

CSIS0230A Principles of Operating Systems (Class A)

Notes for Tutorial 5

Process Structure

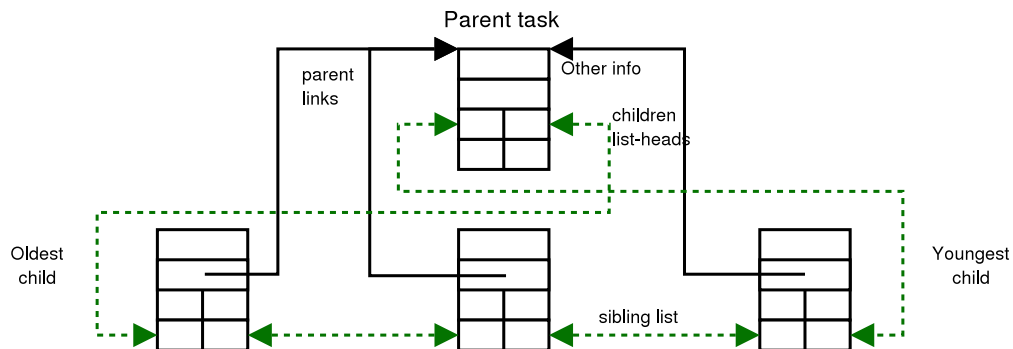
In the lecture, you learnt that a process is represented by a process control block (PCB). In this tutorial, you will learn the structure of a Linux PCB.

1. Linux Task (“Process”) Management

In Linux, processes (more exactly, threads) are called tasks, and PCB is of type **struct** *task_struct*. It is defined in the kernel header *<linux/sched.h>*. Here we introduce some of its members.

```
struct task_struct {  
    ...  
    pid_t pid;  
    char comm[16];  
    struct task_struct *parent;  
    struct list_head children, sibling;  
    ...  
};
```

- **Task ID:** In Linux, each task has an ID, called the task ID. This ID is stored in the *pid* member. For most processes (i.e., those that are not multi-threaded), this is the same as the process ID that we see in the *fork()* and *wait()* system call. Outside the kernel, we identify the task by this number, but to find information relevant to that task in the kernel we must use the *task_struct*. There is a hash-table in the kernel allowing quick lookup of *task_struct*'s given its task ID. The lookup can be performed using the **struct** *task_struct* **find_task_by_pid(pid_t pid)* kernel function. Given a task ID, it returns a pointer to its *task_struct* if it is found, and returns NULL otherwise.
- **Command name:** The first 16 characters of the program executed by the task is stored in the character array *comm*.
- **Task relationships:** There are parent-child relationships among tasks. Each task has a parent, which can be found by the *parent* pointer in the task struct. It is also possible to find the children of a particular task¹. Each task has a “list-head” called *children* that can be used to traverse all the children. All the children of a task form a cyclic doubly linked list using the list-head *sibling*. The cyclic doubly linked list looks like the following...



¹New development kernels (2.5.x) started using a new way to represent the task relations. Although the current stable kernels (2.4.x) are still using the old representation, Redhat modified the kernel it ships so that it uses the new representation. We describe the new representation in this note.

Note that the picture is drawn in a very funny way. While the parent links are simple pointer pointing to the task struct of the parent task, the children and the sibling list-heads are shown in two boxes. A list-head is actually two pointers, *next* and *prev*, used to build a doubly linked list:

```
struct list_head {
    struct list_head *next, *prev;
};
```

This is why we show two boxes instead of one, the box on the left (right) should be interpreted as the *prev* (*next*) pointer.

Perhaps more interestingly, the **arrows do not point to the full task struct**. Instead, they point to **other list-heads**. This is a general characteristic of most Linux complex data structure like linked-list and trees: the full structs (e.g., task struct) will contain small structs (like list-heads) that are used to build the data structure. Only these small structs are involved in the data structure. The full structs can be found from small structs using address manipulations. So in order to find all the children of a task, we can start from *children* list-head, find the *next* pointer of the list-head (which give you another list head) repeatedly until you see the *children* list-head again. Each time you should use address manipulation to subtract the distance between the full task struct and the *sibling* member in order to find the full struct. This is somewhat clumsy, so there are macros to do all these things for you.

2. Handling list-heads

The list-head handling macros are defined in `<linux/list.h>`, which is already included if you use `<linux/sched.h>`. For us, the most important macro is one which allows us to scan through a linked list, and also one for finding a full task struct given a list-head. The first task is done by the *list_for_each* macro, defined like this (simplified a bit to avoid confusion):

```
/* pos and head are pointers to struct list_head */
#define list_for_each(pos, head) \
    for (pos = (head)->next; pos != (head); pos = pos->next)
```

So to traverse the children list, one can do something like

```
/* task is a pointer to struct task_struct */
struct list_head *curr;
list_for_each(curr, &task->children) {
    /* work on curr */
}
```

The other task is done by the macro *list_entry*, defined like this:

```
/* ptr is a list_head, type is the type of the full structure, member is the
particular member containing the list_head */
#define list_entry(ptr, type, member) \
    ((type *)((char *)(ptr)-(unsigned long)((type *)0->member)))
```

So given a pointer *curr* to the *sibling* list-head within a task struct, one can use *list_entry(curr, struct task_struct, sibling)* to get the full task struct.

3. Accessing process virtual memory from kernel

System calls usually need to read and write memory buffers provided by the user process (through pointers to the buffers). Since the kernel can access any memory, this poses a security problem: If the kernel naively read and write the memory buffers directly, the user process can “provide” a buffer which is readable and writable by the kernel, but not by the user process. In this way the user process can trick the kernel into reading and writing memory that does not belong to the process, thus violating 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.

unsigned long *copy_from_user*(**void*** to, **void*** from, **int** sizeOfBlocks)

Copy a block of arbitrary size from the memory of the user process. Return 0 on success, and non-zero on failure. In the latter case the returned number is only guaranteed to be non-zero, its exact value is quite arbitrary.

unsigned long *copy_to_user*(**void*** to, **void*** from, **int** sizeOfBlocks)

Copy a block of arbitrary size to user memory. Return value is similar to *copy_from_user*.

There are two other macros that allow you to conveniently use small amount of process memory, i.e., those which are of 1, 2, 4 or 8 bytes. These functions returns a **long** value, which is 0 on success, and `-EFAULT` on failure.

put_user(*x*, *ptr*)

Copy the content of *x* to the user process memory **ptr*. Apart from the extra checking, this is functionally similar to **ptr = x*.

get_user(*x*, *ptr*)

Copy the content of the user process memory **ptr* to *x*. Apart from the extra checking, this is functionally similar to *x = *ptr*.

All these macros perform their tasks in two steps: first, check whether the “user” memory is owned by the kernel, and if so returns an error; second, perform the copying. The latter copying can fail as well, since invalid memory need not be owned by the kernel (it might just point to somewhere non-existent). In that case the error code is returned as well.

Note: If a user-provided buffer is used many times, it is inefficient to perform the first check every time we access the memory. In this case, you can do the first check at the beginning:

access_ok(*type*, *addr*, *size*)

Check whether the *size* bytes of memory beginning at *addr* has any portion which is owned by the kernel. The *type* is either `VERIFY_READ` or `VERIFY_WRITE` depending on whether read or write access is needed. The function return 0 if any part of the memory lies within the kernel memory (i.e., access is “not okay”), and 1 otherwise (access is “okay”).

Then you can use similar function as above to access the memory, except to add two underscores at the beginning (like `__put_user()`) to notify that first checking is not needed.