Operating Systems 2

Process Management

Arkadiusz Chrobot

Department of Information Systems

March 12, 2025

Notes

Outline

Notes

Process Descriptor

Every operating system stores all information about a single process in a *process descriptor*. In case of Linux it is a structure of the `struct task_struct` type, defined in the `linux/sched.h` header file. The structure is allocated by the *slab allocator* (See second laboratory instruction or wait until the 10^{th} lecture for an explanation ⌣) Its size is about 1.7 KiB. Some of the members of the structure are pointers to other structures, equally big or even bigger.

Notes

Process Descriptor

Descriptor Location

In the Linux kernel versions older than 2.6 the process descriptor had been relatively small, and it had been stored at the bottom of the process kernel stack. However, in the 2.6 it became so big that it would occupy too much space in the stack. Starting with this versions of the kernel, another structure of the `struct thread_info` type has replaced the process descriptor at the bottom of the stack, but this structure has a pointer to the descriptor (see Fig. 1). The `thread_info`, like the descriptor is allocated for each process, however it is much smaller than the descriptor, and it stores low-level data related to the process. For the definition of `struct thread_info` type for the x86 CPUs family see the Listing 1. It is taken from the `asm/thread_info.h` header file for the 4.8 version of the kernel.

Notes

Process Descriptor

Descriptor Location

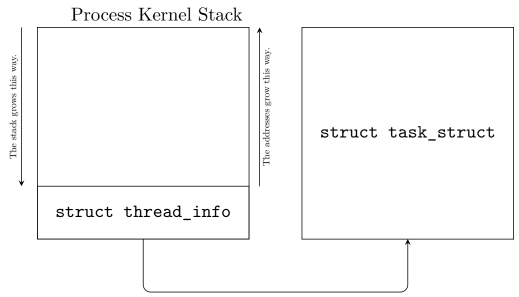

Figure 1: The Process Kernel Stack and The Process Descriptor

Notes

Process Descriptor

The `struct task_info` Type Definition

```
struct thread_info {
        struct task_struct *task; /* main task structure */
        __u32 flags; /* low level flags */
        __u32 status; /* thread synchronous flags */
        __u32 cpu; /* current CPU */
};
```

Listing 1: The definition of `struct thread_info` for the x86 CPUs family in kernel version 4.8

Notes

Process Descriptor

The `current` macro

The kernel code that runs in the process context, particularly a system call, usually needs to quickly locate the descriptor of the process that activated it, because the descriptor stores all the data about that process. In CPUs of the RISC organization the address of the descriptor can be stored in one of the registers, but for CPUs of the CISC organization it has to be calculated on the fly, each time the kernel needs to access the descriptor. To this end kernel programmers created the `current` macro, that returns address of the current process descriptor. In this case the word “current” should be understood as “the one that activated the kernel code”. The `current` macro calls the `current_thread_info()` function, which is processor family specific. The implementation of this function for the family of 32-bit Intel x86 CPUs is given in the Listing 2.

Notes

Process Descriptor

The `current` macro

```
1 movl $-8192, %eax
2 andl %esp, %eax
```

Listing 2: Part of the definition of the `current_thread_info()` function for the x86-32 CPUs family

In the line no. 1 the value -8192 is stored in the `eax` register, which is 32 bits wide. The number 8192 is the size of the stack (two pages, each of the size of 4096 bytes, which gives $2 \times 4096 = 8192$). In binary it is 00000000000000000010000000000000. In the code the value is a negative number, which means that in the two’s complement it is represented as 11111111111111111110000000000000. This value is used in the line no. 2 of the function to mask out the thirteen least-significant bits of the stack pointer, stored in the `esp` register. The resulting value is the address of the bottom of the stack and also the address of the `thread_info` structure.

Notes

Process Descriptor

The `current` macro

After calculating the address of the `thread_info` structure the `current` macro needs only to return the value of the `task` field which stores the address of the current process descriptor:

```
current_thread_info()->task
```

The way of calculating the address of the current process descriptor explains why the process kernel stack has to be linked with the process descriptor.

Notes

Process Descriptor

The `current` macro

Starting with the version 4.9 of the kernel, the linkage between the process descriptor and its kernel stack is preserved only for some CPUs supported by the kernel. In case of the x86 CPUs family, the pointer to the current process descriptor is defined as a so-called per-cpu variable, or (in newer versions) as a member of a structure that is such a variable. Just as its name suggests, the per-cpu variable is created for each of the CPUs separately, which means that it is local to these CPUs. That is convenient in a case of multiprocessor computer systems[1]. The `current` macro needs only to return the address stored in such a variable.

[1] In the OS2 course, we assume that the words *multiprocessor* and *multicore* mean the same. The computer system with only one CPU that has only one core will be called a uniprocessor system.

Notes

Process Descriptor

Process Identifier

Among data stored in the process descriptor is the process identifier (*PID*). This is a natural number that is assigned by the kernel to every process. The upper limit of this number is 32 767. This is because even the newest Linux kernel has to be backward compatible with its older versions and the original Unix. However, this limit can be changed by a privileged user, even when the kernel runs. The PID of the value 1 is assigned to a user process which is an ancestor of all other running processes. Historically this process is named `init`, but in newest distributions of Linux it has been replaced by `upstart` or `systemd`. The identifier is stored in the member of the descriptor named `pid` of the type `pid_t`.

Notes

Process Descriptor

Process State

Any running process changes its state. Inside the kernel the current state of the process is described by a special constant, also called a flag, whose value is stored in the `state` field of the process descriptor. The number, the usage and the names of the flags changed in the history of kernel development, but the most important ones are the following:

TASK_RUNNING describes a process that is either active or ready to run; the kernel doesn't differentiate between those two types of processes,

TASK_INTERRUPTIBLE the process is waiting for some event to happen; in the Linux terminology it sleeps or it is blocked; it can be awaken (unblocked) by the awaited event or by any signal,

TASK_UNINTERRUPTIBLE the process is waiting for some event and it can be awaken only by the event; the state is rarely used, because it prevents aborting the process,

Notes

Process Descriptor

Process State

TASK_KILLABLE the process is waiting for an event; it can be awaken by the event or any signal that causes its abortion; those signals are called *fatal signals*,

TASK_STOPPED the process has been stopped by a signal,

TASK_TRACED the process is being debugged.

There is also a separated field in the process descriptor for storing the state of an exited process. The member is called `exit_state` and can store values of the following flags:

EXIT_ZOMBIE the process exited, but awaits for its parent process to invoke the `wait4()` system call; there are still the kernel stack and descriptor in the RAM that belong to the process,

Notes

Process Descriptor

Process State

EXIT_DEAD the process has exited, its parent called the `wait4()` system call, but kernel hasn't finished removing the process yet; it is used for informing the kernel code running on other CPUs that the process is already being removed.

The state of the current process can be changed with the use of the `set_current_state()` function. To change the state of any process the `set_task_state()` function can be applied.

Notes

Process Family

In Linux the user processes are related to one another. Those connections create a *process family tree*. Each process has a parent, with the exception of the `init` process (or `upstart` or `systemd`), which is the ancestor of all other processes. Every process can have *children*. The direct children of the process are called *siblings*.
Those family connections are mapped in the process descriptor. For example, the address of the descriptor of the process parent is stored in the `parent` field of its descriptor. The `children` field is a list of pointers to descriptors of the process children (if they exist). The following code gets the descriptor of the current process parent:

```
struct task_struct *task = current->parent;
```

This one could be applied for traversing the list of its children:

```
struct task_struct *task;
struct list_head *list;

list_for_each(list, &current->children) {
	task = list_entry(list, struct task_struct, sibling);
}
```

Listing 3: Kernel code for traversing the list of children

Notes

Process Family

The code from the Listing 3 uses the API of generic implementation of list created by the kernel developers. For explanations see the 3^{rd} laboratory instruction. Descriptors of all user processes are connected in a circular doubly linked list. The first element of the list is the descriptor of the `init` process (or its newest replacements). To traverse the list the `for_each_process(task)` macro can be applied (see the 3^{rd} laboratory instruction for details). The macro `next_task(task)` returns the address of the next process descriptor in the list and the `prev_task(task)` returns the address of the previous process descriptor in the list. The kernel has also an array called `pidhash`, that stores pointers to descriptor of all processes. The array has 32 768 elements which means that its indices has the same values as all possible PIDs of processes. The array allows the kernel to quickly obtain the descriptor of any process, provided its PID is known.

Notes

Process Creation

Just like any other Unix-like operating system, Linux allows the user processes the create a new process (a child) by calling the `fork()` or `vfork()` function. There is also a Linux-specific function that can be applied for creating a new process. It is called `clone()`. Linux uses the *copy-on-write* (*COW*) technique to allow the child and the parent to share their address spaces as long as it is possible. If any of the processes starts modifying data, then and only then the address spaces are separated. The parent and the child get their own data segments, but they still share the text (code) segment, which is read-only. The family of the `exec()` functions allows the process to execute a different program than its parent.
Regardless which user-space function is used for creating a new process in Linux, the `clone()` system call is invoked, which calls the `do_fork()` kernel function, which in turn invokes the `copy_process()` function.

Notes

Process Creation

The `copy_process()` function performs the following tasks:

1. creates for the new process the kernel stack and the descriptor,
2. verifies if the new process won't exceed the limit of the number of processes for the current user (the owner of the parent process),
3. sets the state of the new process to `TASK_UNINTERRUPTIBLE`,
4. sets the process flags, obtains a PID for the process,
5. depending on the arguments passed to the invocation of the `clone()` it copies from parent or creates anew structures for managing resources and handling signals,
6. returns pointer to the descriptor of the new process.

After the `copy_process()` exits the control returns to the `do_fork()` function, which eventually awakes and runs the child process.

Notes

Process Creation
User-Space Threads

Linux allows user-space processes to create threads, but unlike other operating systems it doesn't have any subsystems dedicated to handling those threads. For Linux, a user-space thread is just a user-process that always shares majority of its resources, most notably the address space, with other processes (its parent and, potentially, siblings). To create a user-space thread the `clone()` system call has to be invoked with following arguments:

```
clone(CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND,0);
```

while the `fork()` function calls `clone()` like this:

```
clone(SIGCHLD,0);
```

and the `vfork()` function like this:

```
clone(CLONE_VFORK|CLONE_VM|SIGCHLD,0);
```


Notes

Process Creation
Arguments for the `clone()` System Calls

There are defined several flags in the `linux/sched.h` header file that serve as arguments for the `clone()` system call:

- CLONE_FILES parent and child share open files,
- CLONE_FS parent and child share file system data,
- CLONE_IDLETASK set PID of the child to 0 (it will be an idle task),
- CLONE_NEWNS create a new namespace for the child,
- CLONE_PARENT the child grandparent will become its parent,
- CLONE_PTRACE continue tracing the child,
- CLONE_SETTID write the TID (Thread Identifier) back to the user-space,
- CLONE_SETTLS create a new TLS (Thread Local Storage) for the child,
- CLONE_SIGHAND parent and child share signal handling,

Notes

Process Creation

Arguments for the `clone()` System Calls

- CLONE_SYSVSEM parent and child share System V `SEM_UNDO` semantics,
- CLONE_THREAD parent and child belong to the same group of threads,
- CLONE_VFORK the parent will sleep until the child wakes it (the child has been created via `vfork()` function call),
- CLONE_UNTRACED prevents setting the `CLONE_PTRACE` flag for the child,
- CLONE_STOP start the child in the `TASK_STOPPED` state,
- CLONE_CHILD_CLEARTID clear the TID for the child,
- CLONE_CHILD_SETTID set the TID for the child,
- CLONE_PARENT_SETTID set the TID for the parent,
- CLONE_VM parent and child share address space.

Notes

Process Creation

Kernel-Space Threads

The kernel also can create its threads. Examples of such threads are *ksoftirqd* and *kworker*. Kernel threads don't have their own address spaces, they share it with the rest of the kernel. The code of the thread is implemented as a function that usually runs a loop. Most of the kernel threads exit when the system is shutdown or rebooted or the kernel module where they are implemented is unloaded form the kernel. A single kernel thread can be created with the use of the `kernel_thread()` function. In the 2.6.1 kernel version a patch by Rusty Russell was added that introduces a more convenient API for managing kernel threads.

Notes

Process Creation

Kernel-Space Threads

The API consists of following functions and macros:

- `kthread_create()` creates a new kernel thread and returns pointer to its descriptor,
- `kthread_run()` creates and activates the new kernel thread,
- `kthread_stop()` sends a signal to the kernel thread suggesting that it should terminate,
- `kthread_should_stop()` is a function invoked inside the kernel thread and used as a condition in the thread's main loop that checks if the thread should terminate,
- `kthread_bind()` assigns the thread to one or more CPUs.

More detailed description of the API is available in the 5^{th} laboratory instruction.

Notes

Process Termination

The process terminates by calling directly or indirectly the `exit()` function which invokes the `_exit()` system call. The system call then calls the `do_exit()` function. The latter is responsible for releasing most of the kernel data structures related to the process and messaging the parent process that its child has terminated. Only the process descriptor and kernel stack are left in the computer memory. Those structures are freed by the `release_task()` function called by the `wait4()` system call. The function decrements the counter of processes belonging to the user, removes the process descriptor from the `pidhash` array, from the list of traced processes (if the process was traced), from the process list, and finally deallocates the process descriptor and the `thread_info` structure.

Notes

Process Termination

If the parent of the exited process terminated before its child, the process would stuck in the zombie state. In this case the exited process is *adopted* by the `init` process (or its replacements) or by a process that belongs to the same group as the parent of the exited process. The adoption is performed by the `forget_original_parent()` function invoked by the `do_exit()` function. The former checks the list of processes and the list of traced processes to find the new parent.

Notes

Questions

?

Notes

The End

Thank You for Your attention!

Notes

Notes