Linux Kernel Programming
Process Management

Pierre Olivier

Systems Software Research Group @ Virginia Tech

February 9, 2017

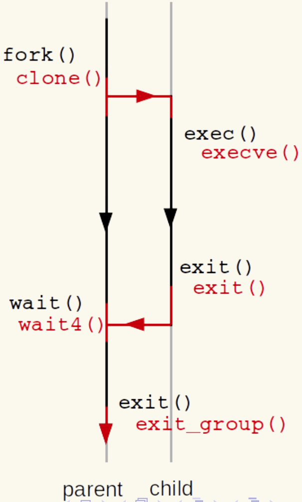
Outline

Outline

Process

Definition

- Refers to a **program currently executing in the system**
  - CPU registers
  - Location and state of memory segments (text, data, stack, etc.)
  - Kernel resources (open files, pending signals, etc.)
  - Threads
- Managed on a per-program way:
  - *Virtualization* of the processor and the memory
- Let's check an example with `strace (-f)`



Process

Sample program

```c
/* process.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void)
{
  pid_t pid = -42;
  int wstatus = -42;
  int ret = -1;

  pid = fork();
  switch(pid)
  {
    case -1:
      perror("fork");
      return EXIT_FAILURE;

    case 0:
      sleep(1);
      printf("Noooooooo!\n");
      exit(0);

    default:
      printf("I am your father!\n");
      break;
  }

  ret = waitpid(pid, &wstatus, 0);
  if(ret == -1)
  {
    perror("waitpid");
    return EXIT_FAILURE;
  }
  printf("Child exit status: %d\n",
     WEXITSTATUS(wstatus));

  return EXIT_SUCCESS;
}
```

```
gcc -Wall -Werror process.c -o process
./process
strace -f ./process > /dev/null
```

Process

fork() & exec() usage

- Tutorial on fork() usage:
  - http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html
- Combining fork() and exec():
  - https://ece.uwaterloo.ca/~dwharder/icsrts/Tutorials/fork_exec/

Outline

The process descriptor: task_struct
Presentation

- List of processes implemented as a linked list of task_struct

```c
struct tastk_struct {
  volatile long state;
  void *stack;
  /* ... */
  int prio;
  /* ... */
  cpumask_t cpus_allowed;
  /* ... */
  struct list_head tasks;
  /* ... */
  struct mm_struct *mm;
  /* ... */
  pid_t pid;
  /* ... */
  struct task_struct *parent;
  struct list_head children;
  struct list_head sibling;
  /* ... */
}
```

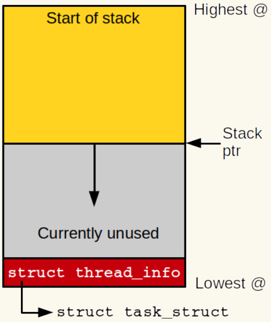
- Total size (Linux 4.8): 6976 bytes
- Full structure definition in linux/sched.h

The process descriptor: task_struct

Allocation & storage

- Prior to 2.6: `task_struct` allocated at the end of the kernel stack of each process
  - Allows to retrieve it without storing its location in a register
- Now dynamically allocated (heap) through the *slab allocator*
  - A `struct thread_info` living at the bottom of the stack

```
struct thread_info {
        struct task_struct      *task;
        __u32                   flags;
        __u32                   status;
        __u32                   cpu;
};
```



- Moved off the stack in 4.9 [2] because of potential exploit [1] when overflowing the kernel stack

The process descriptor: task_struct

Allocation & storage (2)

- **Process Identifier (PID):** `pid_t` (`int`)
  - Max: 32768, can be increased to 4 millions
  - Wraps around when maximum reached
- Quick access to `task_struct` of the task currently running on a core: `current`
  - `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()
```

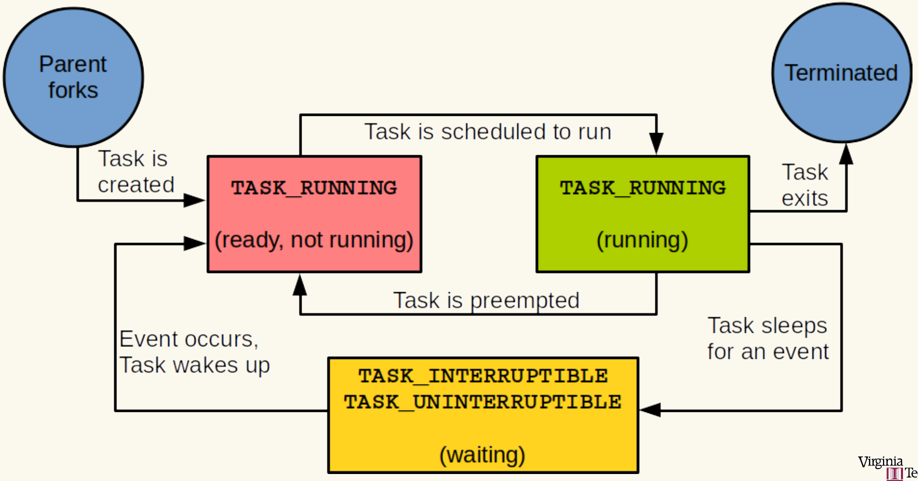
The process descriptor: task_struct

Process states

- **state** field of the task_struct
  - **TASK_RUNNING:**
    - Process is runnable (running or in a CPU run queue)
    - In user or kernel space
  - **TASK_INTERRUPTIBLE:**
    - Process is sleeping waiting for some condition
    - Switched to TASK_RUNNING on condition true or signal received
  - **TASK_UNINTERRUPTIBLE:**
    - Same as TASK_INTERRUPTIBLE but does not wake up on signal
  - __TASK_TRACED: Traced by another process (ex: debugger)
  - __TASK_STOPPED: Not running nor waiting, result of the reception of some signals to pause the process

The process descriptor: task_struct

Process states: flowchart



The process descriptor: task_struct
Process context and family tree

- The kernel can executes in **process** vs **interrupt** context
  - `current` is meaningful only when the kernel executes in *process context*
    - I.e. following a system call or an exception
- **Process hierarchy**
  - Root: `init`, PID 1
    - Launched by the kernel as the last step of the boot process
  - `fork`-based process creation:
    - Each process has a parent: `parent` pointer in the `task_struct`
    - Processes may have children: `children` field (`list_head`)
    - Processes may have siblings: `siblings` field
    - List of all tasks: `tasks` field
      - Easy manipulation through `next_task(t)` and `for_each_process(t)`
  - Let's check it out with the `pstree` command

Outline

Process creation
Presentation, Copy-On-Write

- Linux does not implements creating a tasks from nothing (*spawn*)
- **`fork()` & `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
- On `fork()`, Linux duplicates the parent page tables and creates a new process descriptor
  - It's *fast*, as **the address space is not copied**
    - Page table access bits: read-only
    - **Copy-On-Write** (COW): memory pages are copied only when they are referenced for write operations

Process creation

Forking: fork() and vfork()

- fork() is implemented by the **clone()** system call

1. sys_clone() calls _do_fork(), which calls **copy_process()** and starts the new task
2. **copy_process()**:
   1. Calls dup_task_struct()
      - Duplicates kernel stack, task_struct and thread_info
   2. Checks that we do not overflow the processes number limit
   3. Small amount of values are modified in the task_struct
   4. Calls sched_fork() to set the child state set to TASK_NEW
   5. Copies parent info: files, signal handlers, etc.
   6. Gets a new PID through alloc_pid()
   7. Returns a pointer to the created child task_struct
3. Finally, _do_fork() calls wake_up_new_task()
   - State becomes TASK_RUNNING

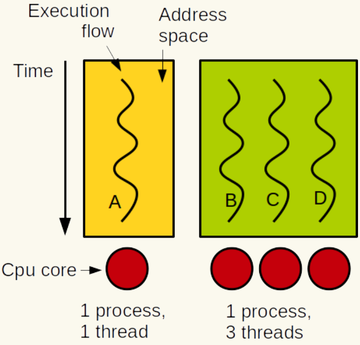
- vfork(): alternative without copy of the address space

Outline

Threads

Presentation

- Theory:



- **Threads** are concurrent flows of execution belonging to the same program **sharing the same address space**
- In Linux there is no concept of a thread
  - No scheduling particularity
  - A thread is just another process sharing some information with other processes
  - Each thread has its own `task_struct`
  - Created through `clone()` with specific flags indicating sharing

Threads

Kernel threads

- To perform background operations in the kernel: **kernel threads**
- 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`
- Used for several tasks:
  - Work queues (`kworker`)
  - Load balancing between CPU scheduling runqueues (`migration`)
  - etc.
  - List of all them with `ps --ppid 2`

Threads

Kernel threads: creation

- Kernel threads are all forked from the `kthread` kernel thread (PID 2), using `clone()`
  - To create a kernel thread, use `kthread_create()`
  - `include/linux/kthread.h`:

```
#define kthread_create(threadfn, data, namefmt, arg...) \
  kthread_create_on_node(threadfn, data, NUMA_NO_NODE, namefmt, ##arg)
```

```
struct task_struct *kthread_create_on_node(int (*threadfn)(void *data),
            void *data,
            int node,
            const char namefmt[], ...);
```

- When created through `kthread_create()`, the thread is not in a runnable state
  - Need to call `wake_up_process()`:

```
int wake_up_process(struct task_struct *p);
```

  - Or use `kthread_run()`

Threads

Kernel threads: creation (2)

- `kthread_run()`:

```
#define kthread_run(threadfn, data, namefmt, ...)                    \
({                                                                    \
  struct task_struct *__k                                             \
    = kthread_create(threadfn, data, namefmt, ## __VA_ARGS__);       \
  if (!IS_ERR(__k))                                                   \
    wake_up_process(__k);                                             \
  __k;                                                                \
})
```

- Thread termination:
  - Thread runs until it calls `do_exit()`:

```
void do_exit(long error_code) __noreturn;
```

  - Or until another part of the kernel calls `kthread_stop()`:

```
int kthread_stop(struct task_struct *k);
```

Virginia Tech

Outline

Process termination

Termination steps: do_exit()

- 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() (kernel/exit.c):
  1. Calls exit_signals() which set the PF_EXITING flag in the task_struct
  2. Set the exit code in the exit_code field of the task_struct
     - To be retrieved by the parent
  3. Calls exit_mm() to release the mm_struct for the task
     - If it is not shared with any other process, it is destroyed
  4. Calls exit_sem(): process dequeued from potential semaphores queues
  5. Calls exit_fs() and exit_files() to update accounting information
     - Potential data structures that are not used anymore are freed

Process termination

Termination steps: do_exit() (2)

- do_exit() (continued):
  6. Calls exit_notify()
     - Sends signals to parent
     - Reparent potential children
     - Set the exit_state of the task_struct to EXIT_ZOMBIE
  7. Calls do_task_dead()
     - Sets the state to TASK_DEAD
     - Calls __schedule() and never returns
- At that point, what is left is the task_struct, thread_info and kernel stack
  - To provide information to the parent
  - Parent notifies the kernel when everything can be freed

Process termination

task_struct cleanup

- Separated from the process of exiting because of the need to pass exit information to the parent
  - `task_struct` must survive a little bit before being deallocated
    - Until the parent grab the exit information through `wait4()`
- Cleanup implemented in `release_task()` called from the `wait4()` implementation
  - Remove the task from the task list
  - Release and free remaining resources

Process termination

Parentless tasks

- **A parent exits before its child**
  - Child must be *reparented*
    - To another process in the current thread group ...
    - ... or `init` if that fails
- `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 the one from `init`
  - Then, all the children of the currently dying task are reparented to the reaper

Bibliography I

[1] Exploiting stack overflow in the linux kernel.
https://jon.oberheide.org/blog/2010/11/29/exploiting-stack-overflows-in-the-linux-kernel/.
Accessed: 2017-01-23.

[2] Security things in linux v4.9.
https://outflux.net/blog/archives/2016/12/12/security-things-in-linux-v4-9/.
Accessed: 2017-01-23.