

CSIS0230A Principle of Operating Systems (Class A)

Notes for Tutorial 5

Process Structure

In the last tutorial, you learnt how to add a trivial system call. In this tutorial, you will learn the structure of process.

1. Process Management

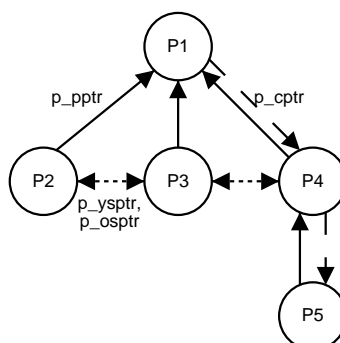
In the lecture, you learn that a process is a unit that resources are allocated in the kernel. Each process descriptors is implemented by a *task_struct* data structure (task and process are terms that Linux uses interchangeably). There are many members in *task_struct* (about 82 members). The whole structure of *task_struct* is shown in Chapter 3 of “Linux Kernel Internals”. Here we introduce some of them.

```
struct task_struct {
    int pid;
    struct task_struct *next_task, *prev_task;
    struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
    char comm[16];
    ...
};
```

- **Process ID:** Every process has its unique process ID called pid. We identify the process by this number. (Note that *pid_t* is **typedef**’ed to **int**).

pid_t pid;

- **Process list:** All processes forms a doubly linked list with *next_task* and *prev_task*.
- **Process relationships:** There are ‘family relationships’ between the processes, which are represented by the *p_Xptr* pointers. The simplest one is *p_pptr*, which points to the parent process’s task structure. To enable a process to access all its child processes, *p_cptr* points to the task structure of the “youngest child”, i.e., the last created child. The child processes are linked together as a doubly linked list by *p_ysptr* (next younger sibling) and *p_osptr* (next older sibling). The following figure should clarify the ‘family relationships’ between processes.



- **Miscellaneous:** The name of the program executed by the process is stored in *comm*.

2. The process table

Conceptually, every process occupies exactly one entry in the process table. In earlier versions

of Linux, this is exactly how it is implemented: there is a static array of size `NR_TASKS` that `task_struct` are allocated from.

```
struct task_struct *task[NR_TASKS]
```

This limits the total number of processes to `NR_TASKS`, which is 512 unless you modified the kernel. Newer kernels does not contain a task array, and instead dynamically allocate task structures from physical memory, so this limitation has been removed. Given any process, all the processes currently in a system can present could be traversed by following the doubly-linked list containing all processes. Traditionally, this is done by a macro `for_each_task` defined in `<linux/sched.h>`:

```
#define for_each_task(p) \  
    for(p=&init_task; (p=p->next_task) != &init_task;)
```

Here `init_task` is the first task created during the boot process. Note the strange position in which the `next_task` link is followed: it makes sure that `init_task` itself is not visited.

Many system calls receives a pid and needs to find the task structure with that pid. This can be done by traversing all processes, by something like

```
for_each_task(p) if (p->pid == XXX) break;
```

But this requires time linear to the current number of processes. Newer kernels contains a hash table of task structures, allowing the task structures to be found in nearly constant time by the `find_task_by_pid` function defined in `<linux/sched.h>`:

```
extern __inline__ struct task_struct *find_task_by_pid(int pid) {  
    struct task_struct *p, **htable = &pidhash[pid_hashfn(pid)];  
    for (p = *htable; p && p->pid != pid; p = p->pidhash_next);  
    return p;  
}
```

Finally, the task structure of the currently running task can be obtained via the variable `current`.

3. Accessing the process address space

System calls usually need to read or write to memory buffers provided by the caller, i.e., the user process. Though, this pose a security problem. If the kernel naively uses its full right to read and write the memory buffers, the user can trick the kernel into reading and writing memory that does not belong to the process, thus violating the security barrier. Thus system calls must check whether memory provided by user processes is really owned by the process. Linux has a set of macros defined in `asm/uaccess.h` that perform this automatically.

```
copy_from_user(void* to, void* from, int sizeOfBlocks)
```

Copies a block of arbitrary size from user space. Return 0 on success, and non-zero on failure.

```
copy_to_user(void* to, void* from, int sizeOfBlocks)
```

Copies a block of arbitrary size to user space. Return 0 on success, and non-zero on failure.

4. Modules

Linux is a monolithic kernel; that is, it is one single large program where all the functional components of the kernel have access to all of its internal data structures and routines. This made it rather difficult to add new components into the kernel, since it involves a kernel compilation. To ease this, Linux has a system of kernel modules, which allows you to dynamically load and unload components into the kernel. Linux modules are lumps of code to be dynamically linked into kernel. They can be unlinked from the kernel and removed when they are no longer needed. Mostly kernel modules are device drives, pseudo-device drivers such as network drivers, or file-systems.

You can either explicitly call `insmod` and `rmmmod` commands to load and unload kernel modules, or use the kernel daemon (`kernelld`) to automatically recognize the need for device drivers and load modules accordingly.

A kernel module has at least two functions: *init_module* which is called when the module is inserted into the kernel, and *cleanup_module* which is called just before it is removed. Typically, *init_module* either registers a device or filesystem handler, or replaces one of the system call with its own code. The *cleanup_module* function should undo whatever *init_module* did, so the module can be unloaded safely. The following is a sample program - hello world. You can try it in Linux.

```
/* File: hellomod.c */
/* Should be compiled with
   gcc -c -Wall -I/path/to/kernel/linux hellomod.c
   replacing /path/to/kernel to you actual path to kernel root */

/* These tells the header files that you want to compile a kernel module, and so expose
   the needed macros. */
#ifndef __KERNEL__
#define __KERNEL__
#endif
#ifndef MODULE
#define MODULE
#endif
#include <linux/kernel.h>
/* Must include this, although the real reason is out of scope of this course */
#include <linux/module.h>

/* Initialize the module */
int init_module() {
    printk("Hello, world - this is the kernel speaking\n");
    /* The following tells the kernel that initialization is successful.
       To indicate failure, return a non-zero number instead. */
    return 0;
}

/* Cleanup - undo whatever init_module did */
void cleanup_module() {
    printk("Short is the life of a kernel module\n");
}
```