CS 5264/4224; ECE 5414/4414
(Advanced) Linux Kernel Programming
Lecture 6

Process Management

February 13, 2025

Huaicheng Li

Acknowledgement: Credits to Dr. Changwoo Min for the original LKP lecture slides.

Agenda

- Process
- Linux PCB: task_struct
- Process creation
- Threads
- Kernel thread API

Process

- A program currently executing in the system
- A process consists 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 User-space View

- pid_t fork(void)
  - create 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

fork() Example

```c
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("Nooooooooo!\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));
```

Process Descriptor: task_struct

```c
/* 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_mask;    /* 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. */
}; /* TODO: Let's check `pstree` output. */
```

More about task_struct

- task_struct is dynamically allocated at heap b/c of potential exploit when overflowing the kernel stack
- For efficient access to current task_struct, kernel maintains per-CPU variable, named "current_task"
  - Use "current" to get "current_task"

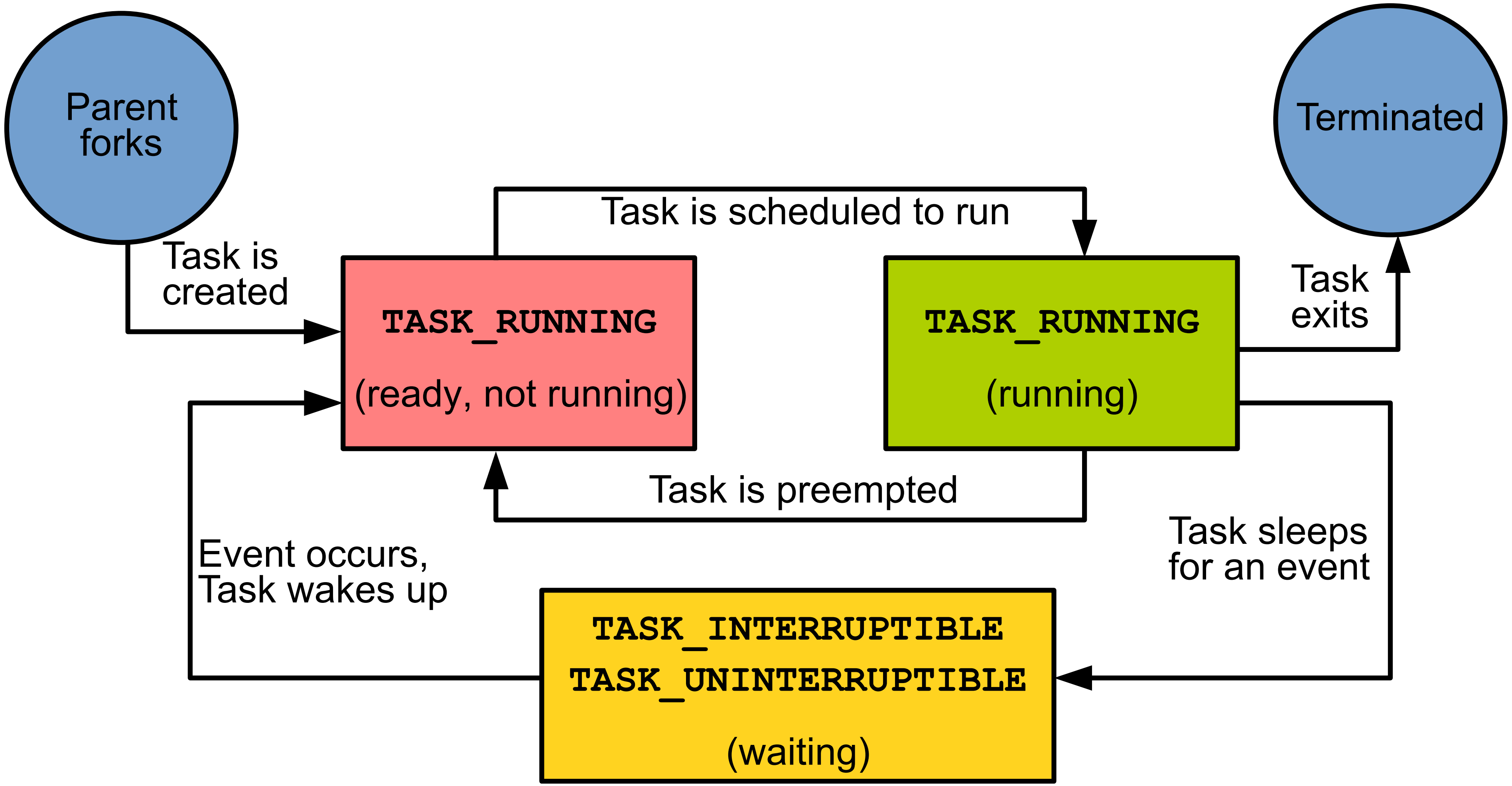
```
/* linux/arch/x86/include/asm/current.h */
DECLARE_PER_CPU(struct task_struct *, current_task);
static __always_inline struct task_struct *get_current(void)
{
    return this_cpu_read_stable(current_task);
}
#define current get_current() /* TODO: Let's check how `current` is used. */
```

PID: pid_t

- Maximum is 32768 (int)
- Can be increased to 4 million
- Wraps around when max 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
- TASK_INTRRUPTIBLE
  - 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 doesn't wake up on signal
- __TASK_TRACED
  - Traced by another process (e.g., GDB)
- __TASK_STOPPED
  - Not running nor waiting, result of the reception of some signals (e.g., SIGSTOP) to pause the process


Parent
forks
Task is
created
TASK_RUNNING
(ready, not running)
Task is scheduled to run
TASK_RUNNING
(running)
Terminated
Task
exits
Task is preempted
Event occurs,
Task wakes up
Task sleeps
for an event
TASK_INTERRUPTIBLE
TASK_UNINTERRUPTIBLE
(waiting)

Producer-Consumer Example

- Producer
  - generate an event and wake up a consumer
- Consumer
  - check if there is an event
  - if so, process all pending events in the list
  - otherwise, sleep until the producer wakes the consumer 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 booting process
  - Its PID is 1
  - Its task_struct is a global variable, named “init_task” (linux/init/init_task.c)

```
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
```

```
➜  ~ pstree -p
systemd(1)─┬─ModemManager(631)─┬─{ModemManager}(653)
           │                   ├─{ModemManager}(664)
           │                   └─{ModemManager}(665)
           ├─agetty(725)
           ├─agetty(733)
           ├─cron(720)
           ├─dbus-daemon(592)
           ├─irqbalance(597)───{irqbalance}(603)
           ├─networkd-dispat(599)
           ├─polkitd(600)─┬─{polkitd}(606)
           │              ├─{polkitd}(609)
           │              └─{polkitd}(610)
           ├─rsyslogd(601)─┬─{rsyslogd}(616)
           │               ├─{rsyslogd}(617)
           │               └─{rsyslogd}(618)
           ├─snapd(605)─┬─{snapd}(634)
           │            ├─{snapd}(635)
           │            ├─{snapd}(636)
           │            ├─{snapd}(637)
           │            ├─{snapd}(639)
           │            ├─{snapd}(676)
           │            ├─{snapd}(678)
           │            ├─{snapd}(679)
           │            ├─{snapd}(680)
           │            ├─{snapd}(682)
           │            ├─{snapd}(683)
           │            ├─{snapd}(694)
           │            ├─{snapd}(695)
           │            ├─{snapd}(696)
           │            └─{snapd}(736)
           ├─sshd(677)───sshd(761)───sshd(847)───zsh(848)───pstree(907)
           ├─systemd(764)───(sd-pam)(765)
           ├─systemd-journal(376)
           ├─systemd-logind(607)
           ├─systemd-network(575)
           ├─systemd-resolve(589)
           ├─systemd-timedat(734)
           ├─systemd-timesyn(578)───{systemd-timesyn}(590)
           ├─systemd-udevd(415)
           ├─udisksd(608)─┬─{udisksd}(620)
           │              ├─{udisksd}(625)
           │              ├─{udisksd}(629)
           │              ├─{udisksd}(654)
           │              └─{udisksd}(668)
           └─unattended-upgr(648)───{unattended-upgr}(701)
```

Process Family Tree

- “fork” based process creation
  - parent task: current->parent
  - children tasks: current->children
  - sibling under the parent: current->siblings
  - list of all tasks in the system: current->tasks
  - macros for easy exploration:
    - » next_task(t), for_each_process(t)
- Check out the implementation!

Process Creation

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

Copy-on-Write (CoW)

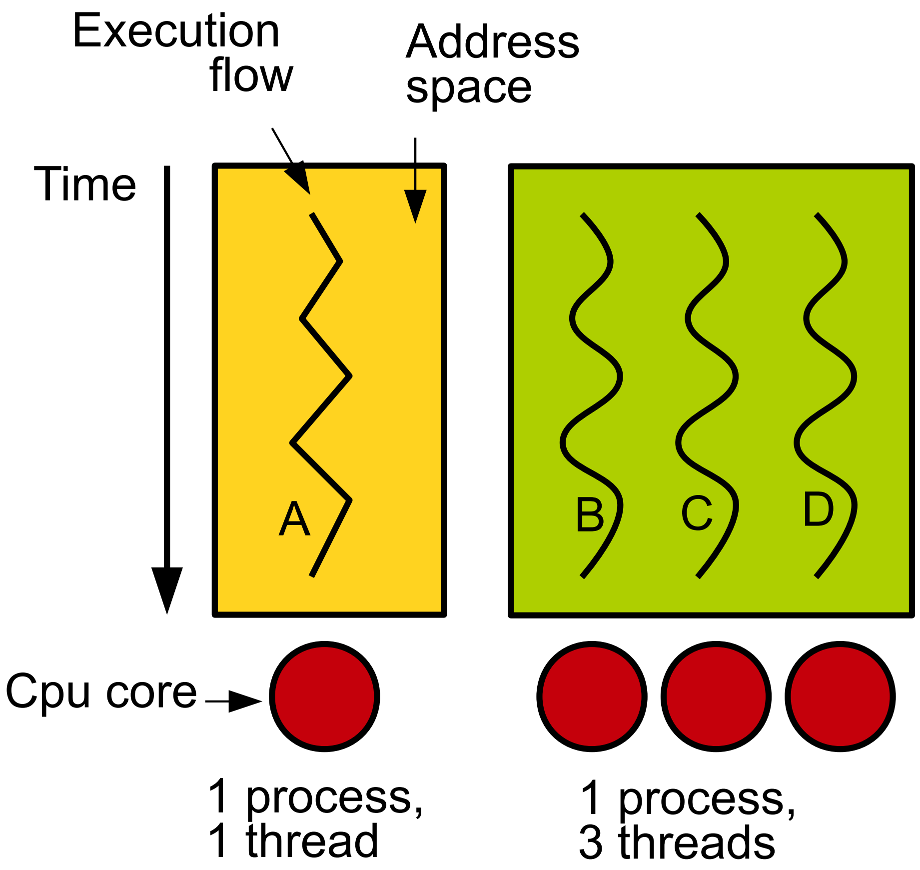
- On fork(), Linux duplicates the parent page tables and creates a new process descriptor
  - Change page table access bits to read-only
  - When a page is accessed for write operations, that page is copied and the corresponding page table entry is changed to read-write
- fork() is fast by delaying or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

Forking

- fork() is implemented by the "clone()" system call
- kernel_clone() calls copy_process() and starts the new task
- copy_process()
  - dup_task_struct(), which duplicates kernel stack, task_struct, and thread_info
  - Check that we do not overflow the process number limit
  - Various members of the task_struct are cleared
  - 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, wake_up_new_task()
  - The new child task becomes TASK_RUNNING

Thread

- Threads are concurrent flows of execution belong to the same process sharing the 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”
  - Create through clone() system call with specific flags indicating sharing
  - clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);

Kernel Thread

- Use to perform background operations in the kernel
- Very similar to sue space threads
  - They are schedulable entities (lie regular processes)
- However, they do not have their own addr space
  - task_struct->mm is NULL
  - why?
- Kernel threads are all forked from the “kthreadd” thread (PID 2)
- Use cases (ps –ppid 2)
  - Work queues (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 asks a kernel thread to stop using kthread_stop()
  - A kernel thread should check kthread_should _stop() to decide to continue or stop

```c
/**
 * 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);
```

```
/**
 * 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;
        }
        /* ... */
```

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)
  - Cals exit_signals() which set the PF_EXITTInG 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
  - Calls exit_mm() to release the mm_struct of the task
  - Calls exit_sem(), if the process is queued waiting for a semaphore, dequeue here
  - Calls exit_files() and exit_fs() to decrement the reference counter of file descriptors and filesystem data, respectively. If a refenrece counter becomes zero, that object is no longer in use by any process, and it is destroyed.

- Calls exit_notify()
  - Sends signals to parent
  - Re-parent any of tis 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 process is now not schedulable, do_exit() never returns.
- At this point, what is left is task_structu, thread_info, and kernel stack
- This is required to provide information to the parent
  - pid_t wait(int *wstatus)
- After the parent retrieves the information, the remaining memory held by the process is freed
- Cleanup implemented in release_task() called from wait()
  - Remove the task from the task list and release remaining resources

Zombie Process

- What happens if a parent task exits before its child?
- A child must be re-parented
- 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, other init
  - Then, all the children of the currently dying task are re-parented to the reaper

Further Readings

- Kernel Korner – Sleeping in the Kernel
- Exploiting Stack Overflows in the Linux Kernel

Next Lecture

- Process scheduling!