|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 课程名称 | 操作系统设计与实践 | | | 成 绩 |  | 教师签名 |  |
| 实验名称 | 进程间通信 | | | 实验序号 | 10 | 实验日期 |  |
| 姓 名 | 张子航 | 学 号 | 2021302181026 | | | | 组长 |
| 姓 名 | 辜汝曦 | 学 号 | 2021302141194 | | | | 组员 |
| 姓 名 | 杨馨悦 | 学 号 | 2021302181212 | | | | 组员 |
| 姓 名 | 赵敏 | 学 号 | 2021302181215 | | | | 组员 |

**《操作系统设计与实践》实验报告**

# 实验目的及实验内容

（本次实验所涉及并要求掌握的知识；实验内容等）

## 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

## 参考资料

《操作系统设计实验系列（十） 进程间通信》PPT

## 实验要求

* 验证IPC的实现机理
* 学习分析IPC实现的技巧与细节

# 实验环境及实验步骤

（列出本次实验所使用的软件、工具；简要概括实验步骤）

## 实验环境

虚拟机工具：VMWare Workstation 16

虚拟机版本：Ubuntu 14.04.6(内存4GB，硬盘40GB，双核处理器)

开发与调试工具：bochs 2.6.8

## 实验步骤

阅读章节资料，根据课件内容进行理解并调试，最终自行实现相应内容

# 实验过程分析

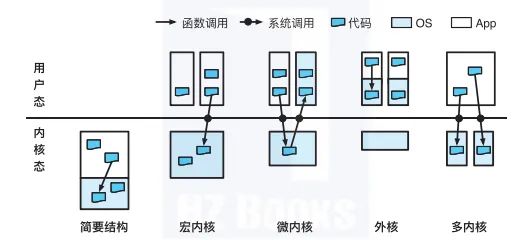
（详细记录实验过程，通过截图展示得到的结果。特别是对于实验中发生的故障和问题，要进行故障分析，说明故障排除的过程及方法。）

## 阅读8.1节资料回答以下问题：

### 微内核与宏内核在系统调用角度差异是什么？

首先我们介绍微内核和宏内核的基本概念。

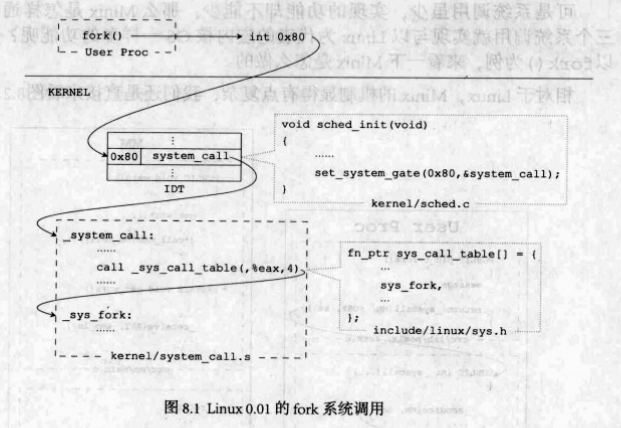
宏内核（Monolithic kernel）又称单内核，其特征是操作系统内核的所有模块（包括进程调度、内存管理、文件系统、设备驱动等）均运行在内核态，具备直接操作硬件的能力，这类操作系统包括UNIX/Linux、FreeBSD等。



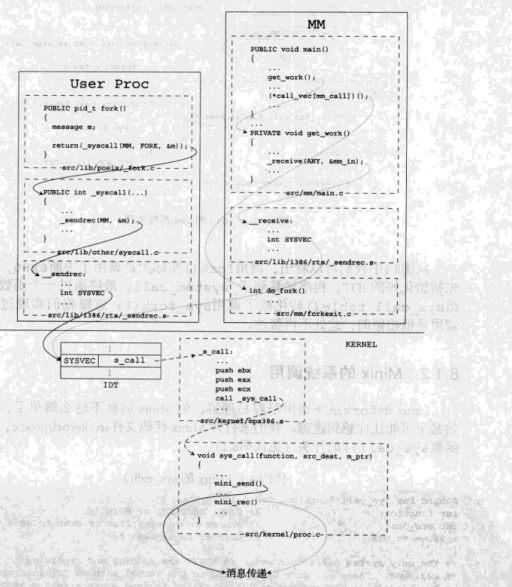
随着宏内核操作系统的内核功能不断增长，系统的复杂度也持续增加，在可靠性、安全性等方面导致了更多的问题。这是因为在宏内核架构下，所有内核模块均运行在特权空间，一个单点的错误就可能会导致整个系统崩溃或者被攻破。

因此，研究人员尝试将单个功能或模块从内核中拆分出来，作为一个独立的服务部署到独立的运行环境中；内核仅保留极少的功能，为这些服务提供通信等基础能力，使其能够互相协作以完成操作系统所必需的功能。这种架构被称为微内核（Microkernel）。

从系统调用方面来说，宏内核因为集成了许多复杂的功能，因此系统调用会直接调用相应功能，直接在内核解决问题；而微内核将功能拆分出来，对于系统调用，它只起到通信的作用，并调用内核之外的模块完成任务。



对于宏内核，以Linux内核的fork()为例，调用fork()实际上是调用了中断0x80，通过事先初始化好的IDT，程序转移到了\_sytem\_call，最终通过一个函数指针数组sys\_call\_table[]转化成了调用sys\_fork()。从这里可以看出，由于操作系统功能复杂，宏内核需要实现的系统调用也较多。



对于微内核，同样以fork()为例，Minix中fork的过程不在内核中完成，而是内核将系统调用的信息转移到内核之外的MM模块中，由MM负责fork()的具体工作。Minix中的系统调用仅有三个：SEND、RECEIVE、BOTH，以实现消息的接收与发送。

### 我们之前的实验实现，更类似哪种架构？

我们之前的实验实现更类似宏内核。我们实现的键盘中断、设备驱动均运行在内核中，并设置了对应的系统调用，并没有分配内核之外的独立模块来处理这些功能。

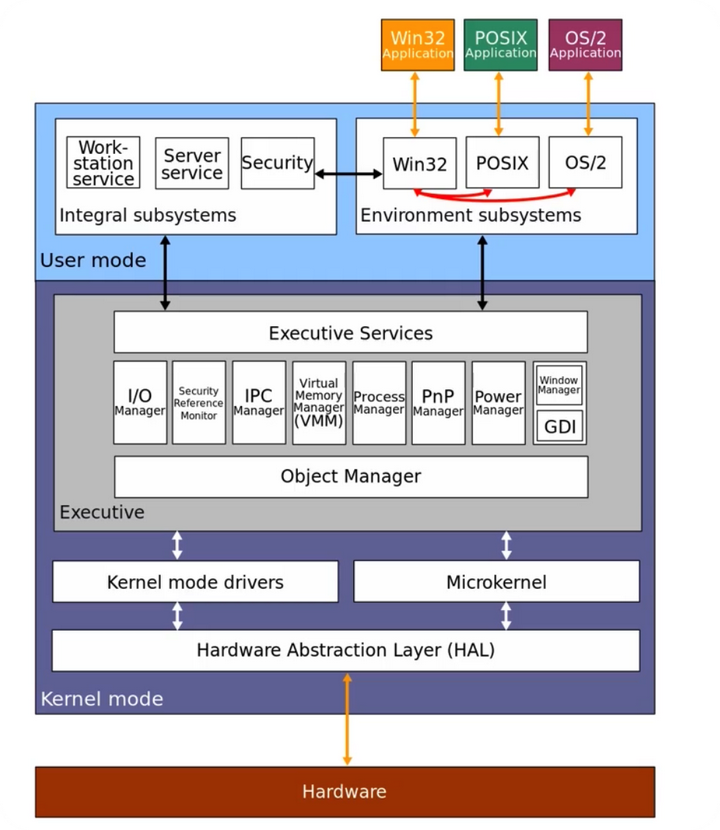
### 调研一下，目前的主流桌面OS，如windows, linux, mac OS都是怎样的内核架构

由于技术发展，目前的操作系统大多数融合了微内核和宏内核的思想，因此难以武断地说哪一种操作系统是完全基于微内核或者宏内核的，我们统一将其称为混合内核架构。

当然，以我个人的观点，如果一个内核，只在Kernel Mode模式里放置IPC，MM，Scheduling模块，那么就是微内核；反之，如果把其它的功能，比如file system, VFS, device drivers放到Kernel Mode模式中，那么就是宏内核。混合内核本质上就是宏内核。

* Windows

Windows 的内核设计属于混合类型。可以看到，内核中有一个 Microkernel 模块。而整个内核实现又像宏内核一样，含有许多其他模块，譬如Virtual Memory Manager、I/O Manager等，是一个完整的整体。



Windows内核分为三层，硬件抽象层(Hardware Abstraction Layer ,简称HAL)， HAL之上是内核层，有时候也称为微内核，在内核层之上则是执行体(executive)层。

硬件抽象层的作用是把所有与硬件相关联的代码逻辑隔离到一个专门的模块中，从而使上面的层次做到独立于硬件平台。

内核层和执行层的分工是，内核层实现操作系统的基本机制，而所有的策略决定则留给执行体。

内核层是内核模块ntoskrml.exe 中的下层部分，最接近HAL层，负责现场调度和中断、异常的处理；对于多处理器系统，还负责同步处理器之间的行为。

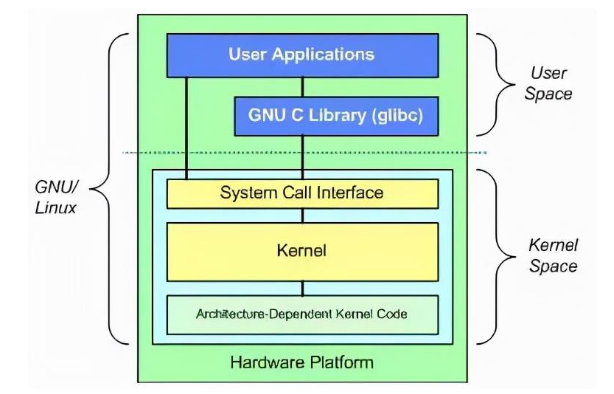
执行体是内核模块ntoskrnl.exe 的上层部分，包含5种类型的函数：

* + 被导出的，可在用户模式下调用的函数，如ntdll.dll；
  + 虽已被导出并且可在用户模式下调用，但无法通过任何一个Windows API调用的函数；
  + 只能在内核模式下调用的导出函数；
  + 供执行体组件之间相互调用，但未被文档化的函数；
  + 属于一个组件的内部函数。
* Linux

Linux 是一个宏内核架构的操作系统。在 Linux 内核中，核心功能和服务被实现为一个单一的内核，包括进程调度、内存管理、设备驱动、文件系统等。这些功能模块共享相同的内核地址空间和数据结构，通过直接的函数调用和共享数据来进行通信和调度。

虽然 Linux 内核是宏内核架构，但它也具有一些微内核架构的特性。例如，Linux 内核提供了动态加载的模块机制，允许在运行时加载和卸载特定的功能模块，以实现系统的灵活性和可扩展性。

Linux结构如下：

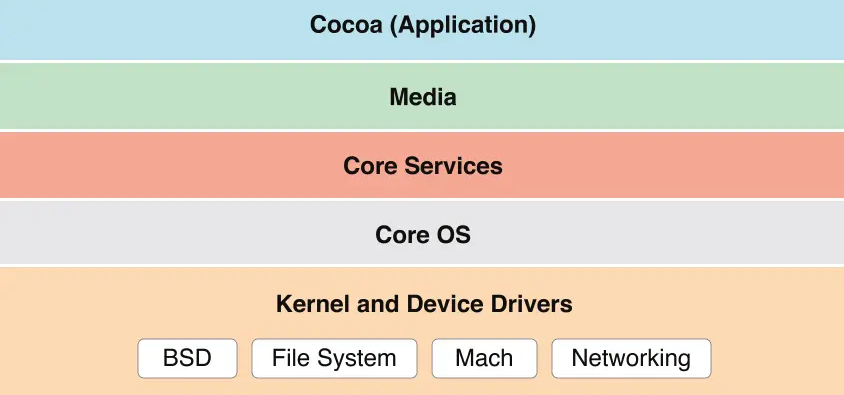


系统最上面是用户空间，这是用户应用程序执行的地方。用户空间之下是内核空间，glibc也在此处。它提供了连接内核的系统调用接口，还提供了在用户空间应用程序和内核之间进行转换的机制，因为内核和用户空间的应用程序使用的是不同的保护地址空间。每个用户空间的进程都使用自己的虚拟地址空间，而内核则占用单独的地址空间。

Linux 内核可以进一步划分成 3 层。最上面是系统调用接口，它实现了一些基本的功能，例如read和write。系统调用接口之下是内核代码，更准确地说，是独立于体系结构的内核代码。这些代码是 Linux 所支持的所有处理器体系结构所通用的。在这些代码之下是依赖于体系结构的代码，构成了BSP（Board Support Package）的部分。这些代码可以使得内核核心代码独立于硬件平台，作用类似于Windows中的HAL层。

* Mac OS

XNU是macOS的核心部分，它是一种结合了微内核与宏内核特性的混合内核，主要由两部分构成：BSD（FS，NKE）＋Mach（I/O Kit，Drivers）。



最底层的Mach管理进程资源，诸如CPU和内存、handle调度、内存保护，以及向其他操作系统层提供一个以消息为中心的基础设施。包括以下几个组件：

* untyped interprocess communication (IPC )－进程间通信
* remote procedure calls (RPC )－ 远程程序调用
* scheduler support for symmetric multiprocessing (SMP ) －对称多处理器调度
* support for real-time services－实时服务
* virtual memory support－虚拟内存
* support for pagers－页式管理
* modular architecture－模块化架构

基于Mach层的BSD层提供 OS personality APIs and services。BSD 层是基于BSD kernel, 主要是 FreeBSD。Mac OS中的BSD的组件如下：

* file systems－文件系统
* networking (except for the hardware device level)－网络功能（不包括硬件设备级别）
* UNIX security model－UNIX安全模块
* syscall support－系统调用
* the BSD process model, including process IDs and signals－BSD进程模块，包括进程ID和信号
* FreeBSD kernel APIs－FreeBSD内核API
* kernel support for pthreads (POSIX threads)－pthreads的内核支持（POSIX线程）

Mac OS X提供了多种类型的文件系统，包括HFS、HFS +、UFS、NFS、ISO 9660等。默认的文件系统类型是HFS+。这部分与内核关系不大，因此不做分析。

I/O Kit提供了一个简化的驱动程序开发框架，支持多种设备类型。I/O Kit具有面向对象的I/O架构，并支持模块化和可扩展性。其组件包括：

* true plug and play－即插即用
* dynamic device management－动态设备管理
* power management for desktop systems as well as portables－电源管理
* multiprocessor capabilities－多处理器能力

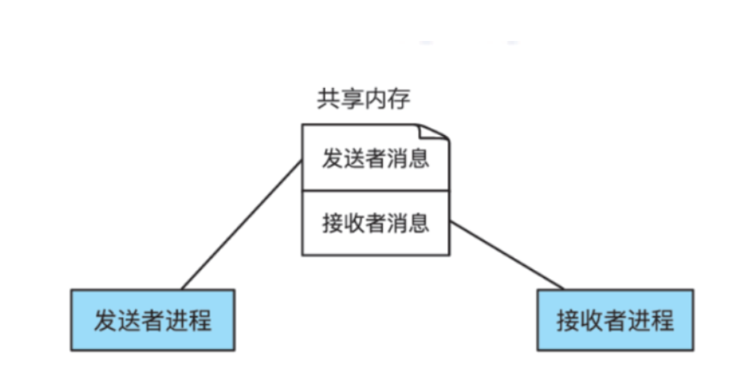
## 阅读8.2—8.5，并对代码目录a进行分析，完成以下任务：

### 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

本章实验中使用的是同步IPC

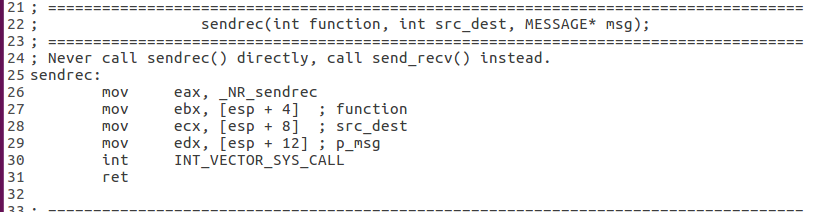
IPC至少需要两方（如两个进程）参与。根据信息流动的方向，这两方通常被称为发送者和接收者。在实际使用中，IPC经常被用于服务调用，因此参与IPC的两方又被称为调用者和被调用者，或者客户端和服务端。

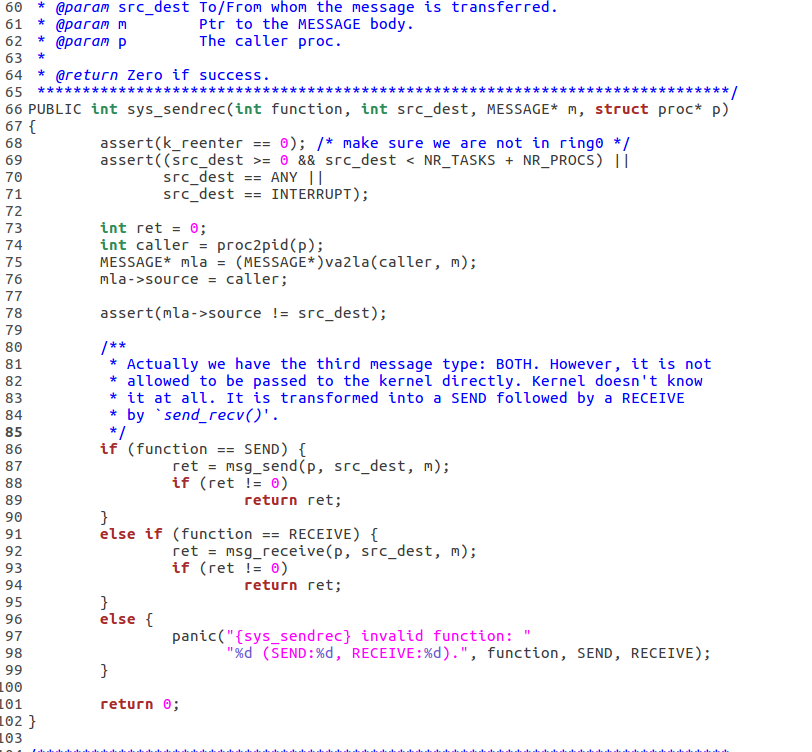
如图是一个简单的IPC设计。它假设内核已为两个进程映射了一段共享内存，且共享内存刚好可以存放两个消息（发送者消息和接受者消息）它的IPC操作（如Send）会阻塞进程直到该操作完成



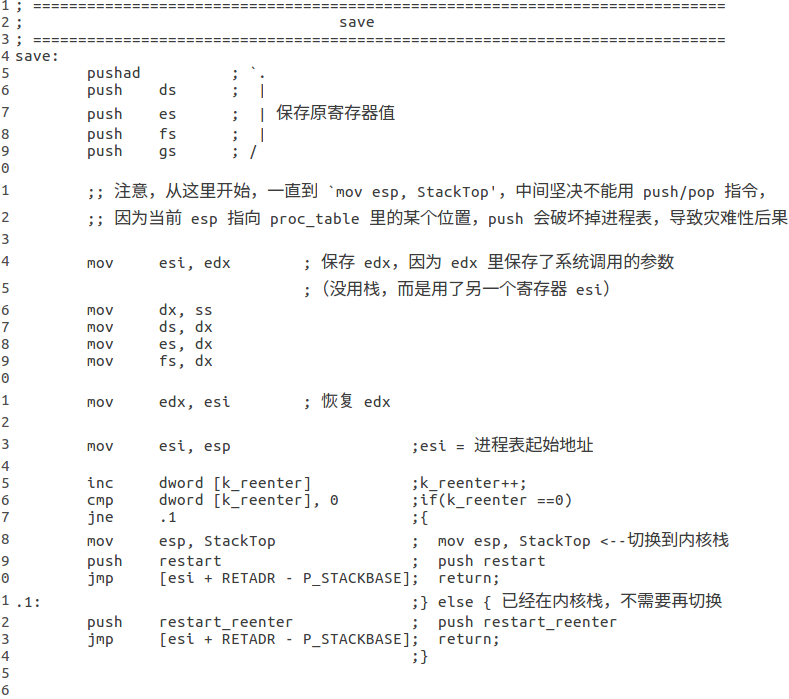
### 简要描述该处涉及系统调用的流程与作用

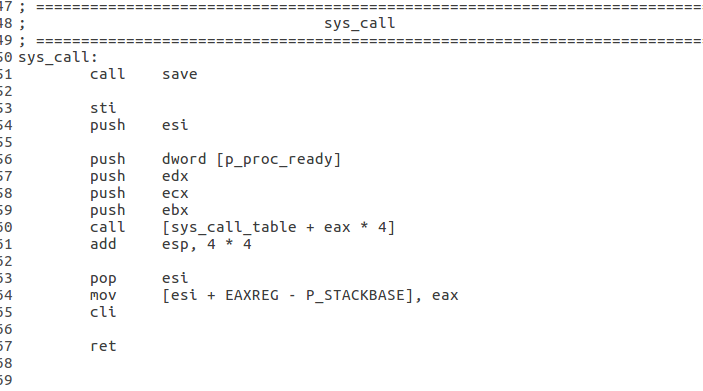
这个新的系统调用起名为sendrec





新加的参数是通过edx这个参数传递的，而save这个函数中也用到了寄存器dx，所以我们同时需要修改save

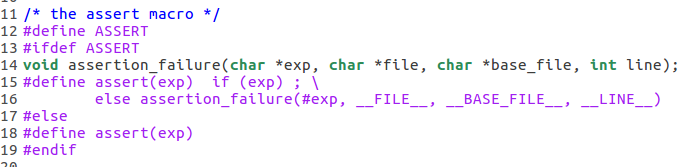




Sys\_sendrec把SEND消息交给msg\_send( )处理，把RECEIVE消息交给

msg\_receive( )处理

创建ass ert( )和panic ( )函数



添加宏定义

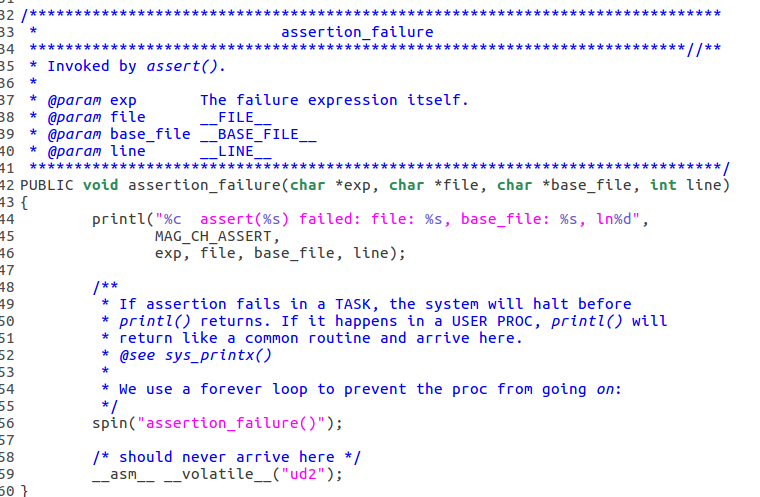
注意其中的\_\_FILE\_\_、\_\_BASE\_FILE\_\_和\_\_LINE\_\_这三个宏，它们的意义如下

\_\_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。

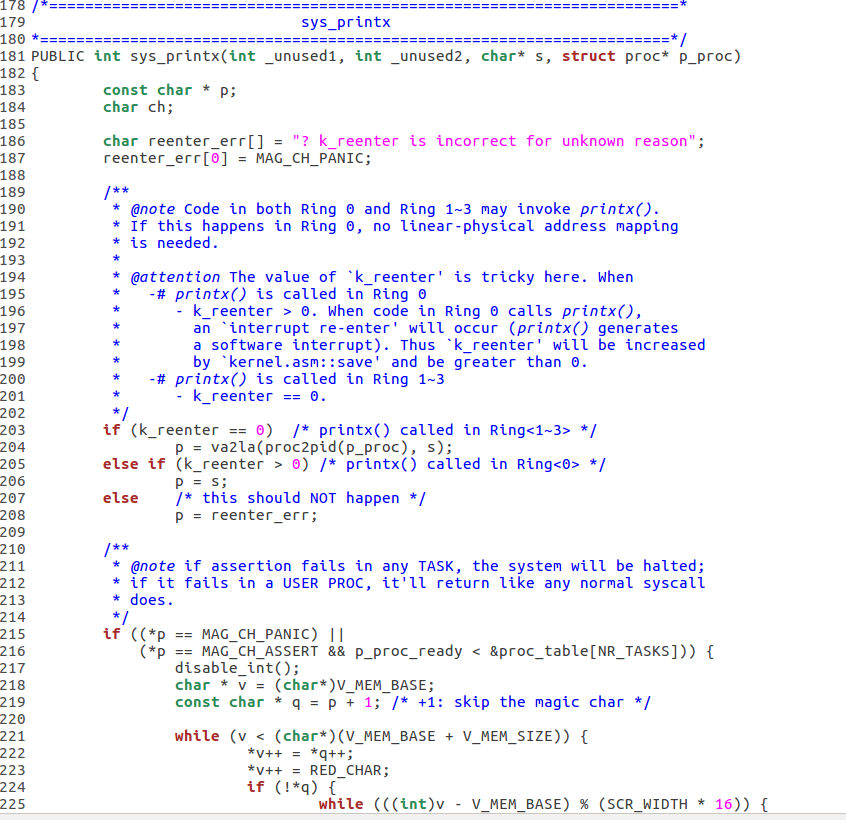
\_\_BASE\_FILE\_\_可被认为是传递给编译器的那个文件名。比如你在m.c中包含了n.h，而n.h中的某一个assert函数失败了，则\_\_FILE\_\_为n.h，\_\_BASE\_FILE\_\_为m.c。

\_\_LINE\_\_将被展开成当前的行号。

明白了这几个宏的意义，剩下的assertion\_failure( )这个函数就显得容易了，它的作用就是将错误发生的位置打印出来



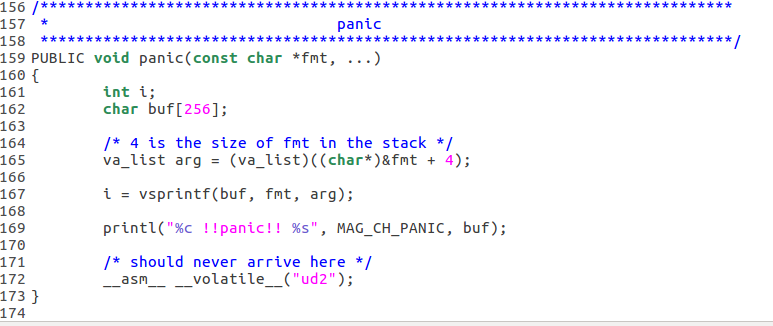
使用了一个改进后的打印函数，叫做printl( )，它其实就是一个定义成printf的宏，不过这里的printf跟上一章中的稍有不同，它将调用一个叫做printx的系统调用，并最终调用函数sys\_printx( )，它位于tty.c中



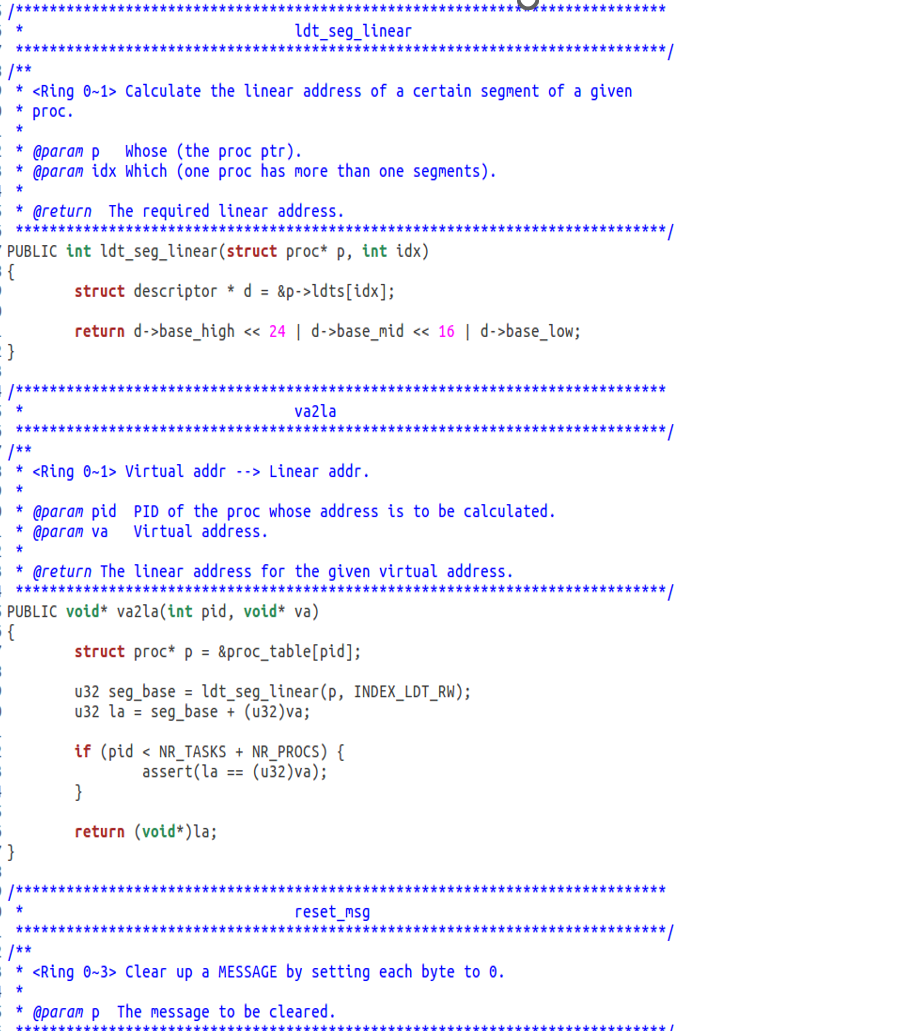
sys\_printx( )将首先判断首字符是否为预先设定的“Magic Char”，如果是的话，则做响应的特殊处理。我们的assertion\_failure( )就使用了MAG\_CH\_ASSERT作为“Magic Char”。当sys\_printx( )发现传入字符串的第一个字符是MAG\_CH\_ASSERT时，会同时判断调用系统调用的进程是系统进程（TASK）还是用户进程（USER PROC），如果是系统进程，则停止整个系统的运转，并将要打印的字符串打印在显存的各处；如果是用户进程，则打印之后像一个普通的printx调用一样返回，届时该用户进程会因为assertion\_failure( )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。

Panic（）函数



Msg\_send和msg\_receive



围绕msg\_send( )和msg\_receive( )，代码中还列出了其他几个必要的函数，它们是：

ldt\_se g\_l inear( )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。

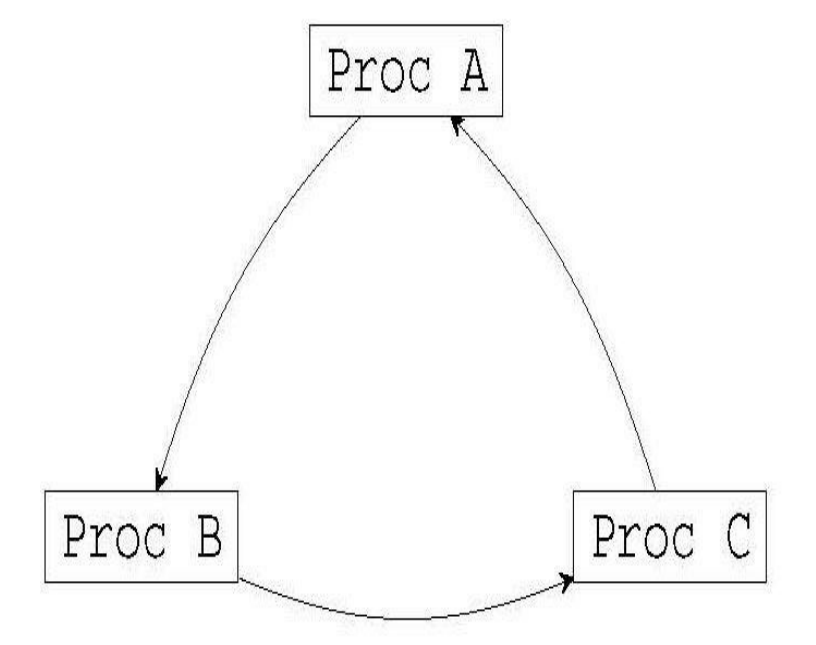
va2la( )用来由虚拟地址求线性地址，它用到了ldt\_seg\_linear( )。

reset\_ msg ( )用于把一个消息的每个字节清零。

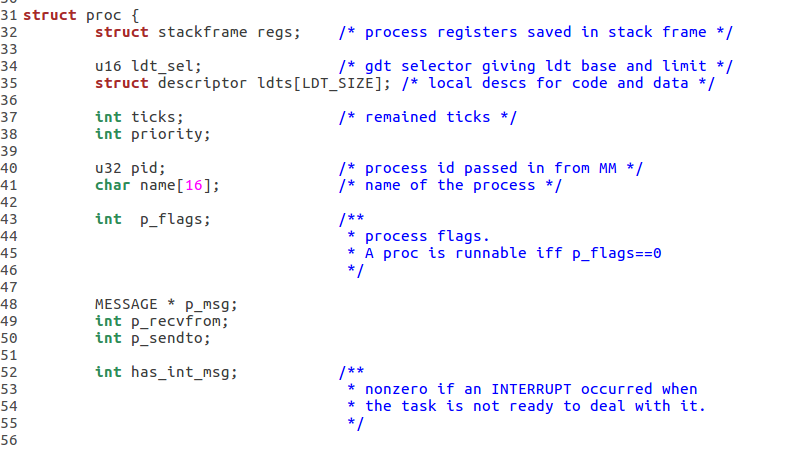
block( )阻塞一个进程。

unbloc k( )解除一个进程的阻塞。

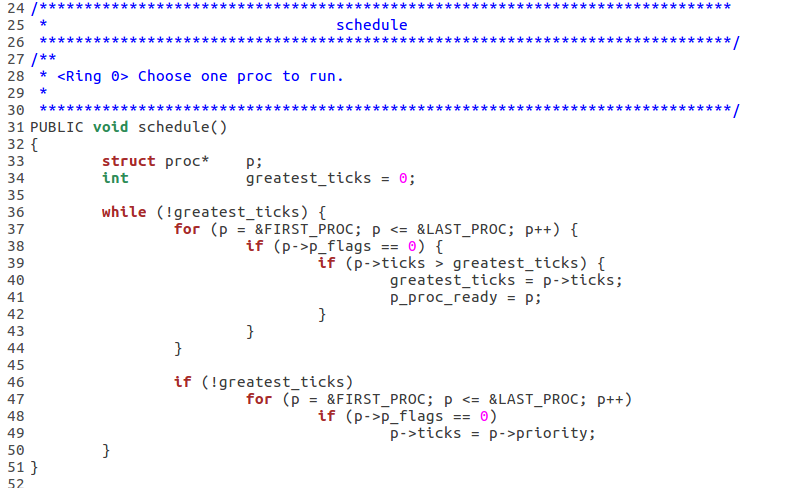
deadlo ck( )简单地判断是否发生死锁。方法是判断消息的发送是否构成一个环，如果构成环则意味着发生死锁，比如A试图发消息给B，同时B试图给C，C试图给A发消息，那么死锁就发生了，因为A、B和C三个进程都将无限等待下去



创建进程表



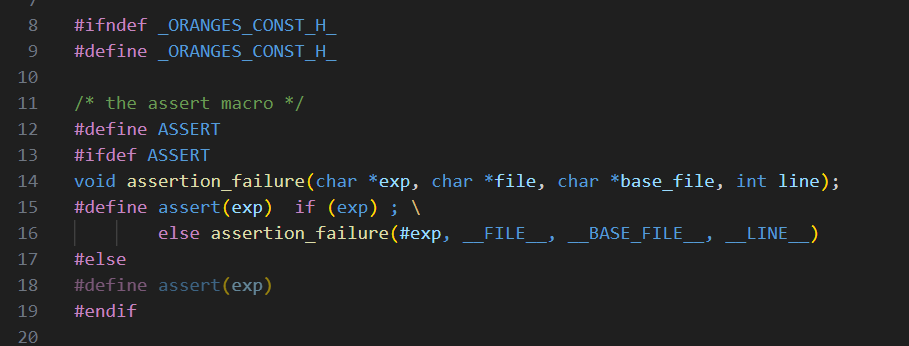
增加消息机制之后的进程调度



可以看到，当且仅当p\_flags为零时，一个进程才可能获得运行的机会

### 解释一下assert、Panic的实现过程

**（1）Assert：**



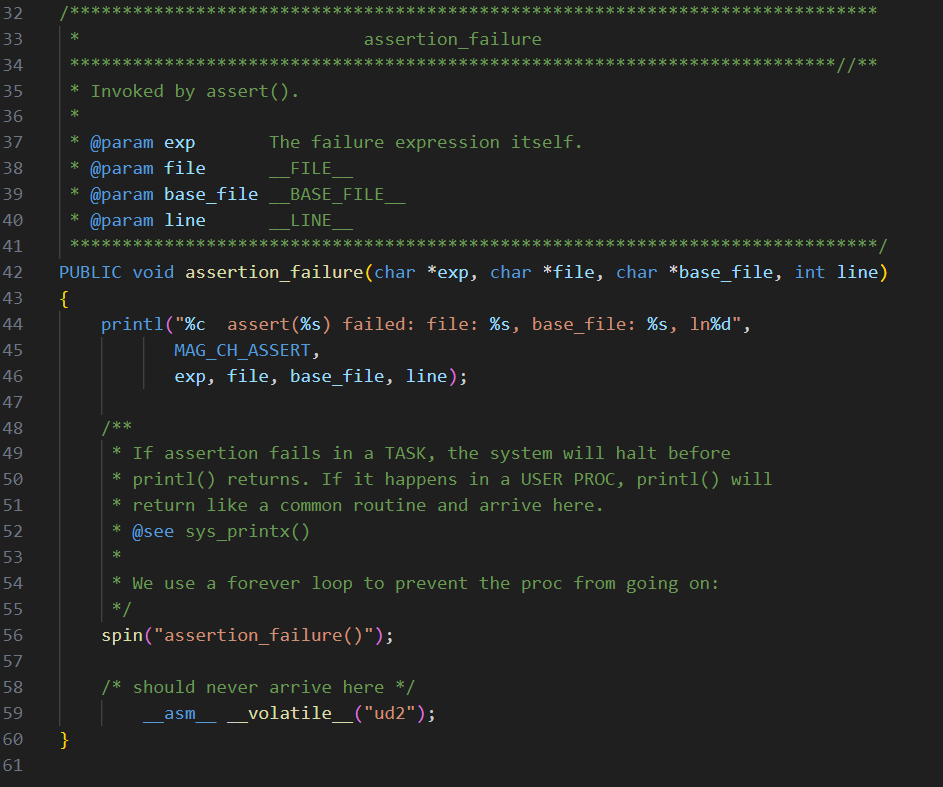
其中的\_ FILE\_、\_ BASE FILE\_ 和\_LINE\_三个宏的意义如下:

• \_ 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中的某一一个assert函数失败了，则\_FILE \_为n.h,\_BASE FILE\_ 为m.co

• \_LINE\_ 将被展开成当前的行号。

然后调用assertion failure ()这个函数，作用是将错误发生的位置打印出来。函数如下：



函数代码其实很简单，使用了一个改进后的打印函数printl（），它实际上是一个定义成printf的宏，将调用一个叫做printx的系统调用并最终调用函数sys\_printx()(它位于tty.c中）。

sys\_printx()内容如下图：

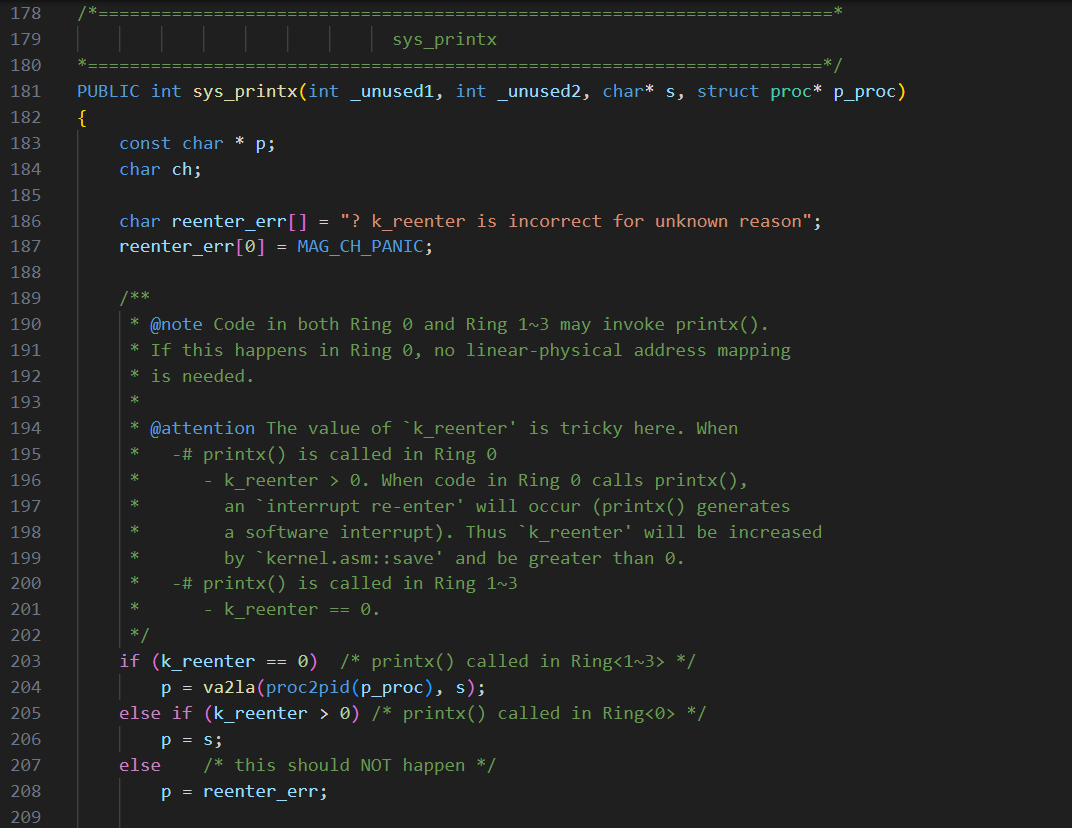
容易看到，sys\_ printx() 将首先判断首字符是否为预先设定的“ Magic Char”，如果是的话，则做响应的特殊处理。

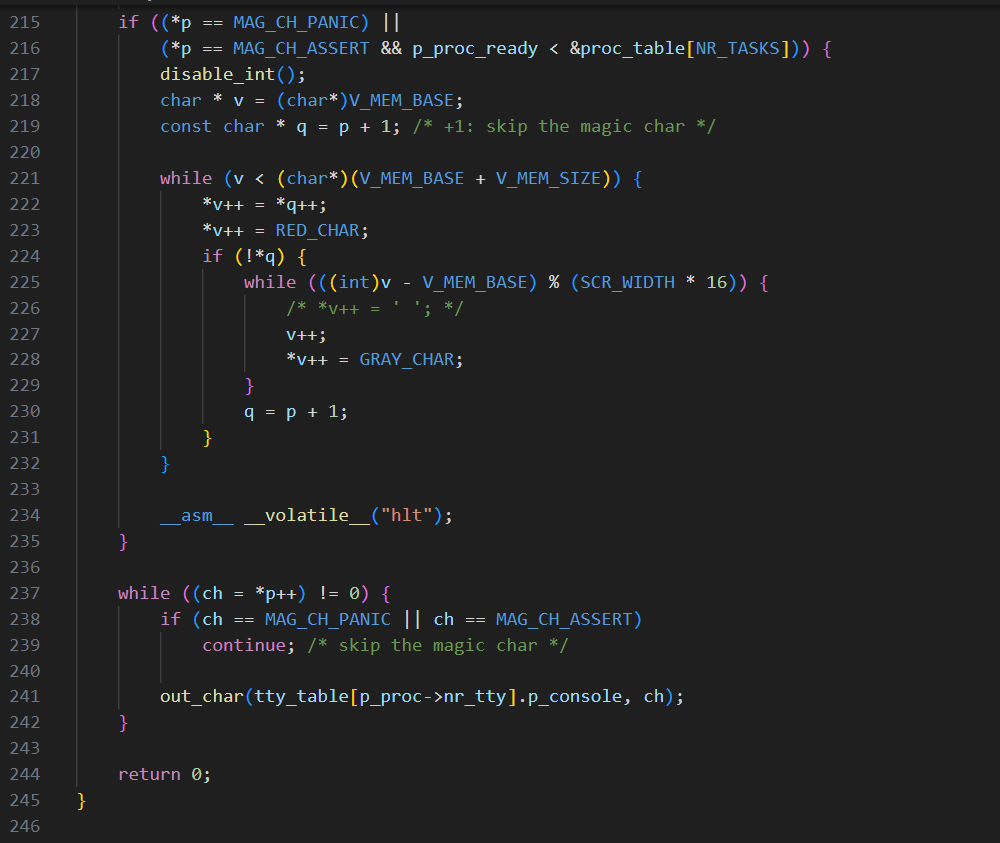
而assertion failure()就使用了MAG\_CH\_ASSERT作为“Magic Char" 。当sys\_ printx() 发现传入字符串的第一个字符是MAG\_CH\_ASSERT时，会同时判断调用系统调用的进程是系统进程(TASK) 还是用户进程(USER PROC)。

• 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。





**（2）panic**

panic跟assert类似，也使用了sys\_printx()和“Magic Char”，不过它要更简单一些。



由于panic只会用在系统任务所处的Ringl或Ring0,所以sys\_ printx()遇到MaG\_ CH\_ANIC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。

### 在本部分的消息机制中，如何实现同步和异步通信的？对进程如何调度管理的？

在这一部分，进程间只能进行同步通信。也就是说发送方发送消息后会等待接收方处理完毕并返回消息后才继续执行后续的操作。

消息发送和接收的过程如下：

**假设有进程A想要向B发送消息M，那么过程将会是这样的:**

1. A首先准备好M。

2. A通过系统调用sendrec,最终调用msg\_ send.

3.简单判断是否发生死锁。

4.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。

**假设有进程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:**

1. B准备一个空的消息结构体M,用于接收消息。

2. B通过系统调用sendrec,最终调用msg\_ receive。

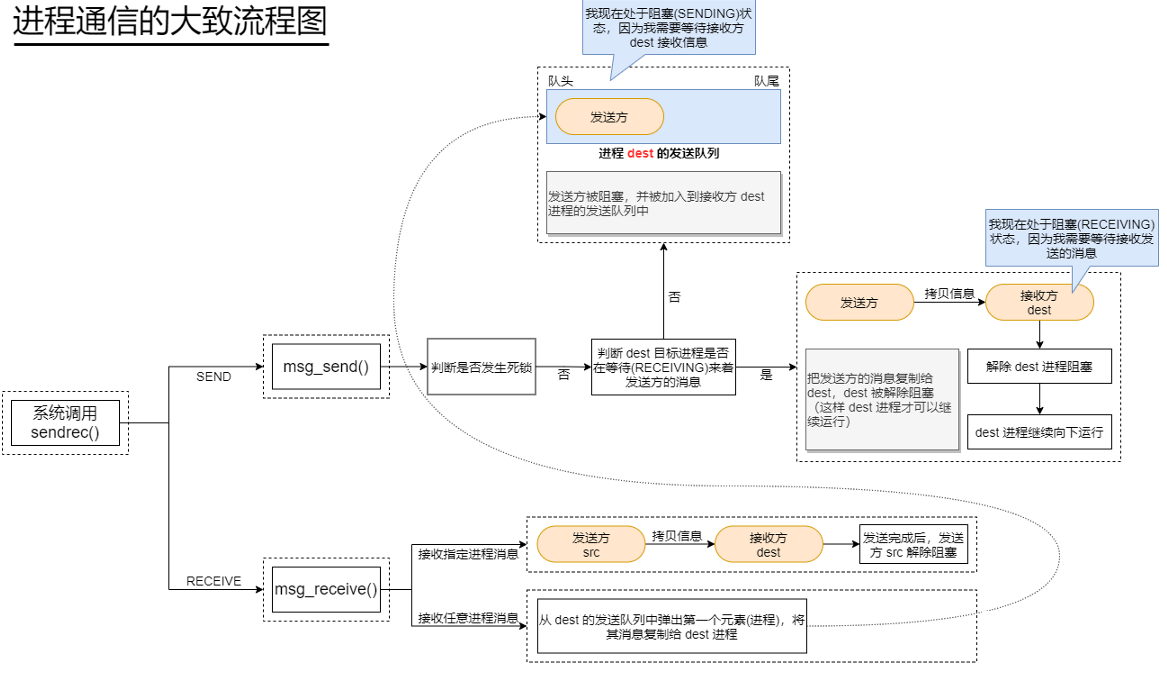
3.判断B是否有个来自硬件的消息(通过has\_ int\_ msg), 如果是，并且B准备接收来自中断的消息或准备接收任意消息，则马上准备一个消息给B,并返回。

4.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- 个卡(如果队列非空的话) ,将其消息复制给M。

5.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。

6.如果此时没有任何进程发消息给B, B会被阻塞。

值得说明的是，不管是接收方还是发送方，都各自维护一个消息结构体，只不过发送方的结构体是携带了消息内容的而接收方是空的。由于我们使用同步IPC,一方的需求一发送或接收一 只有被满足之后才会继续运行， 所以操作系统不需要维护任何的消息缓冲，实现起来也就相对简单。



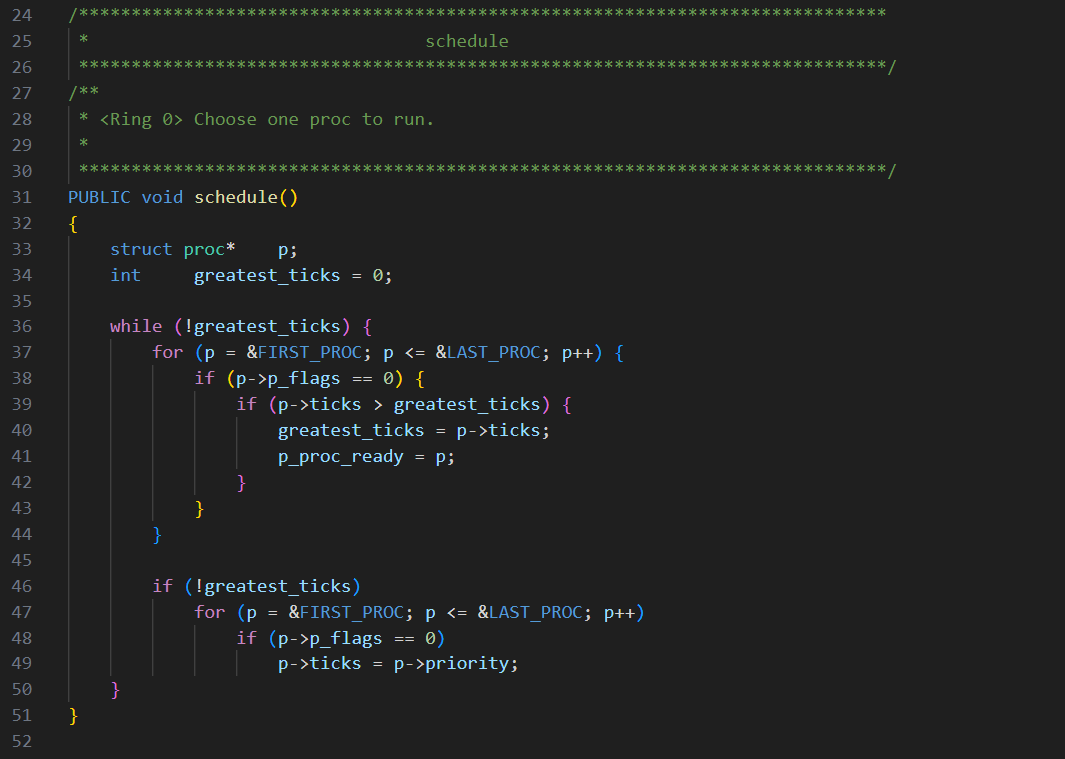
**对进程如何调度管理：**

如今的每个进程增加了两种可能的状态: SENDING和RECEIVING。

相应的，需要在进程调度的时候区别对待。凡是处于SENDING或RECEIVING状态的进程，不再让它们获得CPU。也就是说，将它们“阻塞”了。

一个进程是否阻塞 ，已经由进程表中的p\_ flags项决定，我们不需要额外做什么工作。

修改后的调度函数：



函数解释如下：

这是一个简单的调度函数，用于在多个进程之间选择下一个要执行的进程。

代码开始通过一个 while 循环，条件 !greatest\_ticks 表明只要 greatest\_ticks 为 0，就会一直执行下去。

在循环中，通过遍历从 FIRST\_PROC 到 LAST\_PROC 的进程，寻找未被标记为正在运行（p\_flags == 0）的进程。

对于每个找到的未运行进程，检查其 ticks 是否大于 greatest\_ticks，如果是，则更新 greatest\_ticks 为当前进程的 ticks 值，并将 p\_proc\_ready 指向这个进程。

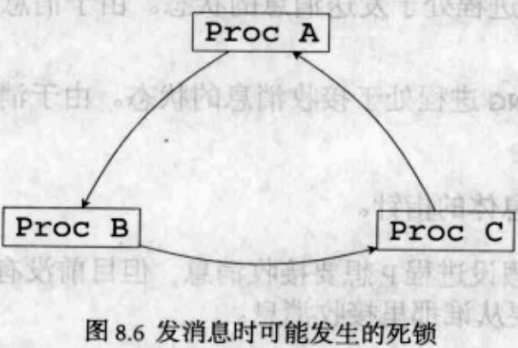
如果找不到 greatest\_ticks 大于 0 的进程，表示所有进程都没有可执行的时间片（ticks 都为 0），那么会再次遍历所有进程，并将它们的 ticks 设置为它们的优先级 priority。

循环会持续执行，直到找到一个有时间片的进程（greatest\_ticks 大于 0），然后将 p\_proc\_ready 指向该进程，从而选择它作为下一个要执行的进程。

这段代码的目的是在多个进程之间进行轮转调度，并且给予已经消耗完时间片的进程一个新的时间片（与其优先级相关），以确保所有进程都有机会执行。

### 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. 首先，死锁产生的四个必要条件是：互斥、请求和保持、不可剥夺、循环等待
2. 本章节中，使用deadlock()函数来简单地判断是否发生死锁，方法原理如下：判断消息传递图中是否存在循环等待赖，即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。举个例子，如下图所示，如果A试图发消息给B，同时B试图发消息给C，C试图发消息给A，则发生死锁，因为A、B、C三个进程都将无限等待下去。



1. 简单地判断是否发生死锁的代码如下：

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) {

*/\* print the chain \*/*

                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;

}

1. 代码分析及正确性验证：
2. 在操作系统中，进程之间通过消息传递来进行通信，如果存在循环等待，就可能导致死锁。这段代码就是一个用于检测消息传递图中是否存在循环等待的函数。
3. 函数deadlock接受两个参数src和dest，分别表示想要发送消息的进程和消息的接收者。函数使用了一个while循环来遍历消息传递图，检查是否存在循环等待。具体步骤如下：
4. 从接收者dest开始，通过p\_sendto指针向上追溯发送消息的路径。
5. 如果遇到进程的p\_flags标志中包含SENDING标志，说明该进程正在发送消息，继续向上追溯。
6. 如果发现某个进程的p\_sendto等于src，则表示消息传递图中存在循环依赖，这会导致死锁。此时会打印消息传递的路径，并返回1。
7. 如果没有发现循环依赖，返回0表示安全。
8. 在这段代码中，它只是判断消息传递中是否会产生死锁，而并没有直接提供解决死锁的代码，因此可能需要在其他地方实现解决死锁的策略，比如资源分配策略、破坏循环等待、死锁检测与恢复等。

### 简要分析基于IPC，如何扩展get\_ticks的方法

1. 使用IPC替换系统调用get\_ticks，用收发消息的方法重新实现。由于收发消息需要有两方参与，所以我们需要在IPC机制中定义一个新的消息类型，用于请求获取系统时钟的信息，在这里我们新建一个系统进程SYSTASK来接收用户进程的消息，并且返回ticks值。
2. 新增一个系统进程task\_sys，和TESTA进行通信。该task\_sys()是SYSTASK的主循环
3. 对task\_sys来说主要完成以下3件事：接收任意进程发送的消息、识别消息、根据消息特性转发到特定进程。其中，调用的send\_recv()函数实现了收发消息的功能，task\_sys代码如下：

*/\*task\_sys\*/*

*/\*<Ring 1> The main loop of TASK SYS.\*/*

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;

        }

    }

}

1. 如果首先执行到系统进程，发送消息，触发中断到ring0级，完成消息发送，之后回到系统进程
2. 待进程切换到TESTA，接受消息，触发中断到ring0级，完成消息接受，之后回到TESTA
3. 系统进程SYSTASK就绪后，修改一下函数get\_ticks

PUBLIC int **get\_ticks**()

{

    MESSAGE msg;

**reset\_msg**(&msg);

    msg.type = GET\_TICKS;

**send\_recv**(BOTH, TASK\_SYS, &msg);

    return msg.RETVAL;

}

void **TestA**()

{

    while (1) {

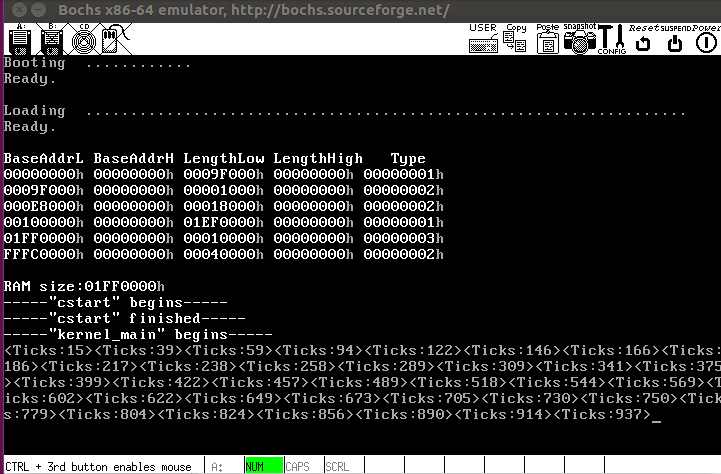
**printf**("<Ticks:%d>", **get\_ticks**());

**milli\_delay**(200);

    }

}

1. 我们以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值放人消息并发给用户进程，用户进程会接收到它，完成整个任务。
2. 我们来运行一下，结果如图所示。进程TestA调用get\_ticks之后，成功地打印出了它们的值，这表明我们的消息机制工作良好。



# 实验结果总结

（对实验结果进行分析。并理论联系实际，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。）

微内核与宏内核：

宏内核的优势在于其逻辑简单，直截了当，实现起来也容易，而且也因为它的直接，避免了像微内核那样在消息传递时占用资源

而微内核的优势在于，它的逻辑虽相对复杂但非常严谨，结构上显得非常优雅和精致，而且程序更容易模块化，从而更容易移植。

从编程的难易程度上来看，宏内核看上去具有一定优势，因为它很直接，不需要绕弯子，但从长期来看，当内核逐渐变大，微内核的结构会更加清晰。虽然选择微内核意味着有调试起来有些困难的消息机制摆在面前，但从设计理念上来看，微内核更加 “摩登”，更“酷”

分析IPC实现的技巧与细节

IPC（进程间通信）是指在操作系统中，不同进程之间进行数据交换和通信的机制。

共享内存：共享内存是一种高效的IPC方式，它允许多个进程直接访问同一块内存区域。通过共享内存，不同进程可以在内存中读写数据，实现快速的数据传输。在使用共享内存时，需要注意同步机制，以避免数据竞争和一致性问题。

信号量：信号量是一种用于进程同步的机制，通过对信号量的操作（如P操作和V操作），不同进程可以实现互斥访问共享资源。信号量可以用于实现临界区保护、进程间同步等功能。

消息队列：消息队列是一种基于消息的通信方式，发送进程将消息发送到消息队列，接收进程从消息队列中接收消息。消息队列可以实现点对点通信和发布/订阅模式，适用于异步通信和解耦合的场景。

管道：管道是一种单向的通信机制，它可以在相关进程之间传递数据。管道可以分为匿名管道和有名管道两种形式，匿名管道只能在具有亲缘关系的进程之间使用，而有名管道则可以在无关进程之间使用。

套接字：套接字是一种网络编程中常用的通信机制，它可以在不同主机上的进程之间进行通信。套接字提供了一种基于TCP/IP协议的可靠、双向的通信方式，通过套接字，进程可以实现网络通信和远程调用

同步IPC有若干的好处，比如：

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。

# 个人分工及心得体会

（每个人分别填写自己在本次实验中的分工，并总结实验的心得体会。）

## 张子航

完成本次实验要解决的问题的第2.1、2.2题，并撰写实验报告的对应部分。

在本次实验中，我学习和理解了各种IPC机制的原理和使用方法，包括共享内存、信号量、消息队列、管道等。了解了它们的特点、优缺点以及适用场景，也能够更好地选择和设计合适的IPC方式。同时也意识到了进程间通信需要考虑同步和互斥问题，以确保数据的正确性和一致性，要注意使用适当的同步机制（如信号量）来保护共享资源，避免数据竞争和死锁等问题的发生

## 辜汝曦

完成本次实验要解决的问题的第1.1-1.3题，并撰写实验报告的对应部分。

在本次实验中，我从系统调用的层面，认识了微内核与宏内核的概念及其区别，并了解了目前主流的桌面操作系统，如Windows、Linux、Mac OS的内核结构。从本次实验中，我拓展了自己的视野，认识到一个真正可用的操作系统结构的复杂性。我之前就听说过不少关于操作系统谁更好用的争论，譬如Windows和Mac OS、或者Windows与Linux。之前我的评判标准就只有UI的美观程度，本次实验使我可以在一个更深入的层面来分析并评判目前的主流操作系统。

## 杨馨悦

完成本次实验要解决的问题的第2.5、2.6题，并撰写实验报告的对应部分。

通过本次实验，我了解了微内核架构和宏内核架构的工作原理和各自的优缺点，了解了进程间通信IPC的优点及其实现的技巧与细节。这次实验让我掌握了进程间通信的消息机制，掌握了通过收发消息的方法重新实现get\_ticks的基本思路，并对操作系统内核的设计与实现有了更深入的理解。同时还增强了我对操作系统原理的理解，让我对操作系统的内核架构有了更清晰的认识。

## 赵敏

完成本次实验要解决的问题的第2.3、2.4题，并撰写实验报告的对应部分。

在本次实验中，我认识了微内核与宏内核的概念及其区别，了解了它们各自的优缺点和目前业界对于它们的看法。学习了实现进程间同步通信的代码方法，并且对于两个通信进程间的不同状态转化有了详实的了解并且再次回顾了进程调度的方法。还学习到assert、panic的实现过程，了解到系统进程和用户进程出现错误的不同处理方法。这次实验使得我对于操作系统内核的设计与实现有了更深入的理解。