

操作系统课内实验



姓名 钟臻

班级 软件 82 班

学号 2183612059

日期 2020-12-13

一、进程管理实验

实验目的

- 1) 加深对进程概念的理解，明确进程和程序的区别。
- 2) 进一步认识并发执行的实质。
- 3) 分析进程争用资源的现象，学习解决进程互斥的方法。
- 4) 了解 Linux 系统中进程通信的基本原理。

进程是操作系统中最重要概念，贯穿始终，也是学习现代操作系统的关键。通过本次实验，要求理解进程的实质和进程管理的机制。在 Linux 系统下实现进程从创建到终止的全过程，从中体会进程的创建过程、父进程和子进程的关系、进程状态的变化、进程之间的同步机制、进程调度的原理和以信号和管道为代表的进程间通信方式的实现。

1 编制实现软中断通信的程序

实验内容

使用系统调用 `fork()` 创建两个子进程，再用系统调用 `signal()` 让父进程捕捉键盘上发出的中断信号（即按 `delete` 键），当父进程接收到这两个软中断的某一个后，父进程用系统调用 `kill()` 向两个子进程分别发出整数值为 16 和 17 软中断信号，子进程获得对应软中断信号，然后分别输出下列信息后终止：

Child process 1 is killed by parent !!

Child process 2 is killed by parent !!

父进程调用 `wait()` 函数等待两个子进程终止后，输入以下信息，结束进程执行：

Parent process is killed!!

多运行几次编写的程序，简略分析出现不同结果的原因。

程序清单

头文件：CHILDPROCESS.H

这个头文件定义了一个子进程类，借鉴了 JavaScript 的 `promise` 的经典设计，将子进程包装成类，通过回调函数设置创建成功之后主进程的操作(`then`)、设置创建子进程错误之后主进程的操作(`err`)、设置子进程的操作(`task`)。设置成功之后，使用 `create`，自动创建子进程执行，我使用了一个 `static` 变量来记录是否为主进程，在子进程中，`create` 是不会执行的。有了这个类，撰写之后的代码就变得非常方便了。

```
1. #include <unistd.h>
2. #include <functional>
3.
4. // 子进程类
5. class ChildProcess
6. {
7. private:
8.     static bool _isInFatherProcess;
9.     pid_t pid;
10.     std::function<void()> err = []{};
11.     std::function<void()> then = []{};
12.     std::function<void()> task = []{};
13.
14. public:
```

```

15. static bool isInFatherProcess()
16. {
17.     return ChildProcess::_isInFatherProcess;
18. }
19. pid_t getPid()
20. {
21.     return this->pid;
22. }
23. ChildProcess &setErr(std::function<void()> err)
24. {
25.     this->err = err;
26.     return *this;
27. }
28. ChildProcess &setThen(std::function<void()> then)
29. {
30.     this->then = then;
31.     return *this;
32. }
33. ChildProcess &setTask(std::function<void()> task)
34. {
35.     this->task = task;
36.     return *this;
37. }
38. ChildProcess &create()
39. {
40.     if (_isInFatherProcess)
41.     {
42.         this->pid = fork();
43.         if (pid == -1)
44.         {
45.             err();
46.         }
47.         else if (pid == 0)
48.         {
49.             _isInFatherProcess = false;
50.             task();
51.         }
52.         else
53.         {
54.             then();
55.         }
56.     }
57.     return *this;
58. }

```

```
59. };
60. // 用于判断是否在父进程中
61. bool ChildProcess::_isInFatherProcess = true;
```

CPP 文件：SIGNALTEST.CPP

这个文件，主要是撰写了主体代码，两个子进程注册了 signal，分别响应 16、17，而父进程注册了 signal，响应 SIGINT，也就是 delete 或者 ctrl+c 发送的信号。这里我使用了 c++ 的新特性：lambda 表达式，这个特性可以让你在任意位置创建函数，这样方便使用，可以临时性创建函数。

```
1. #include <signal.h>
2. #include <sys/wait.h>
3. #include <stdio.h>
4. #include <unistd.h>
5. #include <stdlib.h>
6.
7. #include "ChildProcess.h"
8.
9. pid_t pid1, pid2;
10.
11. int main()
12. {
13.     ChildProcess childProcess1 = ChildProcess()
14.         .setThen([&]O {
15.             fprintf(stdout, "Create child process 1 success...\n");
16.         })
17.         .setErr([&]O {
18.             fprintf(stderr, "Create child process 1 error...\n");
19.         })
20.         .setTask([&]O {
21.             signal(16, [](int) {
22.                 fprintf(stdout, "Child process 1 is killed by parent!!\n");
23.                 exit(0);
24.             });
25.             for (;;)
26.                 ;
27.         })
28.         .create();
29.
30.     ChildProcess childProcess2 = ChildProcess()
31.         .setThen([&]O {
32.             fprintf(stdout, "Create child process 2 success...\n");
33.         })
34.         .setErr([&]O {
```

```

35.         fprintf(stderr, "Create child process 2 error...\n");
36.     })
37.     .setTask([&]O {
38.         signal(17, [](int) {
39.             fprintf(stdout, "Child process 2 is killed by parent!!\n");
40.             exit(0);
41.         });
42.         for (;;)
43.             ;
44.     })
45.     .create();
46.
47. if (ChildProcess::isInFatherProcess())
48. {
49.     printf("parentpid=%d\n", getpid());
50.     pid1 = childProcess1.getPid();
51.     pid2 = childProcess2.getPid();
52.
53.     signal(SIGINT, [](int) {
54.         sleep(1);
55.         kill(pid1, 16);
56.         kill(pid2, 17);
57.
58.         wait(0);
59.         wait(0);
60.         fprintf(stdout, "Parent process is killed...\n");
61.         exit(0);
62.     });
63.
64.     while (true)
65.         ;
66. }
67.
68. return 0;
69. }

```

结果：

```

[ucalan@localhost workspace]$ g++ signaltest.cpp -std=c++11
[ucalan@localhost workspace]$ ./a.out
Create child process 1 success...
Create child process 2 success...
parentpid=7747
Child process 1 is killed by parent!!
Child process 2 is killed by parent!!
Parent process is killed...
[ucalan@localhost workspace]$ ./a.out
Create child process 1 success...
Create child process 2 success...
parentpid=7773
Child process 2 is killed by parent!!
Child process 1 is killed by parent!!
Parent process is killed...
[ucalan@localhost workspace]$ █

```

出现了不同结果：有时候 2 设置的回调函数先打印，有时候 1 设置的回调函数先打印。这是由于我们连续使用 kill 发送信号给子进程，子进程几乎是同时收到信号，由于我们同时使用 stdout 输出，那么会出现一个问题，就是两个进程争抢 stdout，有时候 1 先抢到，有时候 2 先抢到，或者我们可以这样说，进程 1 和进程 2 是并行的，并行具有不可预测性。

2 编制实现进程的管道通信的程序

实验内容

使用系统调用 pipe() 建立一条管道线，两个子进程分别向管道写一句话：

Child process 1 is sending a message!

Child process 2 is sending a message!

而父进程则从管道中读出来自于两个子进程的信息，显示在屏幕上。

要求：父进程先接收子进程 P1 发来的消息，然后再接收子进程 P2 发来的消息。

程序清单

头文件：CHILDPROCESS.H

这个头文件定义了一个子进程类，借鉴了 JavaScript 的 promise 的经典设计，将子进程包装成类，通过回调函数设置创建成功之后主进程的操作(then)、设置创建子进程错误之后主进程的操作(err)、设置子进程的操作(task)。设置成功之后，使用 create，自动创建子进程执行，我使用了一个 static 变量来记录是否为主进程，在子进程中，create 是不会执行的。有了这个类，撰写之后的代码就变得非常方便了。

(代码之前已经贴过，此处不占用空间了)

CPP 文件：CHILDPROCESS.H

```
1. #include <signal.h>
2. #include <sys/wait.h>
3. #include <stdio.h>
4. #include <unistd.h>
5. #include <stdlib.h>
6.
7. #include "ChildProcess.h"
8.
9. int main()
10. {
11.     int fd[2];
12.     char pipeBuffer[100];
13.     pipe(fd);
14.
15.     ChildProcess childProcess1 = ChildProcess()
16.         .setThen([&]() {
17.             fprintf(stdout, "Create child process 1 success...\n");
18.         })
19.         .setErr([&]() {
20.             fprintf(stderr, "Create child process 1 error...\n");
21.         })
22.         .setTask([&]() {
23.             lockf(fd[1], 1, 0); //锁定管道
```

```

24.         sprintf(pipeBuffer, "Child process 1 is sending message!\n");
25.
26.         write(fd[1], pipeBuffer, 50); //写入数据
27.         sleep(1);           //等待
28.         lockf(fd[1], 0, 0);   //接触锁定
29.         exit(0);             //结束子进程 1
30.     })
31.     .create();
32. ChildProcess childProcess2 = ChildProcess()
33.     .setThen([&]O {
34.         fprintf(stdout, "Create child process 2 success...\n");
35.     })
36.     .setErr([&]O {
37.         fprintf(stderr, "Create child process 2 error...\n");
38.     })
39.     .setTask([&]O {
40.         lockf(fd[1], 1, 0); //锁定管道
41.         sprintf(pipeBuffer, "Child process 2 is sending message!\n");
42.
43.         write(fd[1], pipeBuffer, 50); //写入数据
44.         sleep(1);           //等待
45.         lockf(fd[1], 0, 0);   //解除锁定
46.         exit(0);             //结束子进程 2
47.     })
48.     .create();
49.
50. if (ChildProcess::isInFatherProcess())
51. {
52.     wait(0);           //等待子进程 1 结束
53.     read(fd[0], pipeBuffer, 50); //读取子进程 1 传递的数据
54.     printf("%s", pipeBuffer); //输出读取到的数据
55.     wait(0);           //等待子进程 2 结束
56.     read(fd[0], pipeBuffer, 50); //读取子进程 2 传递的数据
57.     printf("%s", pipeBuffer); //输出子进程 2 传递的数据
58. }
59. return 0;
60. }

```

如何实现先接收 P1 再接收 P2

我先创建子进程 P1，此时 P1 执行 task 回调函数，这样，P1 锁住了 fd，再创建子进程 P2，此时 P2 由于 fd 被锁住，无法发送数据，只能等 P1 结束后，再占用 fd。

换言之，我使用了 mutex 锁，再加上进程执行的先后顺序，实现了先接收 P1，再接收 P2。

结果：

```
[ucalan@localhost workspace]$ ./a.out
Create child process 1 success...
Create child process 2 success...
Child process 1 is sending message!
Child process 2 is sending message!
```

二、存储器管理实验

实验目的

- 1) 理解内存页面调度的机理
- 2) 掌握几种理论页面置换算法的实现方法
- 3) 了解 HASH 数据结构的使用
- 4) 通过实验比较几种调度算法的性能优劣

页面置换算法是虚拟存储管理实现的关键，通过本次实验理解内存页面调度的机制，在模拟实现 FIFO、LRU、NRU 和 OPT 几种经典页面置换算法的基础上，比较各种页面置换算法的效率及优缺点，从而了解虚拟存储实现的过程。

实验内容

对比以下几种算法的命中率：

- 1) 先进先出算法 FIFO (First In First Out)
- 2) 最近最少使用算法 LRU (Least Recently Used)
- 3) 最近未使用算法 NUR (Never Used Recently)
- 4) 最佳置换算法 OPT (Optimal Replacement)

程序清单

JAVA 文件：CENTER.JAVA

这个文件是承载 main 函数的文件，我们使用这个文件进行调用其他文件的方法。

我是设计如下：

各个算法分别创建一个文件，各自创建一个类，类中只有一个方法，叫做 run，run 方法有两个函数，其一是物理内存能容纳多少页框(pageFrameCount)，其二是调页顺序(pageRoutes)。

由 center.java 调用四个类，并显示结果。

Center.java 设置了内容大小为四个 frame 大小，调页顺序有 50 个调用。

```
1. package jdemo;
2.
3. public class Center {
4.     public static void main(String[] args) {
5.         int[] pageRoutes = { 1, 3, 1, 4, 2, 5, 3, 1, 4, 0, 2, 2, 1, 1, 4, 5, 2, 4, 1, 0, 3, 1, 0, 4, 4, 3, 4, 3, 5, 1,
6.                               0, 0, 4, 2, 0, 5, 1, 0, 2, 0, 3, 5, 1, 3, 1, 3, 2, 3, 4, 2 };
7.         // 1 3 1 4 2 5 3 1 4 0 2 2 1 1 4 5 2 4 1 0 3 1 0 4 4 3 4 3 5 1 0 0 4 2 0 5 1 0 2 0 3 5 1 3 1 3 2 3 4 2
8.         int pageFrameCount = 4;
9.
10.        int lruResult = LRUAlgorithm.run(pageFrameCount, pageRoutes);
11.        int nurResult = NURAlgorithm.run(pageFrameCount, pageRoutes);
12.        int optResult = OPTAlgorithm.run(pageFrameCount, pageRoutes);
```



```

13.     int fifoResult = FIFOAlgorithm.run(pageFrameCount, pageRoutes);
14.     System.out.printf("LRU 算法命中率=%.2f\n", 1- (double)lruResult / pageRoutes.length);
15.     System.out.printf("OPT 算法命中率=%.2f\n", 1- (double)optResult / pageRoutes.length);
16.     System.out.printf("NUR 算法命中率=%.2f\n", 1- (double)nurResult / pageRoutes.length);
17.     System.out.printf("FIFO 算法命中率=%.2f\n", 1- (double)fifoResult / pageRoutes.length);
18. }
19. }

```

JAVA 文件：LRUALGORITHM.JAVA

这个类是 LRU 算法的实现类，在 java 标准库中，有一个类叫 LinkedHashMap，这个类是专门用来实现 LRU 算法的类。设定了最大容量之后，我们往里面增加数据时，会自动将符合 LRU 被替换条件的数据替换。

```

1.  package jdemo;
2.
3.  import java.util.LinkedHashMap;
4.
5.  class LRUCache<K,V> extends LinkedHashMap<K,V>
6.  {
7.      public LRUCache(int maxSize)
8.      {
9.          super(maxSize, 0.75F, true);
10.         maxElements = maxSize;
11.     }
12.
13.     protected boolean removeEldestEntry(java.util.Map.Entry<K,V> eldest)
14.     {
15.         return size() > maxElements;
16.     }
17.
18.     private static final long serialVersionUID = 1L;
19.     protected int maxElements;
20. }
21.
22. public class LRUAlgorithm {
23.     public static int run(int pageFrameCount, int[] pageRoutes) {
24.         LRUCache<Integer, Integer> cache = new LRUCache<Integer, Integer>(pageFrameCount);
25.
26.         int faultCount = 0;
27.         for (int i : pageRoutes) {
28.             if (cache.containsKey(i)) {
29.                 // 在内存中
30.                 cache.put(i, cache.get(i) + 1);
31.             } else {

```

```

32.         // 不在内存中
33.         faultCount++;
34.         cache.put(i, 1);
35.     }
36. }
37. return faultCount;
38. }
39. }

```

JAVA 文件: NURALGORITHM.JAVA

```

1. package demo;
2.
3. import java.util.HashMap;
4.
5. public class NURAlgorithm {
6.     public static int run(int pageFrameCount, int[] pageRoutes) {
7.         HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
8.
9.         int faultCount = 0;
10.        int arr[] = new int[pageFrameCount]; // 记录所有页最近是否被访问, 未被访问对应位置值
        为 0, 否则为 1
11.        int arr1[] = new int[pageFrameCount]; // 记录所有在内存中的页
12.        for (int i = 0; i < pageFrameCount; i++) {
13.            arr[i] = 0; // 赋初值
14.            arr1[i] = 0; // 赋初值
15.        }
16.
17.        int curr = 0; // 记录当前访问位置, 避免一直从一个位置访问
18.        for (int i = 0; i < pageRoutes.length; i++) {
19.            int pageToBeVisited = pageRoutes[i]; // 待访问页
20.            if (!map.containsKey(pageToBeVisited)) {
21.                // 判断不在内存时, 缺页次数加 1
22.                faultCount++;
23.                if (map.size() < pageFrameCount) {
24.                    // 还有空闲帧时
25.                    int temp = map.size();
26.                    map.put(pageToBeVisited, temp); // 将该页写入
27.                    arr[temp] = 1; // 将该页状态设为最近被访问
28.                    arr1[temp] = pageToBeVisited; // 记录内存中的页
29.                } else {
30.                    // 没有空闲帧时
31.                    for (int j = 0; j < pageFrameCount + 1; j++) {
32.                        // 顺序旋转, 最多 n+1 次肯定能找到牺牲帧
33.                        if (arr[curr % pageFrameCount] == 1) {

```

```

34.         // 状态位为 1 时，将状态为置 0
35.         arr[curr % pageFrameCount] = 0;
36.         curr++;
37.     } else {
38.         map.remove(arr1[curr % pageFrameCount]); // 移出找到的最近未访问页
39.         map.put(pageToBeVisited, curr % pageFrameCount); // 将待访问页写入空闲帧
40.         arr1[curr % pageFrameCount] = 1; // 记录页
41.         arr[curr % pageFrameCount] = pageToBeVisited; // 将新写入页状态改为最近被访问
42.         break;
43.     }
44. }
45. }
46. } else {
47.     // 如果该页在内存中
48.     // 找到该页在 map 的位置
49.     int c = map.get(pageToBeVisited);
50.     // 将该页状态设为最近被访问
51.     arr[c] = 1;
52. }
53. }
54. return faultCount;
55. }
56. }

```

JAVA 文件：OPTALGORITHM.JAVA

```

1. package jdemo;
2.
3. import java.util.HashMap;
4.
5. public class OPTAlgorithm {
6.     public static int run(int pageFrameCount, int[] pageRoutes) {
7.         HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
8.         int faultCount = 0;
9.         int j;
10.        for (int i = 0; i < pageRoutes.length; i++) {
11.            // 获取待访问页
12.            int pageToBeVisited = pageRoutes[i];
13.            if (!map.containsValue(pageToBeVisited)) {
14.                // 检索是否在内存中
15.                // 不在时，缺页次数加 1
16.                faultCount++;
17.                if (map.size() < pageFrameCount) {
18.                    // 如果还有空闲帧

```

```

19.         int temp = map.size();
20.         // 使用空闲帧
21.         map.put(temp, pageToBeVisited);
22.     } else {
23.         // 如果没有空闲帧
24.         int index = 0; // 牺牲帧位置
25.         int max = 0; // 已在内存中的页的最晚出现的时间
26.         // 依次查找在内存中的页，下一次出现的时间
27.         for (int t = 0; t < pageFrameCount; t++) {
28.             for (j = i + 1; j < pageRoutes.length; j++) {
29.                 if (pageRoutes[i] == map.get(t)) {
30.                     // 已在内存中的页后续再次出现
31.                     if (j - i > max) {
32.                         // 判断：如果比现在已找到最后出现的页还要更晚出现，变更当前值
33.                         index = t;
34.                         max = j - i;
35.                     }
36.                     break;
37.                 }
38.             }
39.             if (j == pageRoutes.length) {
40.                 // 如果当前页在之后都不会出现，直接将该页设为牺牲帧
41.                 index = t;
42.                 max = j - i;
43.             }
44.         }
45.         map.remove(index); // 移出牺牲帧的内容
46.         map.put(index, pageToBeVisited); // 将所需页读入空闲帧
47.     }
48. }
49. }
50. return faultCount;
51. }
52. }

```

JAVA 文件：FIFOALGORITHM.JAVA

```

1. package demo;
2.
3. import java.util.LinkedList;
4. import java.util.Queue;
5.
6. public class FIFOAlgorithm {
7.     public static int run(int pageFrameCount, int[] pageRoutes) {

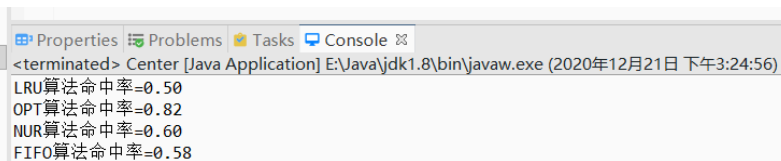
```

```

8.     Queue<Integer> queue = new LinkedList<Integer>();
9.
10.    int faultCount = 0;
11.    for(int i : pageRoutes) {
12.        if (queue.contains(i)) {
13.            // 在内存中
14.            // 假装调页
15.        } else if (queue.size() < pageFrameCount) {
16.            // 不在内存中，内存未装满
17.            faultCount++;
18.            queue.offer(i);
19.        } else {
20.            // 内存装满
21.            faultCount++;
22.            queue.poll();
23.            queue.offer(i);
24.        }
25.    }
26.    return faultCount;
27. }
28. }

```

结果：



```

<terminated> Center [Java Application] E:\Java\jdk1.8\bin\javaw.exe (2020年12月21日 下午3:24:56)
LRU算法命中率=0.50
OPT算法命中率=0.82
NUR算法命中率=0.60
FIFO算法命中率=0.58

```

结论

LRU、OPT、NUR 各有优劣，在各自适合的场景中，能做到最好的命中率。OPT 算法是理论上最好的算法，他的命中率是最高的，但是现实中完全无法实现。

