|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 武汉大学国家网络安全学院教学实验报告 | | | | | |
| 课程名称 | 操作系统设计与实践 | | 实验日期 | | 2024.11.18 |
| 实验名称 | 进程间通信 | | 实验周次 | | 第十二周 |
| 姓名 | 学号 | | 专业 | | 班级 |
| 王亚鹏 | 2022302181161 | | 信息安全 | | 5班 |
| 杨依磊 | 2022302181159 | | 信息安全 | | 5班 |
| 杜泓波 | 2022302181162 | | 信息安全 | | 5班 |
| 侯名扬 | 2022302181165 | | 信息安全 | | 5班 |
| 1. 实验目的及实验内容   （要求掌握的知识；实验内容；原理分析） | | | | | |
| **实验目标：**  1、了解微内核架构和宏内核架构的差异  2、理解微内核架构中IPC的实现机理  3、掌握微内核架构中IPC的实现技巧  **实验内容：**  1. 验证IPC的实现机理  2. 学习分析IPC实现的技巧与细节  **原理分析：**   1. **阅读8.1节资料回答以下问题**   **① 微内核与宏内核在系统调用角度差异是什么？**  1. 系统调用处理流程：  宏内核 (Monolithic Kernel): 在宏内核中，所有的操作系统服务，包括文件系统、网络协议栈、设备驱动等，都运行在内核空间。当应用程序发起系统调用时，内核直接在内核空间内处理请求，无需进行进程间通信 (IPC)。整个处理流程都在内核态完成，效率相对较高。  微内核 (Microkernel): 微内核只包含最基本的功能，如进程调度、内存管理和基本的进程间通信。其他服务，如文件系统、网络协议栈等，都以用户态进程的形式运行在内核之外，称为服务器 (Servers)。当应用程序发起系统调用时，微内核将请求转发给相应的服务器进程。服务器处理完请求后，将结果通过 IPC 返回给微内核，再由微内核传递给应用程序。这意味着系统调用需要经过多次进程间通信，处理流程相对复杂。  2. 系统调用复杂性：  宏内核: 由于所有服务都在内核空间，系统调用处理相对简单直接。内核可以直接访问所有资源，无需进行复杂的IPC。  微内核: 系统调用需要经过微内核和多个服务器进程之间的IPC，处理流程复杂，增加了系统调用的开销。微内核需要维护 IPC 机制，并处理消息传递的细节。  3. 系统调用性能：  宏内核: 由于系统调用处理流程简单，直接在内核空间完成，宏内核的系统调用性能通常较高。但是，如果内核代码庞大且复杂，可能会导致性能下降。  微内核: 系统调用需要进行多次 IPC，增加了开销，导致性能相对较低。但是，由于微内核体积小，代码精简，可以减少内核本身的开销，并且可以通过优化 IPC 机制来提高性能。  4. 系统调用安全性：  宏内核: 所有服务都在内核空间运行，如果某个服务出现错误，可能会影响整个内核的稳定性，甚至导致系统崩溃。此外，由于内核代码庞大，更容易出现安全漏洞。  微内核: 服务以用户态进程的形式运行，即使某个服务出现错误，也不会影响内核的稳定性。微内核的体积小，代码精简，更容易进行安全审计和验证，安全性相对较高。  总结：   | 特性 | 宏内核 | 微内核 | | --- | --- | --- | | 系统调用处理流程 | 直接在内核空间处理 | 通过 IPC 转发给服务器进程处理 | | 系统调用复杂性 | 简单 | 复杂 | | 系统调用性能 | 通常较高 | 通常较低，但可以通过优化 IPC 提高 | | 系统调用安全性 | 相对较低，服务错误可能影响内核稳定性 | 相对较高，服务错误不会影响内核稳定性 |   需要注意的是， 微内核和宏内核并非绝对的二分法，而是两种设计理念的体现。现代操作系统通常采用混合内核的设计，结合了宏内核和微内核的优点。例如，Linux 内核虽然是宏内核，但可以通过模块化设计来提高灵活性和可维护性。  **② 我们之前的实验实现，更类似哪种架构？**  我们之前的实验是微内核的架构。通过内核代码中sys\_call函数负责处理系统调用和中断处理。sys\_call函数调用save函数来保存当前进程的状态，然后跳转到sys\_call\_table中的相应函数来处理系统调用或中断。save函数负责保存当前进程的状态，包括寄存器值和栈顶指针。  save函数：负责保存当前进程的状态。  sys\_call函数：负责处理系统调用和中断处理。  restart函数：负责重新启动当前进程。  exception\_handler函数：负责处理异常和中断。  **③ 主流桌面操作系统内核架构调研**  1. Windows  内核类型: 混合内核 (Hybrid Kernel)  架构细节:  NT 内核: Windows 的核心是 NT 内核，它是一个混合内核，结合了微内核和宏内核的特性。  微内核部分: NT 内核包含一个小的微内核，负责基本的功能，如线程调度、进程间通信 (IPC) 和内存管理。  宏内核部分: NT 内核的其余部分运行在内核模式，包括设备驱动程序、文件系统、网络协议栈等。这些模块直接运行在内核空间，可以提高性能。  执行体 (Executive): 位于微内核之上，提供了一系列系统服务，如 I/O 管理、对象管理、安全管理等。  硬件抽象层 (HAL): 位于内核和硬件之间，提供了一个抽象层，使得 Windows 可以运行在不同的硬件平台上。  驱动程序模型: Windows 使用驱动程序模型来管理硬件设备。驱动程序运行在内核模式，可以直接访问硬件。  优点:  性能: 混合内核结合了微内核和宏内核的优点，在性能和稳定性之间取得了平衡。  兼容性: Windows 支持大量的硬件设备和软件应用程序。  功能丰富: Windows 提供了丰富的功能，如图形用户界面、多媒体支持、网络功能等。  缺点:  安全性: 由于大部分代码运行在内核模式，如果驱动程序出现问题，可能会导致系统崩溃。  稳定性: 混合内核的复杂性可能会导致稳定性问题。  封闭性: Windows 是闭源操作系统，代码不可见，不利于定制和修改。  2. Linux  内核类型: 单内核 (Monolithic Kernel)  架构细节:  单一内核镜像:Linux 内核是一个单一的二进制镜像文件，包含了所有内核功能。  模块化设计: Linux 内核采用模块化设计，可以将一些功能编译成模块，在需要时动态加载。  内核空间和用户空间: Linux 将系统分为内核空间和用户空间。内核运行在内核空间，应用程序运行在用户空间。  系统调用接口: 用户空间程序通过系统调用接口访问内核功能。  进程管理: Linux 使用进程来管理应用程序。每个进程都有自己的地址空间和资源。  文件系统: Linux 支持多种文件系统，如 ext4、XFS、Btrfs 等。  设备驱动程序: Linux 使用设备驱动程序来管理硬件设备。驱动程序运行在内核空间。  优点:  性能: 单内核结构可以提高系统性能，因为所有内核功能都运行在同一个地址空间。  开源: Linux 是开源操作系统，代码可见，可以自由定制和修改。  灵活性: Linux 支持多种硬件平台和架构。  社区支持: Linux 拥有庞大的社区支持，可以快速解决问题。  缺点:  安全性: 如果内核出现问题，可能会导致整个系统崩溃。  复杂性: Linux 内核代码庞大复杂，维护难度较高。  驱动程序兼容性: 部分硬件设备的 Linux 驱动程序可能不够完善。  3. macOS  内核类型: 混合内核 (Hybrid Kernel) - XNU  架构细节:  XNU 内核: macOS 的核心是 XNU 内核，它是一个混合内核，结合了 Mach 微内核和 BSD 内核的特性。  Mach 微内核: Mach 微内核负责基本的功能，如进程管理、内存管理和消息传递。  BSD 内核: BSD 内核提供了 Unix API、网络协议栈、文件系统等功能。  I/O Kit: macOS 使用 I/O Kit 来管理硬件设备。I/O Kit 是一个面向对象的驱动程序框架。  驱动程序模型: macOS 的驱动程序运行在内核模式，可以通过 I/O Kit 访问硬件。  优点:  稳定性: 混合内核结合了微内核和宏内核的优点，提高了系统的稳定性。  性能: macOS 在性能和稳定性之间取得了平衡。  安全性: macOS 的安全性较高，具有沙盒机制和代码签名等安全特性。  用户体验: macOS 提供了良好的用户体验，界面美观，操作流畅。  缺点:  封闭性: macOS 是闭源操作系统，代码不可见，不利于定制和修改。  硬件限制: macOS 只能运行在苹果 | | | | | |
| 1. 实验环境及实验步骤   （本次实验所使用的器件、仪器设备等的情况；具体实验步骤） | | | | | |
| 实验环境：  **X86 32位Linux环境**  实验步骤：  task\_sys()是一个系统任务，运行在Ring 1保护级别（x86架构中，Ring 1 的权限低于 Ring 0，但高于用户级别的 Ring 3）  PUBLIC void task\_sys()  {  MESSAGE msg;  while (1) {  send\_recv(RECEIVE, ANY, &msg);  int src = msg.source;  switch (msg.type) {  case GET\_TICKS:  msg.RETVAL = ticks;  send\_recv(SEND, src, &msg);  break;  default:  panic("unknown msg type");  break;  }  }  }  接收消息：send\_recv(RECEIVE, ANY, &msg);  系统任务调用 send\_recv()，使用 RECEIVE 操作，等待接收来自任意进程的消息。  消息被存储在 msg 结构体中，供后续处理。  处理消息：int src = msg.source;  提取消息的来源进程 ID，存储在变量 src 中。后续需要用它来回复消息。  switch (msg.type) {  检查消息的类型（msg.type），决定如何处理。  处理 GET\_TICKS 消息  case GET\_TICKS:  msg.RETVAL = ticks;  send\_recv(SEND, src, &msg);  break;  如果消息类型是 GET\_TICKS：  将当前的时钟滴答数 ticks（一个全局变量）存储到消息的 RETVAL 字段中。  通过send\_recv(SEND, src, &msg)将消息发送回来源进程（src），完成回复。  处理未知消息类型  default:  panic("unknown msg type");  break;  如果收到未知类型的消息：  调用panic()函数（可能是打印错误信息并停止系统），表明发生了严重的错误。  这个分支是为了防止系统接收到未知消息类型后发生不可预知的行为。  无限循环  整个task\_sys函数是一个无限循环：  while (1) {  ...  }  这保证了task\_sys始终在后台运行，随时响应其他进程的请求。        sendrec:  mov eax, \_NR\_sendrec; 系统调用号，放入 eax 寄存器  Mov ebx, [esp + 4]从栈中取第一个参数 function，放入 ebx 寄存器  mov ecx,[esp + 8];从栈中取第二个参数 src\_dest，放入 ecx 寄存器  mov edx,[esp + 12] ; 从栈中取第三个参数 p\_msg，放入 edx 寄存器  Int INT\_VECTOR\_SYS\_CALL ; 触发中断，执行系统调用  ret ; 返回调用者  功能：  这是一个封装的系统调用，用于进程间通信（发送和接收消息）。  调用时会将 function（操作类型）、src\_dest（消息目标或来源）、以及 p\_msg（指向消息结构的指针）传递给内核。  内核处理完成后，可能通过寄存器返回结果。  参数传递：  函数的参数通过栈传递。调用者会将参数按顺序压入栈中：  function（第一个参数）在 [esp + 4]。  src\_dest（第二个参数）在 [esp + 8]。  p\_msg（第三个参数）在 [esp + 12]。  这些参数被依次加载到寄存器 ebx, ecx, edx 中。  系统调用号：  \_NR\_sendrec 的值是 1，表示这是系统调用编号为 1 的功能。  mov eax, \_NR\_sendrec 将系统调用号放入 eax 寄存器，这是系统调用约定的一部分。  IMG_256 1. ldt\_seg\_linear功能：计算某个进程的线性地址。 该函数根据进程的段描述符（ldt）中的段基址信息（高、中、低部分）计算出该进程指定段的线性地址。  p 是进程指针，idx 是指定的段索引。  通过对段基址进行移位和按位“或”运算，最终得到完整的线性地址。  PUBLIC int ldt\_seg\_linear(struct proc\* p, int idx) {  struct descriptor \*d = &p->ldts[idx];  return d->base\_high << 24 | d->base\_mid << 16 | d->base\_low;  } 2. va2la功能：将虚拟地址转换为线性地址。 该函数通过调用 ldt\_seg\_linear 来获得进程的线性地址，并与给定的虚拟地址相加，得到线性地址。  如果进程 ID 小于任务数和进程数，则会进行断言，确保虚拟地址和线性地址相等。  PUBLIC void\* va2la(int pid, void\* va) {  struct proc\* p = &proc\_table[pid];  u32 seg\_base = ldt\_seg\_linear(p, INDEX\_LDT\_RW);  u32 la = seg\_base + (u32)va;  if (pid < NR\_TASKS + NR\_PROCS) {  assert(la == (u32)va);  }  return (void\*)la;  } 3. reset\_msg功能：清空消息内容。 通过 memset 函数将传入的消息对象清空。  PUBLIC void reset\_msg(MESSAGE\* p) {  memset(p, 0, sizeof(MESSAGE));  } 4. block功能：将进程阻塞。 设置进程的 p\_flags，然后调用 schedule() 调度函数，选择其他进程运行。  注意：block 函数本身并不修改 p\_flags，它只是在调用时需要确保 p\_flags 已经被设置。  PRIVATE void block(struct proc\* p) {  assert(p->p\_flags);  schedule();  } 5. unblock功能：解阻塞进程。 该函数是一个“空操作”函数，实际上并没有做任何事情。调用时需要确保进程的 p\_flags 已经被清除。  PRIVATE void unblock(struct proc\* p) {  assert(p->p\_flags == 0);  } 6. deadlock功能：检查是否存在死锁。 该函数检查进程之间是否存在消息传递的死锁。若存在死锁，返回 1，否则返回 0。  它通过检查消息传递图中是否存在环来判断是否发生死锁。若从目标进程到源进程之间存在环，说明发生了死锁。  PRIVATE int deadlock(int src, int dest) {  struct proc\* p = proc\_table + dest;  while (1) {  if (p->p\_flags & SENDING) {  if (p->p\_sendto == src) {  // 打印链  p = proc\_table + dest;  printl("=\_=%s", p->name);  do {  assert(p->p\_msg);  p = proc\_table + p->p\_sendto;  printl("->%s", p->name);  } while (p != proc\_table + src);  printl("=\_=");  return 1;  }  p = proc\_table + p->p\_sendto;  } else {  break;  }  }  return 0;  } 7. msg\_send功能：发送消息。 该函数用于进程间的消息发送。如果目标进程正在接收消息并且是等待当前发送者的消息，消息将直接发送并解阻塞目标进程。  如果目标进程没有准备好接收消息，则发送者会被阻塞，并加入目标进程的发送队列，直到目标进程准备好接收消息。  PRIVATE int msg\_send(struct proc\* current, int dest, MESSAGE\* m) {  struct proc\* sender = current;  struct proc\* p\_dest = proc\_table + dest;  assert(proc2pid(sender) != dest);  // 检查死锁  if (deadlock(proc2pid(sender), dest)) {  panic(">>DEADLOCK<<%s->%s", sender->name, p\_dest->name);  }  if ((p\_dest->p\_flags & RECEIVING) &&  (p\_dest->p\_recvfrom == proc2pid(sender) || p\_dest->p\_recvfrom == ANY)) {  // 目标进程正在接收消息  assert(p\_dest->p\_msg);  assert(m);    // 将消息从发送者复制到接收者  phys\_copy(va2la(dest, p\_dest->p\_msg), va2la(proc2pid(sender), m), sizeof(MESSAGE));  p\_dest->p\_msg = 0;  p\_dest->p\_flags &= ~RECEIVING;  p\_dest->p\_recvfrom = NO\_TASK;  unblock(p\_dest);  } else {  // 目标进程未准备好接收消息  sender->p\_flags |= SENDING;  sender->p\_sendto = dest;  sender->p\_msg = m;  // 将发送者加入目标进程的发送队列  struct proc \*p;  if (p\_dest->q\_sending) {  p = p\_dest->q\_sending;  while (p->next\_sending)  p = p->next\_sending;  p->next\_sending = sender;  } else {  p\_dest->q\_sending = sender;  }  sender->next\_sending = 0;  block(sender);  }  return 0;  } 8. msg\_receive功能：接收消息。  * 该函数用于接收来自某个进程或任何进程的消息。如果消息队列中已有等待的消息，接收者会将消息从发送者复制过来并解除发送者的阻塞。 * 如果接收者没有消息可接收，它会被阻塞，直到消息到达。   PRIVATE int msg\_receive(struct proc\* current, int src, MESSAGE\* m) {  struct proc\* p\_who\_wanna\_recv = current;  struct proc\* p\_from = 0;  struct proc\* prev = 0;  int copyok = 0;  assert(proc2pid(p\_who\_wanna\_recv) != src);  if ((p\_who\_wanna\_recv->has\_int\_msg) &&  ((src == ANY) || (src == INTERRUPT))) {  // 处理中断消息  MESSAGE msg;  reset\_msg(&msg);  msg.source = INTERRUPT;  msg.type = HARD\_INT;  assert(m);  phys\_copy(va2la(proc2pid(p\_who\_wanna\_recv), m), &msg, sizeof(MESSAGE));  p\_who\_wanna\_recv->has\_int\_msg = 0;  return 0;  }  if (src == ANY) {  // 接收来自任何进程的消息  if (p\_who\_wanna\_recv->q\_sending) {  p\_from = p\_who\_wanna\_recv->q\_sending;  copyok = 1;  }  } else {  // 接收来自特定进程的消息  p\_from = &proc\_table[src];  if ((p\_from->p\_flags & SENDING) &&  (p\_from->p\_sendto == proc2pid(p\_who\_wanna\_recv))) {  copyok = 1;  }  }  if (copyok) {  // 从消息发送者复制消息  if (p\_from == p\_who\_wanna\_recv->q\_sending) {  p\_who\_wanna\_recv->q\_sending = p\_from->next\_sending;  p\_from->next\_sending = 0;  } else {  prev->next\_sending = p\_from->next\_sending;  p\_from->next\_sending = 0;  }  assert(m);  assert(p\_from->p\_msg);  phys\_copy(va2la(proc2pid(p\_who\_wanna\_recv), m),  va2la(proc2pid(p\_from), p\_from->p\_msg), sizeof(MESSAGE));  p\_from->p\_msg = 0;  p\_from->p\_sendto = NO\_TASK;  p\_from->p\_flags &= ~SENDING;  unblock(p\_from);  } else {  // 如果没有可接收的消息，接收者被阻塞  p\_who\_wanna\_recv->p\_flags |= RECEIVING;  p\_who\_wanna\_recv->p\_msg = m;  if (src == ANY) | | | | | |
| 1. 实验过程分析   （详细记录实验过程中发生的故障和问题，进行故障分析，说明故障排除的过程及方法。根据具体实验，记录、整理相应的数据表格等） | | | | | |
| ③ 程序与中断事件的并发保护  当程序与中断事件并发时，代码采用以下保护措施：  中断屏蔽：  在消息传递的关键操作（如发送、接收队列的修改）前禁用中断，确保操作的原子性。  在操作完成后恢复中断。  状态标志：  使用p\_flags标志位标记进程状态（如SENDING、RECEIVING）。  中断处理程序检查并避免对关键状态的干扰。  同步机制：  中断处理程序通过特定机制（如消息传递或标志位）与程序同步，避免资源竞争。  ④ assert 和 panic 的实现与验证  实现机制：  assert：  检查程序中的条件是否满足，不满足则打印错误并触发panic。  实现方式：宏定义结合条件检查。  #define assert(expr)  if (!(expr)) {  printf("Assertion failed: %s, file: %s, line: %d\n",  #expr, \_\_FILE\_\_, \_\_LINE\_\_);  panic("Assertion failed");  }  panic：  触发系统的严重错误处理机制，打印错误信息并停止系统。  void panic(const char\* msg) {  printf("PANIC: %s\n", msg);  while (1); // 死循环停止系统  }  验证示例：  #include <stdio.h>#define assert(expr)  if (!(expr))  printf("Assertion failed: %s, file: %s, line: %d\n",  #expr, \_\_FILE\_\_, \_\_LINE\_\_);  panic("Assertion failed");  }  void panic(const char\* msg) {  printf("PANIC: %s\n", msg);  while (1);  }  int main() {  int a = 10, b = 0;  assert(b != 0); // 验证除零错误  printf("Result: %d\n", a / b);  return 0;  }  ⑤ 消息机制的通信与调度管理  通信机制：  消息队列：  每个进程拥有发送队列和接收队列，消息通过队列传递。  msg\_send函数将消息加入接收方队列，msg\_receive函数从队列中提取消息。  同步标志：  使用标志位（p\_flags）同步进程状态。  状态包括SENDING、RECEIVING，避免冲突。  调度管理：  阻塞与解阻塞：  block函数阻塞当前进程，等待条件满足。  unblock函数解除进程阻塞，恢复运行。  调度策略：  schedule函数根据进程状态选择下一个可运行进程。  ⑥ 死锁问题的解决与改进  现有解决方案：  在msg\_send函数中调用deadlock检测循环等待。  deadlock函数遍历消息链路，判断是否存在环（如 A -> B -> C -> A）。  可能的问题：  复杂链路可能导致效率低下。  未考虑优先级和资源分配策略。  改进方案：  更高效的检测算法：采用深度优先搜索（DFS）检测环。  超时机制：为每次通信设置超时时间，避免长期等待。  ⑦ 基于IPC扩展get\_ticks的方法  现有get\_ticks机制：  内核通过系统调用直接返回时钟中断计数。  扩展方法：  增加时间服务器：  创建一个专门的时间服务器进程，负责处理时钟中断。  用户进程通过消息向时间服务器请求时间。  实现示例：  // 时间服务器void time\_server() {  MESSAGE msg;  while (1) {  receive(ANY, &msg); // 接收请求  if (msg.type == GET\_TICKS) {  msg.RETVAL = ticks; // 返回时钟计数  send(msg.source, &msg); // 响应请求  }  }  }  // 用户进程请求时间void user\_process() {  MESSAGE msg;  msg.type = GET\_TICKS;  send(TIME\_SERVER\_PID, &msg); // 发送请求  receive(TIME\_SERVER\_PID, &msg); // 接收响应  printf("Current ticks: %d\n", msg.RETVAL);  }  优势：  提高扩展性：可进一步扩展获取系统时间、运行时间等功能。  集中管理：减少对全局变量ticks的直接访问。 | | | | | |
| 1. 实验结果总结   （对实验结果进行分析，完成思考题目，并提出实验的改进意见） | | | | | |
| **结果分析**：    进程TestA调用get\_ticks之后成功打印出了它们的值这表明消息机制工作良好  **问题描述：** 生产者进程负责生产数据并放入共享缓冲区，消费者进程从共享缓冲区取出数据进行处理。为了确保数据的一致性，需要在以下方面进行同步：  当缓冲区已满时，生产者应等待，直到消费者取走数据，腾出空间。  当缓冲区为空时，消费者应等待，直到生产者放入新的数据。  解决方案： 使用信号量（Semaphore）和互斥锁（Mutex）来实现生产者和消费者之间的同步与互斥。具体步骤如下：  定义信号量和互斥锁：  mutex：互斥锁，确保对共享缓冲区的互斥访问。  empty：信号量，表示缓冲区中空闲位置的数量，初始值为缓冲区大小。  full：信号量，表示缓冲区中已占用位置的数量，初始值为0。  生产者流程：  等待empty信号量，确保有空闲位置。  获取mutex锁，进入临界区。  将数据放入缓冲区。  放mutex锁，离开临界区。  释放full信号量，表示新增一个已占用位置。  消费者流程：  等待full信号量，确保有数据可取。  获取mutex锁，进入临界区。  从缓冲区取出数据。  释放mutex锁，离开临界区。  释放empty信号量，表示新增一个空闲位置。  **示例代码：** 以下是使用POSIX线程和信号量实现生产者-消费者问题的示例代码：  #include <pthread.h>#include <semaphore.h>#include <stdio.h>#include <stdlib.h>  #define BUFFER\_SIZE 5  int buffer[BUFFER\_SIZE];int in = 0;int out = 0;  sem\_t empty;sem\_t full;pthread\_mutex\_t mutex;  void\* producer(void\* arg) {  int item;  while (1) {  item = rand() % 100; // 生产一个随机数  sem\_wait(&empty); // 等待空闲位置  pthread\_mutex\_lock(&mutex); // 进入临界区  buffer[in] = item;  printf("生产者生产了数据 %d\n", item);  in = (in + 1) % BUFFER\_SIZE;  pthread\_mutex\_unlock(&mutex); // 离开临界区  sem\_post(&full); // 增加已占用位置计数  sleep(1);  }  }  void\* consumer(void\* arg) {  int item;  while (1) {  sem\_wait(&full); // 等待有数据  pthread\_mutex\_lock(&mutex); // 进入临界区  item = buffer[out];  printf("消费者消费了数据 %d\n", item);  out = (out + 1) % BUFFER\_SIZE;  pthread\_mutex\_unlock(&mutex); // 离开临界区  sem\_post(&empty); // 增加空闲位置计数  sleep(1);  }  }  int main() {  pthread\_t prod, cons;  sem\_init(&empty, 0, BUFFER\_SIZE);  sem\_init(&full, 0, 0);  pthread\_mutex\_init(&mutex, NULL);  pthread\_create(&prod, NULL, producer, NULL);  pthread\_create(&cons, NULL, consumer, NULL);  pthread\_join(prod, NULL);  pthread\_join(cons, NULL);  sem\_destroy(&empty);  sem\_destroy(&full);  pthread\_mutex\_destroy(&mutex);  return 0;  }  **说明：**   * sem\_init用于初始化信号量，sem\_wait和sem\_post分别用于等待和释放信号量。 * pthread\_mutex\_init用于初始化互斥锁，pthread\_mutex\_lock和pthread\_mutex\_unlock分别用于加锁和解锁。 * pthread\_create用于创建线程，pthread\_join用于等待线程结束。   通过上述实现，生产者和消费者可以在并发环境下安全地访问共享缓冲区，避免数据竞争和资源冲突。 | | | | | |
| 1. 各人实验贡献与体会（每人各自撰写） | | | | | |
| 实际上每个人都参与讨论，共同完成了本次实验  1.王亚鹏：全程参与实验  **（1）不同内核之间的差异**   **宏内核**通过将所有操作系统服务集成在内核空间，提供了高效的服务调用机制，但模块间耦合较高，潜在安全隐患。   **微内核**仅保留进程调度、内存管理和基本IPC功能，将其他服务以用户态运行的方式实现，从而提升了系统稳定性和安全性，但通信效率成为主要挑战。  **（2）实验中使用的技术**   **IPC优化技术**： 学习了**零拷贝技术**、快速路径优化等技术如何在实际应用中提升通信效率。零拷贝通过共享内存减少数据传输的拷贝开销，而快速路径优化则为常见的通信模式提供高效路径，显著提升了性能   **死锁检测与解决**： 通过死锁检测函数deadlock验证进程间通信的正确性，理解消息传递中的环路问题。同时学习了在消息传递中通过引入超时机制和优先级调度来减少死锁问题的实际改进方法   **多任务并发保护**： 学会了如何通过中断屏蔽、标志位同步和调度机制来实现多任务间的并发保护，确保系统的稳定运行  **（3）实验中的问题和动手做**   **在send\_recv函数中**，通过调试明确参数传递和中断触发的具体细节，明白了函数的运行方式   使用实际案例验证**assert和panic的功能**   查找资料，学习了**get\_ticks**的扩展方法  2.杨依磊：全程参与实验  1.微内核与宏内核架构宏内核架构将所有功能集成在内核空间中，虽然性能较高，但模块间耦合度高，安全性较低。而微内核架构仅保留核心功能，将其他服务移至用户空间，提高了安全性和稳定性。我通过实验，验证了这两种架构在系统调用处理流程、复杂性、性能和安全性上的差异。2.IPC机制的实现我在实验过程中进行了消息传递的机制，包括发送消息、接收消息以及同步与调度的实现。在实验中，我用标志位记录进程状态，block和unblock函数实现进程的阻塞与解阻塞操作 ，通过deadlock函数检测循环依赖，避免了死锁问题。代码实现与功能扩展3.通过分析示例代码，我了解了以下关键功能的实现方法：sys\_sendrec函数实现了消息的发送与接收，以及对目标进程状态的判断和消息队列的操作。代码用assert与panic机制进行异常处理。在get\_ticks扩展中，代码通过时间服务器的设计实现了基于IPC扩展功能。4.实验过程中遇到了一些问题：1）并发保护：中断事件与进程程序同时操作消息队列时发生了冲突。我通过在关键操作前屏蔽中断、恢复后再开启中断，以及使用标志位确保消息的一致性，成功解决了该问题。2）死锁现象：测试时观察到某些进程在发送消息后未能解阻塞。我通过检查deadlock函数逻辑，发现问题出在消息链路未正确清理，修复后解决了此问题。  3.杜泓波：全程参与实验  通过本次实验，我系统性地理解了微内核和宏内核架构的核心差异，深入探讨了微内核中进程间通信（IPC）的实现机理，并学习了优化IPC性能的多种技巧。宏内核架构通过将大部分操作系统服务集成到内核空间，提供高效的服务调用机制，但由于模块间耦合度高，安全性和稳定性受到挑战。而微内核架构通过模块化设计，将非核心功能移至用户空间，仅保留进程调度、内存管理和IPC等核心功能，从而实现了更高的安全性和系统稳定性，但需要通过优化IPC机制克服性能损耗。  在实验中，我验证了微内核中IPC的实现原理，包括同步通信和异步通信的模型，消息传递的实现方式，以及使用共享内存进行大规模数据交换的策略。我还通过实际代码分析，学习了如零拷贝技术、快速路径优化和轻量级消息传递等技巧。零拷贝技术通过共享内存或页面映射减少数据拷贝的开销，快速路径优化则为常见的通信模式（如请求-响应）提供了高效的执行路径。  此外，通过实验，我深刻认识到上下文切换与调度对IPC性能的影响，理解了如何通过减少系统调用和内核用户态切换来提升通信效率。本次实验不仅让我加深了对微内核架构设计理念的理解，还让我意识到其在实际应用场景中的挑战与优势。总的来说，微内核架构在安全性和稳定性上的优越性使其成为现代操作系统发展的重要方向，而高效的IPC机制是其成功的关键所在。此次实验为我进一步研究操作系统优化及分布式系统设计奠定了坚实的基础。  4.侯名扬：全程参与实验  我了解了微内核与宏内核在系统调用角度差异：在宏内核中，所有的操作系统服务，包括文件系统、网络协议栈、设备驱动等，都运行在内核空间。微内核只包含最基本的功能，如进程调度、内存管理和基本的进程间通信。其他服务，如文件系统、网络协议栈等，都以用户态进程的形式运行在内核之外，称为服务器 (Servers)。我们之前的实验是微内核的架构。通过内核代码中sys\_call函数负责处理系统调用和中断处理。sys\_call函数调用save函数来保存当前进程的状态，然后跳转到sys\_call\_table中的相应函数来处理系统调用或中断。save函数负责保存当前进程的状态，包括寄存器值和栈顶指针。  其次，我学习了如零拷贝技术、快速路径优化和轻量级消息传递等技术在实际应用场景中的挑战与优势。总的来说，微内核架构在安全性和稳定性上的优越性使其成为现代操作系统发展的重要方向，而高效的IPC机制是其成功的关键所在。此次实验为我进一步研究操作系统优化及分布式系统设计奠定了坚实的基础。 | | | | | |
| 1. 教师评语   （实验报告的考评：依据实验内容完整度、实验步骤清晰度、实验结果与分析正确性、实验心得与思考的全面性、实验报告文档的规范性等五个维度综合考评）   |  |  | | --- | --- | | 85-100 | * 实验内容完整或者有超出课程实验大纲的内容； * 实验步骤详尽，能够体现完整的实验过程； * 实验结果正确且实验数据分析得当； * 实验心得与思考全面并且有自己的独立思考； * 实验报告文档规范、排版整齐。 | | 75-84 | * 实验内容较为完整； * 实验步骤较为详尽，能够体现实验过程； * 实验结果正确且实验数据分析较为得当； * 实验心得与思考全面； * 实验报告文档规范、排版较为整齐。 | | 60-74 | * 实验内容有缺失； * 实验步骤不够详尽，不能够体现完整的实验过程； * 实验结果部分正确； * 实验心得与思考无或者不够深入； * 实验报告文档规范性有待增强。 | | 60以下 | * 实验内容严重缺失、实验态度不够端正 * 实验步骤不够详尽，不能够体现完整的实验过程； * 实验结果部分正确； * 实验心得与思考无或者不够深入； * 实验报告文档规范性有待增强。 | | | | | | |
|  | | | | | |
| **教师评分（请填写好姓名、学号）** | | | | | |
| 姓名 | | 学号 | | 分数 | |
| 王亚鹏 | | 2022302181161 | |  | |
| 杨依磊 | | 2022302181159 | |  | |
| 杜泓波 | | 20223021811 | |  | |
| 侯名扬 | | 20223021811 | |  | |
| 教师签名：  年 月 日 | | | | | |