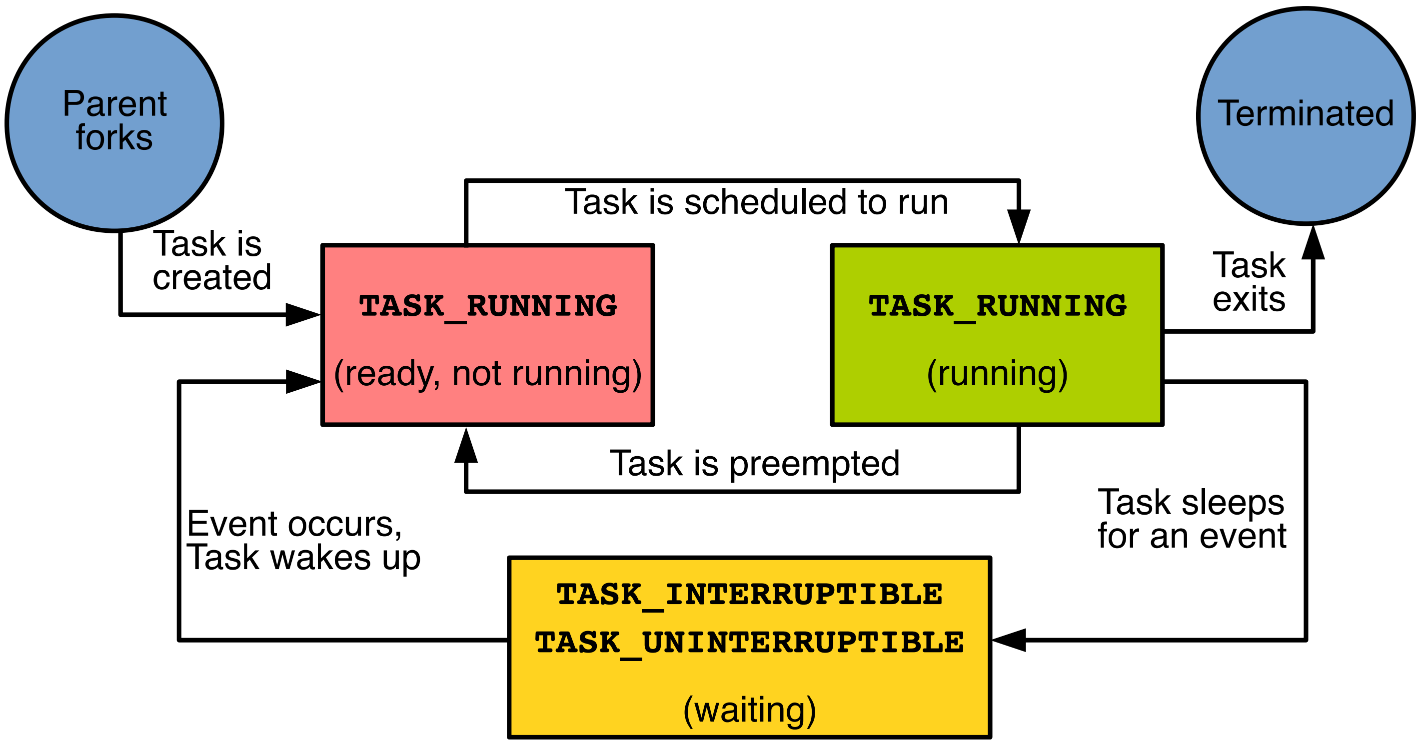
Process status: `task->state`

- `TASK_INTERRUPTIBLE`
  - Process is sleeping waiting for some condition
  - Switched to `TASK_RUNNING` when the waiting condition becomes true or a signal is received
- `TASK_UNINTERRUPTIBLE`
  - Same as `TASK_INTERRUPTIBLE` but does not wake up on signal

Process status: `task->state`

- `__TASK_TRACED`
  - Traced by another process (ex: debugger)
- `__TASK_STOPPED`
  - Not running nor waiting, result of the reception of some signals (e.g., `SIGSTOP`) to pause the process

Process status: `task->state`



Producer-consumer example

- Producer
  - generate an event and wake up a consumer
- Consumer
  - check if there is an event
  - if so, process all pending event in the list
  - otherwise, sleep until the producer wakes me up

Sleeping in the kernel

```
Producer task:
spin_lock(&list_lock);
list_add_tail(&list_head, new_event);  /* append an event to the list */
spin_unlock(&list_lock);
wake_up_process(consumer_task);        /* and wake up the consumer task */

Consumer task:
set_current_state(TASK_INTERRUPTIBLE); /* set status to TASK_INTERUPTIBLE */
spin_lock(&list_lock);
if(list_empty(&list_head)) {           /* if there is no item in the list */
       spin_unlock(&list_lock);
       schedule();            /* sleep until the producer task wakes this */
       spin_lock(&list_lock); /* this task is waken up by the producer */
}
set_current_state(TASK_RUNNING); /* change status to TASK_RUNNING */

list_for_each(pos, list_head) {
       list_del(&pos)
       /* process an item */
       /* ... */
}
spin_unlock(&list_lock);
```

Process context

- The kernel can execute in a **process context** or **interrupt context**
    - `current` is meaningful only when the kernel executes in a process context such as executing a system call
    - Interrupt has its own context

Process family tree

- `init` process is the root of all processes
  - Launched by the kernel as the last step of the boot process
  - Reads the system `initscripts` and executes more programs, such as daemons, eventually completing the boot process
  - Its `PID` is 1
  - Its `task_sturct` is a global variable, named `init_task` (linux/init/init_task.c)

Let’s check process tree using `pstree`

```
21:15 $ pstree
init─┬─apache2───2*[apache2───26*[{apache2}</mark>
     ├─collectl
     ├─cron
     ├─dbus-daemon
     ├─6*[getty]
     ├─irqbalance
     ├─lxcfs───6*[{lxcfs}]
     ├─mdadm
     ├─memcached───5*[{memcached}]
     ├─mosh-server───bash───tmux: client
     ├─mpssd───10*[{mpssd}]
     ├─netserver
     ├─nullmailer-send───smtp
     ├─rpc.idmapd
     ├─rpc.mountd
     ├─rpc.statd
     ├─rpcbind
     ├─rsyslogd───3*[{rsyslogd}]
     ├─sshd───sshd───sshd───bash───pstree
     ├─systemd-logind
     ├─systemd-udevd
     ├─tmux: server─┬─bash───vim───bash
```

Process family tree

- `fork`-based process creation
    - my parent task: `current->parent`
    - my children tasks: `current->children`
    - siblings under the parent: `current->siblings`
    - list of all tasks in the system: `current->tasks`
    - macros to explore:
        - `next_task(t)`, `for_each_process(t)`
- Let’s check how these macros are implemented

Process creation

- Linux does not implements creating a tasks from nothing (`spawn` or `CreateProcess`)
- `fork()` and `exec()`
  - `fork()` creates a child, copy of the parent process
    - Only PID, PPID and some resources/stats differ
  - `exec()` loads into a process address space a new executable

Copy-on-Write (CoW)

- On `fork()`, Linux duplicates the parent page tables and creates a new process descriptor
    - Change page table access bits to *read-only*
    - When a page is accessed *for write operations*, that page is copied and the corresponding page table entry is changed to *read-write*
- `fork()` is fast by delaying or altogether preventing copying of data
- `fork()` saves memory by sharing read-only pages among descendants

Forking

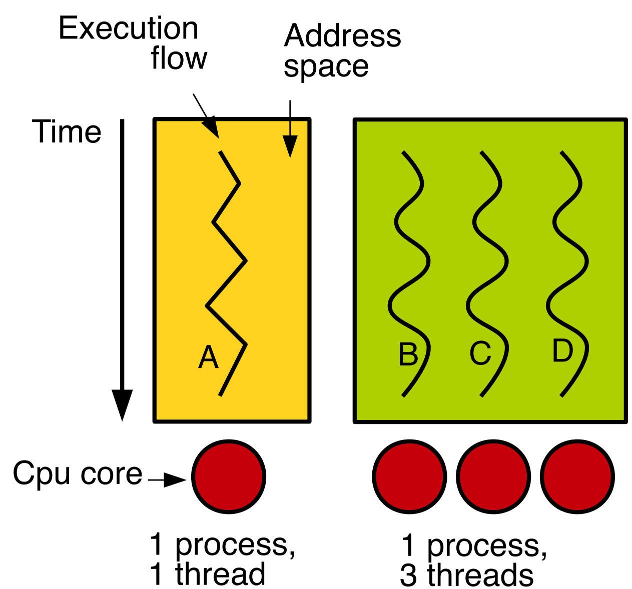
- `fork()` is implemented by the `clone()` system call
- `sys_clone()` calls `_do_fork()`, which calls `copy_process()` and starts the new task
- `copy_process()`
  - `dup_task_struct()`, which duplicates kernel stack, `task_struct`, and `thread_info`
  - Checks that we do not overflow the processes number limit
  - Various members of the `task_struct` are cleared

Forking (cont'd)

- `copy_process()`
  - Calls `sched_fork()` to set the child `state` set to `TASK_NEW`
  - Copies parent information such as files, signal handlers, etc.
  - Gets a new PID using `alloc_pid()`
  - Returns a pointer to the new child `task_struct`
- Finally, `_do_fork()` calls `wake_up_new_task()`
  - The new child task becomes `TASK_RUNNING`

Thread

- Threads are concurrent flows of execution belonging to the same program *sharing the same address space*



Thread

- There is no concept of a thread in Linux kernel
  - No scheduling for threads
- Linux implements all threads as standard processes
  - A thread is just another process sharing some information with other processes so each thread has its own `task_struct`
  - Created through `clone()` system call with specific flags indicating sharing
  - `clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);`

Kernel thread

- Used to perform background operations in the kernel
- Very similar to user space threads
  - They are schedulable entities (like regular processes)
- However they do not have their own address space
  - `mm` in `task_struct` is `NULL`
- Kernel threads are all forked from the `kthreadd` kernel thread (PID 2)
- Use cases (`ps --ppid 2`)
  - Work queue (`kworker`)
  - Load balancing among CPUs (`migration`)

Kernel thread

- To create a kernel thread, use `kthread_create()`
- When created through `kthread_create()`, the thread is not in a runnable state
- Need to call `wake_up_process()` or use `kthread_run()`
- Other threads can ask a kernel thread to stop using `kthread_stop()`
    - A kernel thread should check `kthread_should_stop()` to decide to continue or stop

Kernel thread

```
/**
 * kthread_create - create a kthread on the current node
 * @threadfn: the function to run in the thread
 * @data: data pointer for @threadfn()
 * @namefmt: printf-style format string for the thread name
 * @...: arguments for @namefmt.
 *
 * This macro will create a kthread on the current node, leaving it in
 * the stopped state.
 */
#define kthread_create(threadfn, data, namefmt, arg...) ...

/**
 * wake_up_process - Wake up a specific process
 * @p: The process to be woken up.
 *
 * Attempt to wake up the nominated process and move it to the set of runnable
 * processes.
 *
 * Return: 1 if the process was woken up, 0 if it was already running.
 */
int wake_up_process(struct task_struct *p);
```

Kernel thread

```
/**
 * kthread_run - create and wake a thread.
 * @threadfn: the function to run until signal_pending(current).
 * @data: data ptr for @threadfn.
 * @namefmt: printf-style name for the thread.
 *
 * Description: Convenient wrapper for kthread_create() followed by
 * wake_up_process().  Returns the kthread or ERR_PTR(-ENOMEM).
 */
#define kthread_run(threadfn, data, namefmt, ...) ...

/**
 * kthread_stop - stop a thread created by kthread_create().
 * @k: thread created by kthread_create().
 *
 * Sets kthread_should_stop() for @k to return true, wakes it, and
 * waits for it to exit. If threadfn() may call do_exit() itself,
 * the caller must ensure task_struct can't go away.
 */
int kthread_stop(struct task_struct *k);
```

Kernel thread example

- Ext4 file system uses a kernel thread to finish file system initialization in the background

```
/* linux/fs/ext4/super.c */
static int ext4_run_lazyinit_thread(void)
{
    ext4_lazyinit_task = kthread_run(ext4_lazyinit_thread,
                         ext4_li_info, "ext4lazyinit");
    /* ... */
}

static int ext4_lazyinit_thread(void *arg)
{
    while (true) {
        if (kthread_should_stop()) {
            goto exit_thread;
        }
        /* ... */
    }
}
```

Kernel thread example

```
static void ext4_destroy_lazyinit_thread(void)
{
    /* ... */
    kthread_stop(ext4_lazyinit_task);
}

static void __exit ext4_exit_fs(void)
{
    ext4_destroy_lazyinit_thread();
    /* ... */
}

module_exit(ext4_exit_fs)
```

Process termination

- Termination on invoking the `exit()` system call
  - Can be implicitly inserted by the compiler on `return` from `main()`
  - `sys_exit()` calls `do_exit()`
- `do_exit()` (linux/kernel/exit.c)
  - Calls `exit_signals()` which set the `PF_EXITING` flag in the `task_struct`
  - Set the exit code in the `exit_code` field of the `task_struct`, which will be retrieved by the parent

Process termination (cont'd)

- `do_exit()` (linux/kernel/exit.c)
  - Calls `exit_mm()` to release the `mm_struct` of the task
  - Calls `exit_sem()`. If the process is queued waiting for a semaphore, it is dequeued here.
  - Calls `exit_files()` and `exit_fs()` to decrement the reference counter of file descriptors and filesystem data, respectively. If a reference counter becomes zero, that object is no longer in use by any process, and it is destroyed.

Process termination (cont'd)

- Calls `exit_notify()`
  - Sends signals to parent
  - Reparents any of its children to another thread in the thread group or the init process
  - Set `exit_state` in `task_struct` to `EXIT_ZOMBIE`
- Calls `do_task_dead()`
  - Set the `state` to `TASK_DEAD`
  - Calls `schedule()` to switch to a new process. Because the process is now not scalable, `do_exit()` never returns.

Process termination (cont'd)

- At that point, what is left is `task_struct`, `thread_info` and kernel stack
- This is required to provide information to the parent
  - `pid_t wait(int *wstatus)`
- After the parent retrives the information, the remaining memory held by the process is freed
- Clean up implemented in `release_task()` called from the `wait()` implementation
  - Remove the task from the task list and release remaining resources

Zombie (or parentless) process

- **Q: What happens if a parent task exits before its child?**
- A child task must be *reparented*
- `exit_notify()` calls `forget_original_parent()`, that calls `find_new_reaper()`
  - Returns the `task_struct` of another task in the thread group if it exists, otherwise `init`
  - Then, all the children of the currently dying task are reparented to the reaper

Further readings

- Kernel Korner - Sleeping in the Kernel
- Exploiting Stack Overflows in the Linux Kernel
- security things in Linux v4.9

Next lecture

- Process scheduling