**电子科技大学**信息与软件工程**学院**

**标 准 实 验 报 告**

**（实验）课程名称 操作系统原理与实践实验（一）**

**电子科技大学教务处制表**

**电 子 科 技 大 学**

**电 子 科 技 大 学**

**实 验 报 告**

**学生姓名：李金雨 学 号：2021090922015 指导教师：杨珊**

**实验地点：信软学院楼西304 实验时间： 2023-04-25**

# 实验项目名称：

1. 哲学家就餐问题
2. 生产者/消费者问题的实现
3. 基于管道的进程间数据传输

# 实验学时：

每个实验4学时，共12学时。

# 实验原理：

# 哲学家就餐问题

本实验主要是学习线程的同步机制。本实验研究线程同步的经典问题-哲学家进餐问题。哲学家进餐问题是一大类并发控制问题的典型例子，涉及信号量机制、管程机制以及死锁等操作系统中关键问题的应用，在操作系统文化史上具有非常重要的地位。对该问题的剖析有助于深刻地理解计算机系统中的资源共享、进程同步机制、死锁等问题，并能熟练地将该问题的解决思想应用于生活中的控制流程。

哲学家进餐问题描述有五个哲学家，他们的生活方式是交替地进行思考和进餐，哲学家们共用一张圆桌，分别坐在周围的五张椅子上，在圆桌上有五个碗和五支筷子，平时哲学家进行思考，饥饿时便试图取其左、右最靠近他的筷子，只有在他拿到两支筷子时才能进餐，进餐完毕，放下筷子又继续思考。

约束条件

(1)只有拿到两只筷子时，哲学家才能吃饭。

(2)如果筷子已被别人拿走，则必须等别人吃完之后才能拿到筷子。

(3)任一哲学家在自己未拿到两只筷子吃饭前，不会放下手中拿到的筷子。

信号量机制

筷子是临界资源，一段时间只允许一位哲学家使用。为了表示互斥，用一个信号量表示一只筷子，五个信号量构成信号量数组。本文中算法用类C语言描述伪码算法。算法描述如下：n用五支筷子的信号量构成信号量数组：

Semaphore chopstick[5]={1，l，1，l，1}；

p(stick[i])；

p(stick[(i+1) % 5])；

进餐；

v(stick[i])；

v(stick[(i+1) % 5])；

思考；

当哲学家饥饿时，总是先去拿他左边的筷子，执行wait(chopstick[I])，成功后，再去拿他右边的筷子，执行wait(chopstick[I+1]%5)；成功后便可进餐。进餐毕，先放下他左边的筷子，然后再放下右边的筷子。当五个哲学家同时去取他左边的筷子，每人拿到一只筷子且不释放，即五个哲学家只得无限等待下去，引起死锁。程序应该用相应措施避免死锁或者在程序发生死锁时解除死锁。

本实验涉及到的线程同步机制是Linux互斥量机制。

确保同一时间里只有一个线程访问共享资源或临界区域。互斥量（mutex）本质上是一把锁，在访问共享资源后临界区域前，对互斥量进行加锁，在访问完成后释放互斥量上的锁，对互斥量进行加锁后，任何其他试图再次对互斥量加锁的线程将会被阻塞，直到锁被释放。

互斥量变量由pthread\_mutex\_t数据类型表示。在我们使用一个mutex变量之前，我们必须首先初始化它，通过把它设置为常量PTHREAD\_MUTEX\_INITIALIZER（只针对于静态分配的互斥量）或调用pthread\_mutex\_init。如果我们动态地分配互斥量（例如通过调用malloc），那么我们需要在释放内存前调用pthread\_mutex\_destroy。

#include <pthread.h>

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex, const pthread\_mutexattr\_t \*restrict attr);

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

成功返回0，失败返回错误码。

为了使用默认属性初始化一个互斥量，我们把attr设置为NULL。

为了锁住一个互斥量，我们调用pthread\_mutex\_lock。如果互斥量已经被锁了，那么调用线程将阻塞，直到互斥量被解锁。要解锁一个互斥量，我们调用pthread\_mutex\_unlock。

#include <pthread.h>

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

成功返回0，失败返回错误号。

如果一个线程不能容许阻塞，那么它可以使用pthread\_mutex\_trylock来有条件地锁这个互斥量。如果互斥量在pthread\_mutex\_trylock被调用时已经解锁，那么pthread\_mutex\_trylock会不阻塞地锁住这个互斥量并返回0。否则，pthread\_mutex\_trylock将会失败，返回EBUSY而不锁住这个互斥量。

　 多个任务循环等待它方占有的资源而无限期地僵持下去的局面。即：行进中的任务，等待永远不可能发生的事件。死锁是一种CPU的状态。

产生死锁的原因是系统资源不足（有限）；进程推进顺序不合适（操作不当）

　 同时具备以下四个条件时，死锁就可能发生。

1、互斥条件：

一个资源，在一个时刻只能由一个进程占有；

2、不可抢占条件：

一个资源被占有，在末释放之前，其它进程不得抢占；

3、占有且申请条件：

进程已占有某种资源，又申请新的资源；

4、循环等待条件：

进程已占有某种资源，又要等待其它进程已拥有的资源。

解决死锁的办法有：预防、避免、检测与恢复三种。

死锁预防的基本思想：要求进程申请资源时必须遵守某种协议，从而打破死锁产生的四个必要条件之一。

1、打破互斥条件：

　　允许进程同时访问某些资源，但有些资源不允许同时被访问；

2、打破不可抢占条件：

允许进程强行从占有者处抢占某些资源；

＊以上两个条件的打破都对系统管理不利，所以不易实行，不可取。

3、打破占有且申请条件：

　　采用资源预先分配策略。

　　这个策略是：只有当进程申请的所有资源都得到满足后，才调度其运行。

4、打破循环等待条件：

　　实行资源有序分配策略。

这个策略是将资源按类编号，按序分配。进程申请资源必须按号递增提出。若一个号已被占有，它只能申请它以后的号。

# 生产者/消费者问题的实现

本实验主要是学习进程间的通信机制。本实验研究进程的经典问题-生产者消费者问题。

生产者消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。

1、Linux信号量机制

操作系统中需要一个地位高于进程的管理者来解决公有资源的使用问题，信号量就是操作系统提供的管理公有资源的有效手段，作为操作系统核心代码执行，不受进程调度的打断，信号量代表可用资源实体的数量。

每个信号量s除一个整数值s.count（计数）外，还有一个进程阻塞队列s.queue，信号量只能通过初始化和两个标准的原语（P操作，V操作）来访问，初始化信号量指定s.count为一个非负整数值，表示空闲资源总数（又称为“资源信号量”），在程序运行中如果s.count若为非负值表示当前的空闲资源数，若为负值其绝对值表示当前等待临界区的进程数。二进制信号量：只允许信号量取0或1值（此时信号量功能和互斥量类似）。

在Linux系统中，使用信号量通常分为以下4个步骤：

   ①  创建信号量或获得在系统中已存在的信号量，此时需要调用 semget() 函数。不同进程通过使用同一个信号量键值来获得同一个信号量。

   ②  初始化信号量，此时使用 semctl() 函数的SETVAL操作。当使用二维信号量时，通常将信号量初始化为1。

   ③  进行信号量的PV操作，此时，调用 semop()函数。这一步是实现进程间的同步和互斥的核心工作部分。

   ④  如果不需要信号量，则从系统中删除它，此时使用semctl()函数的 IPC\_RMID操作。需要注意的是，在程序中不应该出现对已经被删除的信号量的操作。

2、Linux共享内存机制

共享内存是内核为进程创建的一个特殊内存段，它将出现在自己的地址空间中，其它进程可以将同一段共享内存连接(attach)到自己的地址空间，它是最快的进程间通信方式，不提供任何同步功能。

对共享内存的主要操作如下：

打开或创建共享内存

int semget(key\_t key, int size, int flag);

将共享内存连接到进程空间

void \*shmat(int shmid, void \*addr, int flag);

断开进程空间和共享内存的连接

int shmdt(void \*addr);

共享内存控制操作

int shmctl(int shmid, int cmd, struct shmid\_ds \*buf);

# 基于管道的进程间数据传输

1）熟悉Linux操作系统下的编程

2）熟悉Linux下进程间的通信过程

3）熟悉Linux环境下进程的控制和同步

# 实验目的：

# 哲学家就餐问题

掌握哲学家就餐问题，掌握预防死锁的实现方法；了解Linux系统下进程和线程的实现；掌握通过互斥量、POSIX信号量、XSI信号量集实现多线程/多进程同步控制的方法

# 生产者/消费者问题的实现

掌握进程、线程的概念，熟悉相关的控制语；掌握进程、线程间的同步原理和方法；掌握进程、线程间的互斥原理和方法。

# 基于管道的进程间数据传输

1.管道是一种古老的通信机制，是一种特殊的文件，只支持半双工，一个进程写，一个进程读，并且只能在父子进程间使用。

2.消息队列就是一个消息的链表。可以把消息看作一个记录，具有特定的格式以及特定的优先级。对消息队列有写权限的进程可以向中按照一定的规则添加新消息；对消息队列有读权限的进程则可以从消息队列中读走消息。

# 实验内容：

# 哲学家就餐问题

熟悉Ubuntu系统环境和命令；熟悉Ubuntu系统下的多线程/多进程编程；在Ubuntu系统下编程实现哲学家就餐问题。实现教材2.5.2节中所描述的哲学家就餐问题。要求显示出每个哲学家的工作状态，如吃饭，思考。连续运行30次以上都未出现死锁现象。

# 生产者/消费者问题的实现

有一群生产者进程在生产产品，并将这些产品提供给消费者进程去消费。为使生产者进程与消费者进程能并发执行，在两者之间设置了一个具有n个缓冲区的缓冲池：生产者进程从文件中读取一个数据，并将它存放到一个缓冲区中； 消费者进程从一个缓冲区中取走数据，并输出此数据。生产者和消费者之间必须保持同步原则：不允许消费者进程到一个空缓冲区去取产品；也不允许生产者进程向一个已装满产品且尚未被取走的缓冲区中投放产品。

创建3进程（或者线程）作为生产者，4个进程（或者线程）作为消费者。创建一个文件作为数据源，文件中事先写入一些内容作为数据。

生产者和消费者进程（或者线程）都具有相同的优先级。

# 基于管道的进程间数据传输

第一部分：利用管道实现两个进程的通信、

一：

1. 父进程首先使用系统调pipe()建立一个管道，然后使用系统调用fork()创建子进程1，
2. 子进程1关闭管道读文件，
3. 子进程1通过文件I/O操作向管道写文件写一句话（向文件中写入字符串）：Child process 1 is sending a message!
4. 然后子进程1调用exit()结束运行。

二：

1. 父进程再次使用系统调用fork()创建子进程2，
2. 子进程2关闭管道读文件，
3. 子进程2通过文件I/O操作向管道写文件写一句话（向文件中写入字符串）：Child process 2 is sending a message!
4. 然后子进程2调用exit()结束运行。
5. .父进程关闭管道写文件，父进程通过文件I/O操作从管道读文件中读出来自于两个子进程的信息，通过printf语句打印输出在屏幕上。

第二部分：利用消息队列实现进程间的通信

1.编程实现两个进程通过消息队列进行通信，

2.一个server进程，一个client进程。server进程向client进程发送其进程ID，client进程同时也向server进程发送其进程ID，

3.双方接收到消息后，将所接收到的进程ID输出到屏幕上。

# 实验器材（设备、元器件）：

PC计算机，操作系统：Ubuntu

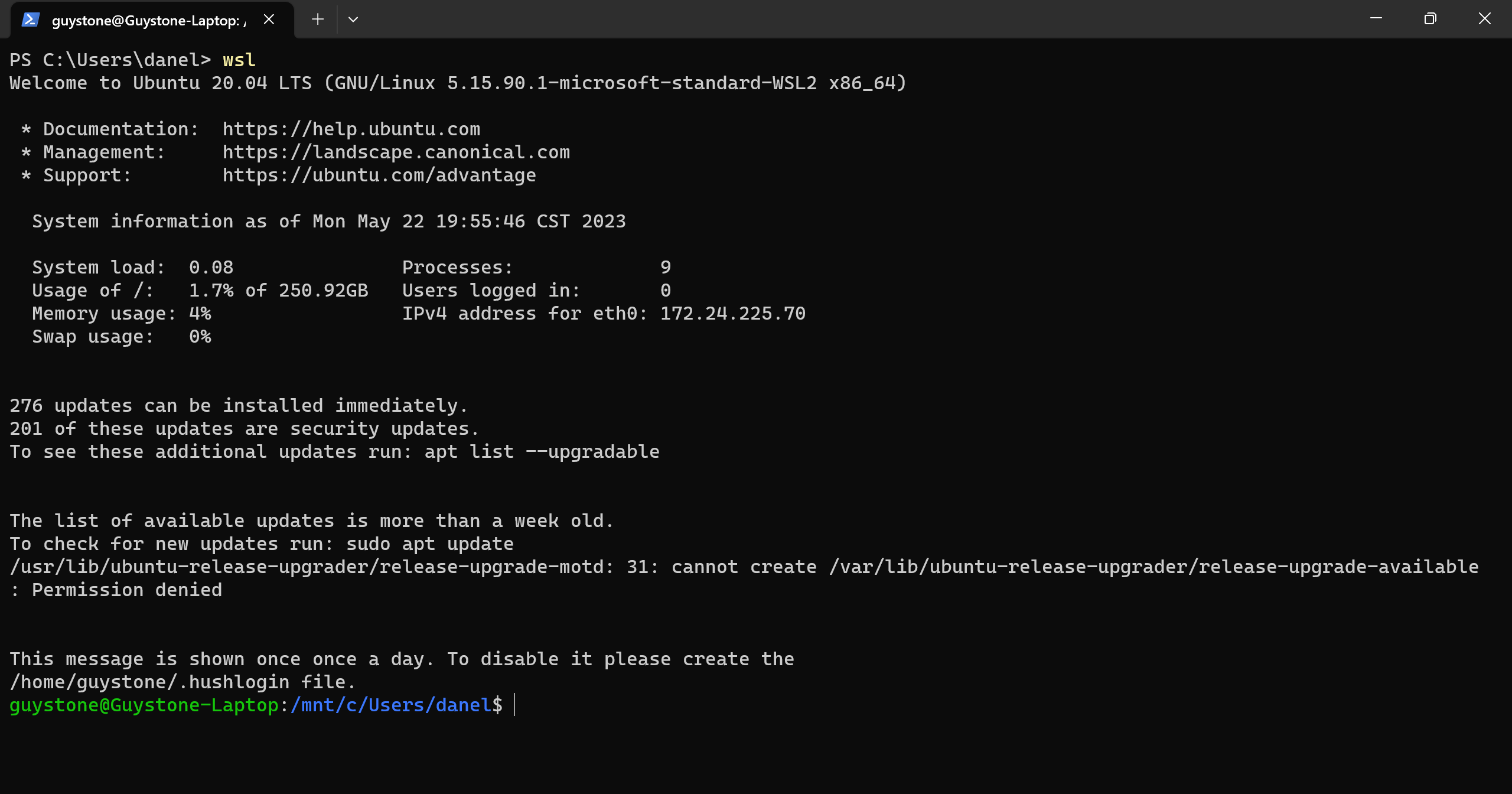
# 实验步骤：

1. 搭建开发环境

只需要安装好Ubuntu系统即可，这里使用了Windows的Linux子系统（Ubuntu内核）。

Ubuntu环境的安装：

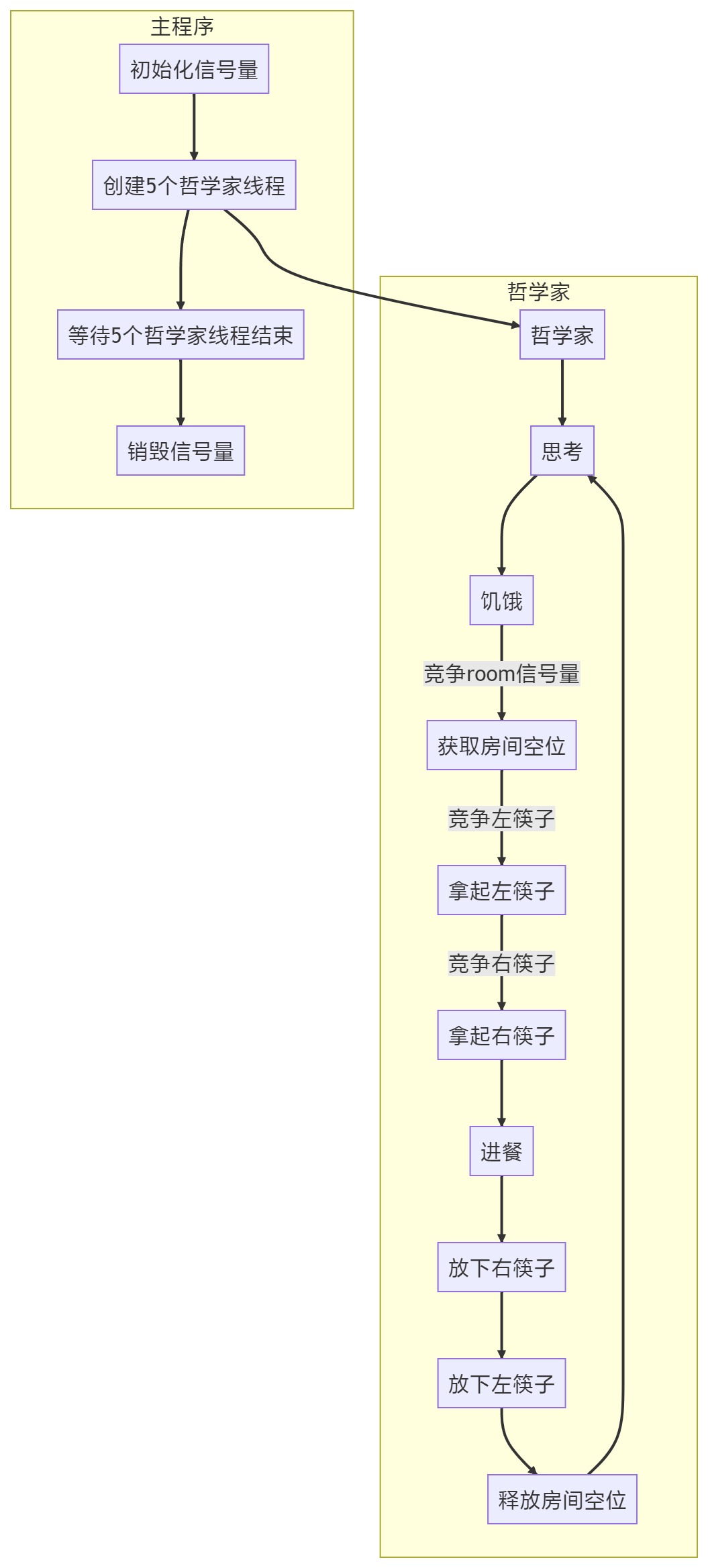
* 1. 启动Powershell
  2. 执行wsl –install命令
  3. 如图，安装完成后执行wsl命令即可进入Ubuntu系统命令行



2) 程序开发：

1. 哲学家就餐问题

流程图如图：



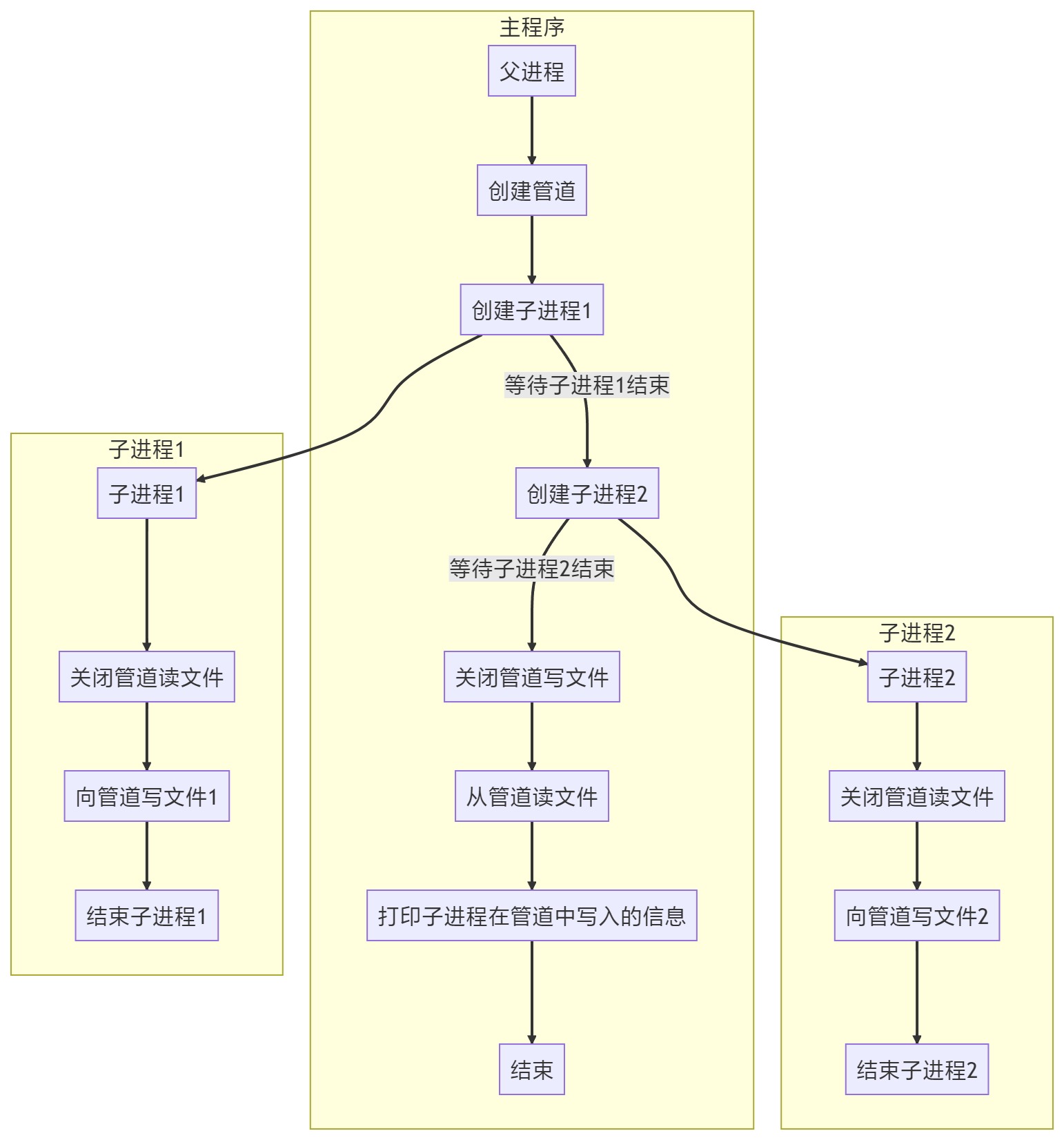
1. 生产者/消费者问题的实现

流程图如图：  
图示, 示意图

描述已自动生成

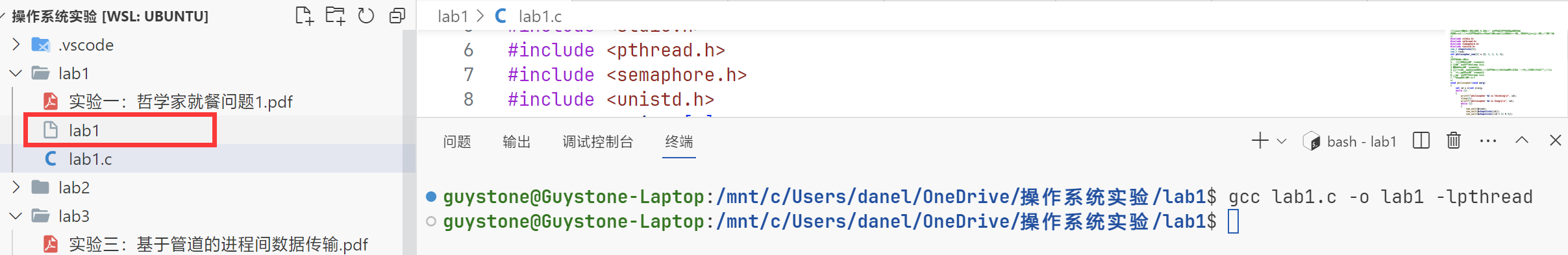
1. 基于管道的进程间数据传输

流程图如图：



3) 编译、调试、运行

* 命令行下的编译运行
  + C语言的编译：使用gcc命令（对于使用了线程的，需要加上-lpthread选项），如对lab1文件，执行gcc lab1.c -o lab1 -lpthread，可以看到生成了lab1可执行文件：



* 可执行文件（包括C语言编译生成的文件和sh文件）的执行：如对lab1文件，执行./lab1即可：

文本

中度可信度描述已自动生成

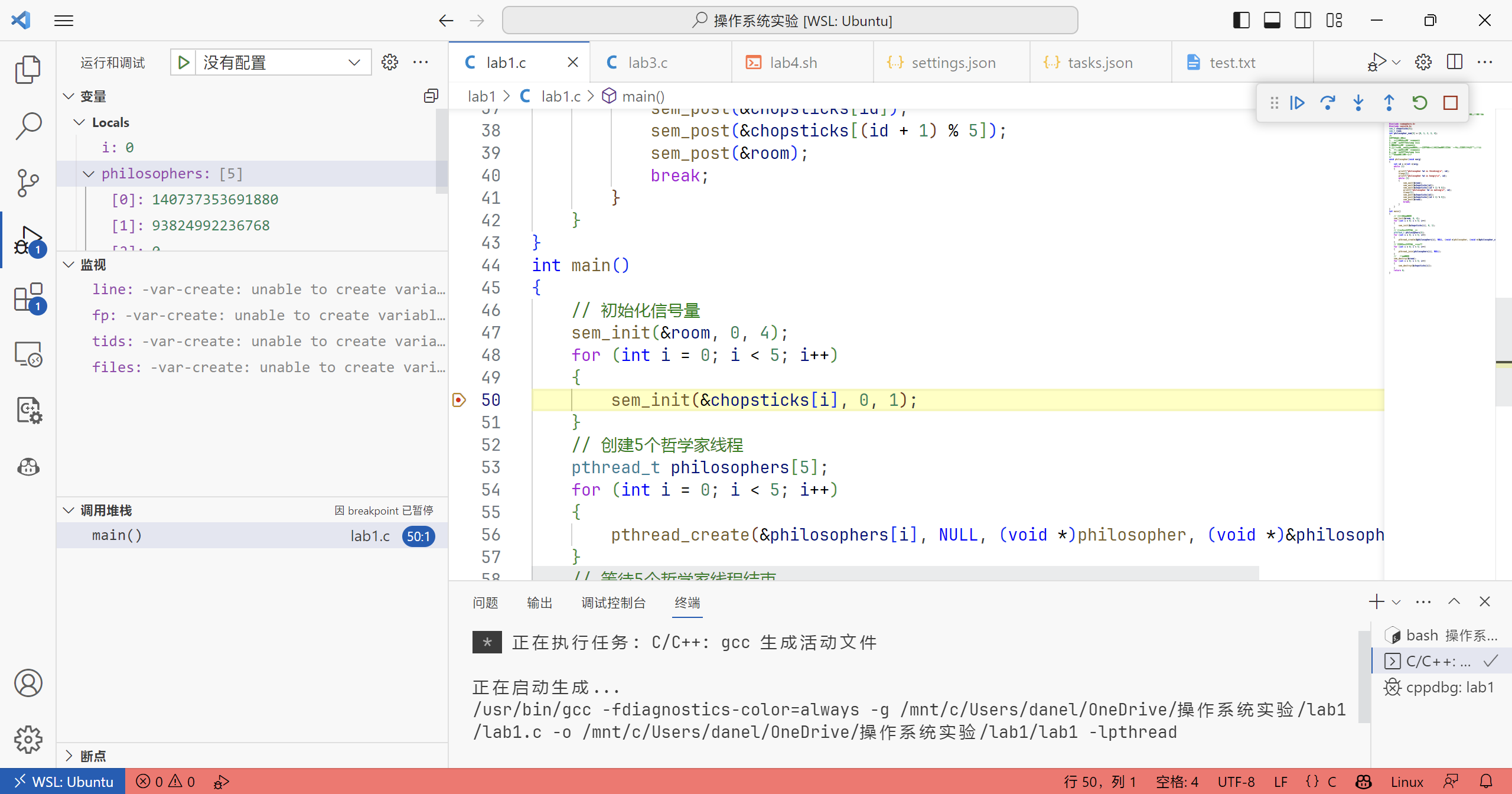
* VSCode下的编译、调试、运行
  + C语言的运行：直接点击运行按钮即可，由于VSCode可以直接支持使用 WSL 进行远程开发和调试，点击运行按钮后会自动在Ubuntu系统下编译和运行程序：



图形用户界面, 文本, 应用程序

描述已自动生成

* + C语言的调试：直接点击调试按钮即可，由于VSCode可以直接支持使用 WSL 进行远程开发和调试，点击运行按钮后会自动在Ubuntu系统下编译和调试程序（下图设置了一个断点）：



# 实验数据及结果分析：

# 哲学家就餐问题

**1.该实验主要程序段如下：**

|  |
| --- |
| #include <stdio.h>  #include <pthread.h>  #include <semaphore.h>  #include <unistd.h>  sem\_t chopsticks[5];  sem\_t room;  int philosopher\_num[5] = {0, 1, 2, 3, 4};  void philosopher(void \*arg)  {      int id = \*(int \*)arg;      while (1)      {          printf("philosopher %d is thinking\n", id);          sleep(1);          printf("philosopher %d is hungry\n", id);          while (1)          {              sem\_wait(&room);              sem\_wait(&chopsticks[id]);              sem\_wait(&chopsticks[(id + 1) % 5]);              printf("philosopher %d is eating\n", id);              sleep(1);              sem\_post(&chopsticks[id]);              sem\_post(&chopsticks[(id + 1) % 5]);              sem\_post(&room);              break;          }      }  }  int main()  {      // 初始化信号量      sem\_init(&room, 0, 4);      for (int i = 0; i < 5; i++)      {          sem\_init(&chopsticks[i], 0, 1);      }      // 创建5个哲学家线程      pthread\_t philosophers[5];      for (int i = 0; i < 5; i++)      {          pthread\_create(&philosophers[i], NULL, (void \*)philosopher, (void \*)&philosopher\_num[i]);      }      // 等待5个哲学家线程结束      for (int i = 0; i < 5; i++)      {          pthread\_join(philosophers[i], NULL);      }      // 销毁信号量      sem\_destroy(&room);      for (int i = 0; i < 5; i++)      {          sem\_destroy(&chopsticks[i]);      }      return 0;  } |

**2.应用程序运行状态截图**

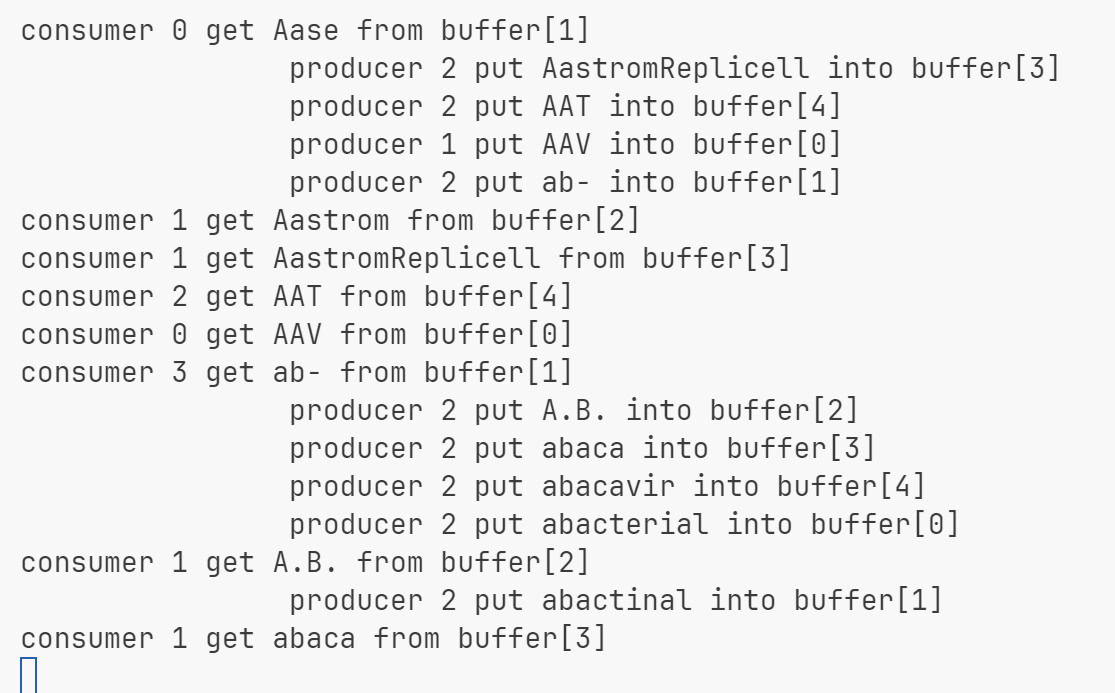


# 生产者/消费者问题的实现

**1.该实验主要程序段如下：**

|  |
| --- |
| #include <stdio.h>  #include <pthread.h>  #include <semaphore.h>  #include <unistd.h>  #include <string.h>  #define N 5            // 缓冲区个数  pthread\_mutex\_t mutex; // 互斥锁，保证缓冲区的互斥访问  sem\_t empty;           // 信号量，表示空缓冲区个数  sem\_t full;            // 信号量，表示满缓冲区个数  char buffer[N][1000];  // 缓冲区  int line = 0;                       // 读取文件的行数  int in = 0, out = 0;                // 缓冲区的写入指针和读取指针  int producer\_num[3] = {0, 1, 2};    // 生产者标号  int consumer\_num[4] = {0, 1, 2, 3}; // 消费者标号  FILE \*fp; // 文件指针  struct paras  {      int \*num;      FILE \*fp;  };  void producer(void \*arg)  {      while (1)      {          // 从文件中读取一个数据          char data[1000];          // 初始化参数          struct paras \*p = (struct paras \*)arg;          int \*num = p->num;          FILE \*fp = p->fp;          sem\_wait(&empty);          pthread\_mutex\_lock(&mutex);          memset(data, 0, sizeof(data));          if(fscanf(fp, "%s", data)==EOF) // 读取文件结束          {              fseek(fp, 0, SEEK\_SET); // 从头开始读取文件              fscanf(fp, "%s", data);          }          printf("                producer %d put %s into buffer[%d]\n", \*num, data, in); // 为方便调试，输出数据          memset(buffer[in], 0, sizeof(buffer[in]));                             // 清空缓冲区          strcpy(buffer[in], data);          in = (in + 1) % N; // 循环队列，in指向下一个空缓冲区          sleep(1);          // 为方便调试，每次生产者生产数据后休眠1秒          pthread\_mutex\_unlock(&mutex);          sem\_post(&full);      }  }  void consumer(void \*arg)  {      while (1)      {          // 从一个缓冲区中取走数据          sem\_wait(&full);          pthread\_mutex\_lock(&mutex);          printf("consumer %d get %s from buffer[%d]\n", \*(int \*)arg, buffer[out], out);          out = (out + 1) % N; // 循环队列，out指向下一个存放数据的缓冲区          sleep(1);            // 为方便调试，每次消费者消费数据后休眠1秒          pthread\_mutex\_unlock(&mutex);          sem\_post(&empty);      }  }  int main()  {      // 打开文件      FILE \*fp = fopen("data.txt", "r");      // 初始化互斥锁      pthread\_mutex\_init(&mutex, NULL);      // 初始化信号量      sem\_init(&empty, 0, N);      sem\_init(&full, 0, 0);      // 初始化缓冲区      for (int i = 0; i < N; i++)      {          memset(buffer[i], 0, sizeof(buffer[i]));      }      // 创建3个生产者线程      pthread\_t producers[3];      for (int i = 0; i < 3; i++)      {          struct paras p;          p.num = &producer\_num[i];          p.fp = fp;          pthread\_create(&producers[i], NULL, (void \*)producer, (void \*)&p);      }      // 创建4个消费者线程      pthread\_t consumers[4];      for (int i = 0; i < 4; i++)      {          pthread\_create(&consumers[i], NULL, (void \*)consumer, (void \*)&consumer\_num[i]);      }      // 等待3个生产者线程结束      for (int i = 0; i < 3; i++)      {          pthread\_join(producers[i], NULL);      }      // 等待4个消费者线程结束      for (int i = 0; i < 4; i++)      {          pthread\_join(consumers[i], NULL);      }      // 销毁互斥锁      pthread\_mutex\_destroy(&mutex);      // 销毁信号量      sem\_destroy(&empty);      sem\_destroy(&full);      return 0;  } |

**2.应用程序运行状态截图**



# 基于管道的进程间数据传输

**1.该实验主要程序段如下：**

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  #include <sys/types.h>  #include <string.h>  #include <stdlib.h>  #include <sys/wait.h>  #define BUFSIZE 100  #define MAX\_PROCESS\_NUM 2  int process\_num = 0;  int main(void)  {      char bufin[BUFSIZE] = "empty";      int bytesin;      pid\_t childpid[MAX\_PROCESS\_NUM];      int fd[2];          // fd[0] 读数据, fd[1] 写数据      if (pipe(fd) == -1) // 创建管道      {          perror("Failed to create the pipe");          return 1;      }      bytesin = strlen(bufin);      for (int i = 0; i < MAX\_PROCESS\_NUM; i++)      {          process\_num++; // 子进程标号          childpid[i] = fork(); // 创建子进程          switch (childpid[i])          {          case -1:              perror("Failed to fork");              return 1;          case 0:          {              char bufout[] = "Child process %d is sending a message!\n"; // 子进程写入的字符串              sprintf(bufout, bufout, process\_num);              write(fd[1], bufout, strlen(bufout) + 1); // 写入管道              printf("Child process %d is sending a message!\n", process\_num);              exit(0); // 子进程退出          }          default:              // 等待子进程i结束              waitpid(childpid[i], NULL, 0);              break;          }      }      printf("Parent process is reading from pipe...\n");      bytesin = read(fd[0], bufin, BUFSIZE);      for(int i = 0; i < BUFSIZE; i++)      {          printf("%c", bufin[i]); // 读取管道，由于字符串在bufin中连续存储，不能直接printf(bufin)，否则只会输出第一个字符串      }      return 0;  } |

**2.应用程序运行状态截图**

文本

描述已自动生成

# 总结及心得体会：

通过这次实验，我加深了对互斥量和信号量的理解，使用了Linux提供的互斥量和信号量定义来解决临界区问题和进程同步问题。在哲学家进餐问题和生产者消费者问题中,我通过信号量实现了线程间的协调,避免了死锁发生。

我还掌握了Linux环境下多进程编程的方法。通过fork创建子进程,然后在父子进程间交换数据,我熟悉了Linux特有的进程间通信API，这使我还加深了对进程间通信的理解。我通过管道这种方式,实现了父子进程间的数据交换,更加直观地了解了进程间通信的工作机制。

另外,我熟悉了Ubuntu环境下C语言编程和在WSL环境下使用VScode进行调试的方法。这为我以后的Linux编程打下了基础。

通过这次实验,我对进程和线程有了更加形象的认识。通过操作和实际编程,我对进程与线程的区别有了更准确的理解。

# 对本实验过程及方法、手段的改进建议：

1. 本次实验使用的是同一个源文件,进程(线程)间通信相对复杂。我建议可以将不同的进程(线程)逻辑写在不同的源文件中,分工合理,利于编码和维护。
2. 信号量和互斥量的初始化和释放使用的是资源初始化和销毁的方法,可以改用RAII(资源获取即初始化)方式封装,更加安全。
3. 可读性不高。进程(线程)间的通信逻辑、同步逻辑混在一起,可读性较差。可以通过函数模块化来提高可读性。

**报告评分：**

**指导教师签字：**