Programming Project 3

SWU-OS-计科中外 34 班

Oct 17, 2024

Experimental topic

Programming Projects in Chapter 3

Page P-17 to P-14 of Operating System Concepts (10th)

Experimental report naming format:

StudentID-name-Project-X (e.g., 2220183211-张三-Project-1)

Experimental objectives

1. (Course objective 4) Master the kernel-level module development capability and establish a system view.

Experimental report submission

Please upload your report with naming format to the ftp server within two weeks.

1 Linux Kernel Module for Listing Tasks

In this project, you will write a kernel module that lists all current tasks in a Linux system. You will iterate through the tasks both linearly and depth first.

2 Part I—Iterating over Tasks Linearly

In the Linux kernel, the `for_each_process()` macro easily allows iteration over all current tasks in the system:

```
#include <linux/sched.h>

struct task struct *task;

for each process(task) {
   /* on each iteration task points to the next task */
}
```

The various fields in task struct can then be displayed as the program loops through the `for_each_process()` macro.

2.1 Assignment

Design a kernel module that iterates through all tasks in the system using the `for_each_process()` macro. In particular, output the task command, state, and process id of each task. (You will probably have to read through the task struct structure in `<linux/sched.h>` to obtain the names of these fields.) Write this code in the module entry point so that its contents will appear in the kernel log buffer, which can be viewed using the dmesg command. To verify that your code is working correctly, compare the contents of the kernel log buffer with the output of the following command, which lists all tasks in the system: .. code-block:: bash

    ps -el

The two values should be very similar. Because tasks are dynamic, however, it is possible that a few tasks may appear in one listing but not the other.

3 Part II—Iterating over Tasks with a Depth-First Search Tree

The second portion of this project involves iterating over all tasks in the system using a depth-first search (DFS) tree. (As an example: the DFS iteration of the processes in Figure 3.7 is 1, 8415, 8416, 9298, 9204, 2808, 3028, 3610, 4005.) Linux maintains its process tree as a series of lists. Examining the task struct in `<linux/sched.h>`, we see two struct `list_head` objects:

    children

and

    sibling

These objects are pointers to a list of the task' s children, as well as its siblings. Linux also maintains a reference to the initial task in the system —`init_task` —which is of type `task_struct`. Using this information as well as macro operations on lists, we can iterate over the children of init task as follows:

```
struct task struct *task;
struct list head *list;
list for each(list, &init task->children) {
   task = list entry(list, struct task struct, sibling);
   /* task points to the next child in the list */
}
```

The `llist_for_each()` macro is passed two parameters, both of type struct list head: + A pointer to the head of the list to be traversed + A pointer to the head node of the list to be traversed

At each iteration of `list_for_each()`, the first parameter is set to the list structure of the next child. We then use this value to obtain each structure in the list using the `list_entry()` macro.

3.1 Assignment

Beginning from `init_task` task, design a kernel module that iterates over all tasks in the system using a DFS tree. Just as in the first part of this project, output the name, state, and pid of each task. Perform this iteration in the kernel entry module so that its output appears in the kernel log buffer. If you output all tasks in the system, you may see many more tasks than appear with the `ps -ael` command. This is because some threads appear as children but do not show up as ordinary processes. Therefore, to check the output of the DFS tree, use the command

```
ps -eLf
```

This command lists all tasks—including threads—in the system. To verify that you have indeed performed an appropriate DFS iteration, you will have to examine the relationships among the various tasks output by the `ps` command.

4 补充知识-linux 遍历子进程浅析

其中关于 linux 遍历子进程的叙述如下：

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

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

初次阅读时感觉很不理解，后经细细琢磨，方知原来如此，下面是我个人的理解，以与诸君探讨。

4.1 1、进程亲属关系的成员

```
struct tast_struct {
    /* real parent process */
    struct task_struct *real_parent;
    /* recipient of SIGCHLD, wait4() reports */
    struct task_struct *parent;
    /* list of my children */
    struct list_head children;
    /* linkage in my parent's children list */
    struct list_head sibling;
    /* threadgroup leader */
    struct task_struct *group_leader;
}
```

4.2 2、linux 内核的链表操作

```
/*-----------------------------------------------
 * list_for_each(/include/linux/list.h)
 *-----------------------------------------------*/
#define list_for_each(pos,head) \
for(pos=(head)->next; pos!=head;pos=pos->next)

/*-----------------------------------------------
 * list_entry(/include/linux/list.h)
 *-----------------------------------------------*/
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)

/*-----------------------------------------------
 * container_of(include/linux/kernel.h)
 *-----------------------------------------------*/
#define container_of(ptr, type, member) ({          \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
    (type *)( (char *)__mptr - offsetof(type,member) );})

/*-----------------------------------------------
 * offsetof(/include/linux/stddef.h)
 *-----------------------------------------------*/
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
```

4.3 3、下面来解释 linux 遍历子进程的原理

```
list_for_each(list,&current->children)
```

@list 是指向 struct list_head 类型的指针，故它可以指向进程描述符中的 children 或者 sibling @current 当前正在执行的进程, 先对 list_for_each 宏做如下格式替换

```
#define list_for_each(list, &current->children) \
for(list=(&current->children)->next; list!=&current->children; list=list->next)
```

再对 list_entry 进行宏替换

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

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

可以看到进程中的 children 和 sibling 有潜在的关系，我们再来看他们二者的关系。在 task_struct 结构体中有对成员 sibling 的注释为

```
structlist_head sibling; // linkage in my parent' schildren list
```

直译过来是该指针存放本进程的父进程的子进程的 list. 某网站上有段英文解释:

> In order to organize data as linked list using struct list_head you have to declare list root and declare the same type (struct list_head). children entry of struct task_struct entry is a root. sibling entry of sibling are used. Usage of list_for_each for children means what children is a root. Usage of list_entry for sibling means what sibling is a list entry.

该段的大意是指在一个循环链表中，children 指向链表头，sibling 指向链表的链表项。Linux 进程家族关系图如 Fig. 4.1 :

通过上述解说及图示理解，container_of 定义中，首先将通过 list_for_each 获得的 list_head 类型的指针赋予一个临时定义的变量 __mptr, 存放了该子进程在父进程的子进程 list 中的位置，其实也就是该子进程的 task_struct 结构体中的 sibling 指针，而后通过 offsetof 获得偏移量之后，通过减法关系，即获得了该子进程的 task_struct 的指针。

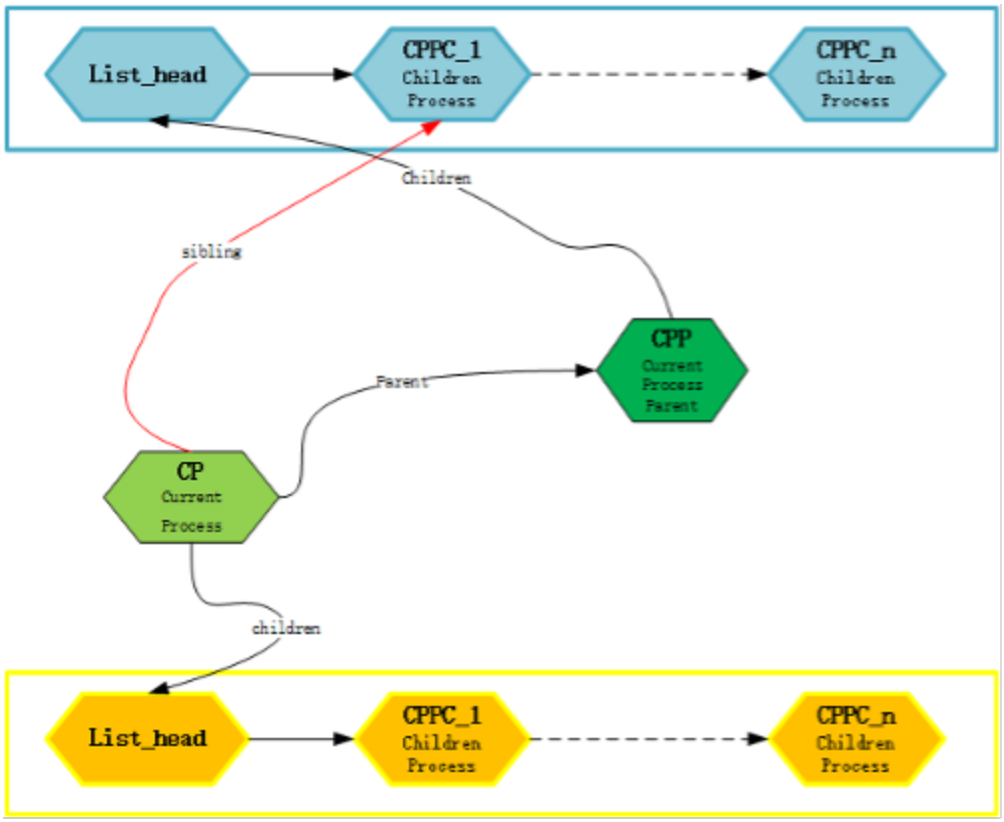

Fig. 4.1: Linux 进程家族关系图

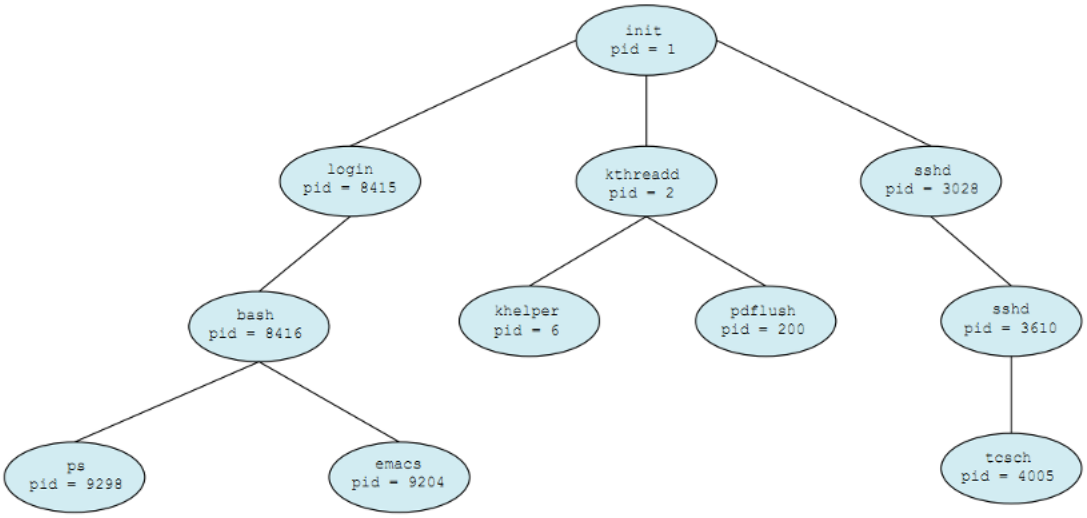

Figure 3.8 A tree of processes on a typical Linux system.