操作系统实验

201616070320 | 物联网1603 | 郭治洪
使用 **MarkDown** 书写

实验 1 实现一个闹钟

Code:

```
/* 编译选项 gcc -lcurses  time.c -o time*/
//详细参考http://www.runoob.com/cprogramming/c-standard-library-time-h.html
#include <stdio.h>
#include <curses.h>
#include <time.h>
int main(void)
{
    time_t t;
    char time_str[256]={0};
    while(1)
    {
        time(&t);
        //time_t time(time_t *t) 以 time_t 对象返回当前日历时间
        struct tm * p = localtime(&t);
        //struct tm *localtime(const time_t *timer) 该函数返回指向 tm 结构的指针，该结构带有被填充的时间信息
        strftime(time_str,100,"%Y-%m-%d %H:%M:%S",p);
        //size_t strftime(char *str, size_t maxsize, const char *format, const struct tm *timeptr)
        //str 目标数组指针，maxsize 最大字符数，format -- 这是 C 字符串，包含了普通字符和特殊格式说明符的任何组合
        printf("%s\n",time_str);
        sleep(1); //休眠1s
        system("clear"); //清屏函数
    }
    return 0;
}
```

Demo:

```
2018-03-24 19:37:59
```

Compile options:

```
gcc -lcurses  time.c -o time
```

参考链接：

> http://www.runoob.com/cprogramming/c-standard-library-time-h.html

实验 2 用多线程计算Pi

Code:

```
 /*
    Compilation:
    编译选项:
    gcc -lpthread -lm pthread.c -o pthread
*/
//利用投针方法计算PI
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
int circle_count = 0; /* the number of hits in the circle 投进圆里的针数*/
pthread_mutex_t mutex;/* mutex lock to protect circle_count 保护循环计数的互斥锁*/
#define NUMBER_OF_DARTS   50000000 //定义投针数
#define NUMBER_OF_THREADS 2 //定义线程数
double random_double()
{
    /*
    * Generates a double precision random number
    * 生成一个双精度（double）随机数
    * 范围-1.0 到 +1.0
    */
    return random() / ((double)RAND_MAX + 1);
};
void *worker(void *param)
{
    int number_of_darts;//接受由创建线程带来的参数
    number_of_darts = *((int *)param);
    int i;
    int hit_count = 0;
    double x,y;
    for (i = 0; i < number_of_darts; i++)
    {//循环计算
```

```
        /* generate random numbers between -1.0 and +1.0 (exclusive) 随机数范围-1.0 到 +1.0*/
        x = random_double() * 2.0 - 1.0;
        y = random_double() * 2.0 - 1.0;
        if ( sqrt(x*x + y*y) < 1.0 )
            ++hit_count;
    }
    pthread_mutex_lock(&mutex);//互斥锁上锁
    circle_count += hit_count;
    pthread_mutex_unlock(&mutex);//互斥锁解锁
    pthread_exit(0);//线程正常退出
};
int main (int argc, const char * argv[])
{
    int darts_per_thread = NUMBER_OF_DARTS/ NUMBER_OF_THREADS; //每个线程的投针数
    int i;
    double estimated_pi; //计算的PI值
    pthread_t workers[NUMBER_OF_THREADS]; //声明线程ID数组
    pthread_mutex_init(&mutex,NULL);
    //int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restric attr);
    //第一个参数互斥锁的的地址，第二个参数互斥锁的属性，若为NULL则为默认的属性
    srandom((unsigned)time(NULL));    /* seed the random number generator 根据时间生成随机数种子*/
    for (i = 0; i < NUMBER_OF_THREADS; i++)
        pthread_create(&workers[i], 0, worker,&darts_per_thread);
        //int pthread_create(pthread_t *tidp,const pthread_attr_t *attr,(void*)(*start_rtn)(void*),void *arg);
        //第一个参数指向线程ID的指针，第二个参数用来设置线程属性，第三个参数是线程运行函数的起始地址，最后一个参数是运行函数的参数
    for (i = 0; i < NUMBER_OF_THREADS; i++)
        pthread_join(workers[i],NULL);
        //int pthread_join(pthread_t tid, void **status);
        //pthread_join()函数会一直阻塞调用线程，直到指定的线程tid终止。当pthread_join()返回之后，应用程序可回收
    /* estimate Pi  计算PI*/
    estimated_pi = 4.0 * circle_count / NUMBER_OF_DARTS;
    printf("Pi = %f\n",estimated_pi);
    pthread_mutex_destroy(&mutex);//互斥锁释放
    return 0;
}
```

Demo:

```
[root@localhost ~]# ./pthread
Pi = 3.141996
```

Compile options:

```
gcc -lpthread -lm pthread.c -o pthread
```

实验 3 简要 Shell 脚本实现

Code:

> 此程序未完工

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/types.h>
#include <string.h>
#define MAX_LINE 80
/* 80 chars per line, per command, should be enough. 每条指令限制80个字符 */
#define MAX_COMMANDS 5
/* size of history 命令历史显示5条*/
char history[MAX_COMMANDS][MAX_LINE];
char display_history[MAX_COMMANDS][MAX_LINE];
int command_count = 0;
//索引号=命令计数%最大历史命令数
void addtohistory(char inputBuffer[])
{/* Add the most recent command to the history. 将最近的命令添加到历史记录中*/
    int i = 0;
    // add the command to history 将该命令添加到历史记录
    strcpy(history[command_count % MAX_COMMANDS], inputBuffer);
    // add the display-style command to history 将要显示的命令添加到历史中
    // char *strcpy(char *dst, const char *src); src为原字符串 dst为要复制的字符串
    while (inputBuffer[i] != '\n' && inputBuffer[i] != '\0')
    {//循环复制字符串
        display_history[command_count % MAX_COMMANDS][i] = inputBuffer[i]; //这是把输入缓冲数组东西给显示历史数组
        i++;
    }
    display_history[command_count % MAX_COMMANDS][i] = '\0'; //显示历史数组加截止符
    ++command_count;//命令计数+1
    return;
}
/**
* The setup function below will not return any value, but it will just: read
* in the next command line; separate it into distinct arguments (using blanks as
* delimiters), and set the args array entries to point to the beginning of what
* will become null-terminated, C-style strings.
* setup函数除了会读取命令，以空格做参数分隔符，并不会返回任何值，但将args参数数组项设置为C的以空字符结尾字符数组
```

```
*/
int setup(char inputBuffer[], char *args[],int *background)
{
    int length,i, start,ct, command_number;
    //字符长度，访问inputBuffer数组的循环索引，下一个命令参数开始处的位置，在参数args[]数组中索引 ，请求的命令号命令编号
    /* # of characters in the command line */
    /*loop index for accessing inputBuffer array */
    /* index where beginning of next command parameter is */
    /* index of where to place the next parameter into args[] */
    /*index of requested command number */
    ct = 0; //在参数args[]数组中索引先设置0
    /*read what the user enters on the command line */
    do
    {
        printf("osh>");
        fflush(stdout);//将标注输出流立即输出
        length = read(STDIN_FILENO,inputBuffer,MAX_LINE);
        //ssize_t read(int fd, void *buf, size_t count); 返回值：成功返回读取的字节数，出错返回-1并设置errno，如果在调read之前已到达文件末尾，则这次
        //从文件输入流读取最大MAX_LINE到输入缓冲数组中
    }
    while (inputBuffer[0] == '\n'); /* swallow newline characters 以换行符结束*/
    /**
    * 0 is the system predefined file descriptor for stdin (standard input),
    * which is the user's screen in this case. inputBuffer by itself is the
    * same as &inputBuffer[0], i.e. the starting address of where to store
    * the command that is read, and length holds the number of characters
    * read in. inputBuffer is not a null terminated C-string.
    */
    //0是stdin（标准输入）的系统预定义文件描述符，inputBuffer本身与&inputBuffer[0]相同，即存储位置的起始地址，读取返回的是字符串长度，它不是空的C的字符串
    start = -1; //下一个命令参数开始处的索引先设置-1
    if (length == 0)
        exit(0); //命令长度为0退出
    /* ^d was entered, end of user command stream */
    /*** the <control><d> signal interrupted the read system call
    * if the process is in the read() system call, read returns -1
    * However, if this occurs, errno is set to EINTR. We can check this value
    * and disregard the -1 value
    * 在输入过程中如果按下Ctrl+D 中断读取，将会将errno变成EINTER，就可以不用判断没有读取任何字符（read函数返回-1）
    */
    if ( (length < 0) && (errno != EINTR) )
    {
        //处理错误的情况，直接报错退出
        perror("error reading the command");
        exit(-1);
        /* terminate with error code of -1 */
    }
    /* Check if they are using history 检查他们是否使用历史记录*/
    if (inputBuffer[0] == '!')
    {//输入缓冲第一个字符是!
        if (command_count == 0)
        {//不存在历史命令
            printf("No history\n");
            return 1;
        }
        else if (inputBuffer[1] == '!')
        {
            //输入缓冲第二个字符是!
            //即命令是!!
            // restore the previous command 恢复上一个的命令
            strcpy(inputBuffer,history[(command_count -1) % MAX_COMMANDS]); //把上一个命令从命令历史数组复制给输入缓冲数组
            length = strlen(inputBuffer) + 1;
        }
        else if (isdigit(inputBuffer[1]))
        {     //输入缓冲第二个字符是!
            //即命令是!n  n为数字
            /* retrieve the nth command 检索第n个命令 */
            command_number = atoi(&inputBuffer[1]);
            strcpy(inputBuffer,history[command_number]);
            length = strlen(inputBuffer) + 1;
        }
    }
    addtohistory(inputBuffer);/*** Add the command to the history 将命令添加到历史记录中 */
    /*** Parse the contents of inputBuffer 解析inputBuffer的内容 */
    for (i=0;i<length;i++)
    {
        /* examine every character in the inputBuffer 检查inputBuffer中的每个字符*/
        switch (inputBuffer[i])
        {
            case ' ':/这也是空格的一种，下面的'\t'是空格转义符
            case '\t' : /* argument separators 参数分隔符（空格）*/
                if(start != -1)
                {
                    args[ct] = &inputBuffer[start]; /* set up pointer 取值给参数数组*/
                    ct++;//参数数组索引+1
                }
                inputBuffer[i] = '\0'; /* add a null char; make a C string C的数组截止字符 */
                start = -1; //重新下一个命令参数开始处的索引先设置-1
                break; //我觉得这可能是continue，而不是break
            case '\n': /* should be the final char examined 检查最后的字符 */
```

```
                if(start != -1)
                {
                    args[ct] = &inputBuffer[start];  * set up pointer 取值给参数数组*/
                    ct++;//参数数组索引+1
                }
                inputBuffer[i] = '\0';//重新下一个命令参数开始处的索引先设置-1
                args[ct] = NULL; /* no more arguments to this command 该命令没有更多的参数 */
                //NULL在C中应该是0，意味着这个参数数组结束
                break;
            default : /* some other character 其他字符 */
                if (start == -1)
                    start = i;//命令参数开始处的索引更新
                if (inputBuffer[i] == '&')
                {
                    /// command & 在shell中为该程序转到后台运行
                    *background = 1;//这个标志变量变为1
                    inputBuffer[i-1] = '\0';//忽略这个符号，将字符串截止符号加到输入缓冲数组command后面的空格
                }
        } /* end of switch */
    } /* end of for */
    /*** If we get &,don't enter it in the args array 在参数数组中忽略& */
    // command & 在shell中为该程序转到后台运行
    if (*background)         //???感觉有问题
        args[--ct] = NULL;
    args[ct] = NULL; /* no more arguments to this command 该命令没有更多的参数 */
    //NULL在C中应该是0，意味着这个参数数组结束
    /* just in case the input line was > 80 */
    return 1;
} /* end of setup routine */
int main(void)
{
    pid_t child;
    child = fork(); /* creates a duplicate process! 创建副本进程*/
    switch (child)
    {
        //待填
    }
    return 0;
}
```

Demo:

> 此程序未完工

Compile options:

> 此程序未完工

参考链接：

> https://blog.csdn.net/chenxun_2010/article/details/46488055