西安電子科技大學

XIDIAN UNIVERSITY

操作系统实验

操作系统上机实验报告

陈德创

19030500217

计算机科学与技术专业

指导教师 柴慧敏

June 7, 2021

操作系统上机实验报告

陈德创　　19030500217

西安电子科技大学

日期：2021 年 6 月 7 日

目录

1 进程的建立 **2**

1.1 实验目的 2

1.2 实验内容 2

1.3 实验要求 2

1.4 实验设计与实现 2

1.5 实验结果分析 3

2 进程间的同步 **4**

2.1 实验目的 4

2.2 实验内容 4

2.3 实验要求 4

2.4 实验设计与实现 4

2.5 实验结果分析 7

3 线程共享进程数据 **8**

3.1 实验目的 8

3.2 实验内容 8

3.3 实验设计与实现 8

3.4 实验结果分析 9

1 进程的建立

1.1 实验目的

创建进程及子进程，在父子进程之间实现进程通信。

1.2 实验内容

创建进程并显示标识等进程控制块的属性信息，显示父子进程的通信信息和相应的应答信息。（进程间通信机制任选）

1.3 实验要求

1). 创建进程
2). 显示进程状态信息
3). 实现父子进程通信

1.4 实验设计与实现

实验主要包括两个方面：创建子进程、实现父子进程之间的通信。

创建父子进程可以用 `fork()` 函数实现。此函数将创建一个当前进程的拷贝，包括数据资源（拷贝，但是不共享）。两个程序并行执行。

此函数有返回值，在父进程中返回子进程的 pid，在子进程中返回 0，如果有错误则返回-1。

我们可以通过`getpid()`函数得到当前进程的 pid，`getppid()`得到父进程的 pid，但是没有函数可以令我们得到子进程的 pid。我们利用管道 pipe 进行父子进程之间的通信。函数原型为 `int pipe (int __pipedes[2]);`。此函数会建立管道，并将文件描述词由参数`__pipedes`数组返回。其中：`__pipedes[0]`为管道的读取端，`__pipedes[1]`为管道的写入端。我们可以利用 `unistd.h` 库中的 `write()` 和 `read()` 函数进行读写操作。具体代码如下：

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

int main() {
  int fd[2], result;
  pid_t pid;
  result = pipe(fd);
  if (-1 == result) {
      printf("Error occurs when pipe!\n");
      return -1;
  }
  pid = fork();
  int r = fd[0], w = fd[1];
```

```
17
18      if (pid == -1) {
19          printf("Error occurs when fork!\n");
20          return -1;
21      } else  if (pid == 0) {
22          // son, reads
23          printf("++++The msg is printed by the son!\n");
24          printf("++++Son's pid: %d, Parent's pid: %d\n", getpid(), getppid());
25          close(w);
26          char msg[105];
27          read(r, msg, sizeof(msg) - 1);
28          printf("++++Ths msg from the parent is \"%s\".\n", msg);
29      } else {
30          printf("----The msg is printed by the parent!\n");
31          printf("----Parent's pid: %d, Son's pid: %d\n", getpid(), pid);
32
33          close(r);
34          char msg[] = "Hello, my son!";
35          write(w, msg, strlen(msg));
36          wait(NULL);
37      }
38      return 0;
39  }
```

1.5 实验结果分析

上述代码编译运行后结果如下：

```
dcac@dcac:~/cpp/task1$ ./main
----The msg is printed by the parent!
----Parent's pid: 224939, Son's pid: 224940
++++The msg is printed by the son!
++++Son's pid: 224940, Parent's pid: 224939
++++Ths msg from the parent is "Hello, my son!".
```

可以看出父进程 pid=224939，子进程 pid=224940。在父子进程中输出的结果一致。并且子进程成功输出了来自父进程的内容。

2 进程间的同步

2.1 实验目的

理解进程同步和互斥模型及其应用.

2.2 实验内容

- 利用通信 API 实现进程之间的同步
- 建立司机和售票员进程
- 实现他们间的同步运行

2.3 实验要求

1). 创建进程
2). 实现同步操作或函数
3). 实现公共汽车司机和售票员开关车门及行车运行过程的同步模型
4). 显示同步运行的结果。

2.4 实验设计与实现

实验的重点和难点主要在于“如何完成进程间的同步”，这需要我们在进程之间建立某种通信方式。

首先考虑线程之间的同步方式，我们可以用信号量 (在 C 语言中定义为`sem_t`) 来完成线程之间的互斥和同步。但是在进程中会存在内存不共享的问题，所以我们需要将为信号量分配公共内存，即使得父子进程都可以访问**同一个**信号量。

进程可以通过调用`shmget()`函数来打开或创建一个共享内存区，函数原型为

```
int shmget(key_t key, size_t size, int flag);
```

。其中参数`size`为共享内存区的大小。[1]

在打开一个共享内存区后，可以通过`shmat()`函数将共享内存区连接到进程上，即要把待使用的共享内存映射到进程空间。该函数函数原型为：

```
void * shmat(int shmid, char __user * shmaddr, int shmflg);
```

其中参数`shmid`为共享内存的标识，由函数`shmget()`返回；参数`shmaddr`为映射地址，如果该值为 0，则由内核决定；参数`shmflg`为共享内存的标志。

由此我们得到了一块父子进程可以同时访问的内存区域，我们定义一个`sem_t*`类型的指针指向此内存，就得到了进程共享的信号量。通过对信号量的 P、V 操作 (对应`sem_wait()`和`sem_post()`操作)，我们可以实现进程间的同步和互斥。

为了实现具体功能我们需要用到两个信号量`door`和`car`。我们可以通过`sem_init()`函数对一个信号量进行初始化，函数原型为

```
int sem_init (sem_t *sem, int pshared, unsigned int value);
```

[1]参考自：【Linux】Linux 的共享内存

其中sem为指向要初始化信号量的指针，pshared设为 1 是该信号量可以在进程间通信，否则不行，value为初始值。

每次循环，driver需要等待conductor让出door信号量时才能继续运行，代表车只有等门关闭才能跑；并且在一次循环结束时让出car信号量，代表车停下来了，门可以打开了。

同理，每次循环conductor需要等待driver让出car信号量才能继续运行，代表车停下来门才能打开；并且在一次循环结束时让出door，代表门关闭了，车可以继续运行了。

初始时我们设门是关闭的，车是运行的，即设door初始值为 1，car初始值为 0。

具体实现如下：

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/shm.h>
#include <pthread.h>
#include <stdlib.h>
#include <time.h>
#include <semaphore.h>

struct Node{
    sem_t door, car;
};

sem_t *door, *car;

void driver() {
    int num = 10;
    while (num--) {
        sem_wait(door);
        printf("driver---Car's running...\n");
        sleep(rand() * 2.0 / RAND_MAX); //Car moves on
        printf("driver---Car's stoping...\n");
        sem_post(car);
    }
}

void conductor() {
    int num = 10;
    while (num--) {
        sem_wait(car);
        printf("conductor---Open the door...\n");
        sleep(rand() * 2.0 / RAND_MAX); // wait for passengers to board...
        printf("conductor---Close the door...\n\n\n");
        sem_post(door);
    }
```

```
}

int main() {
    srand(time(NULL));

    int shmid;

    shmid = shmget(312, sizeof(struct Node), IPC_CREAT | 0666);
    struct Node* t = shmat(shmid, NULL, 0);
    door = &t->door, car = &t->car;

    sem_init(door, 1, 1);
    sem_init(car, 1, 0);

    int pid;
    if ((pid = fork()) == -1) {
        printf("Fork Error!\n");
        return -1;
    }
    if (pid == 0) {
        conductor();
    } else {
        driver();
        wait(NULL);
    }

    sem_destroy(door);
    sem_destroy(car);
    return 0;
}
```

2.5 实验结果分析

上述代码编译运行后结果如下：

```
dcac@dcac:~/cpp/task1$ gcc task2_ac.c -o task2 -pthread -Wall
dcac@dcac:~/cpp/task1$ ./task2
driver---Car's running...
driver---Car's stoping...
conductor---Open the door...
conductor---Close the door...


driver---Car's running...
driver---Car's stoping...
conductor---Open the door...
conductor---Close the door...


driver---Car's running...
driver---Car's stoping...
conductor---Open the door...
conductor---Close the door...


driver---Car's running...
driver---Car's stoping...
```

可以看出进程间的同步符合我们的预期：即当车停止后才会开门，并且关门之后车辆才会继续运行。

3 线程共享进程数据

3.1 实验目的

了解线程与进程之间的数据共享关系。创建一个线程，在线程中更改进程中的数据。

3.2 实验内容

在进程中定义全局共享数据，在线程中直接引用该数据进行更改并输出该数据。

3.3 实验设计与实现

由于同一进程中各线程共享内存地址，所以在主线程定义的全局变量子线程可以直接访问(进程中不行)。

我们可以通过pthread_create()函数新建一个线程并运行线程，该函数函数原型为：

```
int pthread_create(pthread_t *restrict tidp, const pthread_attr_t *restrict attr, void *(*start_rtn)(void),void *restrict arg);
```

第一个参数为指向线程标识符的指针，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，第四个参数是运行函数的参数。当创建线程成功时，函数返回 0，若不为 0 则说明创建线程失败。

在具体实现中我们采用了信号量才保证先执行子线程，以便观察结果。

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

sem_t *metux;
int data;

void* work() {
  printf("Before change: %d\n", data);
  data = 10;
  printf("After change: %d\n", data);
  sem_post(metux);
  return NULL;
}

int main() {
  metux = (sem_t*) malloc(sizeof(sem_t));
  memset(metux, 0, sizeof(sem_t));
  sem_init(metux, 0, 0);

```

```
25      pthread_t pt;
26      int result = pthread_create(&pt, NULL, work, NULL);
27      if (result == -1) {
28        printf("Create thread error!\n");
29        return -1;
30      }
31      sem_wait(metux);
32      printf("Now data = %d\n", data);
33      return 0;
34    }
```

3.4 实验结果分析

上述代码编译运行后结果如下：

```
dcac@dcac:~/cpp/task1$ ./task3
Before change: 0
After change: 10
Now data = 10
```

可以发现子线程确实修改了全局变量data，说明同一进程中的全局数据共享。