《操作系统原理》实验报告

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 姓名 |  | 学号 |  | 专业班级 |  | 时间 | 2023. |

**一、实验目的**

1. 理解进程/线程的概念和应用编程过程；
2. 理解进程/线程的同步机制和应用编程；

**二、实验内容**

1. 在Linux下创建2个线程A和B，循环输出数据或字符串。
2. 在Linux下创建（fork）一个子进程，实验wait/exit函数。
3. 在Linux下利用线程实现“生产者-消费者”同步控制。
4. 在Linux下模拟哲学家就餐，提供死锁和非死锁解法。
5. 研读Linux内核并用printk调试进程创建和调度策略的相关信息。

**三、实验环境和核心代码**

**3.1 Linux线程创建**

实验环境：

操作系统：archlinnux 6.6.1

编译器：GNU gcc 13.2.1

定义线程A与线程B的过程函数

1. **static** **void** \*thread1\_proc(**void** \*arg)
2. {
3. **for** (**int** i = 1; i <= 1000; i++)
4. {
5. printf("A: %d\n", i);
6. usleep(10000);
7. }
8. **return** NULL;
9. }
11. **static** **void** \*thread2\_proc(**void** \*arg)
12. {
13. **for** (**int** i = 1000; i > 0; i--)
14. {
15. printf("B: %d\n", i);
16. usleep(10000);
17. }
18. **return** NULL;
19. }

线程A依次递增输出1-1000，线程B依次递减输出1000-1，输出时每隔10ms输出一个数。

主函数使用pthread\_create创建线程A和线程B，使用pthread\_join等待线程A与线程B的结束。

1. **int** main()
2. {
3. pthread\_t thread1;
4. pthread\_t thread2;
5. // 创建A线程
6. pthread\_create(&thread1, NULL, thread1\_proc, NULL);
7. // 创建B线程
8. pthread\_create(&thread2, NULL, thread2\_proc, NULL);
9. // 等待进程A结束
10. pthread\_join(thread1, NULL);
11. // 等待进程B结束
12. pthread\_join(thread2, NULL);
13. **return** 0;
14. }

**3.2 Linux进程创建**

程序1：父进程不使用wait函数，使父进程先于子进程结束，父进程和子进程中输出对应的进程ID。

1. **int** main()
2. {
3. pid\_t pid = fork();
4. **if** (pid == 0)
5. {
6. printf("[child process]\n");
7. pid\_t parent\_pid = getppid();
8. pid\_t self\_pid = getpid();
9. printf("[child process] parent pid: %d\n", parent\_pid);
10. printf("[child process] self pid : %d\n", self\_pid);
11. **for** (**int** i = 1; i < 10; i++) **for** (**int** j = 1; j > 0; j++);
12. printf("[child process] exit");
13. }
14. **else** **if** (pid > 0)
15. {
16. printf("[parent process]\n");
17. pid\_t self\_pid = getpid();
18. printf("[parent process] self pid : %d\n", self\_pid);
19. printf("[parent process] child pid : %d\n", pid);
20. **for** (**int** i = 1; i < 5; i++) **for** (**int** j = 1; j > 0; j++);
21. printf("[parent process] exit\n");
22. }
23. **else**
24. {
25. printf("fork failed!\n");
26. }
27. **return** 0;
28. }

程序2：父进程使用wait函数，子进程休眠5s后使用exit退出，父进程获取子进程返回值。

1. **int** main()
2. {
3. pid\_t pid = fork();
4. **if** (pid == 0)
5. {
6. printf("[child process]\n");
7. pid\_t parent\_pid = getppid();
8. pid\_t self\_pid = getpid();
9. printf("[child process] parent pid: %d\n", parent\_pid);
10. printf("[child process] self pid : %d\n", self\_pid);
11. sleep(5);
12. printf("[child process] exit\n");
13. exit(111);
14. }
15. **else** **if** (pid > 0)
16. {
17. printf("[parent process]\n");
18. pid\_t self\_pid = getpid();
19. printf("[parent process] self pid : %d\n", self\_pid);
20. printf("[parent process] child pid : %d\n", pid);
21. **int** stat;
22. waitpid(pid, &stat, 0);
23. **int** exit\_status = WEXITSTATUS(stat);
24. printf("[parent process] child process %d die, with status %d\n", pid, exit\_status);
25. printf("[parent process] exit\n");
26. }
27. **else**
28. {
29. printf("fork failed!\n");
30. }
31. **return** 0;
32. }

**3.3 “生产者-消费者”同步控制**

使用pthread创建线程模拟生产者和消费者，使用semaphore信号量控制产品缓冲区数量buf\_num\_sem和产品缓冲区剩余容量buf\_space\_sem，初始缓冲区数量为0，缓冲区容量为10，缓冲区同时只能由一个生产者或消费者访问，因此使用pthread\_mutex互斥量控制缓冲区的访问buf\_mutex。

对于生产者线程，首先等待缓冲区剩余容量信号量，即P(buf\_space\_sem)，当缓冲区有容量时，对缓冲区上锁，即lock(buf\_mutex)，然后向缓冲区存入数据，增加缓冲区数量，即V(buf\_num\_sem)，最后解除缓冲区锁，即unlock(buf\_muxtex)。

生产者线程代码如下：

1. **static** **void** \*thread\_producer(**void** \*arg)
2. {
3. **int** producer\_id = \*(**int** \*)arg;
4. printf("[producer %d] start\n", producer\_id);
5. **for** (**int** i = 0; i < 30; i++)
6. {
7. **int** sleep\_time = ((rand() % 50) + 1) \* 1000;
8. **int** product = (rand() % 1000) + producer\_id \* 1000;
9. usleep(sleep\_time);
10. // 需要缓冲区有空间
11. sem\_wait(&buf\_space\_sem);
12. // 给缓冲区上锁
13. pthread\_mutex\_lock(&buf\_mutex);
15. // 添加产品到缓冲区
16. product\_buf[product\_num] = product;
17. product\_num++;
18. printf("\n");
19. printf("[producer %d] put %d to buf\n", producer\_id, product);
20. printf("[producer %d] product num in buf : %d\n", producer\_id, product\_num);
22. // 释放缓冲区锁
23. pthread\_mutex\_unlock(&buf\_mutex);
24. // 增加缓冲区产品数量
25. sem\_post(&buf\_num\_sem);
26. }
27. **return** NULL;
28. }

对于消费者线程，首先需要等待缓冲区数量信号量，即P(buf\_num\_sem)，当缓冲区中有数据时，对缓冲区上锁，即lock(buf\_mutex)，然后从缓冲区取数据，增加缓冲区空闲空间，即V(buf\_space\_sem)，最后解除缓冲区锁，即unlock(buf\_muxtex)。

消费者线程代码如下：

1. **static** **void** \*thread\_consumer(**void** \*arg)
2. {
3. **int** consumer\_id = \*(**int** \*)arg;
4. printf("[consumer %d] start\n", consumer\_id);
5. **for** (**int** i = 0; i < 20; i++)
6. {
7. **int** sleep\_time = ((rand() % 100) + 1) \* 1000;
8. usleep(sleep\_time);
9. // 需要缓冲区有产品
10. sem\_wait(&buf\_num\_sem);
11. // 给缓冲区上锁
12. pthread\_mutex\_lock(&buf\_mutex);
14. // 从缓冲区取产品
15. product\_num--;
16. **int** product = product\_buf[product\_num];
17. product\_buf[product\_num] = 0;
18. printf("\n");
19. printf("[consumer %d] get %d from buf\n", consumer\_id, product);
20. printf("[consumer %d] product num in buf : %d\n", consumer\_id, product\_num);
22. // 释放缓冲区锁
23. pthread\_mutex\_unlock(&buf\_mutex);
24. // 释放缓冲区空间
25. sem\_post(&buf\_space\_sem);
26. }
27. **return** NULL;
28. }

**3.4 “哲学家就餐”模拟**

在哲学家就餐问题中，共5个哲学家与5根筷子，每一根筷子都是一个互斥资源，同时只能由一个哲学家持有，使用pthread\_mutex模拟每一根筷子，哲学家就餐时，先取左侧筷子，即对左侧筷子上锁，然后取右侧筷子，对右侧筷子上锁，完成就餐后，释放左右筷子。

使用pthread创建线程模拟哲学家，代码如下：

1. **static** **void** \*philospher(**void** \*arg)
2. {
3. **int** id = \*(**int** \*)arg;
4. **int** l\_chopstick = id;
5. **int** r\_chopstick = (id + 1) % NUM;
6. printf("[philospher %d] thread start\n", id);
7. **for** (**int** i = 0; i < 50; i++)
8. {
9. // 思考
10. **int** think\_time = ((rand() % 400) + 101) \* 1000;
11. printf("[philospher %d] thinking\n", id);
12. usleep(think\_time);
13. // 休息
14. **int** sleep\_time = ((rand() % 400) + 101) \* 1000;
15. printf("[philospher %d] sleeping\n", id);
16. usleep(sleep\_time);
18. // 拿起左侧筷子
19. pthread\_mutex\_lock(&chopstick[l\_chopstick]);
20. printf("[philospher %d] pick up left chopstick %d\n", id, l\_chopstick);
21. // 拿起右侧筷子
22. pthread\_mutex\_lock(&chopstick[r\_chopstick]);
23. printf("[philospher %d] pick up right chopstick %d\n", id, r\_chopstick);
24. // 吃饭
25. **int** eat\_time = ((rand() % 400) + 101) \* 1000;
26. usleep(eat\_time);
27. printf("[philospher %d] eatting\n", id);
28. // 放下左侧筷子
29. pthread\_mutex\_unlock(&chopstick[l\_chopstick]);
30. printf("[philospher %d] put down left chopstick %d\n", id, l\_chopstick);
31. // 放下右侧筷子
32. pthread\_mutex\_unlock(&chopstick[r\_chopstick]);
33. printf("[philospher %d] put down right chopstick %d\n", id, r\_chopstick);
34. }
35. **return** NULL;
36. }

在上述思路中，当每个哲学家都拿起左侧筷子时，由于右侧筷子均被另一个哲学家持有，会导致所有的哲学家进入等待状态，产生死锁，解决死锁可以使哲学家同时拿取左右的筷子，先对左侧筷子上锁，再尝试对右侧筷子上锁，当无法对右侧筷子上锁时，主动解除左侧筷子的锁，使用该操作顺序，即可避免死锁的产生。

无死锁的哲学家线程代码如下：

1. **static** **void** \*philospher(**void** \*arg)
2. {
3. **int** id = \*(**int** \*)arg;
4. **int** l\_chopstick = id;
5. **int** r\_chopstick = (id + 1) % NUM;
6. printf("[philospher %d] thread start\n", id);
7. **for** (**int** i = 0; i < 50; i++)
8. {
9. // 思考
10. **int** think\_time = ((rand() % 400) + 101) \* 1000;
11. printf("[philospher %d] thinking\n", id);
12. usleep(think\_time);
13. // 休息
14. **int** sleep\_time = ((rand() % 400) + 101) \* 1000;
15. printf("[philospher %d] sleeping\n", id);
16. usleep(sleep\_time);
18. **while** (1)
19. {
20. // 尝试拿起左侧筷子 如果不可用则等待
21. pthread\_mutex\_lock(&chopstick[l\_chopstick]);
22. // 尝试拿起右侧筷子
23. **int** try\_result = pthread\_mutex\_trylock(&chopstick[r\_chopstick]);
24. **if** (try\_result == 0)
25. {
26. // 当两根筷子都能拿时进入就餐
27. printf("[philospher %d] pick up left chopstick %d\n", id, l\_chopstick);
28. printf("[philospher %d] pick up right chopstick %d\n", id, r\_chopstick);
29. // 吃饭
30. **int** eat\_time = ((rand() % 400) + 101) \* 1000;
31. usleep(eat\_time);
32. printf("[philospher %d] eatting\n", id);
33. // 放下左侧筷子
34. pthread\_mutex\_unlock(&chopstick[l\_chopstick]);
35. printf("[philospher %d] put down left chopstick %d\n", id, l\_chopstick);
36. // 放下右侧筷子
37. pthread\_mutex\_unlock(&chopstick[r\_chopstick]);
38. printf("[philospher %d] put down right chopstick %d\n", id, r\_chopstick);
39. **break**;
40. }
41. **else**
42. {
43. // 不能同时拿到两根筷子则不拿左侧筷子
44. pthread\_mutex\_unlock(&chopstick[l\_chopstick]);
45. }
46. }
47. }
48. **return** NULL;
49. }

**3.5 Linux进程创建与调度策略**

为了方便调试内核，先增加系统调用my\_dbg

1. 550     64      my\_dbg              sys\_my\_dbg
2. #define DBG\_NONE 0
3. #define DBG\_FORK 1
5. **extern** **int** noxke\_global\_dbg;
6. **extern** **int** global\_var[10];
8. asmlinkage **long** sys\_my\_dbg(**int** flag);
9. **int** noxke\_global\_dbg = DBG\_NONE;
11. SYSCALL\_DEFINE1(my\_dbg, **int**, flag)
12. {
13. noxke\_global\_dbg = flag;
14. **return** noxke\_global\_dbg;
15. }

**四、实验结果**

**4.1 Linux线程创建**

编译并运行代码，创建线程A与线程B并运行，查看线程输出结果。

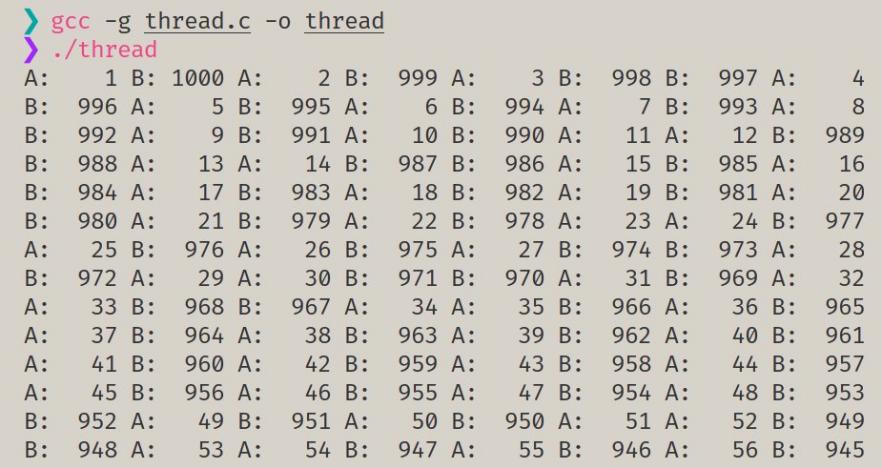


图4-1 线程运行结果

如图4-1所示，线程A与线程B同时运行，即使线程A与线程B的代码、休眠时间相同，两个线程运行的次序并不固定。

**4.2 Linux进程创建**

编译并运行进程创建代码，测试父进程不使用wait，先于子进程返回的情况，如图4-2所示。

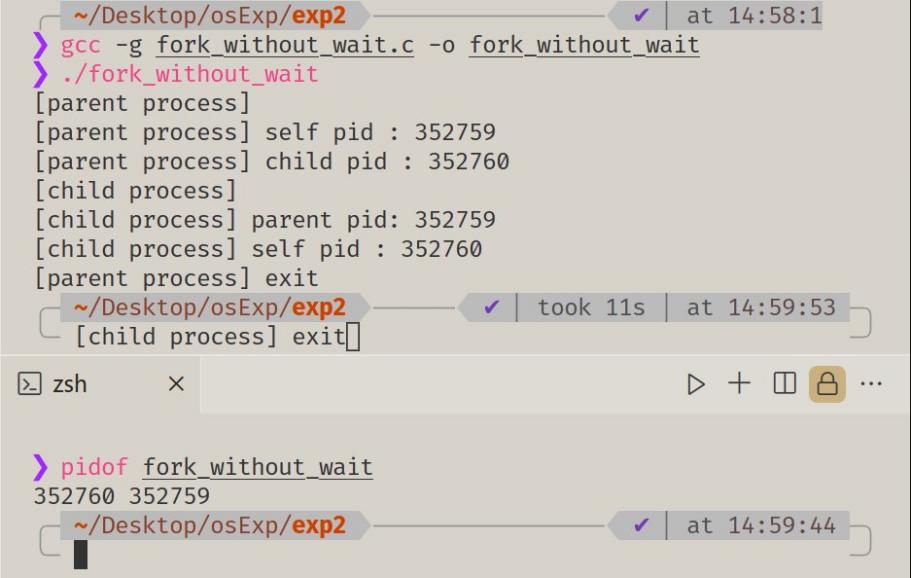


图4-2 父进程不使用wait

根据输出结果显示，父进程352759创建了子进程352760，父进程比子进程先返回退出，使用pidof命令查看程序pid与程序内部使用getpid结果相同。

测试父进程使用wait，等待子进程返回，获取子进程返回参数，如图4-3所示。

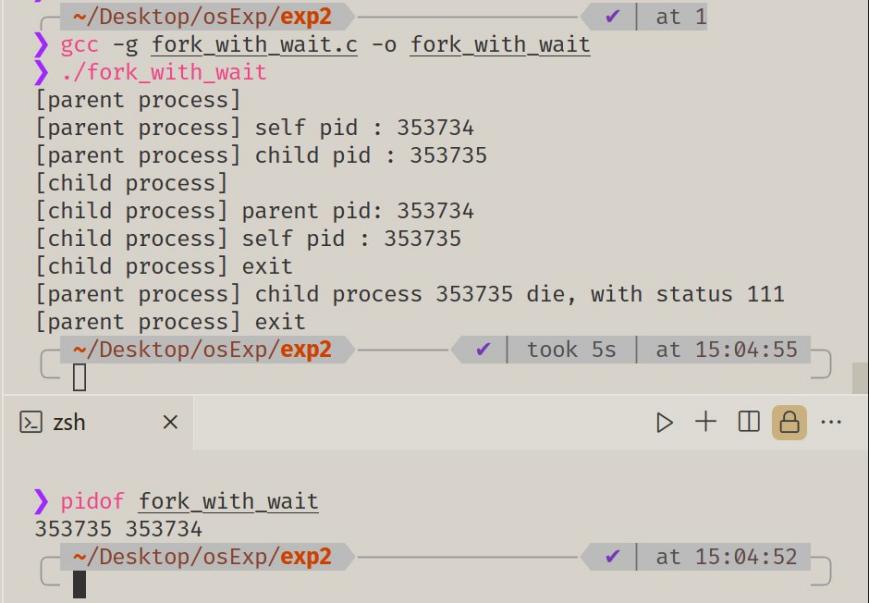


图4-3 父进程使用wait

根据输出结果显示，父进程353734创建了子进程353735,子进程先于父进程退出，子进程使用exit(111)退出后，父进程获取到子进程的退出状态111。

**4.3 “生产者-消费者”同步控制**

编译并运行“生产者-消费者”线程模拟代码，观察其运行结果，如图4-4所示。

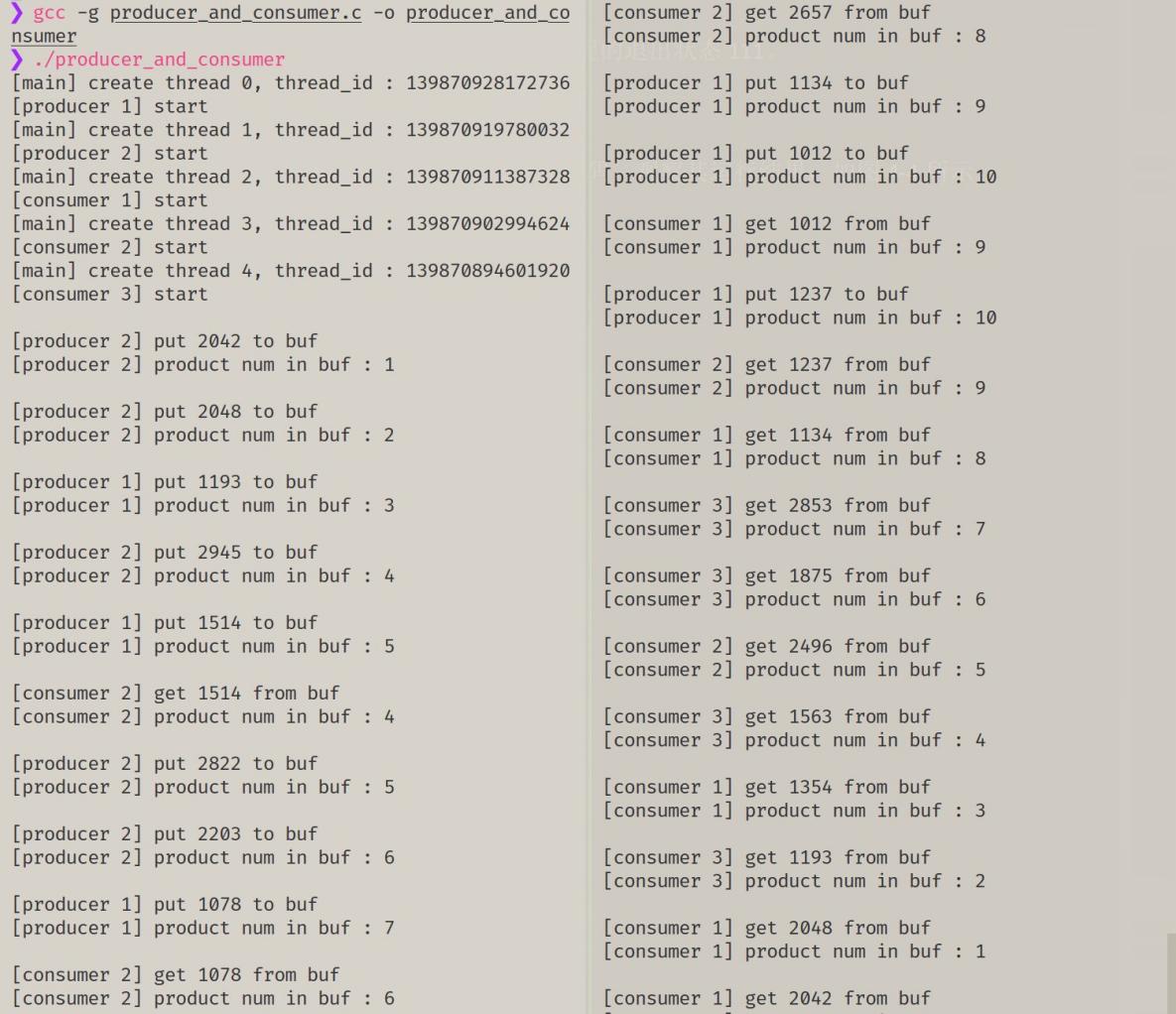


图4-4 生产者-消费者模拟

根据输出结果，生产者依次向缓冲区中加入产品，消费者从缓冲区中取出产品，产品的数量、数值，以及缓冲区状态始终保持正常，未发生数据丢失或失败的情况，说明同步控制功能与预期相符。

**4.4 “哲学家就餐”模拟**

编译并运行哲学家就餐模拟代码，观察程序运行结果，如图4-5所示。

根据输出结果，各个哲学家线程按照固定顺序进行思考、吃饭、睡觉的过程，并且在吃饭过程中，先拿起左侧筷子，再拿起右侧筷子，然后再就餐，最后依次放下左侧和右侧筷子，测试发现，即使程序理论存在死锁的可能，但实际并未观察到死锁现象，说明死锁发生的概率极低。

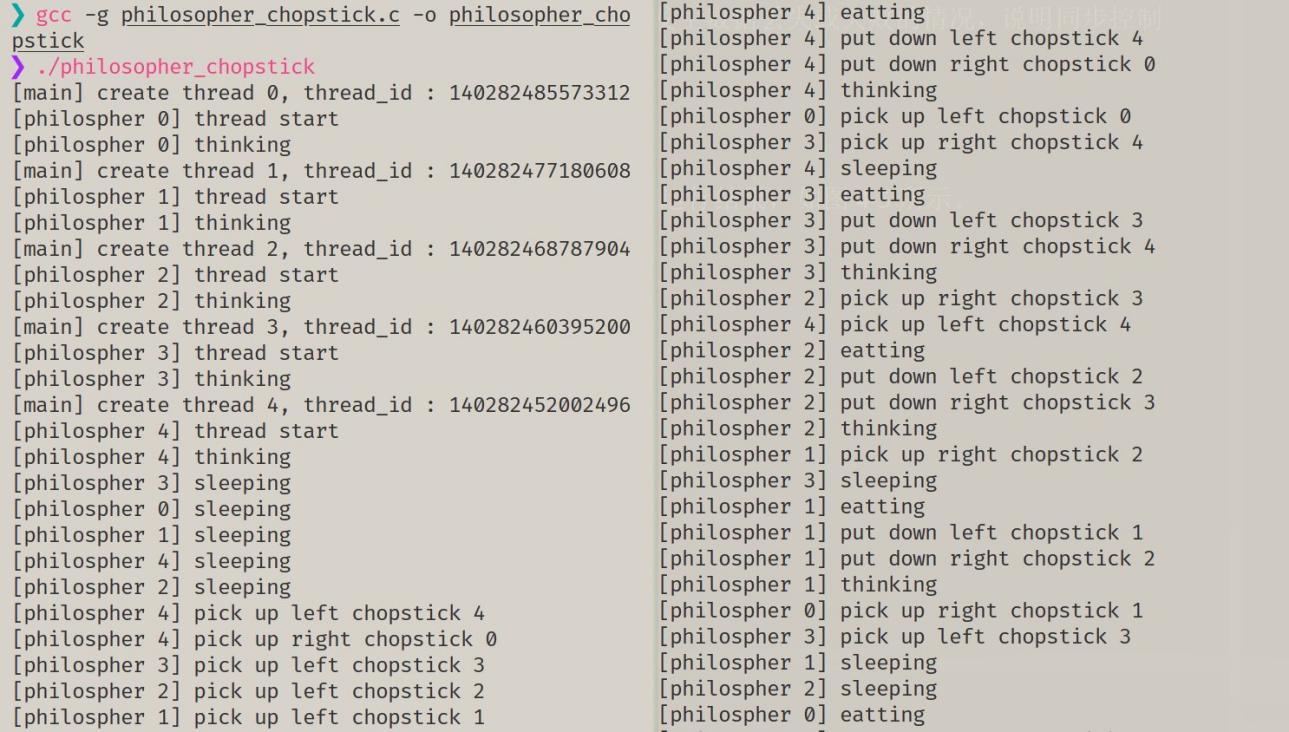


图4-5 哲学家就餐模拟

编译运行不存在死锁的版本，运行结果与图4-5基本相同，均未出现死锁情况。

**4.5 Linux进程创建与调度策略**

阅读linux内核源码，查看fork系统调用定义

1. // linux/kernel/fork.c
2. SYSCALL\_DEFINE0(fork)
3. {
4. **struct** kernel\_clone\_args args = {
5. .exit\_signal = SIGCHLD,
6. };
7. **return** kernel\_clone(&args);
8. }

fork创建子进程的操作主要在kernel\_clone函数中，函数主要代码如下

1. p = copy\_process(NULL, trace, NUMA\_NO\_NODE, args);
2. add\_latent\_entropy();
4. trace\_sched\_process\_fork(current, p);
6. pid = get\_task\_pid(p, PIDTYPE\_PID);
7. nr = pid\_vnr(pid);
9. wake\_up\_new\_task(p);
11. put\_pid(pid);
12. **return** nr;

在kernel\_clone函数返回前添加调试代码，使用printk输出函数返回结果

1. put\_pid(pid);
2. **if** (noxke\_global\_dbg == DBG\_FORK)
3. {
4. pid\_t current\_nr = \*(&current->pid);
5. printk("[noxke dbg][current pid : %d] kernel\_clone return value: %d\n", current\_nr, nr);
6. }
7. **return** nr;

编写代码fork进程测试，如图4-6所示。

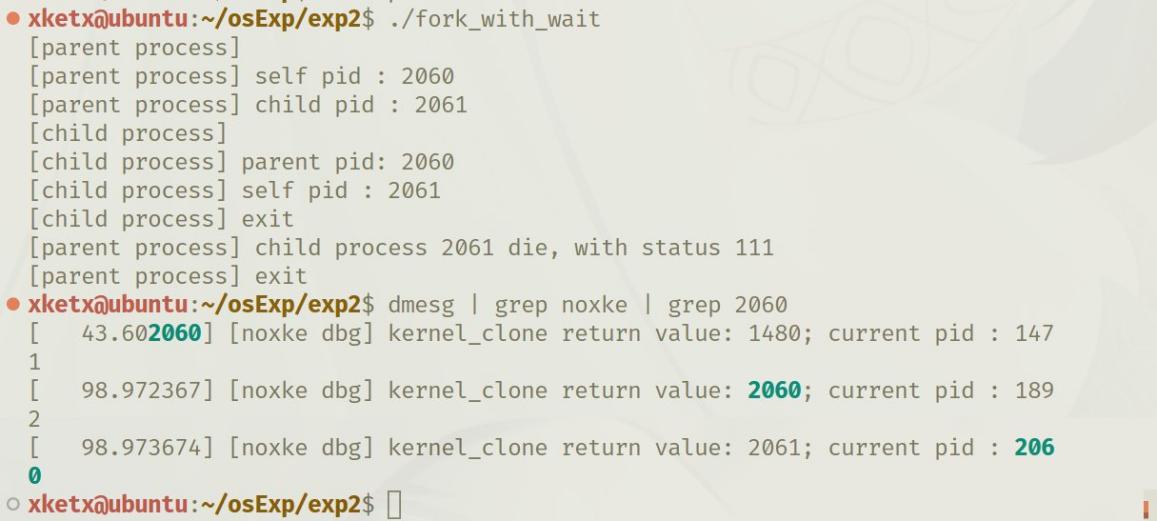


图4-6 kernel\_clone调试

调试可以发现，主进程2060创建了子进程2061，kernel\_clone函数返回值为fork得到的子进程的pid，但是子进程并没有从此处返回，并不能解释子进程fork返回值为0。

分析kernel\_clone中的copy\_process函数调用

1. // linux/kernel/fork.c:2519 copy\_process
2. retval = copy\_thread(p, args);
3. // linux/arch/x86/kernel/process.c:182 copy\_thread
4. frame->ret\_addr = (unsigned **long**) ret\_from\_fork\_asm;

在copy\_thread时调用了ret\_from\_fork\_asm

1. /\*linux/arch/x86/entry/entry\_64.S:225\*/
2. SYM\_CODE\_START(ret\_from\_fork\_asm)
4. UNWIND\_HINT\_END\_OF\_STACK
5. ANNOTATE\_NOENDBR // copy\_thread
6. CALL\_DEPTH\_ACCOUNT
8. movq    %rax, %rdi        /\* prev \*/
9. movq    %rsp, %rsi        /\* regs \*/
10. movq    %rbx, %rdx        /\* fn \*/
11. movq    %r12, %rcx        /\* fn\_arg \*/
12. call    ret\_from\_fork
14. UNWIND\_HINT\_REGS
15. jmp    swapgs\_restore\_regs\_and\_return\_to\_usermode
16. SYM\_CODE\_END(ret\_from\_fork\_asm)
17. // linux/arch/x86/kernel/process.c:140
18. \_\_visible **void** ret\_from\_fork(**struct** task\_struct \*prev, **struct** pt\_regs \*regs,
19. **int** (\*fn)(**void** \*), **void** \*fn\_arg)
20. {
21. schedule\_tail(prev);
23. /\* Is this a kernel thread? \*/
24. **if** (unlikely(fn)) {
25. fn(fn\_arg);
26. regs->ax = 0;
27. }
29. syscall\_exit\_to\_user\_mode(regs);
30. }

追踪源代码发现最终调用了ret\_from\_fork，修改了子进程的ax寄存器为0，以及使子进程由系统调用返回，因此子进程wake\_up之后的返回是0，与主进程返回位置不同，不是kernel\_clone函数。

使用printk调试进程创建过程，如图4-7所示。

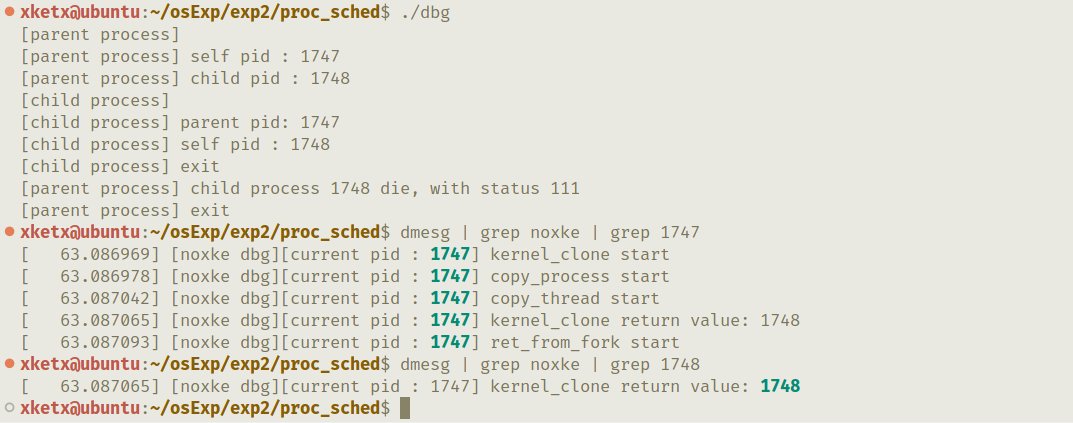


图4-7 printk调试进程创建

linux内核中任务调度的代码位于linux/kernel/sched/中，主调度器为\_\_schedule，主要代码如下：

1. // linux/kernel/sched/core.c:6568 \_\_schedule
2. **struct** task\_struct \*prev, \*next;
4. // 获取当前任务
5. cpu = smp\_processor\_id();
6. rq = cpu\_rq(cpu);
7. prev = rq->curr;
8. // 获取当前任务的切换次数
9. switch\_count = &prev->nivcsw;
10. // 挂起当前任务
11. deactivate\_task(rq, prev, DEQUEUE\_SLEEP | DEQUEUE\_NOCLOCK);
12. switch\_count = &prev->nvcsw;
14. // 获取下一个任务
15. next = pick\_next\_task(rq, prev, &rf);
16. clear\_tsk\_need\_resched(prev);

19. **if** (likely(prev != next)) {
20. rq->nr\_switches++;
21. RCU\_INIT\_POINTER(rq->curr, next);
22. // 上一个人物的切换次数+1
23. ++\*switch\_count;
24. migrate\_disable\_switch(rq, prev);
25. psi\_sched\_switch(prev, next, !task\_on\_rq\_queued(prev));
27. trace\_sched\_switch(sched\_mode & SM\_MASK\_PREEMPT, prev, next, prev\_state);
28. // 切换到新的任务
29. rq = context\_switch(rq, prev, next, &rf);
30. }

**五、实验错误排查和解决方法**

**5.1 Linux线程创建**

使用pthread\_create创建线程后，线程创建成功，但是线程内代码并未全部执行程序就退出了，如图5-1所示。

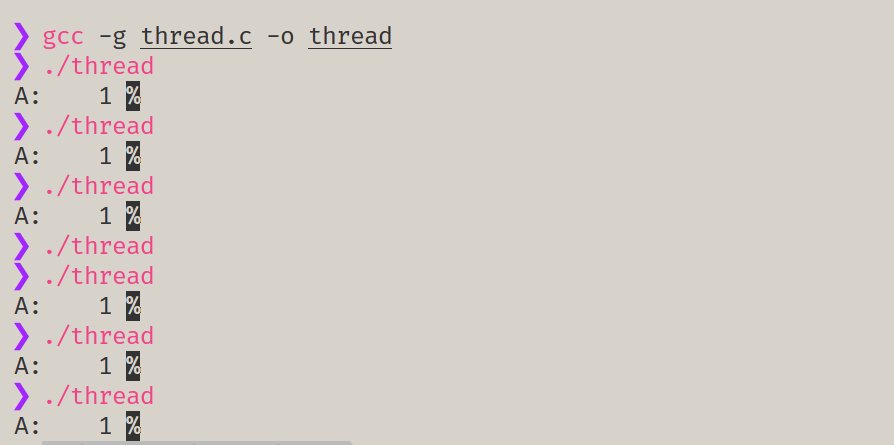


图5-1 进程运行失败

阅读pthread相关文档后得知，需要使用pthread\_join等待线程返回，否则主线程会直接返回，导致新创建的线程中的任务不能全部运行，添加pthread\_join后，主线程创建AB线程后进入阻塞状态，知道A线程与B线程退出，主线程才正常退出。

**5.2 Linux进程创建**

为主进程添加waitpid等待子进程返回时，可以成功等待子进程返回，但是输出状态码stat时，发现与子进程exit返回的状态码不一致，多次测试更换返回值仍存在状态码不一致的情况，阅读linux man手册发现，wait获得的返回状态之需要使用WEXITSTATUS宏处理后才是真正的子进程返回值。

1. **int** stat;
2. waitpid(pid, &stat, 0);
3. **int** exit\_status = WEXITSTATUS(stat);
4. printf("[parent process] child process %d die, with status %d\n", pid, exit\_status);

**5.3 Linux进程创建与调度策略**

在研究fork创建子进程的过程中，分析得到fork系统调用中使用的是kernel\_clone函数，kernel\_clone函数复制当前进程的PCB，唤醒子进程并返回子进程的pid，根据fork函数的返回情况，主进程中kernel\_clone返回值应当是子进程pid,与调试结果相符，但是子进程返回值应当是0，而调试过程中并未发现子进程从kernel\_clone函数返回，无法解释子进程fork函数返回值为0的情况。

详细分析kernel\_clone函数，最终观察到在copy\_process函数中调用copy\_thread时，并非完整地复制父进程的所有信息，copy\_thread中调用ret\_from\_fork\_asm，修改了线程的栈帧，将栈帧恢复到系统调用前的状态，并设置线程的ax寄存器为0，即设置返回值为0，因此子进程并没有从kernel\_clone函数中返回，而是在复制时设置其退出系统调用，wake\_up唤醒子进程后直接从用户态代码执行。

**六、实验参考资料和网址**

**（1）教学课件**

**（2）Linux新增系统调用**

<https://www.kernel.org/doc/html/latest/process/adding-syscalls.html>

**（3）Linux pthread手册**

<https://man7.org/linux/man-pages/man7/pthreads.7.html>

**（4）Linux fork手册**

<https://man7.org/linux/man-pages/man2/fork.2.html>