

**HUBEI UNIVERSITY OF AUTOMOTIVE TECHNOLOGY**



**操作系统**

**实践设计报告**

|  |  |
| --- | --- |
| 课设题目： | 阅读操作系统源代码 |
| 专 业： | 计算机科学与技术 |
| 班 级： | 222 |
| 姓 名： | 章崇文 |
| 完成日期： | 2024.6.25 |
| 指导教师： | 陈艳海 |

目录

[一 编译linux内核源码 4](#_Toc170648110)

[1.1 下载相关内核 4](#_Toc170648111)

[1.2安装相关依赖包 5](#_Toc170648112)

[1.3 执行编译 5](#_Toc170648113)

[1.4 测试 8](#_Toc170648114)

[二 了解linux初始化相关的数据结构，及相关的函数 9](#_Toc170648115)

[2.1 CPU与硬件初始化 9](#_Toc170648116)

[2.2内存管理初始化 9](#_Toc170648117)

[2.3设备驱动初始化 10](#_Toc170648118)

[2.4 Linux内核初始化setup\_arch具体分析 10](#_Toc170648119)

[setup\_initial\_init\_mm() 14](#_Toc170648120)

[early\_ioremap\_setup() 14](#_Toc170648121)

[jump\_label\_init() 15](#_Toc170648122)

[parse\_early\_param() 15](#_Toc170648123)

[paging\_init() 15](#_Toc170648124)

[bootmem\_init() 15](#_Toc170648125)

[kasan\_init() 15](#_Toc170648126)

[三 分析与内核相关的函数 15](#_Toc170648127)

[3.1 SIGNAL.C - 信号处理机制 15](#_Toc170648128)

[3.2 EXIT.C - 进程终止与资源回收 16](#_Toc170648129)

[3.3 FORK.C - 进程创建与克隆 16](#_Toc170648130)

[关联与互动 16](#_Toc170648131)

[四 掌握linux的内存管理机制 17](#_Toc170648132)

[4.1 虚拟内存管理 17](#_Toc170648133)

[4.2 页式管理 17](#_Toc170648134)

[4.3 伙伴系统（Buddy System） 17](#_Toc170648135)

[4.4 分区管理 17](#_Toc170648136)

[4.5 交换与压缩 17](#_Toc170648137)

[4.6 Cache管理 18](#_Toc170648138)

[4.7 内核与用户空间内存 18](#_Toc170648139)

[五 分析linux字符驱动程序 18](#_Toc170648140)

[5.1 驱动结构与注册 18](#_Toc170648141)

[5.2 设备文件与主次设备号 18](#_Toc170648142)

[5.3 打开与关闭函数 18](#_Toc170648143)

[5.4 读写操作 19](#_Toc170648144)

[5.5 IOCTL（Input/Output Control） 19](#_Toc170648145)

[5.6 中断处理（可选） 19](#_Toc170648146)

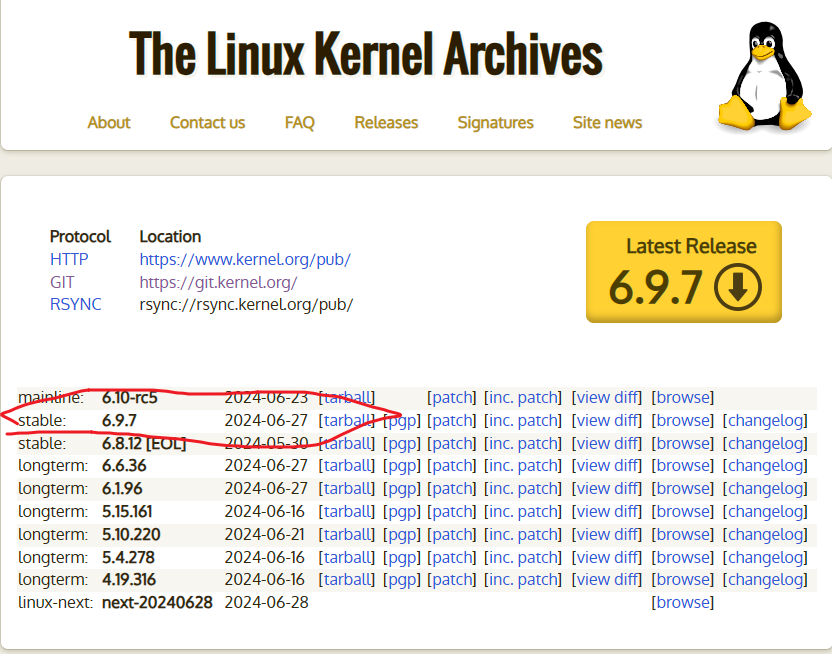
[5.7 同步与异步操作 19](#_Toc170648147)

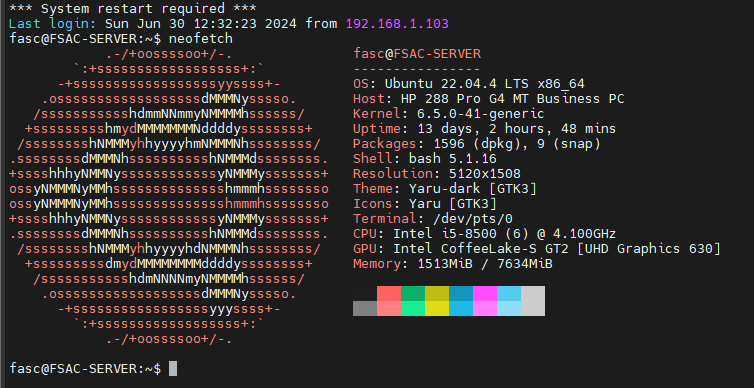
[5.8 错误处理 19](#_Toc170648148)

# 一 编译linux内核源码

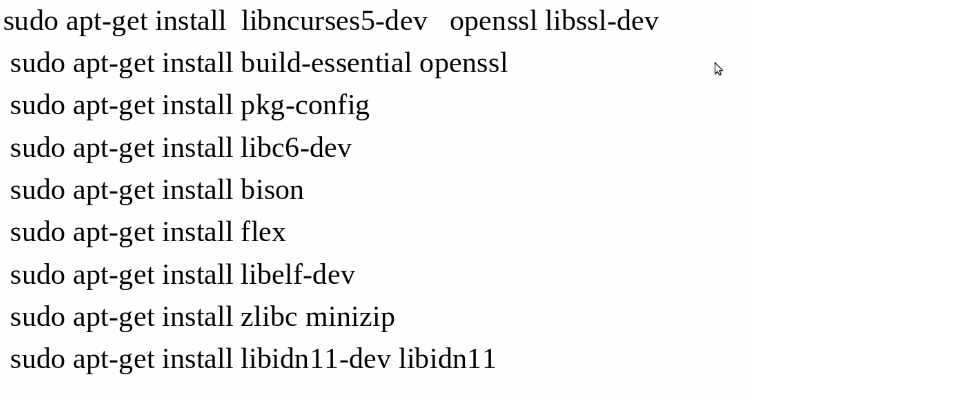
## 1.1 下载相关内核

需要下载一个新的Linux，在linux内核官方网站[www.kernel.org](https://www.kernel.org/)下载。没有使用教材推荐的Linux0.11，而是选择了 Linux 6.9.7作为本次实践。



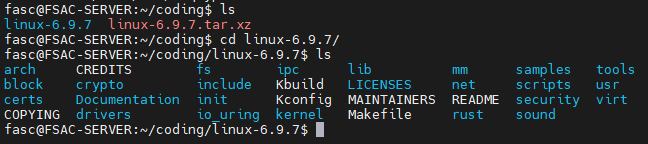
实践平台为个人服务器 ubantu 24.04，使用ssh进行连接编译

## 1.2安装相关依赖包

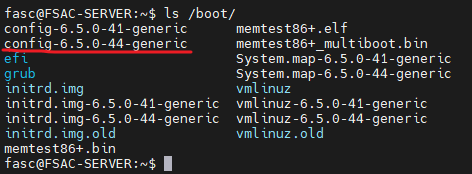


## 1.3 执行编译

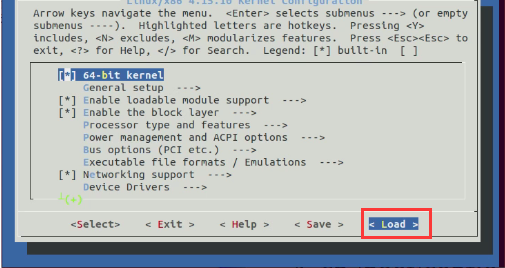
解压缩linux

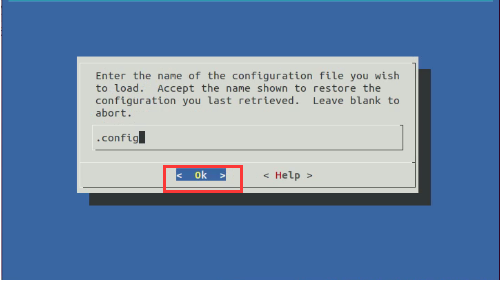


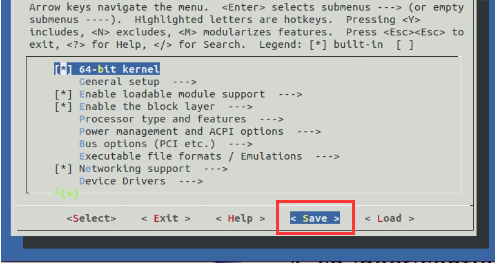
查看旧内核版本

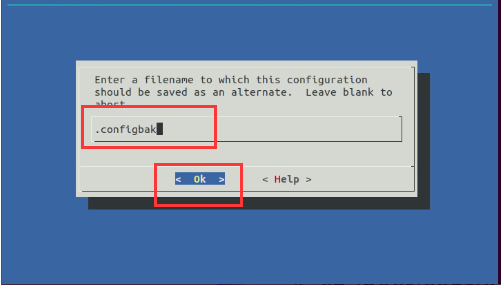


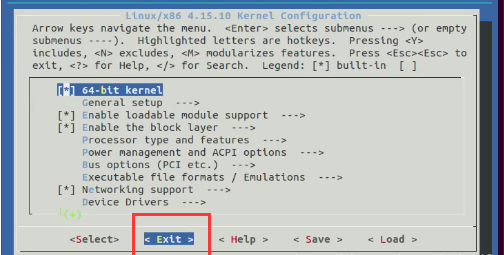
对内核选项进行配置











make clean //删除配置时留下的一些不用的文件

make bzImage -j4 //编译内核

make modules -j4 //编译模块

make INSTALL\_MOD\_STRIPE=1 modules\_install //安装模块

执行完这条命令后， /lib/modules 目录下应该生成一个 6.9.7 （版本号）目录

内核已经全部编译完毕，接下来拷贝内核镜像

把内核镜像和Ststem.map拷贝到/boot目录下

cp /usr/src/linux-6.9.7/arch/x86/boot/bzImage /boot/vmlinuz-6.9.7-generic

cp /usr/src/linux-6.9.7/System.map /boot/System.map-6.9.7

ln -s /boot/System.map-6.9.7 /boot/System.map

最后一步跟新grub配置文件

cd /boot/grub

chmod 777 grub.cfg

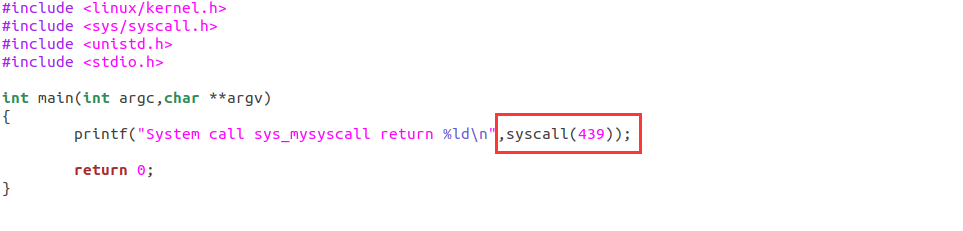
update-grub

重新启动Ubuntu机器可以看到自己的新内核

## 1.4 测试

重新启动机器后，用一个C文件测试

sudo gedit test.c

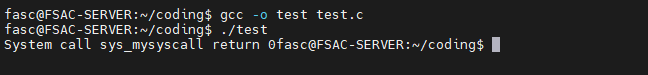


编译运行

gcc -o test test.c

./test

运行结果：



回忆一下系统调用的内核函数：



**返回 0 说明执行成功**，但是现在没有看到打印信息，因为打印信息打在了日志文件中xa

# 二 了解linux初始化相关的数据结构，及相关的函数

## 2.1 CPU与硬件初始化

在Linux系统的启动序列中，CPU自检（POST，Power-On Self Test）完成后，引导加载程序（如GRUB）加载内核映像至内存，并传递控制权给内核。此阶段涉及关键硬件资源的识别与配置，包括但不限于CPU模式设置、内存地址映射、中断控制器初始化，以及基本外设的预备工作，确保内核能在一个稳定且功能完备的硬件环境中启动。

**start\_kernel()函数：内核初始化的核心**

start\_kernel()，位于init/main.c，是Linux内核启动序列的中枢，它负责协调和执行内核各子系统的初始化。该函数的执行流程涵盖了架构特定配置、内存管理准备、进程与调度机制建立、设备驱动加载，以及文件系统初始化等多个层面，具体包括但不限于：

* **架构初始化**：通过setup\_arch()调用实现特定于硬件平台的配置。
* **内存管理准备**：构建内存区域划分、页表设置及伙伴系统初始化。
* **进程与调度器设置**：创建初始进程（PID=0，Swapper/Idle进程）和初始化调度策略。
* **驱动与设备模型搭建**：为后续的设备驱动加载与管理打下基础。

## 2.2内存管理初始化

**内存域与页帧分配**

* **max\_zone\_pfn**：标示了每个内存区域（zone）的最大物理页帧号，是内存管理初始化中至关重要的参数，它帮助系统理解各个内存区域的边界。
* **early\_node\_map**：在早期启动阶段，用于快速映射内存节点，为后续内存管理提供基础框架，确保内存分配策略能有效利用系统内存资源。

**伙伴系统（Buddy System）**

伙伴算法是一种高效的内存块分配策略，通过将内存分为多个大小相同的块链表来管理空闲内存区域。free\_area\_init\_nodes函数遍历所有内存区域，初始化伙伴系统的管理数据结构（free\_area数组），为动态内存分配提供支撑。

**slab分配器**

Slab分配器优化了内核频繁分配和释放小对象的性能。它预先分配和缓存一定数量的对象，减少分配时的碎片化问题。通过维护多个slab列表（部分满、满、空等状态），slab分配器能快速响应内核对特定类型对象的请求，提升系统整体效率。

**进程与调度器初始化**

**初始化进程0 (swapper/Idle)**

进程0，也称作Swapper或Idle进程，是系统中首个创建的内核线程，其主要职责包括维护空闲进程状态、执行CPU空闲循环等。它参与构建最初的页表、内存管理结构，为后续进程的创建与运行奠定基础。

**kernel\_init()与kthreadd**

* **kernel\_init()**：此函数负责执行系统更深层次的初始化脚本，包括执行用户态的初始化进程（如systemd或init），标志着内核向用户空间过渡的关键步骤。
* **kthreadd**：作为内核线程管理的中心，负责创建和销毁其他内核线程，是内核中众多后台服务的基础。

**文件系统与设备驱动初始化**

**挂载根文件系统**

根文件系统的挂载是Linux启动过程中的一项核心任务，它涉及文件系统类型检测、必要模块的自动加载，以及通过mount\_root()函数完成实际挂载操作。这一过程确保了系统能访问基本的系统文件和配置，为运行用户空间程序提供环境。

## 2.3设备驱动初始化

设备驱动的初始化包括硬件探测、匹配与加载相应的驱动模块，以及配置设备参数。这一系列操作由内核的设备模型框架统一管理，通过udev或其他设备管理器实现动态加载，保证了系统能够识别并启用所有可用硬件设备，为用户提供全面的硬件支持。

## 2.4 Linux内核初始化setup\_arch具体分析

C语言代码的入口是start\_kernel这个函数，本文要介绍其中的set\_arch这个函数，该函数的作用是查找给定机器ID的数据结构信息、配置内存条信息、解析bootloader传递命令行参数，然后根据machine\_desc结构体所记录的信息对机器进行一些必要的设置，最后开始正式建立完整的页表。

setup\_arch这个重磅函数，该函数是一个架构相关函数。文本以ARM架构，[linux版本](https://link.zhihu.com/?target=https%3A//so.csdn.net/so/search%3Fq%3Dlinux%25E7%2589%2588%25E6%259C%25AC%26spm%3D1001.2101.3001.7020)6.9.7为例，来分析该函数。先来一份该函数的概览：

函数定义如下（/ubantu/arm/setup.c）：

void \_\_init \_\_no\_sanitize\_address setup\_arch(char \*\*cmdline\_p)

{ //init\_mm是init进程（0号进程）的内存描述符

//初始化内核的mm结构体的代码段、数据段和栈的结束地址

setup\_initial\_init\_mm(\_stext, \_etext, \_edata, \_end);

//cmdline\_p指向boot启动参数

\*cmdline\_p = boot\_command\_line;

kaslr\_init();

/\*

\* If know now we are going to need KPTI then use non-global

\* mappings from the start, avoiding the cost of rewriting

\* everything later.

\*/

//这里根据是否开启kpli(内核页表隔离)决定是否创建非全局页表

arm64\_use\_ng\_mappings = kaslr\_requires\_kpti();

//早期固定映射初始化，搞了个框架，还没有填充pte页表项

early\_fixmap\_init();

//早期io映射初始化

early\_ioremap\_init();

//解析fdt中内存，加入到memblock中

setup\_machine\_fdt(\_\_fdt\_pointer);

/\*

\* Initialise the static keys early as they may be enabled by the

\* cpufeature code and early parameters.

\*/

//初始化jump-label子系统

jump\_label\_init();

//解析early options这两个参数

parse\_early\_param();

dynamic\_scs\_init();

/\*

\* Unmask asynchronous aborts and fiq after bringing up possible

\* earlycon. (Report possible System Errors once we can report this

\* occurred).

\*/

//恢复daif为不可以IRQ

local\_daif\_restore(DAIF\_PROCCTX\_NOIRQ);

/\*

\* TTBR0 is only used for the identity mapping at this stage. Make it

\* point to zero page to avoid speculatively fetching new entries.

\*/

//移除idmap\_pg\_dir页表

cpu\_uninstall\_idmap();

//不支持xen

xen\_early\_init();

efi\_init();

if (!efi\_enabled(EFI\_BOOT)) {

if ((u64)\_text % MIN\_KIMG\_ALIGN)

pr\_warn(FW\_BUG "Kernel image misaligned at boot, please fix your bootloader!");

WARN\_TAINT(mmu\_enabled\_at\_boot, TAINT\_FIRMWARE\_WORKAROUND,

FW\_BUG "Booted with MMU enabled!");

}

//初始化memblock，到这里，不能使用的内存都放入reserved了

arm64\_memblock\_init();

//切换为细粒度映射，释放粗粒度映射页表内存，memblock初始化完成

paging\_init();

//解析acpi表的信息，存放到acpi\_initrd\_files中

acpi\_table\_upgrade();

/\* Parse the ACPI tables for possible boot-time configuration \*/

//解析ACPI表，获得可能的引导时配置

acpi\_boot\_table\_init();

//解析设备树信息，创建设备节点树

if (acpi\_disabled)

unflatten\_device\_tree();

//初始化内存基本数据结构，pg\_data\_t,、zone、page

bootmem\_init();

//KASAN初始化

kasan\_init();

request\_standard\_resources();

//设置after\_paging\_init为1

early\_ioremap\_reset();

//设备树中psci初始化

if (acpi\_disabled)

psci\_dt\_init();

else

psci\_acpi\_init(); //acpi种psci初始化

init\_bootcpu\_ops(); //初始化cpu启动方法集合

smp\_init\_cpus(); //smp的其他cpu初始化

smp\_build\_mpidr\_hash();

/\* Init percpu seeds for random tags after cpus are set up. \*/

kasan\_init\_sw\_tags(); //kasan的东西

#ifdef CONFIG\_ARM64\_SW\_TTBR0\_PAN

/\*

\* Make sure init\_thread\_info.ttbr0 always generates translation

\* faults in case uaccess\_enable() is inadvertently called by the init

\* thread.

\*/

init\_task.thread\_info.ttbr0 = phys\_to\_ttbr(\_\_pa\_symbol(reserved\_pg\_dir));

#endif

if (boot\_args[1] || boot\_args[2] || boot\_args[3]) {

pr\_err("WARNING: x1-x3 nonzero in violation of boot protocol:\n"

"\tx1: %016llx\n\tx2: %016llx\n\tx3: %016llx\n"

"This indicates a broken bootloader or old kernel\n",

boot\_args[1], boot\_args[2], boot\_args[3]);

}

}

setup\_arch函数主要是处理cpu体系相关架构，我们是arm64平台，这个函数处理arm64的一些初始化，主要包括：

* 初始化内核的mm结构体的代码段、数据段和栈的结束地址；
* 调用函数early\_fixmap\_init进行早期固定映射初始化；
* 调用函数early\_ioremap\_init进行早期io映射初始化；
* 调用函数setup\_machine\_fdt解析fdt中内存，加入到memblock中；
* 调用函数parse\_early\_param/解析early options这两个参数；
* 调用函数local\_daif\_restore恢复cpu的daif状态为不可以IRQ；
* 调用函数cpu\_uninstall\_idmap移除idmap\_pg\_dir页表
* 调用函数efi\_init初始化 UEFI 系统中的各个硬件设备以及配置各种环境参数，从而为操作系统的正常启动做好准备
* 调用函数arm64\_memblock\_init初始化memblock，
* 调用函数 paging\_init切换为细粒度映射，释放粗粒度映射页表内存
* 调用函数acpi\_table\_upgrade解析acpi表的信息，存放到acpi\_initrd\_files中
* 调用函数acpi\_boot\_table\_init解析ACPI表，  
  如果找不到ACPI表，调用函数unflatten\_device\_tree()解析设备树信息，创建设备节点树
* 调用函数bootmem\_init初始化内存基本数据结构，pg\_data\_t,、zone、page
* 如果ACPI表不存在，调用函数psci\_dt\_init进行设备树中psci初始化；否则调用函数psci\_acpi\_init进行acpi种psci初始化  
  调用函数init\_bootcpu\_ops初始化cpu启动方法集合
* 调用函数smp\_init\_cpus进行smp的其他cpu初始化

## setup\_initial\_init\_mm()

早期内存的初始化，给出了代码段的起始与结束位置，数据段的结束位置，堆地址结束位置。

void setup\_initial\_init\_mm(void \*start\_code, void \*end\_code,

void \*end\_data, void \*brk)

{

init\_mm.start\_code = (unsigned long)start\_code;

init\_mm.end\_code = (unsigned long)end\_code;

init\_mm.end\_data = (unsigned long)end\_data;

init\_mm.brk = (unsigned long)brk;

}

## early\_ioremap\_setup()

早期 ioremap 初始化,将 I/O 的物理地址映射到虚拟地址。当 CPU 读取一段物理地址时，它可以读取到映射了 I/O 设备的物理 RAM 区域。ioremap 就是用来把设备内存映射到内核地址空间的。  
该函数是一个架构不相关的函数，位于 mm/early\_ioremap.c。

## jump\_label\_init()

架构无关函数，位于 kernel 目录下，初始化 jump-label 子系统，jump-label 用于取消 if 判断分支，通过运行时修改代码，来提高执行的效率。

## parse\_early\_param()

架构无关函数，解析早期传入的参数。

## paging\_init()

完成系统分页机制的初始化工作, 建立页表, 从而内核可以完成虚拟内存的映射和转换工作，这一个函数执行完成之后，就可以通过虚拟地址来访问实际的物理地址了。

## bootmem\_init()

bootmem\_init 是 Linux 内核中用于初始化物理内存管理的函数。在 Linux 内核启动过程中，这个函数负责对物理内存进行初始化和管理，为后续的内存分配和使用做好准备。

## kasan\_init()

初始化 kasan 动态监测内存错误的工具，初始化完成之后，可以在内存使用越界或者释放后访问时，产生出错报告，帮助分析内核异常。

# 三 分析与内核相关的函数

在Linux内核中，signal.c、exit.c、和fork.c这三个文件分别对应着信号处理、进程退出和进程创建这三个核心功能。下面将分别对这三个文件中的关键函数进行分析，探讨它们在内核中的作用、原理以及与其他组件的互动关系。

## 3.1 SIGNAL.C - 信号处理机制

**函数分析**

* **do\_signal()**：当有信号待处理时，此函数负责将信号加入当前进程的待处理信号队列，并根据信号的性质（是否阻塞、是否立即处理等）决定是否立即执行信号处理函数。它是信号处理的核心入口点。
* **send\_signal()**：负责向指定进程发送信号。它检查信号的有效性，更新目标进程的信号位图，并唤醒进程（如果需要）。
* **do\_notify\_resume()**：处理进程从休眠状态恢复时的信号传递，确保信号在合适的时机被进程感知。

## 3.2 EXIT.C - 进程终止与资源回收

**函数分析**

* **do\_exit()**：进程退出的主函数。它执行诸如释放资源（如内存、文件描述符）、通知父进程、执行exit handlers等操作。该函数最终会导致进程被完全移除。
* **wait4()**：这是一个系统调用，允许一个进程等待其子进程的终止，并收集有关子进程的信息。它与do\_wait()函数紧密相关，后者实现了实际的等待逻辑。
* **release\_task()**：在do\_exit()流程的末尾调用，负责释放进程描述符结构本身，完成进程的生命周期。

## 3.3 FORK.C - 进程创建与克隆

**函数分析**

* **do\_fork()**：进程创建的核心函数。它通过复制父进程的内存空间、文件描述符表、信号处理等资源来创建一个新的进程。该函数实现了写时拷贝(COW)机制，以高效地利用内存资源。
* **copy\_process()**：在do\_fork()内部调用，负责实际的进程复制工作，包括分配新的task\_struct、设置进程状态、复制内存管理和文件系统相关结构等。
* **copy\_thread\_tls()**：特用于线程本地存储（TLS）的复制，在多线程环境下，确保新创建的线程拥有独立的TLS段。

## 关联与互动

这三个文件中的函数紧密关联，共同支撑着进程的生命周期管理：

* **信号处理**（signal.c）与进程行为密切相关，它可以中断进程的正常执行流程，要求进程处理特定事件（如用户中断、程序错误等）。
* **进程退出**（exit.c）不仅是进程生命周期的自然终点，也是资源回收的关键环节，影响着系统资源的有效利用。
* **进程创建**（fork.c）是多任务处理的基础，通过fork()或clone()系统调用，不仅支持了新进程的生成，也为多线程编程提供了底层支持，与信号处理和进程退出机制协同工作，确保了系统中进程的动态平衡和稳定运行。

综上所述，signal.c、exit.c、和fork.c中的函数不仅各自承担着核心的系统功能，还通过复杂的交互机制，共同维护着Linux内核中进程管理的高效与稳定。

# 四 掌握linux的内存管理机制

掌握Linux的内存管理机制涉及到理解其如何高效地使用有限的物理内存资源，同时为众多进程提供足够的虚拟地址空间，保证系统性能和稳定性。以下是Linux内存管理机制的关键方面：

## 4.1 虚拟内存管理

* **虚拟地址空间**：Linux为每个进程分配了一个独立的4GB（在32位系统上）或更大的（在64位系统上）虚拟地址空间，使得进程可以独立操作，不受物理内存限制。
* **地址映射**：通过页表和内存管理单元（MMU）将虚拟地址转换为物理地址。这种映射不是静态的，而是按需完成（即页表项的建立），仅当进程尝试访问尚未映射的页面时才发生。

## 4.2 页式管理

* **页面**：内存被分成固定大小的块，称为页面，通常为4KB。这是内存管理的基本单位。
* **页缓存**：Linux使用页缓存来缓存磁盘数据和代码，提高I/O效率，同时也作为未分配给进程的空闲内存的一部分。

## 4.3 伙伴系统（Buddy System）

* **内存分配策略**：Linux内核使用伙伴系统来管理空闲内存页面。内存被划分成11个块链表，每个链表代表不同大小的连续页框集合，便于快速分配和回收内存。
* **分裂与合并**：当请求分配内存时，伙伴系统尝试从现有块中分割出所需大小的空间；当释放内存时，尝试将相邻的空闲块合并以减少碎片。

## 4.4 分区管理

* **Zone**：内存被划分为不同的区域（Zones），如ZONE\_DMA、ZONE\_NORMAL和ZONE\_HIGHMEM，每个区域服务于不同类型的需求，比如对DMA友好的内存或高端内存。
* **Node**：在NUMA（非统一内存访问）架构中，内存被组织成节点，每个节点对应于一组CPU及其直接访问的内存，优化了多处理器系统的内存访问效率。

## 4.5 交换与压缩

* **交换空间**：当物理内存不足时，Linux会将一些不活跃的内存页移到磁盘上的交换空间，释放物理内存给更活跃的进程使用。
* **内存压缩**：Linux内核也支持内存压缩技术，尝试在将页面换出之前压缩它们，以减少实际需要写入磁盘的数据量。

## 4.6 Cache管理

* **LRU算法**：Linux使用最近最少使用（LRU）算法来管理页缓存，确保活跃数据保留在内存中，不活跃数据被替换出去。

## 4.7 内核与用户空间内存

* **内核空间**：保留给操作系统内核使用，不可由用户进程直接访问。
* **用户空间**：分配给各个用户进程，每个进程有自己的私有地址空

五 分析linux字符驱动程序

## 5.1 驱动结构与注册

* **file\_operations结构体**：这是字符驱动程序的核心，定义了一组函数指针，指向驱动提供的操作函数，如打开(open)、关闭(close)、读(read)、写(write)、ioctl（输入输出控制）等。每个函数对应设备的不同操作。
* **驱动注册**：通过register\_chrdev()或更现代的cdev\_add()函数将字符设备驱动注册到内核中。这一步骤会将file\_operations结构体与设备号关联，使得用户空间可以通过设备文件名访问该设备。

## 5.2 设备文件与主次设备号

* **设备文件**：用户空间通过/dev下的设备文件与驱动程序交互。这些文件由mknod或udev规则自动创建。
* **主次设备号**：每个设备文件关联一对设备号，主设备号标识驱动类型，次设备号区分同一类型的多个设备实例。

## 5.3 打开与关闭函数

* **open()**：当用户空间程序打开设备文件时调用，可以用来初始化设备状态，如分配资源、设置设备状态等。
* **release()或close()**：设备文件被关闭时调用，负责释放资源，恢复设备到初始状态。

## 5.4 读写操作

* **read()**：从设备读取数据到用户空间缓冲区。驱动程序需要实现数据的获取逻辑，处理数据传输。
* **write()**：将数据从用户空间缓冲区写入设备。驱动程序需处理数据的接收与存储。

## 5.5 IOCTL（Input/Output Control）

* **ioctl()**：提供了一种灵活的机制，允许驱动程序执行设备特定的操作，如配置设备参数。它通过命令码区分不同的操作。

## 5.6 中断处理（可选）

对于需要中断处理的字符设备，驱动还需要实现中断处理函数，处理硬件中断，确保数据的及时传输和处理。

## 5.7 同步与异步操作

* **同步操作**：在read或write调用中直接完成数据传输，直到操作完成才返回控制权。
* **异步操作**：可以使用异步I/O（AIO）或工作队列、中断处理等方式，使驱动能在后台处理数据传输，提高响应速度。

## 5.8 错误处理

* 在驱动程序中，合理处理并返回错误代码（如-EINVAL, -ENOMEM等）是至关重要的，确保上层调用者能正确响应错误情况。