**深 圳 大 学 实 验 报 告**

**课程名称： 操作系统**

**实验项目名称： 实验2处理机调度**

**学院： 数学科学学院**

**专业： 信息与计算科学（数学与计算机实验班）**

**指导教师： 张滇**

**报告人： 王曦 学号： 2021192010 班级： 数计**

**实验时间： 2023年4月4日-2023年4月30日**

**实验报告提交时间： 2024年4月14日**

**教务部制**

|  |
| --- |
| **实验目的与要求：**  **实验目的：**  掌握进程同步与通信实验  加深对进程同步于通信操作的直观认识；  掌握Linux操作系统的进程、线程机制和编程接口；  掌握Linux操作系统的进程和线程间的同步和通信机制；  掌握经典同步问题的编程方法；  **实验要求：**  操作部分（参考）：   1. 借助google工具查找资料，学习使用Linux进程间通信：管道、消息队列、共享内存； 2. 设计编写以下程序，着重考虑其同步问题：    1. 一个程序（进程）从客户端读入按键信息，一次将“一整行”按键信息保存到一个共享存储的缓冲区内并等待读取进程将数据读走，不断重复上面的操作；    2. 另一个程序（进程）生成两个进程，用于显示缓冲区内的信息，这两个进程并发读取缓冲区信息后将缓冲区清空（一个进程的两次显示操作之间可以加入适当的时延以便于观察）。    3. 在两个独立的终端窗口上分别运行上述两个程序，展示其同步与通信功能，要求一次只有一个任务在操作缓冲区。    4. 运行程序，记录操作过程的截屏并给出文字说名。   要求使用posix信号量来完成这里的生产者和消费者的同步关系。  实验报告要求：   1. 按学校统一格式 2. 需要给出具体命令和自行编写的程序的源代码 3. 程序的设计需要给出设计思路或流程框图 4. 实验操作的截图需要有必要的说明文字   **说明：**  （1）本次实验课作业满分为100分，占总成绩的比例（待定）。  （2）本次实验课作业截至时间2024年4月30日（周二）23:59。  （3）报告正文：请在指定位置填写，本次实验不需要单独提交源程序文件。  （4）个人信息：WORD文件名中的“姓名”、“学号”，请改为你的姓名和学号；实验报告的首页，请准确填写“学院”、“专业”、“报告人”、“学号”、“班级”、“实验报告提交时间”等信息。  （5）提交方式：请在BLACKBOARD平台中按时提交；延迟提交不得分。  （6）发现抄袭（包括复制&粘贴整句话、整张图），该次作业记零分。  （7）期末考试阶段补交无效。 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1. 预备知识**  **1.1 Linux 的进程通信**  Linux 提供了三种进程通信方式：管道、消息队列、共享内存。  **1.1.1 管道**  Linux 的管道分为无名管道（pipe）、命名管道（named pipe，或称FIFO），其中前者用于父子进程间，后者可用于任意进程间。  管道用于单向通信，若需双向通信，则需建立两个方向相反的管道。  **1.1.2 消息队列**  消息队列可用于多进程通信，它类似邮箱，消息类似信件，消息间通过类型区分。  **1.1.3 共享内存**  共享内存是内核提供的一段内存，可映射到多个进程的虚存空间上，进而通过内存的读写操作完成进程间的数据共享。  **1.2 Linux 的共享内存**  本次实验的进程通信采用共享内存实现。下面介绍 Linux 中有关共享内存的指令和C库。  **1.2.1 查看IPC资源**  可用ipcs命令查看系统的IPC通信资源。**图1.1**展示的IPC通信资源分别为消息队列、共享内存和信号量集。  图1.1：ipcs命令  可使用ipcs命令的选项-q、-m、-s分别实现只展示消息队列、共享内存和信号量集。以-m选项为例。如**图1.2**所示。  图1.2：ipcs -m命令  perms列展示IPC资源的权限，类似于文件的权限。  **1.2.2 共享内存**  Linux 的共享内存通过tmpfs文件系统实现，其目录为/dev/shm。它常驻内存，默认尺寸是系统内存大小的一半，但只有有文件时才会真正占用这块内存。  C语言提供了如下函数来操作Linux 的共享内存：   |  |  | | --- | --- | | **原型** | int shm\_open(const char \*name, int oflag, mode\_t mode); | | **功能** | 打开或创建tmpfs文件系统中的共享内存文件。 | | **返回值** | 成功返回fd > 0，失败返回 fd < 0。 | | **参数** | 1. name：要打开或创建的共享内存文件名，由于shm\_open()打开或操作的文件都是位于/dev/shm目录的，因此name不能带路径，例如：/var/myshare 这样的名称是错误的，而 myshare 是正确的，因为 myshare 不带任何路径。如果一定要在name添加路径，可在/dev/shm目录里创建一个目录，如想创建一个bill/myshare 的共享内存文件，先在/dev/shm目录里创建 bill子目录。注意不同Linux系统的tmpfs的位置不都是/dev/shm，因此带路径的名称可能在别的环境下打开不成功。 2. oflag：打开的文件操作属性，是O\_CREAT、O\_RDWR、O\_EXCL的按位或组合。 3. mode：文件共享模式，如 0666 表示所有人都可读可写。 |  |  |  | | --- | --- | | **原型** | void \*mmap(void \*addr, size\_t length, int prot, int flags,int fd, off\_t offset); | | **功能** | 将打开的文件映射到内存，一般是将shm\_open()打开的文件映射到内存，但也可将硬盘上的用open()函数打开的文件映射到内存。该函数将文件映射到内存中，使得用户可用操作内存指针的方式来操作文件数据。 | | **返回值** | 成功返回映射的内存地址指针，可以用该地址指针对映射的文件内容进行读写操作，读写文件数据如同操作内存一样；失败则返回NULL。 | | **参数** | 1. addr：要将文件映射到的内存地址，一般应该传递NULL，由Linux内核指定。 2. length：要映射的文件数据长度。 3. prot：映射的内存区域的操作权限（保护属性），包括PROT\_READ、PROT\_WRITE、PROT\_READ|PROT\_WRITE 。 4. flags：标志位参数，包括MAP\_SHARED、MAP\_PRIVATE与MAP\_ANONYMOUS。 MAP\_SHARED: 建立共享，用于进程间通信。若无该标志，则别的进程即使能打开文件，也看不到数据。 MAP\_PRIVATE: 只有进程自己用的内存区域。 MAP\_ANONYMOUS:匿名映射区。 5. fd：用来建立映射区的文件描述符，用 shm\_open()或open()打开的文件。 6. offset：映射文件相对于文件头的偏移位置，按4096字节对齐。 |  |  |  | | --- | --- | | **原型** | int munmap(void \*addr, size\_t length); | | **功能** | 取消内存映射，addr是由mmap()成功返回的地址，length是要取消的内存长度。munmap ()将映射的内存从进程的地址空间撤销，若不调用该函数，进程终止前，该片区域得不到释放。 |  |  |  | | --- | --- | | **原型** | int shm\_unlink(const char \*name); | | **功能** | 删除/dev/shm目录的文件。shm\_unlink() 删除的文件是由shm\_open()函数创建于/dev/shm目录的。也可以用系统函数unlink()达到同样的效果，用/dev/shm + name 组成完整的路径即可，但一般不这么做，因为系统的tmpfs的位置可能不是/dev/shm。用shm\_open()创建的文件，若不调用此函数删除，则共享内存会一直存在于/dev/shm目录里，直至操作系统重启或者调用linux命令rm删除为止。 |  |  |  | | --- | --- | | **原型** | int ftruncate(int fd, off\_t length); | | **功能** | 重置文件大小。任何open()打开的文件都可以用这个函数，不限于shm\_open()打开的文件。 |   **1.3 POSIX 信号量**  POSIX是System V IPC提供的用于进程间同步的信号量，分为有名信号量和无名信号量，前者与一个文件的路径名关联，创建后不随进程结束而消失；后者只在进程声明周期内存在且只能在该进程创建的线程间使用。  本次实验使用POSIX无名信号量完成，下面介绍Linux中有关操作POSIX无名信号量的函数。   |  |  | | --- | --- | | **函数** | **作用** | | sem\_init() | 创建无名信号量。  原型：int sem\_init(sem\_t \*sem, int pshared, unsigned int value);  参数：  pshared表示该信号量是否用于进程间同步。pshared = 0 表示该信号量只用于进程内的线程间的同步；pshared != 0表示该信号量存放在共享内存中，用于实现访问该共享内存区的进程的同步。 | | sem\_destroy() | 撤掉无名信号量。 | | sem\_wait() | 对无名信号量做P操作。 | | sem\_post() | 对无名信号量做V操作。 |   **2. 生产者-消费者问题**  **2.1 生产者-消费者问题简介**  生产者-消费者问题是多线程编程中的经典同步问题之一，涉及到多个线程对共享资源的访问和操作。通常情况下，有两种类型的线程：生产者和消费者。生产者负责生成一些数据，并将其放入共享资源中，而消费者则负责从共享资源中获取数据并进行处理。  这个问题的挑战在于确保生产者和消费者线程之间的同步，以避免以下情况发生：  （1）生产者速度快于消费者：生产者可能会在消费者处理数据之前生成大量数据，导致资源溢出或浪费。  （2）消费者速度快于生产者：消费者可能会尝试从空资源中获取数据，导致错误或不必要的等待。  为了解决这些问题，可以使用各种同步机制，如互斥锁、条件变量或信号量。通过这些机制，可以确保生产者和消费者之间的协调，使得生产者只在资源可用时才生成数据，而消费者只在资源中有数据可供消费时才获取和处理数据。  **2.2 生产者-消费者问题实现**  **2.2.1 要求**  分别编写生产者、消费者进程，用POSIX信号量实现同步。  要求：   1. 生产者进程从客户端读取按键信息，每次将“一整行”按键信息保存到一个共享存储的缓冲区中，并等待消费者进程将数据读走，不断重复该操作。 2. 消费者进程生成两个子进程，用于显示缓冲区中的消息，两进程并发读取缓冲区消息后清空缓冲区。一个进程的两次显示操作间可加入适当延时以便观察。 3. 在两个独立的终端上分别运行生产者、消费者进程，展现其同步和通信功能，要求每次只有一个进程在操作缓冲区。 4. 运行程序，记录操作过程，并给出文字说明。   **2.2.2 伪代码**  信号量：   |  | | --- | | Semaphore mutex = 1; // 临界互斥信号量  Semaphore empty = n; // 空闲缓冲区  Semaphore full = 0; // 缓冲区初始为空 |   生产者（Producer）：   |  | | --- | | Producer() {  while (1) {  生产数据;  P(empty); // 获取空缓冲区  P(mutex); // 进入临界区  将数据放入缓冲区;  V(mutex); // 退出临界区  V(full); // 满缓冲区数 + 1  }  } |   消费者（Consumer）：   |  | | --- | | Consumer() {  while (1) {  P(full); // 获取满缓冲区  P(mutex); // 进入临界区  从缓冲区中读数据;  V(mutex); // 退出临界区  V(empty); // 空缓冲区数 + 1  消费数据;  }  } |   **2.2.3 C语言实现**  本次实验采用进程级的共享内存文件和POSIX无名信号量实现。  头文件与结构体（SharedMemory.h）：   |  | | --- | | #include <fcntl.h>  #include <sys/stat.h>  #include <semaphore.h>  #include <unistd.h>  #include <stdlib.h>  #include <stdio.h>  #include <stdbool.h>  #include <string.h>  #include <sys/shm.h>  #include <sys/mman.h>  enum {  BUFFER\_SIZE = 100  };  struct SharedMemory {  char data[BUFFER\_SIZE];  sem\_t mutex; // 互斥信号量  sem\_t empty; // 空缓冲区数  sem\_t full; // 满缓冲区数  }; |   生产者（producer.c）：其中使用 fgets() 函数、strncpy() 函数以避免安全问题。   |  | | --- | | #include "SharedMemory.h"  char line[BUFFER\_SIZE];  int main() {  // 创建或打开共享内存文件  int shm\_fd = shm\_open("exp2", O\_CREAT | O\_RDWR, 0666); // 创建、所有人可读可写  if (shm\_fd < 0) {  printf("Failed to create shared memory.\n");  exit(-1);  }  printf("Successfully created shared memory: %d\n", shm\_fd);  // 修改共享内存文件大小  if (ftruncate(shm\_fd, sizeof(struct SharedMemory)) == -1) {  printf("Failed to set the size of the shared memory.\n");  exit(-1);  }  // 将共享内存文件映射到内存  struct SharedMemory\* sm = (struct SharedMemory\*)mmap(  NULL, // 要将文件映射到的地址, 传 NULL 由 Linux 内核指定  sizeof(struct SharedMemory), // 文件长度  PROT\_READ | PROT\_WRITE, // 映射的内存的权限  MAP\_SHARED, // 建立共享, 用于进程间通信  shm\_fd, // 文件描述符  0 // 偏移量  );    if (sm == MAP\_FAILED) {  printf("Failed to map the shared memory.\n");  exit(-1);  }  // 初始化三个信号量  if (sem\_init(&sm->mutex, 1, 1) == -1) { // mutex = 1  printf("Failed to initialize semaphore mutex.\n");  exit(-1);  }  if (sem\_init(&sm->empty, 1, BUFFER\_SIZE) == -1) { // empty = n  printf("Failed to initialize semaphore empty.\n");  exit(-1);  }  if (sem\_init(&sm->full, 1, 0) == -1) { // full = 1  printf("Failed to initialize semaphore full.\n");  exit(-1);  }  while (1) {  printf("[2021192010@Producer ~]$ ");  // gets(line); // 忽略缓冲区溢出  fgets(line, BUFFER\_SIZE, stdin);  // 获取空缓冲区  sem\_wait(&sm->empty); // P(empty)  // 进入临界区  sem\_wait(&sm->mutex); // P(mutex)  // 将行写入缓冲区  // strcpy(sm->data, line); // 忽略缓冲区溢出  strncpy(sm->data, line, BUFFER\_SIZE);  // 退出临界区  sem\_post(&sm->mutex); // V(mutex)  // 满缓冲区数 + 1  sem\_post(&sm->full); // V(full)  // 退出  // if (!strcmp(line, "quit")) {  if (!strcmp(line, "quit\n")) {  break;  }  }  // 撤销信号量  sem\_destroy(&sm->mutex);  sem\_destroy(&sm->empty);  sem\_destroy(&sm->full);  // 解除共享内存文件到内存的映射  munmap(sm, sizeof(struct SharedMemory));  // 关闭共享内存文件  close(shm\_fd);  // 删除共享内存文件  shm\_unlink("exp2");  return 0;  } |   消费者（consumer.c）：注意生产者输入“quit”时实际上是两个消费者进程竞争，收到“quit”的消费者进程能顺利退出，但另一进程会阻塞。为解决该问题，可在收到“quit”时不清空缓冲区，同时保持 full 的值为1，empty的值为0。即：假装消费一下。   |  | | --- | | #include "SharedMemory.h"  char line[BUFFER\_SIZE];  int main() {  // 创建或打开共享内存文件  int shm\_fd = shm\_open("/exp2", O\_RDWR, 0666); // 创建、所有人可读可写  if (shm\_fd == -1) {  printf("Failed to create shared memory.\n");  exit(-1);  }  // 将共享内存文件映射到内存  struct SharedMemory\* sm = (struct SharedMemory\*)mmap(  NULL, // 要将文件映射到的地址, 传 NULL 由 Linux 内核指定  sizeof(struct SharedMemory), // 文件长度  PROT\_READ | PROT\_WRITE, // 映射的内存的权限  MAP\_SHARED, // 建立共享, 用于进程间通信  shm\_fd, // 文件描述符  0 // 偏移量  );  if (sm == MAP\_FAILED) {  printf("Failed to map the shared memory.\n");  exit(-1);  }  // 创建子进程  pid\_t pid = fork();  if (pid < 0) {  printf("Failed to fork a child process.\n");  exit(-1);  }  else if (pid == 0) { // 子进程  while (1) {  // 获取满缓冲区  sem\_wait(&sm->full); // P(full)  // 进入临界区  sem\_wait(&sm->mutex); // P(mutex)  // 检查是否为 quit  bool quit = false;  if (!strcmp(sm->data, "quit\n")) {  quit = true;  }    // 消费缓冲区数据  printf("[2021192010@Consumer\_child ~]$ Received data: %s", sm->data);  if (!quit) {  // 清空缓冲区  memset(sm->data, 0, sizeof(sm->data));  }  // 退出临界区  sem\_post(&sm->mutex); // V(mutex)  // 退出  if (quit) {  // 满缓冲区数 + 1  sem\_post(&sm->full); // V(full)  break;  }  // 空缓冲区数 + 1  sem\_post(&sm->empty); // V(empty)  // 加入延时以便观察  sleep(3);  }  }  else { // 父进程  while (1) {  // 获取满缓冲区  sem\_wait(&sm->full); // P(full)  // 进入临界区  sem\_wait(&sm->mutex); // P(mutex)  // 检查是否为 quit  bool quit = false;  if (!strcmp(sm->data, "quit\n")) {  quit = true;  }  // 消费缓冲区数据  printf("[2021192010@Consumer\_parent ~]$ Received data: %s", sm->data);  if (!quit) {  // 清空缓冲区  memset(sm->data, 0, sizeof(sm->data));  }  // 退出临界区  sem\_post(&sm->mutex); // V(mutex)  // 退出  if (quit) {  // 满缓冲区数 + 1  sem\_post(&sm->full); // V(full)  break;  }  // 空缓冲区数 + 1  sem\_post(&sm->empty); // V(empty)  }  }  // 解除共享内存文件到内存的映射  munmap(sm, sizeof(struct SharedMemory));  // 关闭共享内存文件  close(shm\_fd);  return 0;  } |   **2.3 生产者-消费者问题演示**  **2.3.1 编译**  使用如下命令编译，注意编译时需要链接lrt库。   |  | | --- | | cd /home/hytidel/OSexp/exp2  gcc producer.c -o producer -lrt  gcc consumer.c -o consumer -lrt |   **2.3.2 运行**  在终端1运行生产者进程：   |  | | --- | | /home/hytidel/OSexp/exp2/producer |   运行结果如**图2.3.1**所示。  图2.3.1：生产者进程  在终端2运行消费者进程：   |  | | --- | | /home/hytidel/OSexp/exp2/consumer |   运行结果如**图2.3.2**所示。  图2.3.2：消费者进程  在终端3查看进程和状态：   |  | | --- | | ps aux |   运行结果如**图2.3.3**所示，观察到有1个生产者进程和2个消费者进程，它们都处于阻塞态（S+）。  图2.3.3：3个进程都处于阻塞态  在终端3查看共享内存：   |  | | --- | | ipcs -m |   运行结果如**图2.3.4**所示，没有观察到共享内存段。  图2.3.4：没有观察到共享内存段  检查 /dev/shm 目录，发现目录下确有 exp2 这个共享文件。如**图2.3.5**所示。这表明：说明 shm\_open() 创建的共享内存是进程级的，而非系统级的，故不会在系统的 IPC 资源中显示。  图2.3.5：/dev/shm目录下有共享文件exp2  下面会看到，进程级的共享文件也能用于进程通信。  **2.3.3 输入**  在生产者进程中多次输入除“quit”外的任意文本，观察到两个消费者进程并发地消费缓冲区中的数据。  在生产者进程中匀速地发送数据时，消费者进程消费数据的数量近似父进程：子进程 = 3：1，与子进程 sleep(3) 相符。如**图2.3.6**所示。  图2.3.6：进程通信  **2.3.4 退出**  在生产者进程中输入“quit”，观察到生产者进程和消费者进程都退出。如**图2.3.6**所示。  图2.3.6：生产者进程和消费者进程都退出  回到 /dev/shm 目录，发现共享文件 exp2 也被删除。如**图2.3.7**所示。  图2.3.7：共享文件 exp2 也被删除  **3. 实验总结**  **3.1 实验结论**  **3.1.1 Linux的进程通信**  Linux提供了多种进程间通信的方式，其中包括管道、消息队列和共享内存。  1. 管道（Pipe）：  （1）管道是一种半双工的通信方式，允许一个进程将其输出连接到另一个进程的输入。  （2）在Linux中，管道使用pipe()系统调用创建，通常用于在父子进程或者兄弟进程之间传递数据。  （3）管道的特点是数据单向流动，即一端写入，另一端读取，且具有固定的读写方向。  2. 消息队列（Message Queue）：  （1）消息队列是一种可以实现进程间异步通信的机制，它允许一个或多个进程向队列中发送消息，并由其他进程从队列中接收消息。  （2）在Linux中，消息队列使用msgget()、msgsnd()和msgrcv()等系统调用进行创建、发送和接收消息。  （3）消息队列的特点是消息具有优先级，可以按照先进先出（FIFO）的顺序处理。  3. 共享内存（Shared Memory）：  （1）共享内存是一种高效的进程间通信方式，它允许多个进程共享同一块内存区域，从而实现数据的共享和交换。  （2）在Linux中，共享内存通过shmget()、shmat()和shmdt()等系统调用来创建、连接和断开共享内存段。  （3）共享内存的特点是数据直接存储在共享内存区域中，进程可以直接读写共享内存，而无需进行数据的复制，因此效率较高。  这些进程通信方式各有特点，可以根据具体的需求选择合适的方式进行进程间通信。  **3.1.2 对比 shmget() 与 shm\_open()**  shmget() 和 shm\_open() 是用于创建共享内存的两种不同的系统调用，它们在使用方式和功能上有一些区别。  1. shmget()：  （1）shmget()是一个系统调用，用于在Linux系统级别创建共享内存段。  （2）创建的共享内存段是系统的IPC（Inter-Process Communication，进程间通信）资源，可以被系统管理和查询。  （3）可以使用ipcs命令查询和管理通过shmget()创建的共享内存段。  （4）在使用shmget()创建共享内存时，需要指定一个键值作为共享内存标识符，以便其他进程能够通过该标识符连接到同一块共享内存。  2. shm\_open()：  （1）shm\_open()是一个POSIX函数，用于在进程级别创建共享内存对象。  （2）创建的共享内存对象是与文件描述符相关联的，因此可以像文件一样通过路径进行标识和访问。  （3）由于共享内存对象是进程级别的，无法被ipcs命令查询或管理，它不属于系统的IPC资源。  （4）使用shm\_open()创建的共享内存对象可以被其他进程通过mmap()等方法映射到自己的地址空间，从而实现进程间的共享内存通信。  总的来说，shmget()和shm\_open()都可以用于创建共享内存，但是它们的使用方式和管理方式有所不同。shmget()创建的共享内存是系统级别的，可以被系统管理和查询；而shm\_open()创建的共享内存是进程级别的，无法被系统管理和查询，只能通过文件路径访问。  **3.2 实验心得**  通过本次实验，我深入学习了进程同步与通信的实现方法，加深了对Linux操作系统下进程与线程机制的理解。通过编写生产者和消费者程序，并使用POSIX信号量实现了它们之间的同步关系，我成功地完成了实验目标。这次实验不仅提高了我对操作系统原理的理解，也锻炼了我在Linux环境下进行编程的能力，为今后的学习和工作打下了良好的基础。 |

深圳大学学生实验报告用纸

|  |
| --- |
| 指导教师批阅意见：  成绩评定：  指导教师签字：张滇  2024年 4 月 30 日 |
| 备注： |

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后10日内。