计算机操作系统原理

课程设计

题　　目 进程和线程

学生姓名 _____

专业班级 计算机科学与技术　班

完成时间 _____

指导教师 _____

青岛大学　计算机学院

2019 年 7 月

设计目的

1、加深理解进程和程序、进程和线程的联系与区别；

2、深入理解进程及线程的重要数据结构及实现机制；

3、熟悉进程及线程的创建、执行、阻塞、唤醒、终止等控制方法；

4、学会使用进程及线程开发应用程序。

进程与线程的相关知识点

一、进程与线程的结构

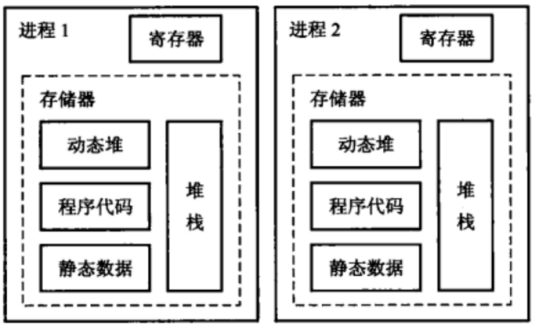

图 2-1　并发进程结构

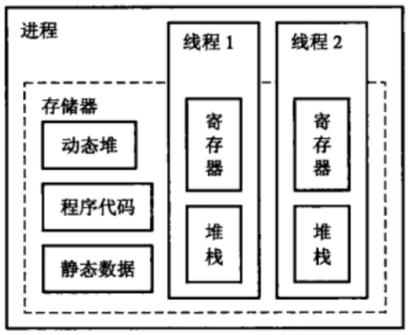

图 2-2　并发线程结构

二、多进程编程

1. Linux 进程控制块 task_struct

```
struct task_struct *task[NR_TASKS] = {&init_task};
#define NR_TASKS 512
 struct task_struct {
        volatile long state;                    /*进程状态*/
        struct thread_info *thread_info;        /*指向进程的 thread_info 指针*/
        atomic_t usage;
        unsigned long flags;                    /*进程标志共 23 种，如正被创建或被信号杀死等*/
        unsigned long ptrace;                   /*跟踪标志*/

        int lock_depth;                         /*锁的深度*/

        int prio, static_prio, normal_prio;     /*动态、静态和正常优先级*/
        struct list_head run_list;              /*链接优先级数组 prio_array 中的元素*/
        struct prio_array *array;               /*当前 CPU 活动的就绪队列*/

        unsigned long sleep_avg;                /*进程的平均等待时间(以毫微秒为单位)*/
        long interactive_credit;                /*进程发生调度事件的时间(以毫微秒为单位)*/
        unsigned long long timestamp, last_ran;
        int activated;                          /*进程进入就绪态的原因*/
```

```
unsigned long policy;                    /*调度策略*/
cpumask_t cpus_allowed;
unsigned int time_slice, first_time_slice;/*时间片余额：0/1 表示是否第 1 次分得时间片*/

struct list_head tasks;                  /*任务队列*/

struct list_head ptrace_children;        /*跟踪进程使用情况*/
struct list_head ptrace_list;
struct mm_struct *mm, *active_mm;        /*进程虚拟主存信息；内核线程借用的地址空间*/
struct Linux_binfmt *binfmt;
int exit_code, exit_signal;              /*退出信号，系统强行退出时发出的信号*/
int pdeath_signal;                       /*父进程死亡时发出该信号*/

pid_t pid;                               /*进程 ID 和组 ID*/
pid_t tgid;                              /*sessionID*/
struct task_struct *real_parent;         /*调试时的真父进程*/
struct task_struct *parent;              /*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;        /*线程组的领导者*/

struct pid_link pids[PIDTYPE_MAX];       /*pid 哈希表链*/
wait_queue_head_t wait_chldexit;         /*wait4( )使用*/
struct completion *vfork_done;           /*vfork( )使用*/
int _ _user *set_child_tid;              /*CLONE_CHILD_SETTID*/
int _ _user *clear_child_tid;            /*CLONE_CHILE_CLEARED*/

struct list_head thread_group;           /*线程队列链*/
unsigned long rt_priority;               /*实时进程优先级*/
unsigned long it_real_value，it_prof_value，it_virt_value;
unsigned long it_real_incr，it_prof_incr，it_virt_incr;
struct timer_list real_timer;
unsigned long utime,stime;
unsigned long nvcsw, nivcsw;             /*上下文切换计数*/
```

```
struct timespec start_time;              /*进程创建时间*/

cputime_t it_prof_expires, it_virt_expires;/*进程的 ITIMER_PROF 和 ITIMER_VIRTUAL 定时器*/
unsigned long long it_sched_expires;
struct list_head cpu_timers[3];

uid_t uid,euid,suid,fsuid; /*用户的标识符、有效标识符、备份标识符、网络环境下文件标识符*/
gid_t gid,egid,sgid,fsgid; /*组的标识符、有效标识符、备份标识符、网络环境下文件标识符*/
struct group_info *group_info;
kernel_cap_t cap_effective, cap_inheritable, cap_permitted;
unsigned keep_capabilities:1;
struct user_struct *user; /*进程定义的用户信息，包括该用户使用的进程数目、打开文件数等*/
struct rlimit rlim[RLIM_NLIMITS];
unsigned short used_math;
char comm[16];
```

```
    struct signal_struct *signal;        /*信号结构，对每种信号，各进程由 signal 属性选择处理函数。
                                          信号的检查在函数结束后或在“慢中断”中断服务程序结束后进行*/
    struct sighand_struct *sighand;
    sigset_t blocked, real_blocked;      /*进程接收信号的位掩码。置位表示屏蔽，复位表示不屏蔽*/
    sigset_t saved_sigmask;              /*与 TIF_RESTORE_SIGMASK 共同恢复*/
    struct sigpending pending;

    unsigned long asa_ss_sp;
    size_t sas_ss_size;
    int (*notifier) (void &priv);
    void *notifier_mask;

    void security;
    struct audit_context *audit_context;
    u32 parent_exec_id;                  /*线程组跟踪*/
    u32 self_exec_id;
```

```
    spinlock_t alloc_lock;               /*分配 mm、files、fs、tty 等的自旋保护锁*/
    spinlock_t proc_lock;                /*保护 proc_dentry*/
    spinlock_t switch_lock;              /*上下文切换的自旋保护锁*/

    void *journal_info;
    struct reclaim_state *reclaim_state;
    struct dentry *proc_dentry;
    struct backing_dev_info *backing_dev_info;

    struct io_context *io_context;
    unsigned long ptrace_message;
    siginfo_t *last_siginfo;             /*跟踪使用情况*/
    wait_queue_t *io_wait;
    …
}
```

2、进程标识符信息

所有进程标识符信息都记录在 task_struct 中，相关函数的原型定义如下：

```
#include <sys/types.h>
#include <unistd.h>
uid_t getpid(void)          /*获取进程 ID*/
uid_t getppid(void)         /*获取父进程 ID*/
pid_t getpgrp(void)         /*获取进程组 ID*/
pid_t getpgid(pid_t pid);   /*获得指定 pid 进程所属组的 ID*/
uid_t getuid(void)          /*获取进程所有者 ID*/
uid_t geteuid(void)         /*获取进程的有效用户 ID*/
git_t getegid(void);        /*获取进程的有效组 ID*/
```

【示例】

```
#include <sys/types.h>

int main(int argc, char **argv)

{

    pid_t    my_pid, parent_pid;
```

```
    uid_t my_uid, my_euid;
    gid_t my_gid, my_egid;

    my_pid = getpid();
    parent_pid = getppid();
    my_uid = getuid();
    my_euid = geteuid();
    my_gid = getgid();
    my_egid = getegid();

    printf("Process ID:%ld\n", my_pid);
    printf("Parent ID:%ld\n", parent_pid);
    printf("User ID:%ld\n", my_uid);
    printf("Effective User ID:%ld\n", my_euid);
    printf("Group ID:%ld\n", my_gid);
    printf("Effective Group ID:%ld\n", my_egid);
}
```

3、创建进程函数 - fork()、vfork()

【示例】理解 fork()函数的功能

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
void main()
{
    int i, p_id;
    if((p_id = fork()) == 0) {
        for(i = 1; i < 3; i++)
            printf("This is child    process\n");
    }
    else if(p_id == -1) {
        printf("fork new process error\n");
        exit(-1);
    }
    else {
        for(i = 1; i < 3; i++)
            printf("This is parent process\n");
    }
}
```

【示例】理解 vfork()函数的功能

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
```

```
{
    int data = 0 ;
    pid_t pid ;
    int choose = 0 ;
    while((choose = getchar( )) != 'q') {
        switch(choose) {
            case '1':
                pid = fork( );
                if(pid < 0 ) {
                    printf("Error !\n");
                }
                if(pid == 0 ) {
                    data++;
                    exit(0);
                }
                wait(pid);
                if(pid > 0 ) {
                    printf("data is %d\n", data);
                }
                break;
            case '2' :
                pid = vfork( );
                if(pid < 0 ) {
                    perror("Error !\n");
                }
                if(pid == 0 ) {
                    data++;
                    exit(0);
                }
                wait(pid);
                if(pid > 0 ) {
                    printf("data is %d\n", data);
                }
                break;
            default :
                break;
        }
    }
}
```

【示例】理解 wait()函数的功能

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
```

```
#include <stdlib.h>
main()
{
        pid_t pc,pr;
        pc=fork();
        if(pc<0)        // 如果出错 */
                printf("error ocurred!\n");
        else if(pc==0){         // 如果是子进程 */
                printf("This is child process with pid of %d\n",getpid());
                sleep(10);      // 睡眠 10 秒钟 */
        }
        else{   // 如果是父进程 */
                pr=wait(NULL);        // 在这里等待 */
                printf("I catched a child process with pid of %d\n",pr);
        }
        exit(0);
}
```

三、多线程编程

1、pthread_create() 和 clone()函数

```
int pthread_create(pthread_t *restrict tidp,const pthread_attr_t *restrict_attr,void*(*start_rtn)(void*),void *restrict arg);
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);
```

2、示例

【示例 1】线程的创建 threadcreatetest.c

```
#include <pthread.h>
#include <stdio.h>
void *create(void *arg)
{
    printf("new thread created ...");
}
int main(int argc, char *argv[])
{
    pthread_t tidp;
    int error;
    error = pthread_create(&tidp, NULL, create, NULL);
    //error=clone(&tidp,NULL,create,NULL);
    printf("error = %d\n", error);
    if(error != 0) {
        printf("pthread_create is not created...");
        return -1;
    }
    printf("pthread_create is created...");
}
```

【说明】:

$ gcc -o tc threadcreatetest.c -lpthread

因为 pthread 库不是 Linux 系统的库， 所以在进行编译的时候要加上-lpthread， 否则编译通不过。

【示例 2】线程同步

pthread 线程库中有许多种同步原语，包括信号量、互斥锁、条件变量和读/写锁。

线程信号量同步原语的函数名都以 sem_开头，而不以 pthread_开头。线程中使用的基本信号量函数有 4 个，其原型定义如下：

```
#include <semaphore.h>
int sem_init(sem_t*sem,int pshared,unsigned int value);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_destroy(sem_t *sem);
```

线程互斥量同步原语，主要解决临界区的使用，线程可以通过相应函数来创建、销毁和操作互斥量。它们的函数原型定义如下：

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex,const pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread__mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

【示例】同步互斥

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

//互斥量，保护工作区及额外变量 time_to_exit //
pthread_mutex_t work_mutex;
#define WORK_SIZE 1024
char work_area[WORK_SIZE]; //工作区
int time_to_exit = 0;
//新线程
void *thread_function(void *arg)
{
    sleep(1);
    pthread_mutex_lock(&work_mutex);
    while(strncmp("end", work_area, 3) != 0) {
        printf("You input %d characters\n", strlen(work_area)-1);
        work_area[0] = '\0';
        pthread_mutex_unlock(&work_mutex);
        sleep(1);
        pthread_mutex_lock(&work_mutex);
        while (work_area[0] == '\0' ) {
            pthread_mutex_unlock(&work_mutex);
```

```
                sleep(1);
                pthread_mutex_lock(&work_mutex);
            }
        }
        time_to_exit = 1;
        work_area[0] = '\0';
        pthread_mutex_unlock(&work_mutex);
        pthread_exit(0);
}

int main( ) {
        int res;
        pthread_t a_thread;
        void *thread_result;
        res = pthread_mutex_init(&work_mutex, NULL); //初始化工作区
        if (res != 0) {
            perror("Mutex initialization failed");
            exit(EXIT_FAILURE);
        }
        res = pthread_create(&a_thread, NULL, thread_function, NULL); //创建新线程
        if (res != 0) {
            perror("Thread creation failed");
            exit(EXIT_FAILURE);
        }
        //主线程
        pthread_mutex_lock(&work_mutex);
        printf("Input some text. Enter 'end' to finish\n");
        while(!time_to_exit) {
            fgets(work_area, WORK_SIZE, stdin);
            pthread_mutex_unlock(&work_mutex);
            while(1) {
                pthread_mutex_lock(&work_mutex);
                if (work_area[0] != '\0') {
                    pthread_mutex_unlock(&work_mutex);
                    sleep(1);
                }
                else {
                    break;
                }
            }
        }
        pthread_mutex_unlock(&work_mutex);
                printf("\nWaiting for thread to finish...\n");
        res = pthread_join(a_thread, &thread_result);
```

```
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("Thread joined\n");
    pthread_mutex_destroy(&work_mutex);
    exit(EXIT_SUCCESS);
}
```

四、设计 – 多线程实现单词统计工具

1、实验说明

设有两个文本文件 file1.txt、file2.txt，统计两个文件中单词的总数。

2、解决方案

区分单词原则：凡是一个非字母或数字的字符跟在字母或数字的后面，那么这个字母或数字就是单词的结尾。

允许线程使用互斥锁来修改临界资源，确保线程间的同步与协作。如果两个线程需要安全地共享一个公共计数器，需要把公共计数器加锁。线程需要访问称为互斥锁的变量，它可以使线程间很好地合作，避免对于资源的访问冲突。

3、程序框架