Lab3-Linux进程控制_lockf()

学号	姓名	专业班级	课程名称	学期	任课教师	完成日期
23020011046	翟一航	软件工程23	操作系统	2025秋	李晓慧	1.5

一、实验目的及要求

- 了解进程与程序的区别，加深对进程概念的理解；
- 进一步认识进程并发执行的原理，理解进程并发执行的特点，区别进程并发执行与顺序执行；
- 分析进程争用临界资源的现象，学习解决进程互斥的方法；
- 学习lockf()函数使用原理和范例,对指定区域进行加锁或解锁，以实现进程的同步或互斥。

二、实验内容与源码

`lockf()` 头文件及调用

```c
# include <unistd.h>

int lockf(int fd, int cmd, off_t len);
```

函数说明

- `lockf()` : apply, test or remove a POSIX lock on an open file
  - 参数 `fd` : 文件描述符
  - 参数 `cmd` : 指定action
    - `F_LOCK` : 加锁
      - If (part of) this section is already locked, the call blocks until the previous lock is released.
      - If this section overlaps an earlier locked section, both are merged.
    - `F_TLOCK` :
      - Same as `F_LOCK` but the call never blocks and returns an error instead if the file is already locked.
    - `F_ULOCK` : 解锁
      - Unlock the indicated section of the file.
      - This may cause a locked section to be split into two locked sections.
    - `F_TEST` : 测试锁
      - return 0 if the specified section is unlocked or locked by this process
      - return -1, set errno to EAGAIN (EACCES on some other systems), if another process holds a lock.
  - 参数 `len` : 从文件当前位置的起始要锁住（解锁）的长度

        - 正数：向前
        - 负数：向后
        - 0：从当前偏移量到最大可能的文件偏移集的部分
    - 返回值：
        - On success, zero is returned.

示例1：

```c
# include<stdio.h>
# include<sys/types.h>
# include<unistd.h>
#include <sys/wait.h>
int main(void)
{
    int pid1,pid2;

    // lockf(1,1,0);  // 加锁
    printf("Parent process:a\n");
    // lockf(1,0,0);  // 解锁

    if((pid1=fork())<0) {
        printf("child1 fail create\n");
        return 1;
    }
    else if(pid1==0) {
        // lockf(1,1,0);  // 子进程1加锁
        printf("This is child1(pid=%d) process:b\n",getpid());
        // lockf(1,0,0);  // 子进程1解锁
        return 0;
    }

    if((pid2=fork())<0) {
        printf("child2 fail create\n");
        return 1;
    }
    else if(pid2==0) {
        // lockf(1,1,0);  // 子进程2加锁
        printf("This is child2(pid=%d) process:c\n",getpid());
        // lockf(1,0,0);  // 子进程2解锁
        return 0;
    }

    // 父进程等待子进程结束
    wait(NULL);
    wait(NULL);
    return 0;
}
```

示例2：

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

int main(){
    int pid, cnt;
        cnt = 10;
    if((pid = fork())<0){
        printf("child fails to create\n");
        return 0;
    }
    else if (pid ==0){
        // lockf(1,1,0);
        for(int i = 0;i < cnt;i++)
            printf("This is child (pid = %d)process:b\n",getpid());
        // lockf(1,0,0);
        return 0;
    }
    else{
        // lockf(1,1,0);
        for(int i = 0;i < cnt;i++)
            printf("Parent process:a\n");
        // lockf(1,0,0);
    return 0;
    }
}
```

Task--练习1

- 运行以上示例程序，观察并分析运行结果

Task--练习2

- 比较 `lockf()` 与 `flock()` 的区别

三、实验结果分析

Task—练习1

示例1

观察程序不难发现，如果不使用 `lockf` 首先一定会先输出 `Parent process:a`，之后可能会先输出 `This is child1(pid=xxx) process:b`，也可能会先输出 `This is child2(pid=yyy) process:c`，这取决于系统对于进程的调度。原因正如我们在实验1中所分析的：

- 父进程先创建child1，然后创建child2

- 两个子进程谁先运行取决于系统调度
- 每个子进程只输出一次就结束
- 输出结果相对简短

如果取消掉注释掉的 `lockf` 函数，我们会发现程序中其实是存在一个关键问题：

- 父进程的 `lockf(1,0,0)` 在最后才解锁，但实际上是在 `fork()` 之后才执行。程序执行流程为：
    - 父进程先锁住标准输出
    - 父进程输出"Parent process:a"
    - 父进程创建子进程1（此时锁状态被继承）
    - 父进程创建子进程2（此时锁状态被继承）
    - 子进程1尝试加锁，但可能被阻塞
    - 子进程2尝试加锁，但可能被阻塞
    - 父进程最后解锁
- 这可能导致死锁或不确定行为

因此，我对示例1的程序作出了一些修改，正如上述源码中展示，将父进程的 `lockf` 提前，这样就避免了死锁问题的产生，同时也能够更好地测试 `lockf` 的应用。此时运行结果如下：

```
Documents/OS/Lab3 via C v17.0.0-clang
❯ for i in {1..10}; do ./test1; echo "--- 运行 $i 结束 ---"; done
Parent process:a
This is child1(pid=83814) process:b
This is child2(pid=83815) process:c
--- 运行 1 结束 ---
Parent process:a
This is child1(pid=83817) process:b
This is child2(pid=83818) process:c
--- 运行 2 结束 ---
Parent process:a
This is child1(pid=83820) process:b
This is child2(pid=83821) process:c
--- 运行 3 结束 ---
Parent process:a
This is child1(pid=83823) process:b
This is child2(pid=83824) process:c
--- 运行 4 结束 ---
Parent process:a
This is child1(pid=83826) process:b
This is child2(pid=83827) process:c
--- 运行 5 结束 ---
Parent process:a
This is child1(pid=83829) process:b
This is child2(pid=83830) process:c
--- 运行 6 结束 ---
Parent process:a
This is child1(pid=83832) process:b
This is child2(pid=83833) process:c
--- 运行 7 结束 ---
Parent process:a
This is child1(pid=83835) process:b
This is child2(pid=83836) process:c
--- 运行 8 结束 ---
Parent process:a
This is child1(pid=83838) process:b
This is child2(pid=83839) process:c
--- 运行 9 结束 ---
Parent process:a
This is child1(pid=83841) process:b
This is child2(pid=83842) process:c
--- 运行 10 结束 ---
❯
```

可以发现，此时每次运行结果固定，总是先调度子进程1再调度子进程2，输出结果固定（pid不同）。这是因为加锁/解锁的操作是原子性的，锁机制强制了进程执行的顺序性，实现了互斥访问——当进程A锁定了文件，其他试图锁定同一文件的进程会被**阻塞**，直到A解锁。

示例2

观察程序可以发现，如果不使用 `lockf` ，那么父进程和子进程并发执行，父进程循环10次输出 `Parent process:a` ，子进程循环10次输出 `This is child...` ，由于没有对进程调度进行控制，所以会出现随机交错的结果，每次运行的输出都不同，父进程与子进程的输出穿插。如下图所示：

```
Documents/OS/Lab3 via C v17.0.0-clang
❯ ./test3
Parent process:a
This is child (pid = 84058)process:b
Parent process:a
This is child (pid = 84058)process:b
Parent process:a
This is child (pid = 84058)process:b
Parent process:a
This is child (pid = 84058)process:b
Parent process:a
This is child (pid = 84058)process:b
Parent process:a
This is child (pid = 84058)process:b
Parent process:a
This is child (pid = 84058)process:b
Parent process:a
This is child (pid = 84058)process:b
Parent process:a
This is child (pid = 84058)process:b
Parent process:a
This is child (pid = 84058)process:b
Parent process:a
This is child (pid = 84058)process:b
```

取消注释，使用 `lockf` ，那么输出结果将固定为父进程先进行10次输出，或者子进程先进行10次输出，中间不会出现交错输出的现象。这是因为父进程与子进程先后创建，OS对于它们调度的先后顺序存在不确定性，但是 `lockf` 写在 `for` 循环的前后，在进入循环之前就会将资源固定，只有当前循环执行完毕，才会将资源释放。运行结果如下：

```
Documents/OS/Lab3 via C v17.0.0-clang
❯ ./test2
Parent process:a
Parent process:a
Parent process:a
Parent process:a
Parent process:a
Parent process:a
Parent process:a
Parent process:a
Parent process:a
Parent process:a
This is child (pid = 84164)process:b
This is child (pid = 84164)process:b
This is child (pid = 84164)process:b
This is child (pid = 84164)process:b
This is child (pid = 84164)process:b
This is child (pid = 84164)process:b
This is child (pid = 84164)process:b
This is child (pid = 84164)process:b
This is child (pid = 84164)process:b
This is child (pid = 84164)process:b
```

Task—练习2

1．基本定义区别

```c
#include <unistd.h>
int lockf(int fd, int cmd, off_t len);

#include <sys/file.h>
int flock(int fd, int operation);
```

`lockf()` 是建议性锁，实际上是对 `fcntl()` 的封装，只能锁定整个文件或文件的一部分区域。

`flock()` 是早期BSD系统的文件锁，可设置为建议性锁或强制性锁，只能锁定整个文件，不能锁定文件区域。

2．主要区别

特性	lockf()	flock()
继承性	子进程不继承锁	子进程继承锁
锁类型	只支持写入锁（排他锁）	支持共享锁和排他锁
进程间	只能用于相关进程间	可用于不相关进程间
线程安全	线程安全	线程不安全
网络文件	支持NFS	某些系统支持NFS
性能	较慢	较快

四、心得总结

在本次Linux进程控制实验中，通过对 `lockf()` 函数的学习和实践，我深刻理解了进程并发执行的特点以及互斥访问的重要性。通过比较有无 `lockf()` 锁机制的程序运行结果，我直观地认识到，不加锁时进程间的执行顺序完全由操作系统调度决定，存在不确定性，而使用 `lockf()` 进行同步控制后，能够有效避免资源竞争，保证临界区的互斥访问，从而实现确定性的输出顺序。

同时，通过与 `flock()` 函数的对比分析，我了解到不同文件锁机制在功能特性、适用范围和性能表现上的差异，这让我明白在实际系统编程中需要根据具体需求选择合适的同步工具。这次实验不仅加深了我对进程并发、同步与互斥等操作系统核心概念的理解，也锻炼了我分析和解决实际并发问题的能力。