1044414: Advanced Operating Systems and Virtualization **Spring 2018**

Lecture 17: May 8

Lecturer: Alessandro Pellegrini *Scribe: Anxhelo Xhebraj*

17.1 Fork syscall

There is a maximum number of threads that can be spawned in linux which is `max_threads` variable in the `fork_init()` function.

Both fork and pthread create rely on do fork function to initialize a process.

Code is from kernel 2.4. In `sys_clone` some flags that tell how the current thread should be cloned are used. Both the flags and the stack pointer are taken from registers.

In order for a new process to be spawned a stack must be setup. `__clone` function setups the registers in the pointer.

`get_pid` is an entry point to the pid allocator subsystem that uses idr.

`copy_mm` copies the memory map.

`sem_undo` is used to ensure that after some process dies, the locks acquired by it are released.

On forking, the process might still have some amount of time of its timeslice. That is halved and split to the parent and the newly created child.

`SET_LINKS` and `hash_pid` are used to initialized the data structures for thread management (namespaces etc).

On kernel 2.4 the information related to the process control block are placed in the kernel level stack.

`copy_thread`

When creating the new process the kernel level stack must be properly initialized respecting the convention of the dispatcher which expects the registers to be found on the kernel level stack.

`copy_mm`

One if case checks whether virtual memory is shared or not. In the code the reference counter of the memory map is increased to ensure that it is not teared down while still in use.

`mm_init` initializes a new PGD.

`dup_mmap` initializes memory management for the new process.

`copy_page_range` sets up the PTE. It also ensures Copy On Write if needed.

17.2 Kernel level threads

It has its own `task_struct` and never switches to user space. The API for creating new threads is `kthread_create`. The interleaved execution of the kernel allows the system to be responsive to user interactivity.

This threads are not related to init. When pid0 spawns init also `kthreadd` is spawned of which childs are kernel level threads.

On context switch from kernel space to user space, it is checked if there are any signals pending. If those are handled by the US then control is passed to the handler. How to go back in the execution path that was interrupted? Theoretically another switch to kernel must happen that then restores the context.

In early days, stack was executable. In signal handling through SIGRETURN the whole PCB was present in stack. Sig return oriented programming. XD, execution deny flag allows to set some memory regions as not executable. Used in modern systems to make the stack not executable.

17.3 Out of Memory Killer

The first thing done is check whether some process has some kill signal pending. If there isn't, the process that gives highest revenue in terms of memory is killed. Kernel threads are skipped in this scenario. `oom_badness` evaluates the various processes.

17.4 Linux Watchdog

Concept derived from embedding systems. Checks whether the system is still running properly.

Two parts implement it. A kernel level module allows to perform hard reset and the user space background daemon that refreshes the timer. As long as user space is living the system won't be reset.

NMI are interrupts delivered to the handler in any case, even if they are masked.

`ioctl` allows operations that are not captured by system call semantics. This function allows to interact with drivers directly through files.

This allows to integrate kernel modules without having to mess systemcalls.

17.5 Loadable Kernel Modules

In order to implement a module, the license must be specified. Two function pointers. When loading a kernel module its image is taken from disk and these functions are "constructors/destructors" of the module. For example watchdog in the init creates the file in `/dev`. Kernel Level Modules can use facilities of other modules. At some point one module calls a function exposed into another module.

Kernel keeps a reference counter for each kernel module. `lsmod` lists all modules that have been loaded by the system. The number is the reference counter telling how many modules use that module. On unmounting some module all its dependent module must be unmounted before.

The reference counter of a specific kernel module is incremented through `try_module_get` which might fail. `module_put` is the opposite that decrements the reference counter. Some modules might be declared as unloadable.