Process Management

Xiaoguang Wang

Summary of past lectures

Get, build, and explore the Linux kernel

System call: interface between applications and kernel

Kernel data structures

Kernel modules

Kernel debugging techniques

Recap: kernel debugging techniques

Print debug message: `printk()`

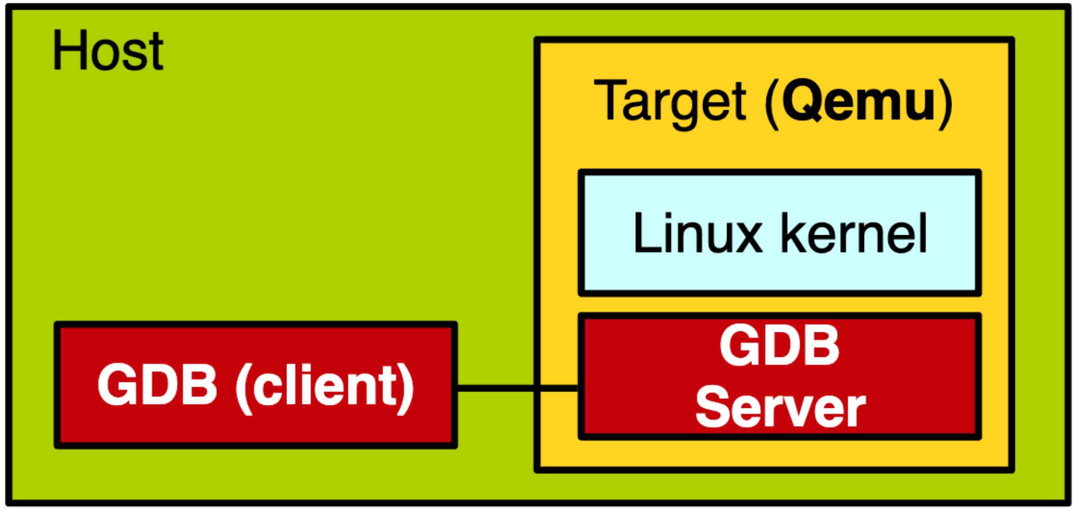
Assert your code: `BUG_ON(c)`, `WARN_ON(c)`

Analyze kernel panic message

Debug with `QEMU/gdb`

Recap: debugging with QEMU/gdb

- Linux kernel runs in a virtual machine (KVM or emulated on QEMU)
- Hardware devices are emulated with QEMU
- GDB server runs in QEMU, contacting with the virtual machine emulation logic
- So, it can fully control Linux kernel running on QEMU



Recap: connect and debug with QEMU/gdb

```
$ cd /path/to/linux-build
$ gdb vmlinux
(gdb) target remote :1234
Remote debugging using :1234
native_irq_disable ()
    at ./arch/x86/include/asm/irqflags.h:37
37              asm volatile("cli": : :"memory");
(gdb) step
(gdb) next
(gdb) b btrfs_init_sysfs
(gdb) lx-dmesg
(gdb) p $lx_current().pid
... ...
(gdb) continue
Continuing.
```

hw4 and hw5

Kernel module and data structures

- hw4 due this Friday (Feb 2nd)
  - linked list
  - kernel module
- hw5 due next Friday (Feb 9nd)
  - hash table, rbtree and xarray
  - /proc file system

No hesitate to ask for help!

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 the user-space view

`pid_t fork(void)`

- creates a new process by duplicating the calling process

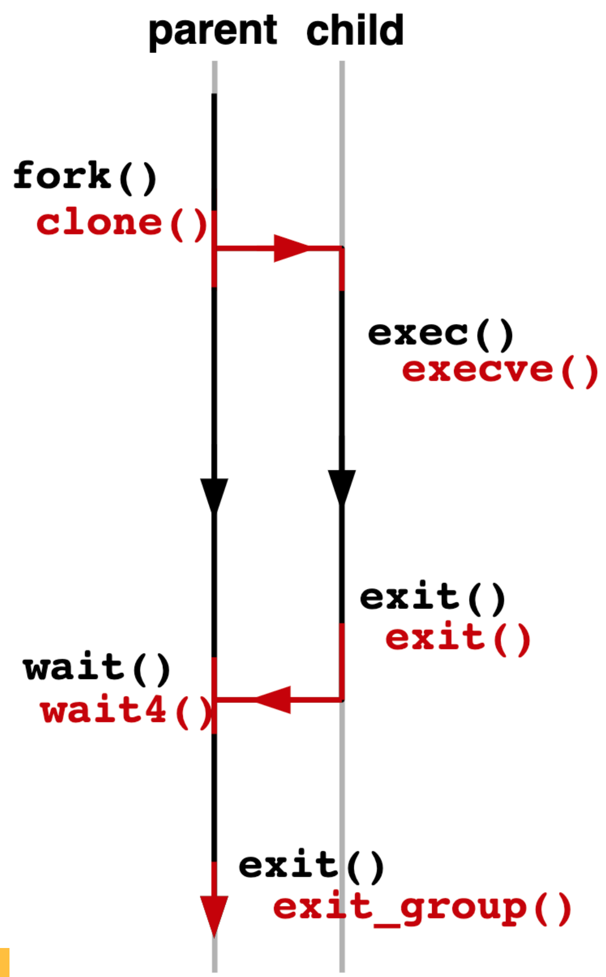
`int execv(const char *path, const char *arg, ...)`

- 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 the user-space view



fork() example

Code on blackboard

- Download and run it
- Try `make run`

```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("I'm a child process!\n");
            exit(99);
        default:  /* pid = pid of child: parent process */
            printf("I'm the parent process! Your pid: %d\n", pid);
            break;
    }
    ret = waitpid(pid, &wstatus, 0); /* A parent wait until child terminates */
    if(ret == -1)
        return EXIT_FAILURE;
    printf("Child exit status: %d\n", WEXITSTATUS(wstatus));
    return EXIT_SUCCESS;
}
```

Processor descriptor: `task_struct`

```c
/* 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. */
```

task_struct

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

```
/* 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. */
```

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`

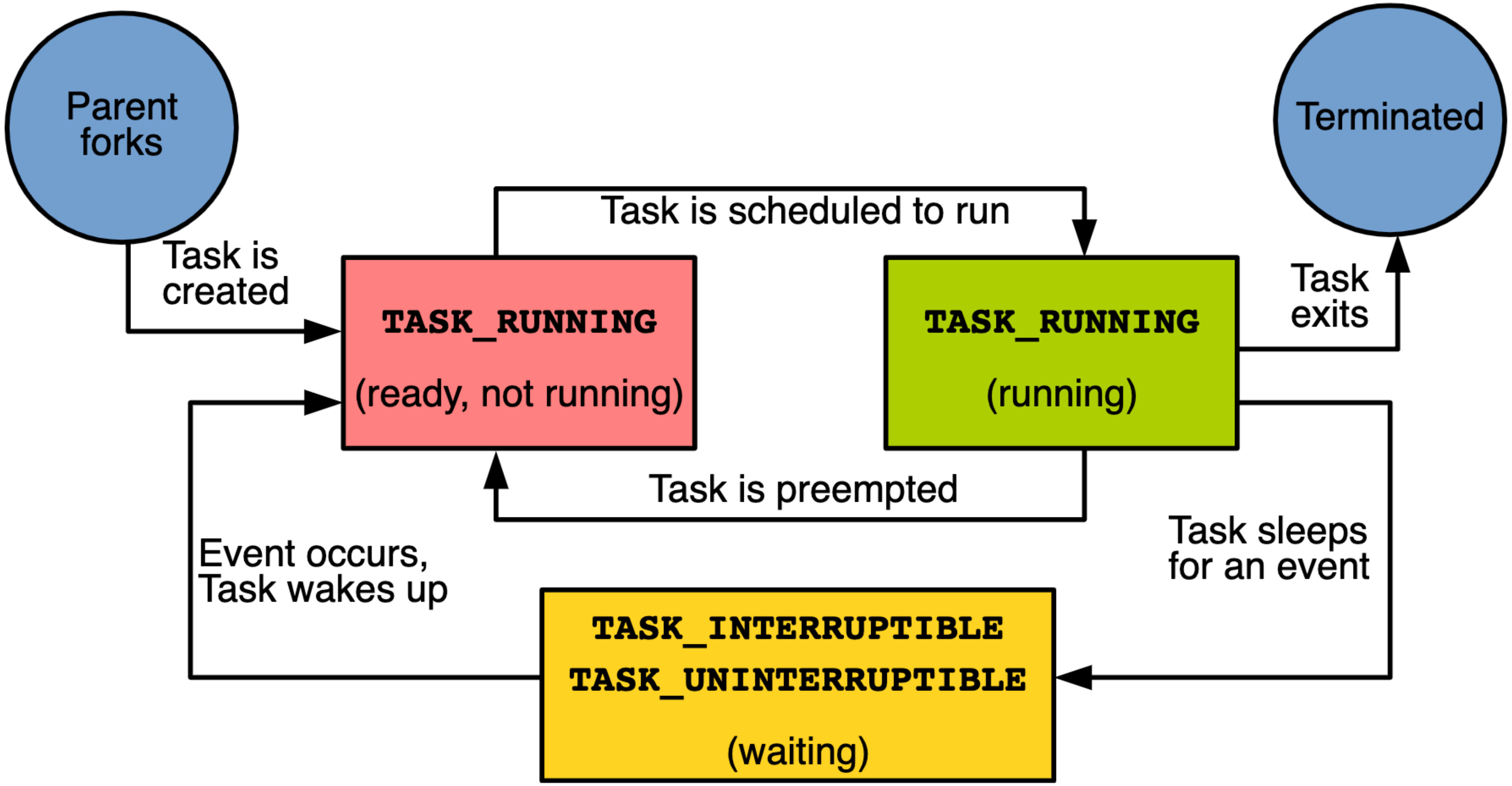
`__TASK_TRACED`

- Traced by another process (i.e., 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`)

Recent Debian-based Linux distributions use `systemd`

Let's check process tree using `pstree`

```
▶ pstree -g
systemd(1)─┬─ModemManager(939)─┬─{ModemManager}(939)
           │                   └─{ModemManager}(939)
           ├─VGAuthService(752)
           ├─agetty(907)
           ├─cron(876)
           ├─dbus-daemon(878)
           ├─dockerd(1115)─┬─containerd(1168)─┬─{containerd}(1168)
           │               │                  ├─{containerd}(1168)
```

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 easy to explore:
    - next_task(t), for_each_process(t)

TODO: Let's ask ChatGPT on how these macros are used!

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

Q: Then how to efficiently create 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 preventing copying of data

`fork()` saves memory by sharing read-only pages among descendants

Fork

`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`
- Checks that we do not overflow the processes number limit
- Various members of the `task_struct` are cleared

Fork

`copy_process()`

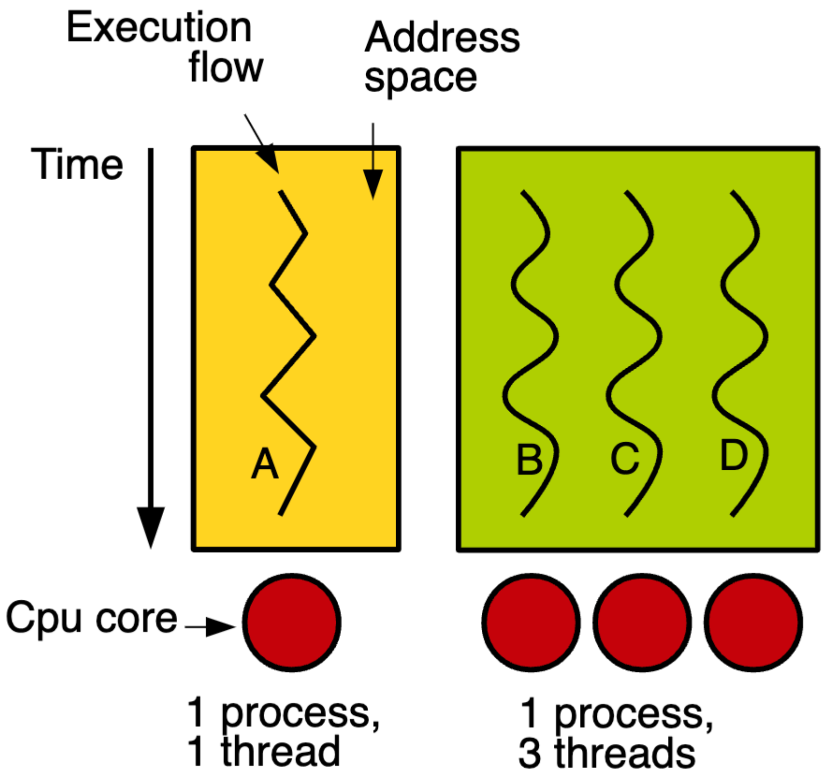
- Calls `sched_fork()` to set the child state set to `TASK_NEW`
- Copies parent information such as files, signal handlers, etc.
- Gets a new PID using `alloc_pid()`
- Returns a pointer to the new child `task_struct`

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` thread (PID 2)

Find all kernel threads? Use `ps --ppid 2` (have a try!)

- 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. 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. Return: 1 if the process was woken up, 0 if it was already running. */*

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

Kernel thread

```c
/* kthread_run - create and wake a thread.   threadfn: the function to run until
signal_pending(current).   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

Let's find a `kthread` usage!

- Can you find a usage of `kthread_create` or `kthread_run`?

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()` (`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

`do_exit()` (`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

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 schedulable, `do_exit()` never returns.

Process termination

At this 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 `wait()`

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 reading

Book: LKD3: Chap 3 Process Management

Kernel Korner - Sleeping in the Kernel

Exploiting Stack Overflows in the Linux Kernel

Feedback