# Lab 2 : Process Management

# 实 验 报 告

刘子涵　518021910690

## 【1】实验要求

1. 为 `task_struct` 结构添加数据成员 `int ctx` ，每当进程被调度一次，`ctx++`。
2. 把 `ctx` 输出到 `/proc/<PID>/ctx` 下，通过 `cat /proc/<PID>/ctx` 可以查看当前指定进程的 `ctx` 的值。

## 【2】实验环境

- Linux 内核版本：5.5.11
- GCC 版本：7.4.0
- 操作系统：Ubuntu 18.04

## 【3】实验思路及具体实现

### 1. 进程管理数据结构

Linux 内核通过进程描述符（Process Descriptor）来管理进程，相应的数据结构是 `task_struct` ，它定义在 `include/linux/sched.h` 中。在第 `673` 行添加数据成员 `ctx`，如下所示：

```
struct task_struct {
    // ...
    int             recent_used_cpu;
    int             wake_cpu;
#endif
    int             ctx;                // Line 673: declare ctx here

    int             on_rq;

    int             prio;
    int             static_prio;
    int             normal_prio;
    unsigned int            rt_priority;
    // ...
}
```

### 2. 进程创建

Linux 内核进程创建实质上是对父进程的复制，进程创建的源码位于 `kernel/fork.c` 中。

- Linux 提供了三个创建进程的系统调用 `clone()`、`fork()`、`vfork()`。分析源码第 `2511` 行到第 `2578` 行，可以发现这三个系统调用均通过 `_do_fork(&args)` 实现。
- 找到函数 `_do_fork()` ，函数头位于第 `2394` 行，注释说这个函数是 main fork-routine ，主要工作是复制进程，核心是第 `2421` 行的 `copy_process()` 函数：

  ```
  p = copy_process(NULL, trace, NUMA_NO_NODE, args); // Line 2421
  ```

- 找到函数 `copy_process()` ，函数头位于第 `1824` 行，注释说这个函数根据父进程创建子进程，但没有真正启动进程，它复制了所有寄存器与运行环境，由调用者来启动进程。在第 `1911` 行的 `dup_task_struct()` 函数复制父进程的进程描述符，`current` 即为指向父进程 `task_struct` 的指针，返回指向子进程 `task_struct` 的指针 `p` 。

  ```
  p = dup_task_struct(current, node); // Line 1911
  ```

- 对子进程 `ctx` 初始化应该是在第 `1911` 行之后，第 `2041` 行开始初始化子进程的调度策略、优先级、调度类等进程调度相关成员，并接着复制父进程的信息（文件、信号、内存等），即 `copy_xxx()` 函数。此处为进程初始化的代码段，将 `ctx` 在此处初始化应该是合理的，所以在第 `2041` 行添加初始化语句如下：

```
static __latent_entropy struct task_struct *copy_process(
    // ...
    p->ctx = 0;         // Line 2041: initialize ctx here

    /* Perform scheduler related setup. Assign this task to a CPU. */
    retval = sched_fork(clone_flags, p);
    if (retval)
        goto bad_fork_cleanup_policy;
}
```

**3. 进程调度**

Linux 内核关于进程调度的源码在 `kernel/sched/core.c` 中，所有的调度都发生在 `schedule()` 函数中。找到 `schedule()` 函数，位于第 `4153` 行。每当进程被调度，这个函数会被执行，获取指向当前进程 `task_struct` 的指针 `tsk` ，然后通过 `preempt_disable()` 关闭内核抢占，然后调用 `__schedule` 函数。因为进程每得到一次调度会执行 `ctx++` 操作，所以直接在 `schedule()` 函数中添加即可。

```
asmlinkage __visible void __sched schedule(void)
{
    struct task_struct *tsk = current;

    tsk->ctx ++;    // Line 4157: increment ctx here

    sched_submit_work(tsk);
    do {
        preempt_disable();
        __schedule(false);
        sched_preempt_enable_no_resched();
    } while (need_resched());
    sched_update_worker(tsk);
}
```

**4. 创建 proc entry**

每个进程都在 `/proc` 下有自己的目录 `/proc/<PID>` ，目录内文件或文件夹的创建源码位于 `fs/proc/base.c` 中。每个进程文件夹下所有文件的静态列表定义在数组 `tgid_base_stuff[]` 中，各元素类型为 `pid_entry` 。在 `/proc/<PID>` 目录下创建一个文件，则需要在这个静态常量数组中增加一项。参照第 `116` 行到第 `149` 行的结构体 `pid_entry` 定义和创建各种类型 `pid_entry` 所定义的宏（`DIR` 创建目录， `LNK` 创建链接， `REG` 和 `ONE` 均可创建文件， `REG` 传入完整的文件操作， `ONE` 可以只有读操作）。此处只需要创建一个可读文件，读取 `ctx` 的值，所以使用 `ONE` 。 将 `ctx` 打印在屏幕

上需要一个函数，参照函数 `proc_pid_personality()` 的实现，定义函数 `proc_pid_ctx()` ，调用 `seq_printf()` 函数将 `task->ctx` 打印在屏幕上。

```
// ...
/* L2993: get task->ctx */
static int proc_pid_ctx(struct seq_file *m, struct pid_namespace *ns,
                struct pid *pid, struct task_struct *task)
{
    int err = lock_trace(task);
    if (!err) {
        seq_printf(m, "%d\n", task->ctx);
        unlock_trace(task);
    }
    return err;
}

// ...

static const struct pid_entry tgid_base_stuff[] = {
    // ...
    ONE("personality", S_IRUSR, proc_pid_personality),
    ONE("limits",     S_IRUGO, proc_pid_limits),

    ONE("ctx",    S_IRUSR, proc_pid_ctx),  // L3026: create /proc/<pid>/ctx
// ...
```

## 【4】实验测试及效果截图

### 1. 重新编译内核

将所有改动过的文件复制替换掉源码目录下 `/usr/src/linux-5.5.11` 对应文件，重新 `make && make install` ，重启计算机。将上述操作写成 `Makefile` ，方便自动化地重新编译内核。代码如下：

```
SRC=/usr/src/linux-$(shell uname -r)

all:
    cp sched.h $(SRC)/include/linux/sched.h
    cp fork.c $(SRC)/kernel/fork.c
    cp core.c $(SRC)/kernel/sched/core.c
    cp base.c $(SRC)/fs/proc/base.c
    make -j2 -C $(SRC)
    make -C $(SRC) install
    reboot
```

### 2. 测试程序

编写一个 `test.c` 程序，循环接收输入。每接收输入一次，进程得到一次调度，对应 `ctx` 加 1 。 `test.c` 如下：

```
#include <stdio.h>

int main() {
    while(1) getchar();
    return 0;
}
```

使用 gcc 编译 `test.c`：

```
gcc test.c -o test
```

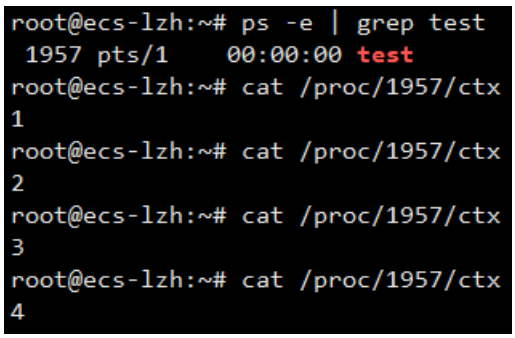
**3. 测试**

① 在终端窗口 A 运行 `test`；

② 在终端窗口 B 查找其 PID；

```
ps -e | grep test
```

③ 在终端窗口 A 中输入字符并回车，然后在终端窗口 B 查看 `/proc/<PID>/ctx` 中的值。反复进行此过程，测试截图如下：




## 【5】实验心得

这次实验基于 Linux 内核进程管理与进程调度理论课，通过实践深入理解 Linux 进程管理的设计思想，对课堂上讲解的数据结构 `task_struct` 有了进一步的理解。这次实验本身修改的代码不多，但需要阅读 Linux 内核的几个源码文件，理解代码的执行顺序，才能在合适的位置插入代码。在 Google 的帮助下，我找到了每个文件的主要数据结构和函数，并根据这些函数所调用的子函数，分析应该在何处插入代码。这一过程中，我提高了我阅读分析大规模系统软件源码的能力和一些 C 语言的编程规范。在修改完代码之后，重新编译需要几个小时的时间，我大概编译了三四次才成功完成实验。总之，这次实验虽然相对简单，但让我收获了很多 Linux 进程管理和调度的知识，提高了我阅读大规模 C 源代码的能力。

---