

# 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	11月20日
实验名称	进程间通信	实验周次	第11周
姓名	学号	专业	班级
黄东威	2022302181148	信息安全	5班
王浚杰	2022302181143	网络空间安全	5班
程序	2022302181131	信息安全	4班
周业营	2022302181145	信息安全	5班

# Contents

<b>1 实验目的及实验内容</b>	<b>4</b>
1.1 实验目的	4
1.2 实验内容	4
1.3 本次实验要解决的问题	4
<b>2 实验环境</b>	<b>4</b>
<b>3 实验步骤</b>	<b>5</b>
3.1 验证 IPC 的实现机理	5
3.2 学习分析 IPC 实现的技巧和细节	5
<b>4 实验过程分析</b>	<b>5</b>
4.1 微内核与宏内核在系统调用角度差异是什么	5
4.1.1 微内核	5
4.1.2 宏内核	6
4.2 我们之前的实验实现，更类似哪种架构	6
4.3 调研一下，目前的主流桌面 OS，如 windows, linux, mac OS 都是怎样的内核架构	6
4.3.1 windows 的内核架构	6
4.3.2 Linux 的内核架构	7
4.3.3 Mac 的内核架构	9
4.4 画出一个逻辑关系图，描述本章实验中 IPC 的实现框架机理，并加以文字解释，特别注意：处理器状态的切换，信息的流向	10
4.5 简要描述 8.2-8.5 涉及系统调用的流程与作用	10
4.6 在 8.2-8.5 代码中，当涉及程序与中断事件的并发时，是如何施加保护的	11
4.7 解释一下 assert、Panic 的实现过程（含涉及的系统调用机理），撰写几个小例子验证其作用	11
4.7.1 assert	11
4.7.2 panic	14
4.7.3 举例子进行测试	14
4.8 在本部分的消息机制中，如何实现通信的？对进程如何调度管理的？	16
4.8.1 通信实现	16
4.8.2 调度管理	23
4.9 死锁问题是如何解决的？是否存在问题，若有改进之，若无说明验证其正确性。	24
4.9.1 死锁检测逻辑	25
4.9.2 死锁处理	25
4.9.3 死锁检测正确性讨论	26
4.9.4 改进方案	26
4.10 简要分析基于 IPC，如何扩展 get_ticks 的方法	28
4.11 针对上学期学习的经典同步互斥问题，试着用 IPC 解决一例	30
<b>5 实验结果总结</b>	<b>33</b>
5.1 IPC	33
5.2 死锁	34
5.3 assert	34
5.4 panic	35

6	各人实验贡献与体会（每人各自撰写）	35
7	教师评语	36
8	教师评分（请填写好姓名、学号）	37

# 1 实验目的及实验内容

## 1.1 实验目的

- 了解微内核架构和宏内核架构的差异
- 理解微内核架构中 IPC 的实现机理
- 掌握微内核架构中 IPC 的实现技巧

## 1.2 实验内容

- 验证 IPC 的实现机理
- 学习分析 IPC 实现的技巧与细节

## 1.3 本次实验要解决的问题

1. 微内核与宏内核在系统调用角度差异是什么？
2. 我们之前的实验实现，更类似哪种架构？
3. 调研一下，目前的主流桌面 OS，如 windows, linux, mac OS 都是怎样的内核架构
4. 画出一个逻辑关系图，描述本章实验中 IPC 的实现框架机理，并加以文字解释，特别注意：处理器状态的切换，信息的流向
5. 简要描述该处涉及系统调用的流程与作用
6. 在该代码中，当涉及程序与中断事件的并发时，是如何施加保护的？
7. 解释一下 assert、Panic 的实现过程（含涉及的系统调用机理），撰写几个小例子验证其作用。
8. 在本部分的消息机制中，如何实现通信的？对进程如何调度管理的？
9. 死锁问题是如何解决的？是否存在问题，若有改进之，若无说明验证其正确性。
10. 简要分析基于 IPC，如何扩展 get\_ticks 的方法
11. 针对上学期学习的经典同步互斥问题，试着用 IPC 解决一例。

# 2 实验环境

- Windows Subsystem for Linux 2 (WSL 2)
- Ubuntu 20.04
- NASM 2.14.02
- Bochs 2.7
- Visual Studio Code (VSCode)

## 3 实验步骤

### 3.1 验证 IPC 的实现机理

IPC 通过统一的系统调用 sys\_sendrec 来实现。

- msg\_send 函数的实现流程：
  - 检查是否死锁
  - 如果目标程序正在处于接收消息状态，则复制发送者的消息到目标程序，设置目标程序相应状态，唤醒。
  - 否则（目标程序没有处于接收消息），则设置发送者进程为发送中，把消息挂载到发送者 proc 的发送队列中，阻塞发送进程。
- msg\_receive 函数的实现流程：
  - 如果有一个硬件中断消息，则把消息交给接收者
  - 如果试图接收任意进程的消息，那么就把第一个消息复制给他。
  - 如果试图接收某个进程的消息，则复制消息
  - 如果没有进程发送消息，则调用阻塞自己

### 3.2 学习分析 IPC 实现的技巧和细节

IPC 的实现过程中还用到了以下这些函数：

- ldt\_seg\_linear( ) 函数：每个进程都有自己的 LDT，位于进程表的中间，这个函数就是根据 LDT 中描述符的索引来求得描述符所指向的段的基地址。
- va2la( ) 用来由虚拟地址求线性地址，它用到了 ldt\_seg\_linear( )。
- reset\_msg( ) 函数：用于把一个消息的每个字节清零。
- block( ) 函数：阻塞一个进程。
- unblock( ) 函数：解除一个进程的阻塞。
- deadlock( ) 函数：简单地判断是否发生死锁。方法是判断消息的发送是否构成一个环，如果构成环则意味着发生死锁，比如 A 试图发消息给 B，同时 B 试图给 C，C 试图给 A 发消息，那么死锁就发生了，因为 A、B 和 C 三个进程都将无限等待下去。

## 4 实验过程分析

### 4.1 微内核与宏内核在系统调用角度差异是什么

#### 4.1.1 微内核

微内核则把操作系统的主要部分（包括设备驱动、文件系统、进程管理、内存管理等）放在内核地址空间之外的其他地方（进程），比如内存管理器 MM，文件系统 FS 等，而系统调用只有发送、接受消息以及一个 both 三个种类，在调用时内核只需要做好消息的来回发送和接收即可。这里内核就像一个消息中转站，连接了用户进程和负责完成具体功能的其他进程，而不需要承担更多更复杂的功能。系统调用的复杂功能模块实际发生在内核外的其他进程中，这些进程不断从内核中接收消息（消息源头是用户进程），据此得出需要完成的系统调用功能是什么，然后尽力完成。

### 4.1.2 宏内核

在宏内核中，整个操作系统是一个运行在核心态的单独的 a.out 文件，这个二进制文件包含进程管理、内存管理、文件系统以及其他。具体实例包括 UNIX、MS-DOS、VMS、MVS、OS/360、MULTICS 等。宏内核将操作系统的主要部分都放在内核空间中运行，内核中有大量系统调用功能模块，系统调用的功能实际发生在内核中，调用时直接使用嵌在内核中的功能模块。

## 4.2 我们之前的实验实现，更类似哪种架构

我认为我们之前的操作系统代码架构更类似于宏内核，之前我们的处理中断、异常请求和系统调用的功能都是内核的一部分。微内核通过消息在不同进程间传递信息，而我们的系统并未采用这种机制，而是直接在内核空间内调用系统函数来提供服务，这比较体现宏内核的特征。

## 4.3 调研一下，目前的主流桌面 OS，如 windows, linux, mac OS 都是怎样的内核架构

### 4.3.1 windows 的内核架构

不同的 Windows 版本的内核架构整体框架是相似的，因为微软的 Windows 操作系统始终基于 Windows NT 内核，但微软通过不断更新，优化了内核的性能、稳定性和安全性，以适应新硬件和新需求。特别是在内存管理、多处理器支持、虚拟化、图形子系统等方面，不同版本的内核实现有显著差异，但这些差异都是在 Windows NT 混合内核基础上不断演化和扩展的结果。

Windows 的内核架构基于混合内核设计，它结合了微内核和宏内核的特点，提供了性能与模块化设计之间的平衡。Windows 采用分层结构以实现高效管理硬件资源与提供操作系统功能，Windows NT 内核主要由以下几个核心模块组成。

1. 执行体模块：执行体模块是 Windows 内核的高级部分，负责实现大多数操作系统的核功能。

- 内存管理器 (MM)：负责管理虚拟内存和物理内存，提供分页 (Paging)、内存保护以及缓存功能。
- 进程和线程管理器 (PTM)：负责管理进程和线程，包括进程创建、终止、线程调度等。
- I/O 管理器 (I/O Manager)：负责处理所有输入输出操作，通过 I/O 请求包 (IRP) 在设备驱动程序之间传递请求。
- 安全参考监视器 (SRM)：负责访问控制、权限检查以及用户身份验证。
- 对象管理器 (OM)：负责管理所有内核对象（如文件、进程、线程等），提供统一的对象接口。
- 配置管理器 (CM)：管理 Windows 注册表。
- 电源管理器 (PM)：负责管理系统的电源状态（如休眠、唤醒、节能模式）。

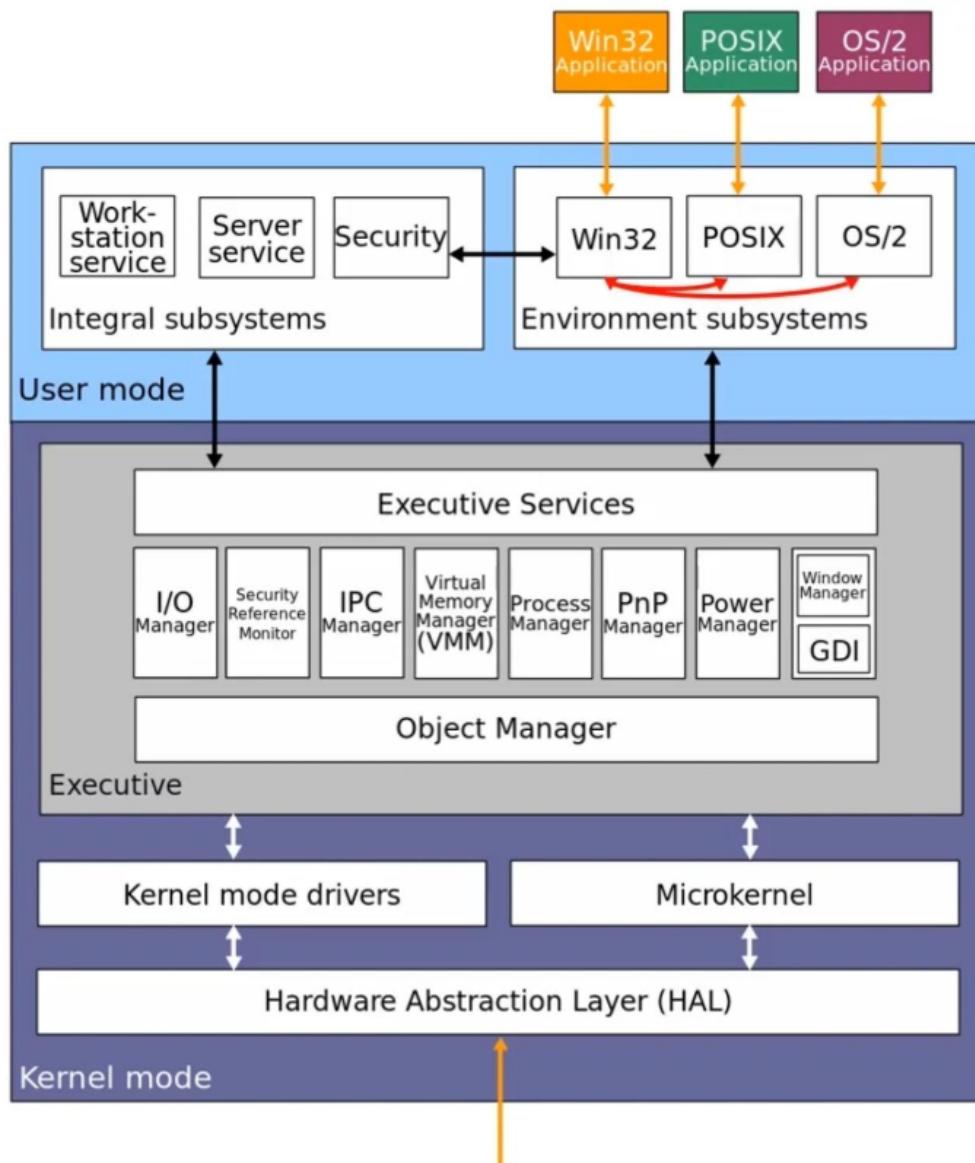
2. 微内核：微内核提供最基本的底层功能，直接与硬件交互。

负责中断和异常处理、线程调度与优先级管理、多处理器 (SMP) 支持、同步机制（如内核锁、信号量）

3. 硬件抽象层 (HAL): 抽象底层硬件的实现细节，提供统一的接口，使得内核和驱动程序无需关心具体硬件差异。能够屏蔽硬件的细节，例如中断控制、时钟管理和处理器通信并支持硬件平台的移植性。

4. 设备驱动程序 (Device Drivers): 驱动程序是内核的一部分，负责操作系统与硬件设备之间的交互。

windows 这样的混合内核架构的设计同时结合了宏内核（性能优先）和微内核（模块化、稳定性优先）的优点。同时，模块化的架构使得各功能模块独立设计，便于扩展和维护。除此之外，windows NT 内核提供了对对称多处理器 (SMP) 的支持，能够实现高效的多核性能，HAL 的抽象层设计使得 Windows 具备强大的硬件兼容性，能够兼容各种硬件平台。



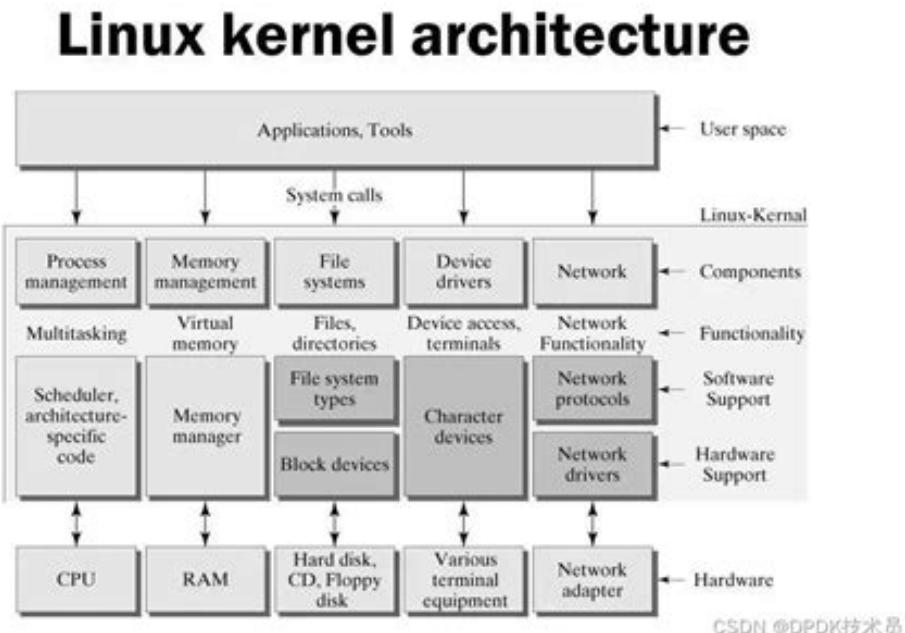
#### 4.3.2 Linux 的内核架构

Linux 内核采用了宏内核 (Monolithic Kernel) 设计，其特点是将所有核心功能模块集成在单一的内核中运行，都运行在内核的地址空间内，同时通过动态模块加载机制提供了一定程度的模块化支持。这种设计在性能和扩展性之间达成了平衡，使 Linux 内核在桌面、服务器、嵌入式设备等多种场景中得到了广泛的应用。

Linux 的内核架构主要分为以下几个核心模块：

1. 进程管理模块：负责对进程的生命周期进行管理，包括创建、调度、终止等。
  - 任务调度器：实现进程调度算法，例如完全公平调度器，确保多任务处理的效率。
  - 信号机制：提供进程间通信（IPC）和同步功能。
2. 内存管理模块：负责管理系统的内存资源。
  - 虚拟内存：支持分页、段式存储以及物理内存管理。
  - 内存分配器：通过 ‘kmalloc’ 等接口为内核分配内存。
  - 缓存管理：对文件系统数据和页面进行缓存，提升 I/O 性能。
3. 文件系统模块：提供对文件和存储设备的统一访问接口。
  - VFS（虚拟文件系统）：抽象具体文件系统的接口。
  - 支持多种文件系统：如 EXT4、XFS、Btrfs。
4. 网络协议栈：实现多层次的网络通信协议。
  - 支持协议：包括 TCP/IP、UDP 等主流网络协议。
  - 网络设备接口：通过 Netfilter 等机制支持防火墙和数据包过滤。
5. 设备驱动模块：通过标准化接口与硬件交互。
  - 字符设备、块设备和网络设备驱动。
  - 动态模块加载机制：支持驱动程序在运行时加载和卸载。

Linux 的宏内核设计在保持高性能的同时，通过动态模块加载、丰富的文件系统支持和可移植性，使其成为广泛使用的操作系统内核。



### 4.3.3 Mac 的内核架构

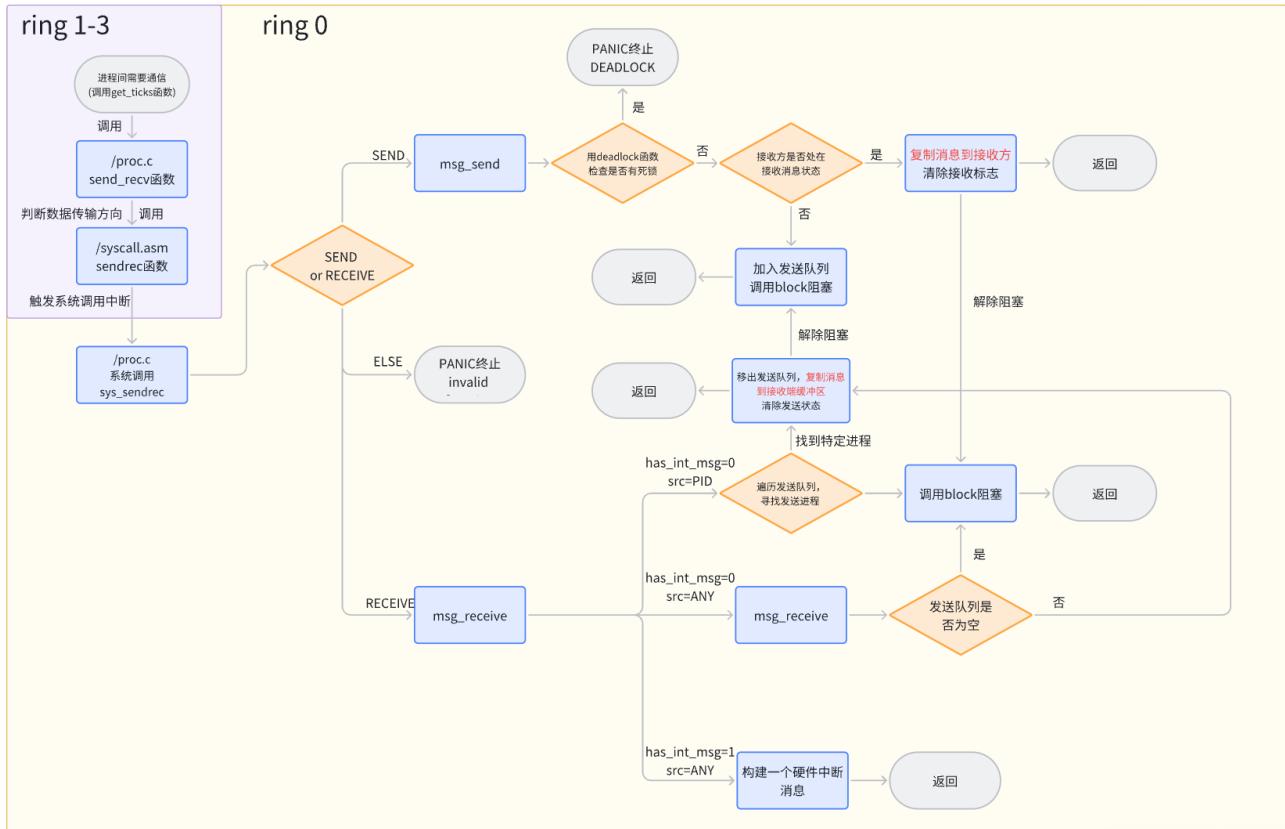
Mac 的内核基于 XNU 内核 (“X is Not Unix”), 它也采用了混合内核设计，结合了微内核和宏内核的特点。XNU 内核最初源自 BSD 和 Mach 微内核的结合，并在其基础上添加了苹果的特定改进。

Mac 内核架构的主要组成部分包括：

1. Mach 微内核：提供基础的底层功能。
  - 任务与线程管理：支持多线程、优先级调度。
  - 内存管理：通过 Mach 消息机制实现虚拟内存和物理内存管理。
  - 中断与异常处理：为硬件和系统事件提供统一的处理接口。
2. BSD 层：为系统提供高级操作系统功能。
  - 文件系统：支持 HFS+、APFS 等文件系统。
  - 网络栈：实现 POSIX 兼容的网络通信协议。
  - 进程管理：为任务提供传统 UNIX 风格的进程模型。
3. I/O Kit：苹果为设备驱动程序设计的框架。
  - 面向对象设计：基于 C++，提供模块化和可扩展的驱动开发框架。
  - 动态驱动管理：支持热插拔和驱动程序的动态加载。
4. 内核扩展 (Kexts)：允许第三方开发的扩展模块加载到内核中。
  - 提供安全沙盒机制，限制扩展模块对系统的影响。

XNU 内核的混合架构设计，使其能够结合 Mach 微内核的灵活性与 BSD 系统的高性能，同时通过 I/O Kit 和内核扩展机制，实现对苹果硬件的深度优化和兼容。

## 4.4 画出一个逻辑关系图，描述本章实验中 IPC 的实现框架机理，并加以文字解释，特别注意：处理器状态的切换，信息的流向



## 4.5 简要描述 8.2-8.5 涉及系统调用的流程与作用

流程：

- 用户进程调用 sendrec() 函数
- 触发 90 号中断，将控制权从用户态切换到内核态
- syscall 调用，通过 NR，确定要调用的系统调用函数
- sysSendrec 调用，将处理器从用户态切换到内核态，并根据传递的参数决定调用 msSend() 还是 msgReceive()
- 处理消息的发送和接受
- 在进程中调用修改后的 get\_ticks()

作用：sys\_sendrec() 系统调用是 sendrec 的核心，把 SEND 消息交给 msg\_send() 处理，把 RECEIVE 消息交给 msg\_receive 处理，并通过 function 参数来确定时发送还是接受消息，并根据 src\_dest 参数指定的目标进程来执行相应的操作。

## 4.6 在 8.2-8.5 代码中，当涉及程序与中断事件的并发时，是如何施加保护的

该部分代码使用同步 IPC 实现并发时的保护。

同步 IPC 的发送者一直需要等到接收者收到信息才结束，接收者亦只能到发送者的消息才能继续。同步 IPC 有若干好处：

- 操作系统不需要另外维护缓冲区来存放正在传递的消息；
- 操作系统不需要保留一份消息副本；
- 操作系统不需要维护接收队列（发送队列还是需要的）；
- 发送者和接收者都可在任何时刻清晰且容易地知道消息是否送达；
- 从实现系统调用的角度来看，同步 IPC 更加合理——当使用系统调用时，我们的确需要等待内核返回结果之后再继续。

IPC 解决了如下的中断与程序间的并发问题：

- 中断事件触发的任务：如果中断触发的是需要主程序进一步处理的任务，IPC 可以作为事件通知机制或数据共享桥梁。
- 主程序和中断间的数据传递：IPC 可以实现数据从中断上下文传递到用户进程（主程序），避免直接共享资源的竞争。
- 复杂任务协调：当需要多个程序模块协同处理中断引发的任务时，IPC 提供了高效的同步手段。

## 4.7 解释一下 assert、Panic 的实现过程（含涉及的系统调用机理），撰写几个小例子验证其作用

### 4.7.1 assert

首先定义 assert 的头文件如下：

```
1 #define ASSERT
2 #ifdef ASSERT
3 void assertion_failure(char *exp, char *file, char *base_file, int line);
4 #define assert(exp) if (exp) \
5             else assertion_failure(#exp, __FILE__, __BASE_FILE__, __LINE__)
6 #else
7 #define assert(exp)
8 #endif
```

其中值得注意的宏的意义如下：

- \_\_FILE\_\_ 将被展开成当前输入的文件。在这里，它告诉我们哪个文件中产生了异常。
- \_\_BASE\_FILE\_\_ 可被认为是传递给编译器的那个文件名。比如你在 m.c 中包含了 n.h，而 n.h 中的某一个 assert 函数失败了，则 \_\_FILE\_\_ 为 n.h，\_\_BASE\_FILE\_\_ 为 m.c。
- \_\_LINE\_\_ 将被展开成当前的行

接下来实现 assertio\_failure() 函数:

```
1 PUBLIC void assertion_failure(char *exp, char *file, char *base_file, int
2   line)
3 {
4   printf("%c assert(%s) failed: file: %s, base_file: %s, ln%d",
5         MAG_CH_ASSERT,
6         exp, file, base_file, line);
7
8   /**
9    * If assertion fails in a TASK, the system will halt before
10   * printf() returns. If it happens in a USER PROC, printf() will
11   * return like a common routine and arrive here.
12   * @see sys_printx()
13   *
14   * We use a forever loop to prevent the proc from going on:
15   */
16   spin("assertion_failure()");
17
18   /* should never arrive here */
19   __asm__ __volatile__("ud2");
}
```

这个函数的作用是将错误产生的位置打印出来，这里并没有使用原来的打印函数，而是使用了改进后的 printf()，它将调用一个叫做 printx 的系统调用，最终调用 sys\_printx()，该函数如下:

```
1 PUBLIC int sys_printx(int _unused1, int _unused2, char* s, struct proc*
2   p_proc)
3 {
4   const char * p;
5   char ch;
6
7   char reenter_err[] = "? k_reenter is incorrect for unknown reason";
8   reenter_err[0] = MAG_CH_PANIC;
9
10  /**
11   * @note Code in both Ring 0 and Ring 1~3 may invoke printx().
12   * If this happens in Ring 0, no linear-physical address mapping
13   * is needed.
14   *
15   * @attention The value of `k_reenter` is tricky here. When
16   * -# printx() is called in Ring 0
17   *   - k_reenter > 0. When code in Ring 0 calls printx(),
18   *     an `interrupt re-enter` will occur (printx() generates
19   *     a software interrupt). Thus `k_reenter` will be increased
20   *     by `kernel.asm::save` and be greater than 0.
21   * -# printx() is called in Ring 1~3
22   *   - k_reenter == 0.
23   */
24   if (k_reenter == 0) /* printx() called in Ring<1~3> */
25     p = va2la(proc2pid(p_proc), s);
  else if (k_reenter > 0) /* printx() called in Ring<0> */
```

```

26     p = s;
27 else /* this should NOT happen */
28     p = reenter_err;
29
30 /**
31 * @note if assertion fails in any TASK, the system will be halted;
32 * if it fails in a USER PROC, it'll return like any normal syscall
33 * does.
34 */
35 if ((*p == MAG_CH_PANIC) ||
36     (*p == MAG_CH_ASSERT && p_proc_ready < &proc_table[NR_TASKS])) {
37     disable_int();
38     char * v = (char*)V_MEM_BASE;
39     const char * q = p + 1; /* +1: skip the magic char */
40
41     while (v < (char*)(V_MEM_BASE + V_MEM_SIZE)) {
42         *v++ = *q++;
43         *v++ = RED_CHAR;
44         if (!*q) {
45             while (((int)v - V_MEM_BASE) % (SCR_WIDTH * 16)) {
46                 /* *v++ = ' ';*/
47                 v++;
48                 *v++ = GRAY_CHAR;
49             }
50             q = p + 1;
51         }
52     }
53
54     __asm__ __volatile__ ("hlt");
55 }
56
57 while ((ch = *p++) != 0) {
58     if (ch == MAG_CH_PANIC || ch == MAG_CH_ASSERT)
59         continue; /* skip the magic char */
60
61     out_char(tty_table[p_proc->nr_tty].p_console, ch);
62 }
63
64 return 0;
65 }

```

该函数先判断首字符是否为预设的”Magic Char”，如果是，就进行特殊的响应处理。当 sys\_printx( ) 发现传入字符串的第一个字符是 MAG\_CH\_ASSERT 时，会同时判断调用系统调用的进程是系统进程 (TASK) 还是用户进程 (USER PROC)，如果是系统进程，则停止整个系统的运转，并将要打印的字符串打印在显存的各处；如果是用户进程，则打印之后像一个普通的 printx 调用一样返回，届时该用户进程会因为 assertion\_failure( ) 中对函数 spin( ) 的调用而进入死循环。换言之，系统进程的 assert 失败会导致系统停转，用户进程的失败仅仅使自己停转。

#### 4.7.2 panic

panic 函数同样用到了之前提到的 sys\_printx() 和 “Magic Char”，它的实现代码如下：

```
1 PUBLIC void panic(const char *fmt, ...)
2 {
3     int i;
4     char buf[256];
5
6     /* 4 is the size of fmt in the stack */
7     va_list arg = (va_list)((char*)&fmt + 4);
8
9     i = vsprintf(buf, fmt, arg);
10
11    printf("%c !!panic!! %s", MAG_CH_PANIC, buf);
12
13    /* should never arrive here */
14    __asm__ __volatile__(#ud2);
15 }
```

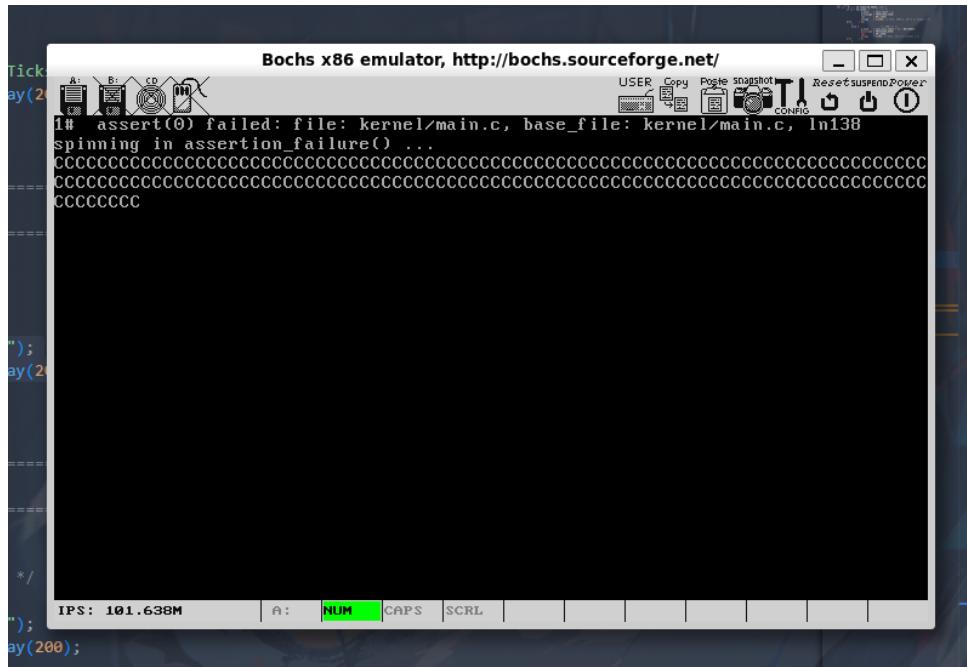
由于 panic 只会用在系统任务所处的 Ring1 或 Ring0，所以 sys\_printx() 遇到 MAG\_CH\_PANIC 就直接叫停整个系统，因为我们使用 panic 的时候，必是发生了严重错误的时候。

#### 4.7.3 举例子进行测试

先对 assert 进行测试，我们在 B 进程执行前加入 assert，如下：

```
1 void TestB()
2 {
3     assert(0);
4     while(1){
5         printf("B");
6         milli_delay(200);
7     }
8 }
```

运行结果如下：

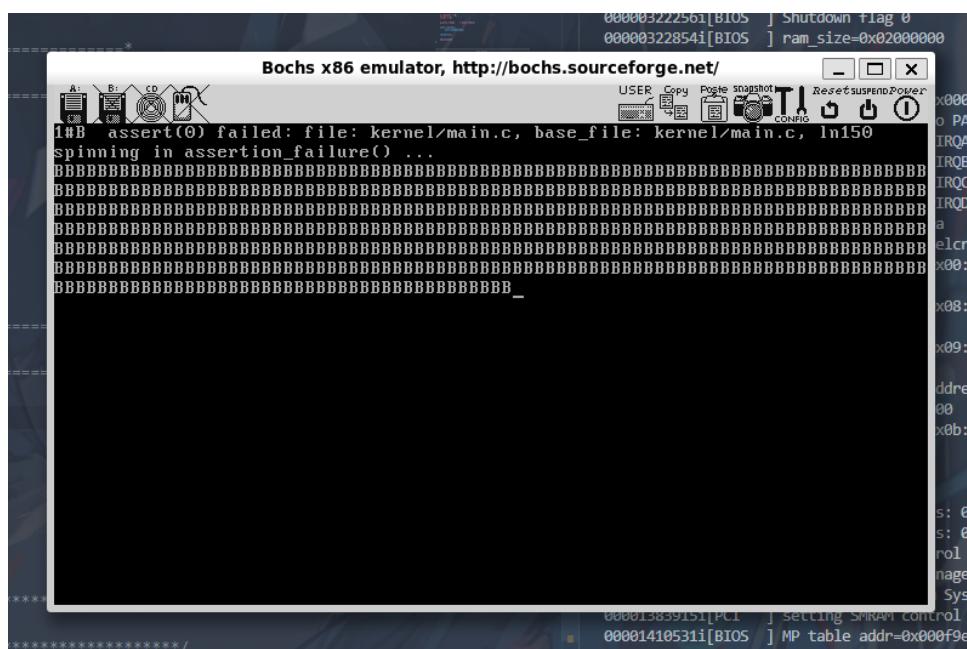


可以看到最顶部打印出现了响应 assert 的文件及其行号，同时没有打印“B”，只有打印了“C”，符合预期。再对 C 进程执行前加入 assert，如下：

```

1 void TestC()
2 {
3     assert(0);
4     while(1){
5         printf("C");
6         milli_delay(200);
7     }
8 }
```

运行结果如下：

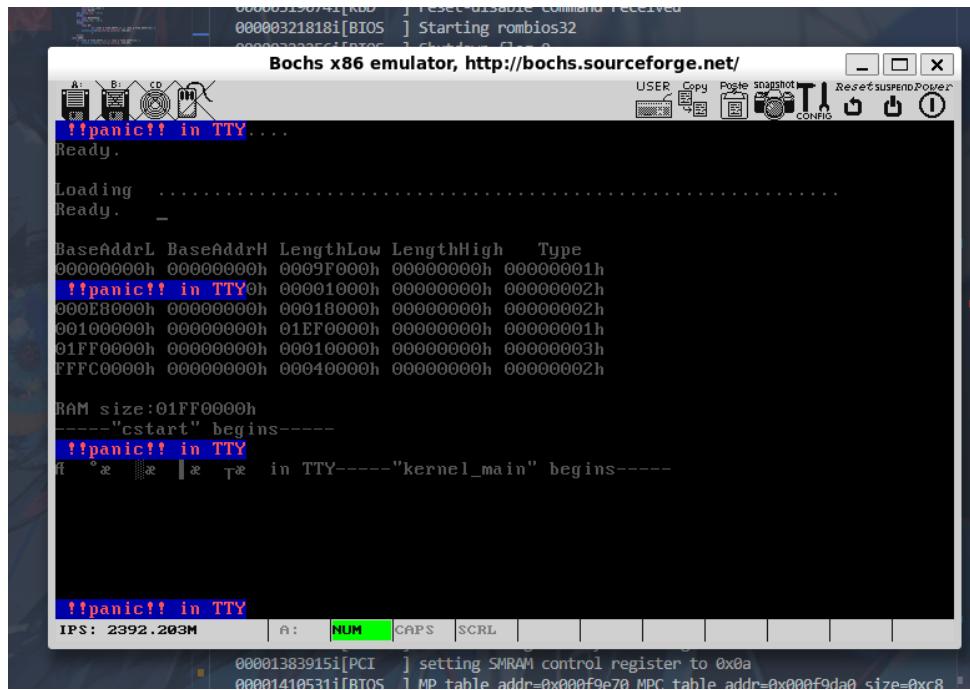


只有打印“C”，没有打印“B”，同时提供了错误的定位信息。

下面对 panic 函数进行测试，在 task\_tty() 中加入 panic，如下：

```
1 PUBLIC void task_tty()
2 {
3     TTY* p_tty;
4
5     panic("in TTY");
6
7     init_keyboard();
8
9     for (p_tty=TTY_FIRST;p_tty<TTY_END;p_tty++) {
10         init_tty(p_tty);
11     }
12     select_console(0);
13     while (1) {
14         for (p_tty=TTY_FIRST;p_tty<TTY_END;p_tty++) {
15             tty_do_read(p_tty);
16             tty_do_write(p_tty);
17         }
18     }
19 }
```

运行结果如下：



可以看到屏幕中出现了多处 panic in tty 的信息提示，同时系统不再响应别的进程，系统停止了下来，没有打印“A、B、C”字符。

## 4.8 在本部分的消息机制中，如何实现通信的？对进程如何调度管理的？

### 4.8.1 通信实现

新增系统调用名为 sendrec，sendrec 代码如下：

```

1 sendrec:
2     mov eax, _NR_sendrec
3     mov ebx, [esp + 4]    ; function
4     mov ecx, [esp + 8]    ; src_dest
5     mov edx, [esp + 12]   ; p_msg
6     int INT_VECTOR_SYS_CALL
7     ret

```

sys\_sendrec 如下：

```

1 PUBLIC int sys_sendrec(int function, int src_dest, MESSAGE* m, struct
2 proc* p)
3 {
4     assert(k_reenter == 0); /* make sure we are not in ring0 */
5     assert((src_dest >= 0 && src_dest < NR_TASKS + NR_PROCS) ||
6            src_dest == ANY ||
7            src_dest == INTERRUPT);
8
9     int ret = 0;
10    int caller = proc2pid(p);
11    MESSAGE* mla = (MESSAGE*)va2la(caller, m);
12    mla->source = caller;
13
14    assert(mla->source != src_dest);
15
16    /**
17     * Actually we have the third message type: BOTH. However, it is not
18     * allowed to be passed to the kernel directly. Kernel doesn't know
19     * it at all. It is transformed into a SEND followed by a RECEIVE
20     * by `send_recv()`'.
21     */
22    if (function == SEND) {
23        ret = msg_send(p, src_dest, m);
24        if (ret != 0)
25            return ret;
26    }
27    else if (function == RECEIVE) {
28        ret = msg_receive(p, src_dest, m);
29        if (ret != 0)
30            return ret;
31    }
32    else {
33        panic("{sys_sendrec} invalid function: "
34              "%d (SEND:%d, RECEIVE:%d).", function, SEND, RECEIVE);
35    }
36
37    return 0;
}

```

sys\_sendrec 函数可以描述为，把 SEND 消息交给 msg\_send 处理，把 RECEIVE 消息交给 msg\_receive 处理。msg\_send 实现如下：

```

1 PRIVATE int msg_send(struct proc* current, int dest, MESSAGE* m)
2 {
3     struct proc* sender = current;
4     struct proc* p_dest = proc_table + dest; /* proc dest */
5
6     assert(proc2pid(sender) != dest);
7
8     /* check for deadlock here */
9     if (deadlock(proc2pid(sender), dest)) {
10         panic(">>DEADLOCK<< %s->%s", sender->name, p_dest->name);
11     }
12
13     if ((p_dest->p_flags & RECEIVING) && /* dest is waiting for the msg */
14         /* */
15         (p_dest->p_recvfrom == proc2pid(sender) ||
16          p_dest->p_recvfrom == ANY)) {
17         assert(p_dest->p_msg);
18         assert(m);
19
20         phys_copy(va2la(dest, p_dest->p_msg),
21                   va2la(proc2pid(sender), m),
22                   sizeof(MESSAGE));
23         p_dest->p_msg = 0;
24         p_dest->p_flags &= ~RECEIVING; /* dest has received the msg */
25         p_dest->p_recvfrom = NO_TASK;
26         unblock(p_dest);
27
28         assert(p_dest->p_flags == 0);
29         assert(p_dest->p_msg == 0);
30         assert(p_dest->p_recvfrom == NO_TASK);
31         assert(p_dest->p_sendto == NO_TASK);
32         assert(sender->p_flags == 0);
33         assert(sender->p_msg == 0);
34         assert(sender->p_recvfrom == NO_TASK);
35         assert(sender->p_sendto == NO_TASK);
36     }
37     else { /* dest is not waiting for the msg */
38         sender->p_flags |= SENDING;
39         assert(sender->p_flags == SENDING);
40         sender->p_sendto = dest;
41         sender->p_msg = m;
42
43         /* append to the sending queue */
44         struct proc * p;
45         if (p_dest->q_sending) {
46             p = p_dest->q_sending;
47             while (p->next_sending)
48                 p = p->next_sending;
49             p->next_sending = sender;
50         }
51         else {

```

```

51     p_dest->q_sending = sender;
52 }
53 sender->next_sending = 0;
54
55 block(sender);
56
57 assert(sender->p_flags == SENDING);
58 assert(sender->p_msg != 0);
59 assert(sender->p_recvfrom == NO_TASK);
60 assert(sender->p_sendto == dest);
61 }
62
63 return 0;
64 }
```

msg\_receive 如下：

```

1 PRIVATE int msg_receive(struct proc* current, int src, MESSAGE* m)
2 {
3     struct proc* p_who_wanna_recv = current; /**
4             * This name is a little bit
5             * weird, but it makes me
6             * think clearly, so I keep
7             * it.
8             */
9     struct proc* p_from = 0; /* from which the message will be fetched */
10    struct proc* prev = 0;
11    int copyok = 0;
12
13    assert(proc2pid(p_who_wanna_recv) != src);
14
15    if ((p_who_wanna_recv->has_int_msg) &&
16        ((src == ANY) || (src == INTERRUPT))) {
17        /* There is an interrupt needs p_who_wanna_recv's handling and
18         * p_who_wanna_recv is ready to handle it.
19         */
20
21        MESSAGE msg;
22        reset_msg(&msg);
23        msg.source = INTERRUPT;
24        msg.type = HARD_INT;
25        assert(m);
26        phys_copy(va2la(proc2pid(p_who_wanna_recv), m), &msg,
27                  sizeof(MESSAGE));
28
29        p_who_wanna_recv->has_int_msg = 0;
30
31        assert(p_who_wanna_recv->p_flags == 0);
32        assert(p_who_wanna_recv->p_msg == 0);
33        assert(p_who_wanna_recv->p_sendto == NO_TASK);
34        assert(p_who_wanna_recv->has_int_msg == 0);
35 }
```

```

36         return 0;
37     }
38
39
40     /* Arrives here if no interrupt for p_who_wanna_recv. */
41     if (src == ANY) {
42         /* p_who_wanna_recv is ready to receive messages from
43          * ANY proc, we'll check the sending queue and pick the
44          * first proc in it.
45         */
46         if (p_who_wanna_recv->q_sending) {
47             p_from = p_who_wanna_recv->q_sending;
48             copyok = 1;
49
50             assert(p_who_wanna_recv->p_flags == 0);
51             assert(p_who_wanna_recv->p_msg == 0);
52             assert(p_who_wanna_recv->p_recvfrom == NO_TASK);
53             assert(p_who_wanna_recv->p_sendto == NO_TASK);
54             assert(p_who_wanna_recv->q_sending != 0);
55             assert(p_from->p_flags == SENDING);
56             assert(p_from->p_msg != 0);
57             assert(p_from->p_recvfrom == NO_TASK);
58             assert(p_from->p_sendto == proc2pid(p_who_wanna_recv));
59         }
60     }
61     else {
62         /* p_who_wanna_recv wants to receive a message from
63          * a certain proc: src.
64         */
65         p_from = &proc_table[src];
66
67         if ((p_from->p_flags & SENDING) &&
68             (p_from->p_sendto == proc2pid(p_who_wanna_recv))) {
69             /* Perfect, src is sending a message to
70              * p_who_wanna_recv.
71              */
72             copyok = 1;
73
74             struct proc* p = p_who_wanna_recv->q_sending;
75             assert(p); /* p_from must have been appended to the
76                         * queue, so the queue must not be NULL
77                         */
78             while (p) {
79                 assert(p_from->p_flags & SENDING);
80                 if (proc2pid(p) == src) { /* if p is the one */
81                     p_from = p;
82                     break;
83                 }
84                 prev = p;
85                 p = p->next_sending;
86             }
}

```

```

87     assert(p_who_wanna_recv->p_flags == 0);
88     assert(p_who_wanna_recv->p_msg == 0);
89     assert(p_who_wanna_recv->p_recvfrom == NO_TASK);
90     assert(p_who_wanna_recv->p_sendto == NO_TASK);
91     assert(p_who_wanna_recv->q_sending != 0);
92     assert(p_from->p_flags == SENDING);
93     assert(p_from->p_msg != 0);
94     assert(p_from->p_recvfrom == NO_TASK);
95     assert(p_from->p_sendto == proc2pid(p_who_wanna_recv));
96   }
97 }
98
99 if (copyok) {
100 /* It's determined from which proc the message will
101 * be copied. Note that this proc must have been
102 * waiting for this moment in the queue, so we should
103 * remove it from the queue.
104 */
105 if (p_from == p_who_wanna_recv->q_sending) { /* the 1st one */
106     assert(prev == 0);
107     p_who_wanna_recv->q_sending = p_from->next_sending;
108     p_from->next_sending = 0;
109 }
110 else {
111     assert(prev);
112     prev->next_sending = p_from->next_sending;
113     p_from->next_sending = 0;
114 }
115
116 assert(m);
117 assert(p_from->p_msg);
118 /* copy the message */
119 phys_copy(va2la(proc2pid(p_who_wanna_recv), m),
120           va2la(proc2pid(p_from), p_from->p_msg),
121           sizeof(MESSAGE));
122
123 p_from->p_msg = 0;
124 p_from->p_sendto = NO_TASK;
125 p_from->p_flags &= ~SENDING;
126 unblock(p_from);
127 }
128 else { /* nobody's sending any msg */
129     /* Set p_flags so that p_who_wanna_recv will not
130      * be scheduled until it is unblocked.
131      */
132     p_who_wanna_recv->p_flags |= RECEIVING;
133
134     p_who_wanna_recv->p_msg = m;
135
136     if (src == ANY)
137

```

```

138     p_who_wanna_recv->p_recvfrom = ANY;
139     else
140         p_who_wanna_recv->p_recvfrom = proc2pid(p_from);
141
142     block(p_who_wanna_recv);
143
144     assert(p_who_wanna_recv->p_flags == RECEIVING);
145     assert(p_who_wanna_recv->p_msg != 0);
146     assert(p_who_wanna_recv->p_recvfrom != NO_TASK);
147     assert(p_who_wanna_recv->p_sendto == NO_TASK);
148     assert(p_who_wanna_recv->has_int_msg == 0);
149 }
150
151     return 0;
152 }
```

代码流程如下：

- msg\_receive 从进程或者中断处接收消息
  - 中断消息处理：如果接收进程有中断消息（has\_int\_msg 标志），会直接处理，并将消息拷贝到用户空间（phys\_copy）
  - 从队列接收消息：如果接收方的 src 参数是 ANY，会从发送队列 q\_sending 中找到第一个进程发送的消息。如果 src 指定了某个进程，遍历队列找到该进程发送的消息。
  - 消息拷贝：从发送方的消息结构体中拷贝消息内容到接收方。从队列中移除发送方，并重置其状态。
  - 阻塞接收方：如果没有可用的消息，设置接收方的状态为 RECEIVING，阻塞该进程，直到有消息到来。
- msg\_send 向目标进程发送消息
  - 检查死锁：调用 deadlock() 检测可能的死锁情形。
  - 直接传递消息：如果目标进程正等待消息，并且接收方允许接收来自发送方的消息（p\_recvfrom 为 ANY 或发送方的 PID），直接传递消息并解除接收方的阻塞状态。
  - 加入发送队列：如果目标进程未准备好接收消息，将当前发送方加入目标进程的发送队列（q\_sending）。设置发送方的状态为 SENDING，阻塞发送方，等待接收方处理。

具体而言，如果进程 A 想要向 B 发送消息 M，那么这两个进程之间的通信过程是这样的：

- A 首先准备好 M。
- A 通过系统调用 sendrec，最终调用 msg\_send。
- 简单判断是否发生死锁。
- 判断目标进程 B 是否正在等待来自 A 的消息：
  - 如果是：消息被复制给 B，B 被解除阻塞，继续运行；

- 如果否：A 被阻塞，并被加入到 B 的发送队列中。

如果有进程 B 想要接收消息（来自特定进程、中断或者任意进程），那么过程将会是这样的：

- B 准备一个空的消息结构体 M，用于接收消息。
- B 通过系统调用 sendrec，最终调用 msg\_receive。
- 判断 B 是否有个来自硬件的消息（通过 has\_int\_msg），如果是，并且 B 准备接收来自中断的消息或准备接收任意消息，则马上准备一个消息给 B，并返回。
- 如果 B 想接收来自任意进程的消息，则从自己的发送队列中选取第一个（如果队列非空的话），将其消息复制给 M。
- 如果 B 是想接收来自特定进程 A 的消息，则先判断 A 是否正在等待向 B 发送消息，若是的话，将其消息复制给 M。
- 如果此时没有任何进程发消息给 B，B 会被阻塞。

#### 4.8.2 调度管理

现在的每个进程增加了两种可能的状态：SENDING 和 RECEIVING。而凡是处于 SENDING 或 RECEIVING 状态的进程，我们就不再让它们获得 CPU 了。一个进程是否阻塞，由进程表中的 p\_flags 项决定。如下是修改后的进程调度：

```

1 PUBLIC void schedule()
2 {
3     struct proc*      p;
4     int      greatest_ticks = 0;
5
6     while (!greatest_ticks) {
7         for (p = &FIRST_PROC; p <= &LAST_PROC; p++) {
8             if (p->p_flags == 0) {
9                 if (p->ticks > greatest_ticks) {
10                     greatest_ticks = p->ticks;
11                     p_proc_ready = p;
12                 }
13             }
14         }
15
16         if (!greatest_ticks)
17             for (p = &FIRST_PROC; p <= &LAST_PROC; p++)
18                 if (p->p_flags == 0)
19                     p->ticks = p->priority;
20     }
21 }
```

可以看到，当且仅当 p\_flags 为 0 时，一个进程才可能获得运行的机会。定义 SENDING 和 RECEIVING 如下：

```

1 #define SENDING 0x02 /* set when proc trying to send */
2 #define RECEIVING 0x04 /* set when proc trying to recv */
```

p\_flags 有三种取值，分别是 0、SENDING、RECEIVING，修改后的 proc 结构体如下：

```

1 struct proc {
2     struct stackframe regs;      /* process registers saved in stack frame
3         */
4
5     u16 ldt_sel;                /* gdt selector giving ldt base and limit
6         */
7     struct descriptor ldts[LDT_SIZE]; /* local descs for code and data */
8
9     int ticks;                  /* remained ticks */
10    int priority;
11
12    u32 pid;                   /* process id passed in from MM */
13    char name[16];             /* name of the process */
14
15    int p_flags;                /* process flags.
16        * A proc is runnable iff p_flags==0
17        */
18
19    MESSAGE * p_msg;
20    int p_recvfrom;
21    int p_sendto;
22
23    int has_int_msg;            /* non zero if an INTERRUPT occurred when
24        * the task is not ready to deal with it.
25        */
26
27    struct proc * q_sending;    /* queue of procs sending messages to
28        * this proc
29        */
30
31    struct proc * next_sending; /* next proc in the sending
32        * queue (q_sending)
33        */
34
35
36    int nr_tty;
37 }

```

## 4.9 死锁问题是如何解决的？是否存在问题，若有改进之，若无说明验证其正确性。

```

1 PRIVATE int deadlock(int src, int dest)
2 {
3     struct proc* p = proc_table + dest;
4     while (1) {
5         if (p->p_flags & SENDING) {
6             if (p->p_sendto == src) {

```

```

7      /* print the chain */
8      p = proc_table + dest;
9      printf("=_=%s", p->name);
10     do {
11         assert(p->p_msg);
12         p = proc_table + p->p_sendto;
13         printf("->%s", p->name);
14     } while (p != proc_table + src);
15     printf("=_=");
16
17     return 1;
18 }
19 p = proc_table + p->p_sendto;
20 }
21 else {
22     break;
23 }
24 }
25 return 0;
26 }
```

#### 4.9.1 死锁检测逻辑

死锁的本质是一个进程等待的资源被另一个进程占用，且这些进程形成了一个环（循环等待）。但在学习的过程中我们也注意到，死锁一定有环，有环不一定有死锁，存在环路是形成死锁的必要条件。

所以在项目的程序中如果存在死锁也就是存在一个循环的链条：src -> dest -> ... -> src，其中每个进程都在等待下一个进程的资源。

根据这个思想，这个死锁检测函数的实现逻辑是：

1. 进程指针 p 从指向接收进程开始，沿发送链向下遍历进程，如果接受进程处于发送状态则将 p 更新为这个接收进程，继续检查发送链上的下一个进程。
2. 如果沿着发送链追溯发现目标进程最终会发送消息给原始发送者 src，说明出现了死锁，也就是 if(p->p\_sendto == src)，则表示 src 和 dest 之间形成了一个循环等待的环路。
3. 出现了环路就意味着出现了死锁，返回 1；否则返回 0。

#### 4.9.2 死锁处理

如果检测到了死锁就打印所有涉及的进程：

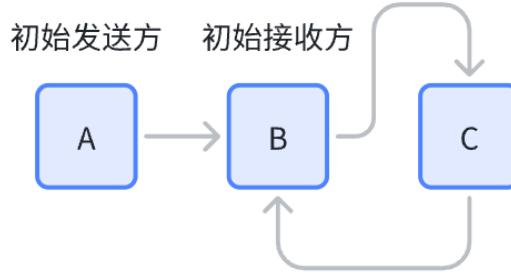
```

1 p = proc_table + dest;
2 printf("=_=%s", p->name);
3 do {
4     assert(p->p_msg);
5     p = proc_table + p->p_sendto;
6     printf("->%s", p->name);
7 } while (p != proc_table + src);
8 printf("=_=");
```

#### 4.9.3 死锁检测正确性讨论

这个死锁检测算法存在问题，但问题不是一开始设想的有环不一定死锁的情况，有环不一定死锁的情况是在使用信号量的异步 IPC 中存在的问题，在应用同步 IPC 且为单核 CPU 的情况下暂时没有想到可能发生有环但不一定死锁的情况。

这个死锁检测算法存在的问题在于这个死锁检测算法只能检测到以初始发送方为起止点的大环路，但如果在发送链中存在不经过初始发送方的小环路，则可能导致程序循环判断，陷入死循环，而无法返回是否有死锁的判断。示意图如下：



#### 4.9.4 改进方案

解决方案是将发送链条上的所有进程 ID 存储到一张哈希表中，如果 p 指针指到的新进程的 ID 与哈希表中的其他 ID 产生了哈希冲突，则表明存在死锁。

```
1 ****
2 *                               deadlock_plus
3 ****
4 /**
5 * <Ring 0> Check whether it is safe to send a message from src to dest.
6 * The routine will detect if the messaging graph contains a cycle. For
7 * instance, if we have procs trying to send messages like this:
8 * A -> B -> C -> B, then a deadlock occurs, because all of them will
9 * wait forever. If no cycles detected, it is considered as safe.
10 *
11 * @param src    Who wants to send message.
12 * @param dest   To whom the message is sent.
13 *
14 * @return Zero if success.
15 ****
16 PRIVATE int deadlock_plus(int src, int dest)
17 {
18     struct proc* p = proc_table + dest;
19     clear_hash_table(); // 每次检测死锁前都要清空哈希表
20
21     while (1) {
22         if (p->p_flags & SENDING) {
23             if (hash(p->p_sendto)) {
24                 /* print the chain */
```

```

25         int curcle = p->p_sendto;
26         int flag = 0;
27         p = proc_table + dest;
28         printf("=_=%s", p->name);
29         do {
30             if(p->p_sendto == curcle) {
31                 flag++;
32             }
33             assert(p->p_msg);
34             p = proc_table + p->p_sendto;
35             printf("->%s", p->name);
36         } while (flag != 2);
37         printf("=_=");
38
39         return 1; // 发生了死锁
40     }
41     p = proc_table + p->p_sendto;
42 }
43 else {
44     break;
45 }
46 }
47 return 0;
}

```

然后具体的 hash 操作是这样实现的：

```

1 ****global.c
2 ****
3 PUBLIC int hash_table[HASH_TABLE_SIZE];
4
5 ****global.h
6 ****
7 extern int hash_table[];
8
9 ****proc.h
10 ****
11 * /pro.c hash
12 ****
13 /**
14 * <Ring 0> Check whether it is safe to send a message from src.
15 * The routine will detect if the src has been appended to hash table.
16 * For
17 * instance, if we have procs trying to send messages like this:
18 * A -> B -> C -> B, then a hash conflict occurs.
19 * @param src Who wants to send message.

```

```

20 *
21 * @return Zero if there isn't deadlock.
22 ****
23
24 int hash(int src) {
25     int index = src % HASH_TABLE_SIZE; // 使用简单取模法生成哈希值
26
27     if (hash_table[index] == src) {
28         return 1; // 冲突，表示可能存在死锁
29     } else if (hash_table[index] == 0) {
30         hash_table[index] = src; // 将进程 ID 存入哈希表
31         return 0; // 正常插入
32     } else {
33         return 1; // 发生了冲突，出现了死锁
34     }
}

```

最后还需要维护这个哈希表，在每次死锁检测时重置一次哈希表。

```

1 ****
2
3 *           clear_hash_table
4 ****
5
6 /**
7 * <Ring 0> Clear the hash table before each time of deadlock detection.
8 *
9 ****
10
11 void clear_hash_table() {
12     for (int i = 0; i < HASH_TABLE_SIZE; i++) {
13         hash_table[i] = 0; // 重置哈希表
14     }
}

```

## 4.10 简要分析基于 IPC，如何扩展 get\_ticks 的方法

使用 IPC 替换 get\_ticks 这个系统调用时为了让收发消息双方能够参与。我们需要一个系统进程来接受用户进程的消息，并且返回 ticks 值，建立一个新的系统调用，叫做“SYSTASK”：

```

1 PUBLIC void task_sys()
2 {
3     MESSAGE msg;
4     while (1) {
5         send_recv(RECEIVE, ANY, &msg);
6         int src = msg.source;
7
8         switch (msg.type) {
9             case GET_TICKS:
10                 msg.RETVAL = ticks;
11                 send_recv(SEND, src, &msg);
12                 break;
}

```

```

13     default:
14         panic("unknown msg type");
15         break;
16     }
17 }

```

通过 send\_recv 接受来自进程的消息，将消息存储在 msg 中，对 GET\_TICKS 使用 send\_recv 将消息发送到进程。其中的 send\_recv() 是将 sendrec 封装实现的：

```

1 PUBLIC int send_recv(int function, int src_dest, MESSAGE* msg)
2 {
3     int ret = 0;
4
5     if (function == RECEIVE)
6         memset(msg, 0, sizeof(MESSAGE));
7
8     switch (function) {
9     case BOTH:
10        ret = sendrec(SEND, src_dest, msg);
11        if (ret == 0)
12            ret = sendrec(RECEIVE, src_dest, msg);
13        break;
14     case SEND:
15     case RECEIVE:
16        ret = sendrec(function, src_dest, msg);
17        break;
18     default:
19         assert((function == BOTH) ||
20                (function == SEND) || (function == RECEIVE));
21         break;
22     }
23
24     return ret;
25 }

```

之后想要收发消息时，就直接使用这个 send\_recv()，而不直接使用系统调用 sendrec，接下来修改函数 get\_ticks：

```

1 PUBLIC int get_ticks()
2 {
3     MESSAGE msg;
4     reset_msg(&msg);
5     msg.type = GET_TICKS;
6     send_recv(BOTH, TASK_SYS, &msg);
7     return msg.RETVAL;
8 }

```

以 GET\_TICKS 为消息类型，不夹带其他任何信息地传递给 SYSTASK，SYSTASK 收到这个消息之后，把当前的 ticks 值放入消息并发给用户进程，用户进程会接收到它，完成整个任务。使用进程 A 调用 get\_ticks 进行测试：

```

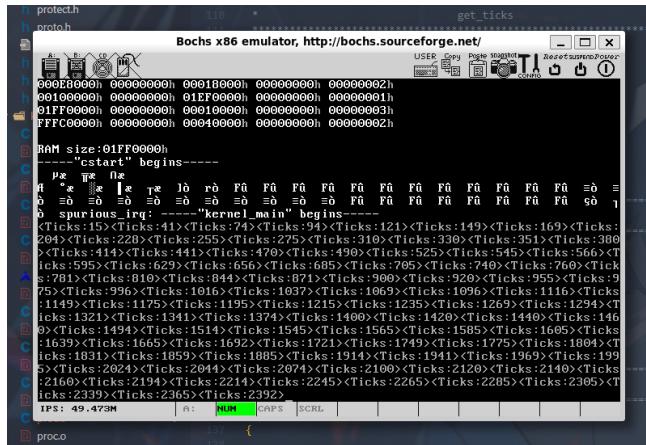
1 void TestA()
2 {

```

```

3     while (1) {
4         printf("<Ticks:%d>", get_ticks());
5         milli_delay(200);
6     }
7 }
```

运行代码：



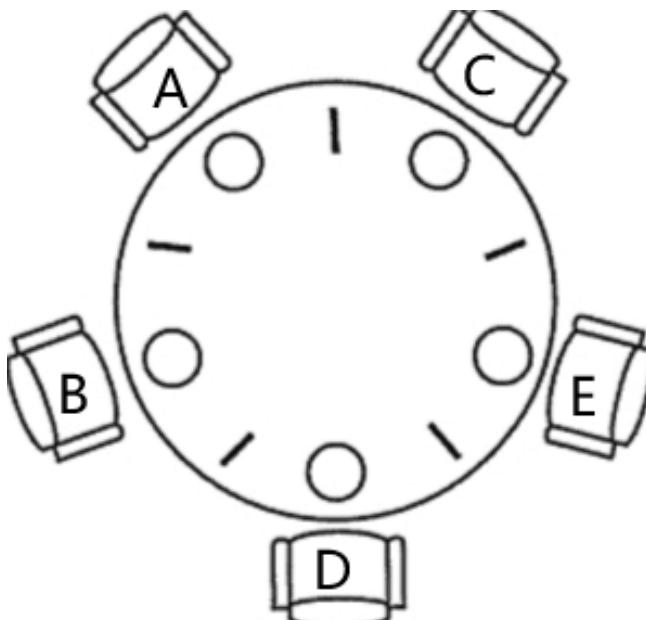
可以看到，在运行进程 A 后，成功的打印了当前的 ticks。

## 4.11 针对上学期学习的经典同步互斥问题，试着用 IPC 解决一例

目标：使用同步 IPC 解决哲学家问题

题目背景：有 5 个哲学家共用一张圆桌，分别坐在周围的 5 张椅子上，在圆桌上有 5 个碗和 5 只筷子，他们的生活方式是交替地进行思考和进餐。平时，每个哲学家进行思考，饥饿时便试图拿起其左右最靠近他的筷子，只有在他拿到两只筷子时才能进餐。进餐完毕，放下筷子继续思考。

思路：利用发送消息给特定进程的 unblock 操作和接受来自特定进程消息时的 block 操作来实现控制进程间的前驱后继关系。



例如，针对哲学家进餐问题的一种可行的解决方案是：A 和 D 先进餐，然后 B 和 E 进餐，最后 C 进餐。

下面展示代码实现：

定义五个进程模拟哲学家进餐问题：

- 进程定义：

```
1     PUBLIC void TestA();
2     PUBLIC void TestB();
3     PUBLIC void TestC();
4     PUBLIC void TestD();
5     PUBLIC void TestE();
```

- 进程常量初始化：

```
1     #define NR_PROCS      5
2     #define STACK_SIZE_TESTA    0x8000
3     #define STACK_SIZE_TESTB    0x8000
4     #define STACK_SIZE_TESTC    0x8000
5     #define STACK_SIZE_TESTD    0x8000
6     #define STACK_SIZE_TESTE    0x8000
7     #define STACK_SIZE_TOTAL   (STACK_SIZE_TTY + \
8                           STACK_SIZE_SYS + \
9                           STACK_SIZE_TESTA + \
10                          STACK_SIZE_TESTB + \
11                          STACK_SIZE_TESTC + \
12                          STACK_SIZE_TESTD + \
13                          STACK_SIZE_TESTE)
```

- 在用户进程表中加入新增进程：

```
1     PUBLIC struct task user_proc_table[NR_PROCS] = {
2         {TestA, STACK_SIZE_TESTA, "TestA"}, 
3         {TestB, STACK_SIZE_TESTB, "TestB"}, 
4         {TestC, STACK_SIZE_TESTC, "TestC"}, 
5         {TestD, STACK_SIZE_TESTD, "TestD"}, 
6         {TestE, STACK_SIZE_TESTE, "TestE"};
```

- 在 main.c 中新增进程，进程间通过 send\_recv 进行通信，哲学家问题中分为 SEND、RECEIVE 两种状态，当哲学家在进食结束后给后面的哲学家 SEND 刀叉，当哲学家未获得刀叉时陷入等待，直到 RECEIVE 前面的哲学家给自己发送的刀叉。初始时 A、D 进程能够拿起周围的刀叉，用餐结束后放下刀叉告知旁边的哲学家，此时哲学家 BE 拿起刀叉进行用餐，BE 结束后，哲学家 C 才能获得刀叉用餐。

```
1     void TestA()
2     {
3         MESSAGE msg;
4         reset_msg(&msg);
5         msg.type = GET_TICKS;
6         while (1) {
7             printf("philosopher A is eating\n");
8             send_recv(SEND, NR_TASKS + 1, &msg);
```

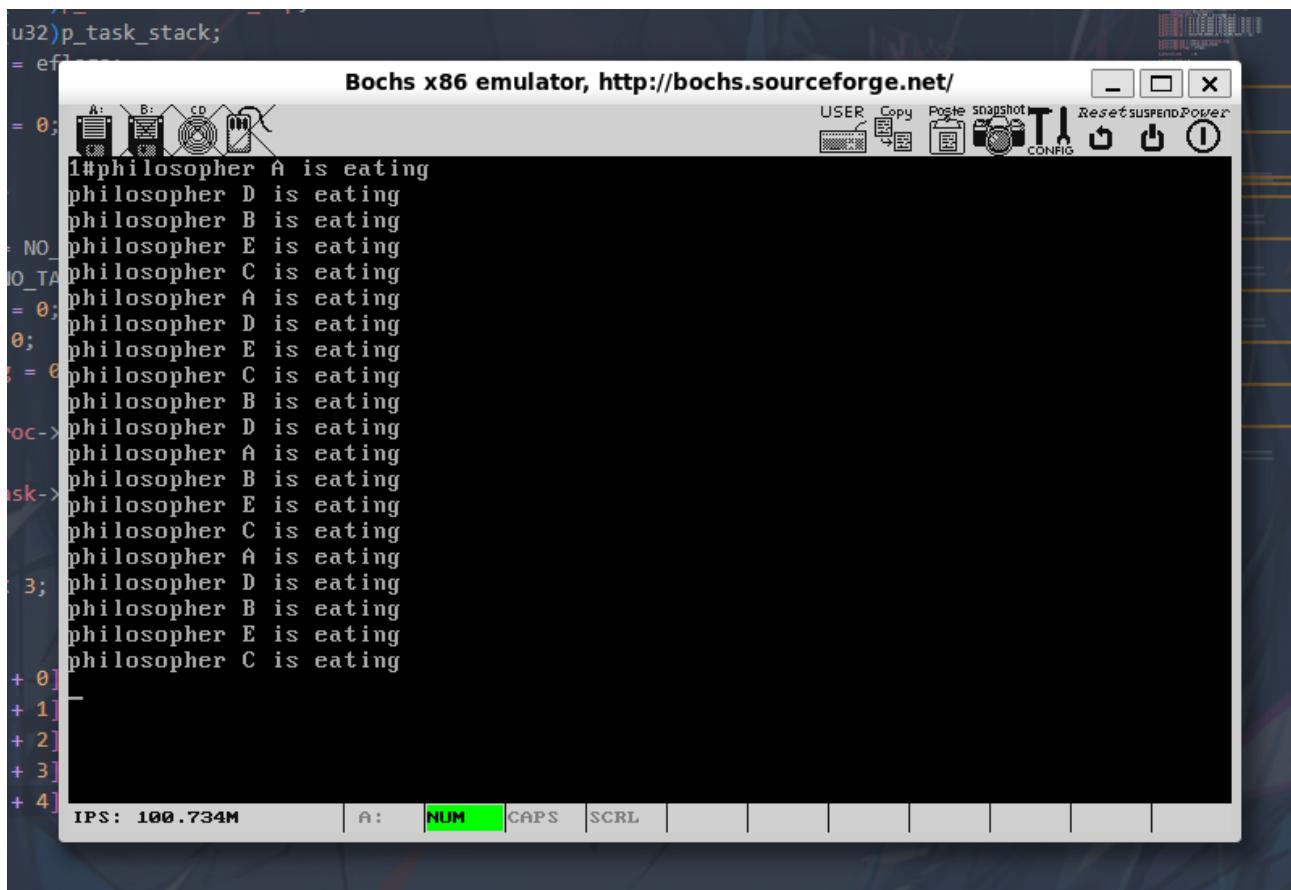
```

9      send_recv(SEND, NR_TASKS + 2, &msg);
10     milli_delay(200000);
11 }
12 }
13 void TestB()
14 {
15     MESSAGE msg;
16     reset_msg(&msg);
17     msg.type = GET_TICKS;
18     while(1){
19         send_recv(RECEIVE, NR_TASKS + 0, &msg);
20         send_recv(RECEIVE, NR_TASKS + 3, &msg);
21         printf("philosopher B is eating\n");
22         milli_delay(200000);
23     }
24 }
25
26 void TestC()
27 {
28     MESSAGE msg;
29     reset_msg(&msg);
30     msg.type = GET_TICKS;
31     while(1){
32         send_recv(RECEIVE, NR_TASKS + 0, &msg);
33         send_recv(RECEIVE, NR_TASKS + 4, &msg);
34         printf("philosopher C is eating\n");
35         milli_delay(200000);
36     }
37 }
38
39 void TestD()
40 {
41     MESSAGE msg;
42     reset_msg(&msg);
43     msg.type = GET_TICKS;
44     while(1){
45         printf("philosopher D is eating\n");
46         send_recv(SEND, NR_TASKS + 1, &msg);
47         send_recv(SEND, NR_TASKS + 4, &msg);
48         milli_delay(200000);
49     }
50 }
51
52 void TestE()
53 {
54     MESSAGE msg;
55     reset_msg(&msg);
56     msg.type = GET_TICKS;
57     while(1){
58         send_recv(RECEIVE, NR_TASKS + 3, &msg);
59         printf("philosopher E is eating\n");

```

```
60         send_recv(SEND, NR_TASKS + 2, &msg);
61         milli_delay(200000);
62     }
63 }
```

最后的输出结果展示了哲学家进餐的顺序：



## 5 实验结果总结

(理论联系实际，思考并列出本实验对应的 OS 原理的知识点，并说明本实验中的实现部分如何对应和体现了原理中的基本概念和关键知识点。)

### 5.1 IPC

IPC (Inter-Process Communication, 进程间通信) 是指计算机中不同进程之间进行数据交换和信息传递的一种机制。由于操作系统中的进程是独立的，其地址空间彼此隔离，因此需要 IPC 来实现它们之间的协作与数据共享。IPC 的应用使得进程能够实现下面四个功能：

1. **数据共享**: 允许一个进程向另一个进程发送或接收数据。
  2. **资源共享**: 多个进程可以通过 IPC 机制访问共享的资源（如共享内存）。
  3. **同步与协调**: 帮助多个进程同步执行，避免竞争条件和死锁问题。
  4. **事件通知**: 一个进程可以通过 IPC 机制向另一个进程发送信号或通知事件的发生。

## 5.2 死锁

在多道程序环境中，当多个进程竞争有限的系统资源时，可能出现某一进程在申请资源时，由于当前资源不可用而进入等待状态。如果这些资源被其他同样处于等待状态的进程占用，则可能导致这些进程无法继续执行，也无法释放其占有的资源，从而进入一种相互等待且永远无法继续的状态。这种现象称为死锁。

总结一下形成死锁的四个必要条件分别是：

- **互斥**，系统中的资源必须至少有一个是不可共享的（即只能被一个进程独占）。
- **占有并等待**，一个进程已经占有了至少一个资源，并且还在请求其他资源，但这些资源被其他进程占有，此时该进程进入等待状态。
- **不可剥夺**，如果一个进程已经获得某个资源，则在未完成使用时，系统无法强制收回这个资源，而只能让其他进程等待。
- **环路等待**，系统中存在一个进程等待资源的循环链，其中的每一个进程都在等待下一个进程占有的资源。

解决死锁问题的策略主要分为以下三种：

1. **死锁预防**：通过设计机制主动破坏死锁发生的四个必要条件之一，确保系统永远不会进入死锁状态。
  - 破坏占有与等待条件：要求进程在申请资源时，必须一次性请求所有所需资源，避免占有部分资源后继续等待。
  - 破坏不可抢占条件：允许系统剥夺已经分配给某进程的资源，使其释放资源后重新申请。
  - 破坏环路等待条件：对资源编号排序，规定进程必须按照固定顺序申请资源，防止资源分配形成环形依赖。
2. **死锁避免**：通过动态分析系统中资源的分配与进程的请求情况，避免进入可能导致死锁的不安全状态。
  - 银行家算法：根据每个进程的最大资源需求和系统当前的可用资源情况，判断是否能够安全分配。如果分配可能导致不安全状态，系统将暂时拒绝分配，让进程等待。
3. **死锁检测与恢复**：系统定期检查当前的资源分配状态，判断是否存在死锁。一旦检测到死锁，可采取以下措施恢复系统的正常运行：
  - 终止进程：选择并终止一个或多个陷入死锁的进程，释放其占有的资源。
  - 回滚进程：将死锁进程回滚至某个安全的检查点状态，释放部分资源。
  - 资源抢占：强制收回部分资源，分配给其他进程，以解除死锁状态。

## 5.3 assert

**assert** 是一种调试工具，用于在程序运行中检查某些条件是否为真。如果条件为假 (false)，程序会中断执行，并输出错误信息。它通常用于验证程序的内部一致性，确保在关键点某些假设成立并且只在开发或调试阶段生效。许多编程语言会在编译时移除 assert 的检查逻辑，以提高生产环境的性能。

## 5.4 panic

panic 是一种严重错误处理机制，用于在程序遇到不可恢复的错误时立即终止程序运行。常用于处理程序的致命错误，当这些错误发生时，继续运行程序可能导致更严重后果的场景比如：未捕获的异常、非法状态、逻辑漏洞或外部不可控因素导致的程序错误。它多用于生产环境，因为它可以直接中断程序运行，防止错误进一步扩散。

## 6 各人实验贡献与体会（每人各自撰写）

1. 黄东威同学：独立完成实验，研究了死锁问题如何解决，编写了同步互斥问题基于 IPC 实现的代码，编写了微内核域宏内核在系统调用角度和同步互斥问题代码相关部分的实验报告撰写。

个人体会：这次实验是 oranges 系统代码的一个高潮，我们终于见到了操作系统中经常提到的并发问题，操作系统虚拟化、并发、调度三大问题，并发问题一向被认为是其中及其复杂的一环。这次进程间的通信，oranges 带领我们使用同步 IPC 逐步的思考解决问题，过程中我体会到了前辈们在实现诸如“管道”等现在看来十分常见的进程通信工具时面临的诸多挑战。本章立足很高，但采用了较为简单的方式实现进程通信，并且带领我们讨论了不同的内核架构在系统调用层面的不同，也算是十分有趣。linus 的反叛精神有了现在 linux 的辉煌，cs 学习需要有质疑精神。这次实验的末尾，我和队友们讨论了如何使用 IPC 实现一个经典同步互斥问题，我们最终选择了哲学家吃饭问题，我们讨论了代码细节，研究了代码运行中出现的 bug，最终顺利完成了这部分内容，有了这次实验的圆满结束。

2. 王浚杰同学：独立完成实验，负责前几问思考题，回答 IPC 的实现框架机理、系统调用的流程与作用以及 Panic 的实现过程等。补充了实验过程以及故障问题。

个人体会：这次实验是 OS 教材小实验的收官之作，主题聚焦于同步 IPC 的实现及其周边问题的探索。在分析微内核和宏内核架构的系统调用差异时，我见到了设计操作系统时的权衡艺术。微内核的模块化设计虽然带来了灵活性，但频繁的进程切换也影响了效率，而宏内核的集中式管理则在性能与复杂度之间找到了平衡。通过实验对比后，我更加理解为何两者在特定场景各有优劣。在 IPC 的同步设计，更需要精妙的同步通信机制，发挥 msg\_send 和 msg\_receive 函数的交互作用，巧妙实现了消息传递的有序性和一致性。并为了其鲁棒性，还引入并改进了死锁检测机制，通过追踪发送链中的循环依赖，确保系统不会因资源竞争而陷入不可恢复的状态。此次实验贯穿了理论与实践，将 OS 前几章的各种核心概念串联起来。从最初的单进程设计，到多进程并发，再到系统调用和进程调度算法的实现，再到底的同步 IPC，我们一路走来真的不容易，多少次熬着夜还不如放弃。但是想到三个队友从不摆烂，以及队长 HDW 在实验里的无私付出，还是坚持在每周倾注大量时间。最后我们较为完美地完成了几乎所有的任务。

3. 程序同学：独立完成实验，调研 Mac 内核架构，完成基于 IPC 如何拓展 get\_ticks 部分的报告部分书写，补充部分实验结果

个人体会：本次实验涉及的书本内容和代码较少，任务量相对较少，主要集中于知识调研和实验改进部分。通过本次实验的调研环节，我了解了 windows、linux、mac 三种平台的特点和基本的内核构成模块。除了课上已经介绍的宏内核和微内核外，还存在以 mac 为代表的混合内核，在设计上结合了微内核和宏内核的特点，既保留了单内核的高性能，又引入了微内核的模块化和灵活性，使得系统具备较好的扩展性和稳定性。

此外，通过本次实验，我学习了进程间通信的基本原理及其实现，通过一个统一的系统调用 sys\_sendrecv 来进行消息的传递。进程间通信时阻塞和解除阻塞的过程与操作系统

理论课中的 PV 操作类似，加深了我对进程间通信的理解。另外，可以看到，本实验采用的进程间通信方法是忙等，我们也可以进一步改进实验，将其变为异步 IPC，使得进程在发送完消息后可以去做别的事，提高运行效率。

4. 周业营同学：调研主流操作系统的内核结构，绘制 IPC 实现框架机理逻辑关系图，讨论死锁实现存在的问题并进行了改进，并完成三个对应部分的实验报告撰写。

个人体会：这是最后一次小实验，也是相对友好的一次实验，在这次实验中讨论和实现了同步 IPC 的实现及其相关的其他问题。IPC 在 OS 理论课上讨论的重点都是异步 IPC，并通过信号量来实现资源的分配和进程的控制，但在 Orange 中实现的是同步 IPC，这相对地提升了一些这次实验的难度。

在调研目前主流的操作系统的内核结构时发现，过去的宏内核和微内核的高下之争其实到了现在已经不再是当务之急，混合内核的应用集两家之长，这才是大势所趋。在绘制 IPC 框图的过程中才逐渐发现了同步 IPC 的精妙之处，这是一种只需要维护发送队列而不需要维护接收队列的机制，msg\_send 与 msg\_receive 环环相扣，组成了一个 IPC 系统。在讨论死锁检测算法的过程中，我也开拓了自己的思维，找到并解决了一个项目的死锁检测算法中存在的问题，这个过程让我受益匪浅。

总算是做完了 OS 实验的小实验任务，赞美 Orange、Jinx、Jayce、Viktor、Vander 和三个给力的队友。

## 7 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)

分数	评语
85-100	<ul style="list-style-type: none"><li>• 实验内容完整或者有超出课程实验大纲的内容；</li><li>• 实验步骤详尽，能够体现完整的实验过程；</li><li>• 实验结果正确且实验数据分析得当；</li><li>• 实验心得与思考全面并且有自己的独立思考；</li><li>• 实验报告文档规范、排版整齐。</li></ul>
75-84	<ul style="list-style-type: none"><li>• 实验内容较为完整；</li><li>• 实验步骤较为详尽，能够体现实验过程；</li><li>• 实验结果正确且实验数据分析较为得当；</li><li>• 实验心得与思考全面；</li><li>• 实验报告文档规范、排版较为整齐。</li></ul>

60-74	<ul style="list-style-type: none"> <li>• 实验内容有缺失；</li> <li>• 实验步骤不够详尽，不能够体现完整的实验过程；</li> <li>• 实验结果部分正确；</li> <li>• 实验心得与思考无或者不够深入；</li> <li>• 实验报告文档规范性有待增强。</li> </ul>
60 以下	<ul style="list-style-type: none"> <li>• 实验内容严重缺失、实验态度不够端正；</li> <li>• 实验步骤不够详尽，不能够体现完整的实验过程；</li> <li>• 实验结果部分正确；</li> <li>• 实验心得与思考无或者不够深入；</li> <li>• 实验报告文档规范性有待增强。</li> </ul>

## 8 教师评分（请填写好姓名、学号）

姓名	学号	分数
黄东威	2022302181148	
王浚杰	2022302181143	
程序	2022302181131	
周业营	2022302181145	

教师签名：

年       月       日