# 实验四:进程管理（二）

姓名：余清鑫
学号：320220921751

## 实验名称：

进程管理（二）

## 实验目的：

1. 进一步学习进程的属性
2. 学习进程管理的系统调用
3. 掌握使用系统调用获取进程的属性、创建进程、实现进程控制等
4. 掌握进程管理的基本原理

## 实验时间

6 学时

## 实验要求：

1. 编写一个程序，打印进程的如下信息：进程标识符，父进程标识符，真实用户 ID，有效用户 ID，真实用户组 ID，有效用户组 ID。并分析真实用户 ID 和有效用户 ID 的区别。

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    printf("Process ID (PID): %d\n", getpid());
    printf("Parent Process ID (PPID): %d\n", getppid());
    printf("Real User ID (RUID): %d\n", getuid());
    printf("Effective User ID (EUID): %d\n", geteuid());
    printf("Real Group ID (RGID): %d\n", getgid());
    printf("Effective Group ID (EGID): %d\n", getegid());

    return 0;
}
```

```
root@markpen-VMware-Virtual-Platform:/home/markpen/code# gcc -o printf_message printf_message.c
root@markpen-VMware-Virtual-Platform:/home/markpen/code# ls
printf_message  printf_message.c
root@markpen-VMware-Virtual-Platform:/home/markpen/code# ./printf_message
Process ID (PID): 8169
Parent Process ID (PPID): 4047
Real User ID (RUID): 0
Effective User ID (EUID): 0
Real Group ID (RGID): 0
Effective Group ID (EGID): 0
root@markpen-VMware-Virtual-Platform:/home/markpen/code#
```

真实用户 ID（RUID）：进程创建者的用户 ID。

有效用户 ID（EUID）：决定进程的权限，可能由于 setuid 提高权限。

真实组 ID（RGID） 和 有效组 ID（EGID） 作用类似。

2. 阅读如下程序：

```
/*    process using time    */
#include<stdio.h>
#include<stdlib.h>
#include<sys/times.h>
#include<time.h>
#include<unistd.h>
void time_print(char *,clock_t);
int main(void)
{
        clock_t start,end;
        struct tms t_start,t_end;
        start = times(&t_start);
        system(“grep the /usr/doc/*/* > /dev/null 2> /dev/null”);
        end=times(&t_end);

        time_print(“elapsed”,end-start);
        puts(“parent times”);
        time_print(“\tuser CPU”,t_end.tms_utime);
        time_print(“\tsys CPU”,t_end.tms_stime);

        puts(“child times”);
        time_print(“\tuser CPU”,t_end.tms_cutime);
        time_print(“\tsys CPU”,t_end.tms_cstime);

        exit(EXIT_SUCCESS);
}
```

void time_print(char *str, clock_t time)

{

long tps = sysconf(_SC_CLK_TCK);

printf("%s: %6.2f secs\n",str,(float)time/tps);

}

编译并运行，分析进程执行过程的时间消耗（总共消耗的时间和 CPU 消耗的时间），并解释执行结果。再编写一个计算密集型的程序替代 grep，比较两次时间的花销。注释程序主要语句。

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/times.h>
#include <unistd.h>
void time_print(char *label, clock_t time) {
    long tps = sysconf(_SC_CLK_TCK);
    printf("%s: %.2f secs\n", label, (float)time / tps);
}
int main(void) {
    clock_t start, end;
    struct tms t_start, t_end;
    start = times(&t_start);
    system("grep the /usr/share/doc/*/* > /dev/null 2> /dev/null");
    end = times(&t_end);
    time_print("Elapsed", end - start);
    puts("Parent process times:");
    time_print("\tUser CPU", t_end.tms_utime);
    time_print("\tSystem CPU", t_end.tms_stime);
    puts("Child process times:");
    time_print("\tUser CPU", t_end.tms_cutime);
    time_print("\tSystem CPU", t_end.tms_cstime);

    return 0;
}
```

```
root@markpen-VMware-Virtual-Platform:/home/markpen/code# ./time_cost
Elapsed: 2.99 secs
Parent process times:
        User CPU: 0.00 secs
        System CPU: 0.00 secs
Child process times:
        User CPU: 0.00 secs
        System CPU: 0.82 secs
```

```c
int fib(int n) {
    return (n <= 1) ? n : fib(n-1) + fib(n-2);
}

int main(void) {
    clock_t start, end;
    struct tms t_start, t_end;

    start = times(&t_start);
    fib(50);
    end = times(&t_end);

    time_print("Elapsed", end - start);
    puts("Parent process times:");
    time_print("\tUser CPU", t_end.tms_utime);
    time_print("\tSystem CPU", t_end.tms_stime);

    puts("Child process times:");
    time_print("\tUser CPU", t_end.tms_cutime);
    time_print("\tSystem CPU", t_end.tms_cstime);

    return 0;
}
```

```
root@markpen-VMware-Virtual-Platform:/home/markpen/code# vim time_cost_fib.c
root@markpen-VMware-Virtual-Platform:/home/markpen/code# gcc -o time_cost_fib time_cost_fib.c
root@markpen-VMware-Virtual-Platform:/home/markpen/code# ./time_cost_fib
Elapsed: 0.40 secs
Parent process times:
        User CPU: 0.40 secs
        System CPU: 0.00 secs
Child process times:
        User CPU: 0.00 secs
        System CPU: 0.00 secs
```

3. 阅读下列程序：

```c
/*   fork usage   */

#include<unistd.h>

#include<stdio.h>

#include<stdlib.h>

int main(void)

{

    pid_t child;

    if((child=fork())==-1) {

        perror("fork");

        exit(EXIT_FAILURE);

    }else if(child==0){

        puts("in child");

        printf("\tchild pid = %d\n",getpid());
```

```
        printf("\tchild ppid = %d\n",getppid());
        exit(EXIT_SUCCESS);
    }else{
        puts("in parent");
        printf("\tparent pid = %d\n",getpid());
        printf("\tparent ppid = %d\n",getppid());
    }
    exit(EXIT_SUCCESS);
}
```

编译并多次运行，观察执行输出次序，说明次序相同（或不同）的原因；观察进程 ID，分析进程 ID 的分配规律。总结 fork()的使用方法。注释程序主要语句。

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    pid_t child = fork();
    if (child == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (child == 0) { // 子进程
        printf("Child Process:\n");
        printf("\tPID: %d\n", getpid());
        printf("\tParent PID: %d\n", getppid());
        exit(EXIT_SUCCESS);
    } else { // 父进程
        printf("Parent Process:\n");
        printf("\tPID: %d\n", getpid());
        printf("\tParent PID: %d\n", getppid());
        sleep(1); // 等待子进程结束
    }
    return 0;
}
```

```
root@markpen-VMware-Virtual-Platform:/home/markpen/code# vim fork_analyze.c
root@markpen-VMware-Virtual-Platform:/home/markpen/code# gcc -o fork_analyze fork_analyze.c
root@markpen-VMware-Virtual-Platform:/home/markpen/code# ./fork_analyze
Parent Process:
        PID: 8306
        Parent PID: 4047
Child Process:
        PID: 8307
        Parent PID: 8306
```

4. 阅读下列程序：

```
/*  usage of kill,signal,wait  */
#include<unistd.h>
#include<stdio.h>
#include<sys/types.h>
#include<signal.h>
#include<stdlib.h>

int flag;
void stop();
int main(void)
{
    int pid1,pid2;
    signal(3,stop);
    while((pid1=fork()) ==-1);
    if(pid1>0){
        while((pid2=fork()) ==-1);
        if(pid2>0){
            flag=1;
            sleep(5);
            kill(pid1,16);
            kill(pid2,17);
            wait(0);
            wait(0);
            printf("\n parent is killed\n");
            exit(EXIT_SUCCESS);
        }else{
            flag=1;
            signal(17,stop);
            printf("\n child2 is killed by parent\n");
```

```
                exit(EXIT_SUCCESS);
            }
        }else{
            flag=1;
            signal(16,stop);
            printf("\n child1 is killed by parent\n");
            exit(EXIT_SUCCESS);
        }
}

void stop(){
        flag = 0;
}
```

编译并运行，等待或者按^C，分别观察执行结果并分析，注释程序主要语句。

flag 有什么作用？通过实验说明。

```c
int flag;  // 进程的运行标志
// 信号处理函数：当进程收到信号时，将 flag 置为 0
void stop() { flag = 0;}
int main(void) {
    int pid1, pid2;
    // 注册信号 3 的处理函数（但 3 不是标准信号，可能无效）
    signal(3, stop);
    // 创建第一个子进程
    while ((pid1 = fork()) == -1);  // 可能由于资源不足而失败，因此使用 while 循环不断尝试
    if (pid1 > 0) {
        // 父进程
        while ((pid2 = fork()) == -1);  // 创建第二个子进程，继续尝试直到成功
        if (pid2 > 0) {
            // 仍然是父进程
            flag = 1;  // 进程运行时 flag 设为 1
            sleep(5);  // 父进程睡眠 5 秒，确保子进程有足够时间运行
            // 发送信号 16 给第一个子进程
            kill(pid1, 16);
            // 发送信号 17 给第二个子进程
            kill(pid2, 17);
            // 等待两个子进程退出
            wait(0);
            wait(0);
            printf("\n Parent is killed\n");
            exit(EXIT_SUCCESS);
        } else {
            // 第二个子进程
            flag = 1;
            signal(17, stop);  // 绑定信号 17 的处理函数
            printf("\n Child2 is killed by parent\n");
            exit(EXIT_SUCCESS);
        }
    } else {
        // 第一个子进程
        flag = 1;
        signal(16, stop);  // 绑定信号 16 的处理函数
        printf("\n Child1 is killed by parent\n");
        exit(EXIT_SUCCESS);
    }
}
```

flag 标志了进程的运行状态，但在流程控制中没有什么作用，因为 exit(EXIT_SUCCESS);直接终止了进程。

```
root@markpen-VMware-Virtual-Platform:/home/markpen/code# vim sign_and_kill.c
root@markpen-VMware-Virtual-Platform:/home/markpen/code# gcc -o sign_and_kill sign_and_kill.c
root@markpen-VMware-Virtual-Platform:/home/markpen/code# ls
fork_analyze    printf_message    sign_and_kill    time_cost    time_cost_fib
fork_analyze.c  printf_message.c  sign_and_kill.c  time_cost.c  time_cost_fib.c
root@markpen-VMware-Virtual-Platform:/home/markpen/code# ./sign_and_kill
Child 1 running...
Child 2 running...
Child 2 exiting...
Child 1 exiting...
Parent exiting...
root@markpen-VMware-Virtual-Platform:/home/markpen/code#
```

5. 编写程序，要求父进程创建一个子进程，使父进程和个子进程各自在屏幕上输出一些信息，但父进程的信息总在子进程的信息之后出现。（分别通过一个程序和两个程序实现）

单程序实现：

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>

int main() {
    pid_t pid = fork(); // 创建子进程

    if (pid < 0) {
        perror("fork failed");
        exit(1);
    } else if (pid == 0) {
        // 子进程代码
        printf("Child process output (PID: %d)\n", getpid());
        exit(0); // 子进程退出
    } else {
        // 父进程代码
        wait(NULL); // 等待子进程结束
        printf("Parent process output (PID: %d, Child PID: %d)\n", getpid(), pid);
    }

    return 0;
}
```

```
markpen@markpen-VMware-Virtual-Platform:~/code$ vim father_after_child.c
markpen@markpen-VMware-Virtual-Platform:~/code$ gcc -o father_after_child father_after_child.c
markpen@markpen-VMware-Virtual-Platform:~/code$ ./father_after_child
Child process output (PID: 11453)
Parent process output (PID: 11452, Child PID: 11453)
```

多程序实现：

child.c

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
    printf("Child process output (PID: %d)\n", getpid());
    return 0;
}
```

father.c

```c
int main() {
    pid_t pid = fork();

    if (pid < 0) {
        perror("fork failed");
        exit(1);
    } else if (pid == 0) {
        // 子进程执行另一个程序
        execl("./child", "child", NULL);
        perror("execl failed"); // 如果execl失败才会执行到这里
        exit(1);
    } else {
        // 父进程等待子进程结束
        wait(NULL);
        printf("Parent process output (PID: %d, Child PID: %d)\n", getpid(), pid);
    }

    return 0;
}
```

执行结果：

```
markpen@markpen-VMware-Virtual-Platform:~/code$ vim child.c
markpen@markpen-VMware-Virtual-Platform:~/code$ vim father.c
markpen@markpen-VMware-Virtual-Platform:~/code$ gcc - child child.c
gcc: error: -E or -x required when input is from standard input
markpen@markpen-VMware-Virtual-Platform:~/code$ gcc -o child child.c
markpen@markpen-VMware-Virtual-Platform:~/code$ gcc -o father father.c
markpen@markpen-VMware-Virtual-Platform:~/code$ ./father
Child process output (PID: 11513)
Parent process output (PID: 11512, Child PID: 11513)
```

6. 编写程序，要求父进程创建一个子进程，子进程执行 shell 命令 find / -name hda* 的功能，子进程结束时由父进程打印子进程结束的信息。执行中父进程改变子进程的优先级。

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/resource.h>

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        setpriority(PRIO_PROCESS, 0, 10); // 降低优先级
        execlp("find", "find", "/", "-name", "hda*", NULL);
        exit(0);
    } else {
        wait(NULL);
        printf("Child process completed.\n");
    }
    return 0;
}
```

```
root@markpen-VMware-Virtual-Platform:/home/markpen/code# vim find_and_setpriority.c
root@markpen-VMware-Virtual-Platform:/home/markpen/code# gcc -o find_and_setpriority find_and_setpriority.c
root@markpen-VMware-Virtual-Platform:/home/markpen/code# ./find_and_setpriority
/var/lib/swcatalog/icons/ubuntu-oracular-universe/64x64/hdate-applet_gnome-calendar.png
/var/lib/swcatalog/icons/ubuntu-oracular-universe/48x48/hdate-applet_gnome-calendar.png
find: '/run/user/1000/doc': Permission denied
find: '/run/user/1000/gvfs': Permission denied
/usr/lib/modules/6.11.0-19-generic/kernel/drivers/platform/x86/hdaps.ko.zst
/usr/lib/modules/6.11.0-19-generic/kernel/sound/pci/hda
/usr/lib/modules/6.11.0-19-generic/kernel/sound/hda
/usr/lib/modules/6.11.0-9-generic/kernel/drivers/platform/x86/hdaps.ko.zst
/usr/lib/modules/6.11.0-9-generic/kernel/sound/pci/hda
/usr/lib/modules/6.11.0-9-generic/kernel/sound/hda
/usr/lib/firmware/intel/avs/hda-generic-tplg.bin.zst
/usr/lib/firmware/intel/avs/hda-808628xx-3ep-tplg.bin.zst
/usr/lib/firmware/intel/avs/hda-8086-generic-tplg.bin.zst
/usr/lib/firmware/intel/avs/hda-generic-1ep-tplg.bin.zst
/usr/src/linux-headers-6.11.0-9/sound/pci/hda
/usr/src/linux-headers-6.11.0-9/sound/hda
/usr/src/linux-headers-6.11.0-9/include/sound/hdaudio_ext.h
/usr/src/linux-headers-6.11.0-9/include/sound/hda_hwdep.h
/usr/src/linux-headers-6.11.0-9/include/sound/hda_component.h
/usr/src/linux-headers-6.11.0-9/include/sound/hda_register.h
/usr/src/linux-headers-6.11.0-9/include/sound/hda_codec.h
/usr/src/linux-headers-6.11.0-9/include/sound/hda_verbs.h
/usr/src/linux-headers-6.11.0-9/include/sound/hda_i915.h
```

```
/usr/src/linux-headers-6.11.0-9/include/sound/hda_chmap.h
/usr/src/linux-headers-6.11.0-9/include/sound/hda-mlink.h
/usr/src/linux-headers-6.11.0-9/include/sound/hda_regmap.h
/usr/src/linux-headers-6.11.0-19/sound/pci/hda
/usr/src/linux-headers-6.11.0-19/sound/hda
/usr/src/linux-headers-6.11.0-19/include/sound/hdaudio_ext.h
/usr/src/linux-headers-6.11.0-19/include/sound/hda_hwdep.h
/usr/src/linux-headers-6.11.0-19/include/sound/hda_component.h
/usr/src/linux-headers-6.11.0-19/include/sound/hda_register.h
/usr/src/linux-headers-6.11.0-19/include/sound/hda_codec.h
/usr/src/linux-headers-6.11.0-19/include/sound/hda_verbs.h
/usr/src/linux-headers-6.11.0-19/include/sound/hda_i915.h
/usr/src/linux-headers-6.11.0-19/include/sound/hdaudio.h
/usr/src/linux-headers-6.11.0-19/include/sound/hda_chmap.h
/usr/src/linux-headers-6.11.0-19/include/sound/hda-mlink.h
/usr/src/linux-headers-6.11.0-19/include/sound/hda_regmap.h
/usr/share/doc/alsa-base/driver/hda_codec.txt.gz
/usr/share/alsa/topology/hda-dsp
/usr/share/alsa/ucm2/codecs/hda
/usr/share/alsa/ucm2/conf.d/hda-dsp
/usr/share/alsa/ucm2/conf.d/hda-dsp/hda-dsp.conf
/usr/share/alsa/ucm2/Intel/hda-dsp
/usr/share/alsa/ucm2/Intel/hda-dsp/hda-dsp.conf
/usr/share/alsa/init/hda
Child process completed.
root@markpen-VMware-Virtual-Platform:/home/markpen/code#
```

7. 编写程序，要求父进程创建一个子进程，子进程对一个 50*50 的字符数组赋值，由父进程改变子进程的优先级，观察不同优先级进程使用 CPU 的时间。

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>
#define SIZE 50
#define ITERATIONS 100000  // 增加迭代次数，使 CPU 使用时间更长

void assign_array() {
    char array[SIZE][SIZE];
    for (int k = 0; k < ITERATIONS; k++) {  // 让子进程执行大量计算
        for (int i = 0; i < SIZE; i++) {
            for (int j = 0; j < SIZE; j++) {
                array[i][j] = 'A' + (i + j) % 26;
            }
        }
    }
}

int main(int argc, char *argv[]) {
    pid_t pid;
    struct rusage usage;

    pid = fork();

    if (pid < 0) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {  // 子进程
        printf("Child process (PID = %d) started.\n", getpid());
        assign_array();
        printf("Child process (PID = %d) completed array assignment.\n", getpid());
        exit(EXIT_SUCCESS);
    } else {  // 父进程
        printf("Parent process (PID = %d) changing child process priority.\n", getpid());

        // 调整子进程优先级 (默认 nice 值为 0, 我们增加 10)
        int priority = atoi(argv[1]);  // nice 值越大，优先级越低
        if (setpriority(PRIO_PROCESS, pid, priority) == -1) {
            perror("Failed to set priority");
        } else {
            printf("Parent process set child priority to %d.\n", priority);
        }

        wait(NULL);  // 等待子进程完成

        // 获取子进程的 CPU 时间
        if (getrusage(RUSAGE_CHILDREN, &usage) == 0) {
            printf("Child process CPU time used:\n");
            printf("  User CPU time: %ld.%06ld seconds\n", usage.ru_utime.tv_sec, usage.ru_utime.tv_usec);
            printf("  System CPU time: %ld.%06ld seconds\n", usage.ru_stime.tv_sec, usage.ru_stime.tv_usec);
        } else {
            perror("Failed to get resource usage");
        }

        printf("Parent process (PID = %d) detected child termination.\n", getpid());
    }

    return 0;
}
```

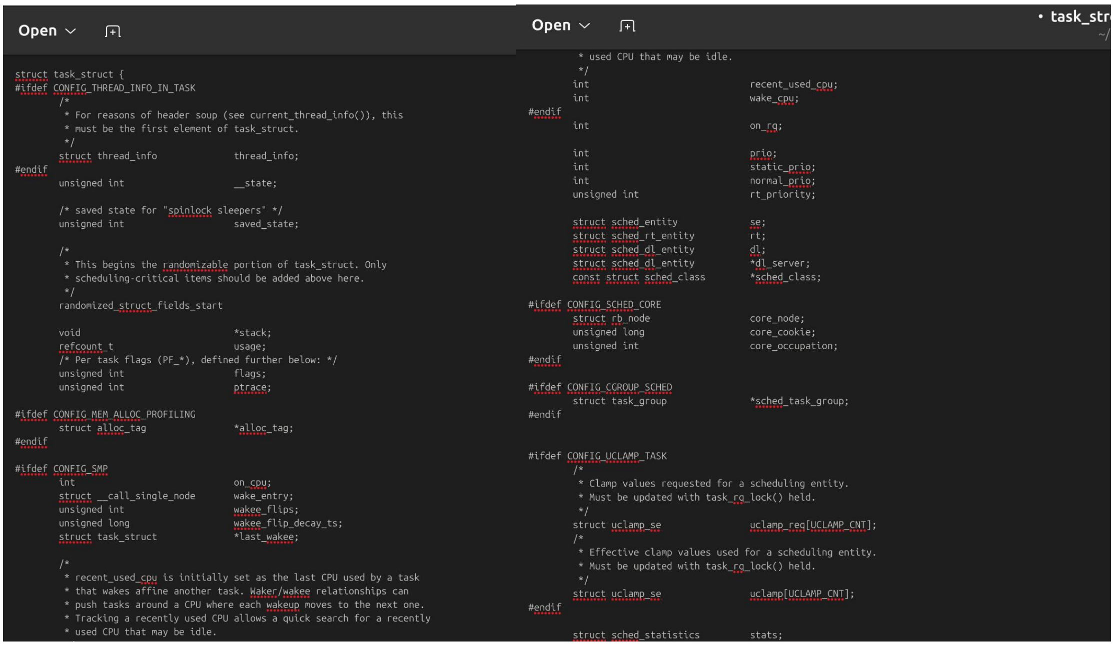
```
root@markpen-VMware-Virtual-Platform:/home/markpen/code# su markpen
markpen@markpen-VMware-Virtual-Platform:~/code$ ./assign_and_setpriority 10
Parent process (PID = 11253) changing child process priority.
Parent process set child priority to 10.
Child process (PID = 11254) started.
Child process (PID = 11254) completed array assignment.
Child process CPU time used:
  User CPU time: 0.709214 seconds
  System CPU time: 0.031661 seconds
Parent process (PID = 11253) detected child termination.
markpen@markpen-VMware-Virtual-Platform:~/code$ sudo ./assign_and_setpriority -10
[sudo] password for markpen:
Parent process (PID = 11264) changing child process priority.
Child process (PID = 11265) started.
Parent process set child priority to -10.
Child process (PID = 11265) completed array assignment.
Child process CPU time used:
  User CPU time: 0.652612 seconds
  System CPU time: 0.074778 seconds
Parent process (PID = 11264) detected child termination.
```

优先级 10（较低优先级）的子进程使用总 CPU 时间为 0.740875 秒，优先级-10（较高优先级）的子进程使用总 CPU 时间为约 0.72739 秒。由此可知，高优先级进程比低优先级进程完成相同工作更快。且主要快在用户态 CPU 时间上，可以推测出 Linux 会优先分配 CPU 资源给高优先级进程，使得它们能更快完成计算任务。

8. 查阅 Linux 系统中 struct task_struct 的定义，说明每项成员的作用。

Linux 中，task_struct 是进程数据块（PCB），包含了进程的信息。关于 task_struct 的定义，位于内核源码目录下的 include/linux/sched.h 文件中

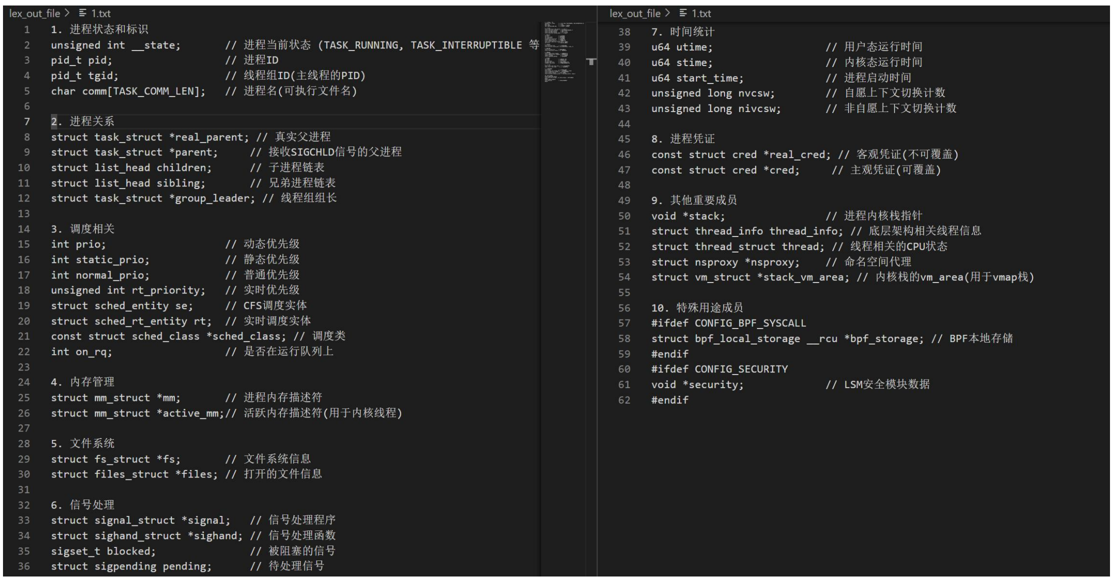
vim /usr/src/linux-headers-$(uname -r)/include/linux/sched.h



```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
	/*
	 * For reasons of header soup (see current_thread_info()), this
	 * must be the first element of task_struct.
	 */
	struct thread_info		thread_info;
#endif
	unsigned int			__state;

	/* saved state for "spinlock sleepers" */
	unsigned int			saved_state;

	/*
	 * This begins the randomizable portion of task_struct. Only
	 * scheduling-critical items should be added above here.
	 */
	randomized_struct_fields_start

	void				*stack;
	refcount_t			usage;
	/* Per task flags (PF_*), defined further below: */
	unsigned int			flags;
	unsigned int			ptrace;

#ifdef CONFIG_MEM_ALLOC_PROFILING
	struct alloc_tag		*alloc_tag;
#endif

#ifdef CONFIG_SMP
	int				on_cpu;
	struct __call_single_node	wake_entry;
	unsigned int			wakee_flips;
	unsigned long			wakee_flip_decay_ts;
	struct task_struct		*last_wakee;

	/*
	 * recent_used_cpu is initially set as the last CPU used by a task
	 * that wakes affine another task. Waker/wakee relationships can
	 * push tasks around a CPU where each wakeup moves to the next one.
	 * Tracking a recently used CPU allows a quick search for a recently
	 * used CPU that may be idle.
```

```
	 * used CPU that may be idle.
	 */
	int				recent_used_cpu;
	int				wake_cpu;
#endif
	int				on_rq;

	int				prio;
	int				static_prio;
	int				normal_prio;
	unsigned int			rt_priority;

	struct sched_entity		se;
	struct sched_rt_entity		rt;
	struct sched_dl_entity		dl;
	struct sched_dl_entity		*dl_server;
	const struct sched_class	*sched_class;

#ifdef CONFIG_SCHED_CORE
	struct rb_node			core_node;
	unsigned long			core_cookie;
	unsigned int			core_occupation;
#endif

#ifdef CONFIG_CGROUP_SCHED
	struct task_group		*sched_task_group;
#endif


#ifdef CONFIG_UCLAMP_TASK
	/*
	 * Clamp values requested for a scheduling entity.
	 * Must be updated with task_rq_lock() held.
	 */
	struct uclamp_se		uclamp_req[UCLAMP_CNT];
	/*
	 * Effective clamp values used for a scheduling entity.
	 * Must be updated with task_rq_lock() held.
	 */
	struct uclamp_se		uclamp[UCLAMP_CNT];
#endif

	struct sched_statistics		stats;
```

分析代码，我们可以得到 task_struct 的主要成员分类：



```
1. 进程状态和标识
unsigned int __state;       // 进程当前状态 (TASK_RUNNING, TASK_INTERRUPTIBLE 等
pid_t pid;                  // 进程ID
pid_t tgid;                 // 线程组ID(主线程的PID)
char comm[TASK_COMM_LEN];   // 进程名(可执行文件名)

2. 进程关系
struct task_struct *real_parent; // 真实父进程
struct task_struct *parent;     // 接收SIGCHLD信号的父进程
struct list_head children;      // 子进程链表
struct list_head sibling;       // 兄弟进程链表
struct task_struct *group_leader; // 线程组组长

3. 调度相关
int prio;                   // 动态优先级
int static_prio;            // 静态优先级
int normal_prio;            // 普通优先级
unsigned int rt_priority;   // 实时优先级
struct sched_entity se;     // CFS调度实体
struct sched_rt_entity rt;  // 实时调度实体
const struct sched_class *sched_class; // 调度类
int on_rq;                  // 是否在运行队列上

4. 内存管理
struct mm_struct *mm;       // 进程内存描述符
struct mm_struct *active_mm;// 活跃内存描述符(用于内核线程)

5. 文件系统
struct fs_struct *fs;       // 文件系统信息
struct files_struct *files; // 打开的文件信息

6. 信号处理
struct signal_struct *signal;   // 信号处理程序
struct sighand_struct *sighand; // 信号处理函数
sigset_t blocked;               // 被阻塞的信号
struct sigpending pending;      // 待处理信号
```

```
7. 时间统计
u64 utime;                  // 用户态运行时间
u64 stime;                  // 内核态运行时间
u64 start_time;             // 进程启动时间
unsigned long nvcsw;        // 自愿上下文切换计数
unsigned long nivcsw;       // 非自愿上下文切换计数

8. 进程凭证
const struct cred *real_cred; // 客观凭证(不可覆盖)
const struct cred *cred;     // 主观凭证(可覆盖)

9. 其他重要成员
void *stack;                // 进程内核栈指针
struct thread_info thread_info; // 底层架构相关线程信息
struct thread_struct thread; // 线程相关的CPU状态
struct nsproxy *nsproxy;    // 命名空间代理
struct vm_struct *stack_vm_area; // 内核栈的vm_area(用于vmap栈)

10. 特殊用途成员
#ifdef CONFIG_BPF_SYSCALL
struct bpf_local_storage __rcu *bpf_storage; // BPF本地存储
#endif
#ifdef CONFIG_SECURITY
void *security;             // LSM安全模块数据
#endif
```