CSIS0230A Principle of Operating Systems (Class A)
Notes for Tutorial 4
Process Structure

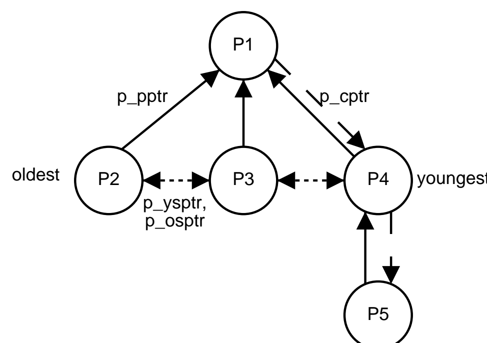
In the last tutorial, you learnt how to add a trivial system call. In this tutorial, you will learn the structure of a Linux task descriptor.

1. Linux Task ("Process") Management

In the lecture, you learn that a process is represented by a process descriptor. In Linux, processes are called tasks, and task descriptors is a structure of type *task_struct*. There are many (more than 80) members in *tast_struct*. It is defined in the kernel header `linux/sched.h`. Here we introduce some of these members.

```
struct task_struct {
    ...
    struct task_struct *next_task, *prev_task;
    ...
    pid_t pid;
    ...
    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, of type *pid_t* (which is **typedef**'ed to **int**). This is the number returned by *fork*() to the parent process when a process is created. Outside the kernel we identify the process by this number, although within the kernel we use the *task_struct*. There is a hash-table of *task_struct* within the kernel, allowing the kernel to quickly lookup a *task_struct* with the *pid* using the *find_task_by_pid* kernel function.
- **Process list:** All processes forms a doubly linked list with *next_task* and *prev_task*.
- **Process relationships:** There are 'family relationships' among 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' among processes.



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

2. Accessing process memory from kernel

System calls usually need to read or write to memory buffers provided by the caller, i.e., the user process. Since the kernel has full right to access any memory, this pose a security problem. If the kernel naively uses read and write the memory buffers directly, the user can trick the kernel into reading and writing memory that does not belong to the process, thus violating the security barrier. (Later we will know that this can be done by reading and writing the "kernel portion" of the address space.)

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.

3. Modules

Linux is a monolithic kernel; that is, every system-related tasks are packed into a large program called the kernel. This made it rather difficult to add new components into the kernal, since it involves recompiling the kernel and rebooting the new kernel. To ease this, Linux has a kernel-module system, 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 from memory when they are no longer needed. Most kernel functionalities can be implemented as modules.

You can either explicitly use `insmod` and `rmmod` commands to load and unload kernel modules, or to configure `module.conf` so that the kernel event daemon (`keventd`) will load the modules accordingly. We will write modules that are loaded and unloaded explicitly.

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* registers a device or filesystem "handler" so that the kernel support one more device or filesystem. It is also possible to replace one of the system call with its own code in *init_module* so that a new system call is added to the kernel. The *cleanup_module* function should undo whatever *init_module* did, so that the module can be safely unloaded. The following is a sample progam—hello world. You can try it in Linux.

```
/* File: hellomod.c */
/* Should be compiled with
    gcc -c -O2 -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");
}
```