

操作系统实验教程

以设计、实现高性能 Web 服务器为例

鲁强

中国石油大学（北京）

# 前言

操作系统课程内容涉及面广，里面讲授的概念、理论和算法较为抽象和难以理解。制约 学生理解、掌握其内容的关键因素是缺少好的实验平台。

目前针对操作系统课程开发的实验平台大体分为两种：一种以复现课程内容中理论和 相关算法实现为主；一种以操作系统内核开发为主。第一种实验平台多是以每章为单位的理 论验证型实验，例如实现银行家算法、实现 LRU 内存替换算法等。这些实验内容仅是复现

了课程中的算法，由于其缺少具体的应用环境，使得学生并不能够体会这些理论和算法在操

作系统或实际系统环境中的真实作用。并且由于实验以每章为单位，各部分实验内容之间缺 少联系，很难使得学生通过这些实验内容来真正理解、掌握和应用这些理论、算法知识。以 内核开发为主的实验平台，大多数已经构造好了基本的内核实现框架，并且已经实现了很大 一部分的代码，仅需要学生补充相关的算法代码即可。这样虽然能够加深学生对课本内容的 理解并增强其阅读代码和系统底层编程能力，但是由于整体架构已经设计好，并不能够较好 地训练学生的系统设计能力（系统设计指的是面临复杂问题时能够在综合考虑各种因素前 提下设计出合理的系统程序以最大化地利用计算机系统性能）。

为克服上述两种实验平台的缺点，本书以设计并实现一个具有较高并发性能的 Web 服

务器系统为目标，将操作系统各个部分的理论和算法知识逐步地、有机地融入到此服务器系 统的设计和开发过程中，使得学生学习到知识能够作用于一个完整的系统，使得他们在实现 每个阶段实验目标基础上逐渐地增强系统的设计能力和分析能力，使得他们得到“学以致理” 和“学以致用”两个层次的提升。

本文具体内容组织如下：

第一章，Web 服务器开发基础，在理解 TCP 和 HTTP 协议基础之上，利用 socket 编程技术实现一个简单的 Web 服务器；

第二章，Web 服务器的多进程和多线程模型，将多进程、多线程、同步与互斥等概念和理论融入到此 Web 服务器的设计中，并探讨了各种提高 Web 服务器并发处理的设计方案；

第三章，Web 服务器的内存管理，在深入分析 Linux 的内核内存管理模型和用户库内存管理模型基础之上，通过介绍 Nginx（一个高性能 Http 服务器）在内存管理中的思考和实现方法，来探讨 Web 服务器管理内容的设计方案；

第四章，Web 服务器的文件存储系统，在深入分析 Linux 的 Ext 文件系统基础之上，通过介绍支持海量小文件高速读取的 TFS（淘宝文件系统）体系结构及其特点，来探讨支持海量 web 文件的存储系统设计方案。

本书中实验内容是作者在多年操作系统实验教学过程中总结、整理并形成的。其最大的 特点为适用性较广，即能够让能力一般的同学经过逐步地学习和训练来完成一个较为完整

的系统软件，也能够让能力较为突出的学生来通过深入钻研操作系统内核和计算机系统结 构相关知识来最大程度地发挥系统软件的性能。其次，每个实验阶段的 Web 服务器性能都会比上一个阶段有所提高，这会极大地激发学生探索新理论和新方法的兴趣。最后，由于此 实验内容具有很好的衡量指标，使得教师可以通过学生完成的 Web 服务器系统性能好坏以及相关实验报告来对学生实验成绩给出客观的评价。

本书不仅可以作为操作系统上机授课教材，也可以作为计算机从业者提升自己的项目 训练手册。希望本书的内容能够对大家有所帮助！同时由于作者知识面有限，难免会出现错 误，请大家予以见谅，并提出改正建议。

鲁强

2017.8.3

**目录**

[第 1 章 Web 服务器开发基础 6](#_TOC_250032)

* 1. Web 服务器简介 6
  2. TCP 与 HTTP 协议 7
     1. [TCP/IP 协议族简介 7](#_TOC_250031)
     2. [HTTP 协议 7](#_TOC_250030)
  3. Socket 编程 11
  4. [开发环境与测试环境 18](#_TOC_250029)
     1. GCC 18
     2. 构建 makefile 23
     3. 调试代码 GDB 23
     4. 服务性能测试工具 29
     5. [性能指标 31](#_TOC_250028)
  5. 实验 1 Web 服务器初步实现 32

[第 2 章 Web 服务器的多进程和多线程模型 34](#_TOC_250027)

* 1. [背景介绍 34](#_TOC_250026)
  2. [进程模型 34](#_TOC_250025)
     1. [Linux 中进程创建相关函数 34](#_TOC_250024)
     2. [Linux 中进程通信相关函数 36](#_TOC_250023)
     3. [多进程 Web 服务器模型 44](#_TOC_250022)
     4. [实验 2 Web 服务器的多进程模型实现 45](#_TOC_250021)
  3. [线程模型 46](#_TOC_250020)
     1. [Linux 线程模型 46](#_TOC_250019)
     2. [POSIX 线程库接口 46](#_TOC_250018)
     3. [Linux 线程间同步与互斥 50](#_TOC_250017)
     4. [Web 服务器的多线程模型 52](#_TOC_250016)
     5. [Web 服务器的多线程模型 57](#_TOC_250015)
  4. [线程池模型 57](#_TOC_250014)
     1. 实验 4 Web 服务器的线程池模型 62
  5. [业务分割模型 62](#_TOC_250013)
     1. [实验 5 Web 服务器的业务分割模型 66](#_TOC_250012)
  6. [混合模型 66](#_TOC_250011)
     1. [实验 6 Web 服务器的混合模型 69](#_TOC_250010)

[第 3 章 Web 服务器的内存管理 70](#_TOC_250009)

* 1. [背景介绍 70](#_TOC_250008)
  2. Web 页面的缓存逻辑结构 71
  3. Web 页面的缓存置换算法 76
  4. Web 服务器的内存管理模型 85
     1. [Linux 内核内存管理模型 86](#_TOC_250007)
     2. [Linux 用户库函数管理内存方法 95](#_TOC_250006)
     3. [Nginx 内存管理模型 106](#_TOC_250005)
     4. [实验 8 Web 服务器的内存管理 107](#_TOC_250004)

[第 4 章 Web 服务器的文件存储系统 109](#_TOC_250003)

* 1. [背景介绍 109](#_TOC_250002)
  2. Linux 中的 Ext 文件系统 109
     1. [Ext2 文件系统结构 109](#_TOC_250001)
     2. [Ext2 文件系统分析 110](#_TOC_250000)
  3. TFS 文件系统 112
     1. TFS 文件系统架构 112
     2. TFS 文件系统性能分析 115
  4. 实验 9 Web 服务器的文件系统 115

# 第 1 章 Web 服务器开发基础

* 1. Web **服务器简介**

Web 服务器是通过 HTTP 协议将客户端请求的文件发送到客户端的软件系统。其主要功能是读取 Web 网页，并将 Web 网页发送到客户端的浏览器中。Web 服务器主要包括两种类型：静态 Web 服务器和动态 Web 服务器。静态 Web 服务器不负责代码脚本的执行， 只是将 Web 文件发送到客户端，例如 Apache、Nginx 和 IIS 等 Web 服务器；动态 Web 服务器，需运行客户请求的代码脚本，并将运行结果发送到客户端。动态 Web 服务器一般也被称为应用服务器。例如，Tomcat 应用服务器负责 JSP 代码脚本的解析和运行；Zend 应用服务器服务 PHP 代码脚本的解析和运行。在目前企业应用架构中，经常将静态服务器与动态服务器混合，以支持灵活的企业应用。例如，Apache、Nginx 和 IIS 等服务器可以通过配置相关的模块，与应用服务结合来达到即能完成静态页面传输，也能完成动态页面解析运行

的功能。

由于本实验主要以实现静态 Web 服务器为目标，在文章以后的内容中出现的“Web 服务器”特指“静态 Web 服务器”。Web 服务器主要包括 Web 文件存储，客户请求路径解析， Web 文件读取和 Web 文件传输四个部分。用户浏览器与 Web 服务器具体的交流流程如下图 1-1 所示。



**HTTP TCP/IP**

**Internet**

**HTTP TCP/IP**

**index.html File System**

[**http://abc.com/index.html**](http://abc.com/index.html)

**Client Web Server**

图 1-1 用户请求 Web 网页过程图

用户在浏览器中输入URL 链接地址[“h](http://abc.com/index.html)t[tp://abc.com/index.html](http://abc.com/index.html)”，浏览器将根据此URL封装成 HTTP 请求消息，并通过 TCP/IP 协议将此请求消息发送到指定地址的 Web 服务器中。Web 服务器接收到此HTTP 请求消息后，首先进行解析并从中获得到用户需要 index.html 文件的信息，然后在文件系统中查找此文件并读取此文件内容，然后将此文件内容封装成

HTTP 消息返回给用户浏览器。浏览器在接收到返回消息后，对里面的 html 内容进行解析， 并展示到浏览器界面中。

* 1. TCP **与** HTTP **协议**

###### TCP/IP 协议族简介

OSI 将计算机网络体系分为七层：物理层、数据链路层、网络层、传输层、会话层、表示层和应用层1。

在 Internet 网络中使用的是五层网络模型：物理层、网络接口层、网络层、传输层和应用层。物理层对应网络的基本硬件；网络接口层中的协议定义了网络中传输的帧格式；网络

层中协议定义了信息包的格式以及这些信息包在网络中的转发机制；传输层中协议用于网 络中两个终端之间的信息传输；应用层中协议指定了具体应用中的信息格式。

TCP/IP 协议族是支撑 Internet 网络的主要协议组，其中应用层包括 DNS、FTP、HTTP、

IMAP、LDAP、RTP、SSH、Telnet、TLS/SSL 等协议；传输层包括 TCP、UDP、RSVP、SCTP 等协议；网络层包括 IP(IPv4,IPv6)、ICMP、ECN、IGMP 等协议；网络接口层包括 ARP、PPP、Ethernet、DSL、ISDN、FDDI 等协议。具体包含协议情况详见 Wikipedia 中的 TCP/IP 词条。

TCP/IP 协议族以传输层中的 TCP(Transmission Control Protocol) 和互联网层中的IP(Internet Protocol)来命名，足以说明这两个协议的重要性。其中，TCP 是一种面向连接的、可靠的、基于字节流的传输协议；IP 定义了寻址方法和数据包的封装结构。

###### HTTP 协议

HTTP(Hypertext Transfer Protocol)协议，是应用层协议，主要负责超文本的交互与传输。而超文本是结构化的文档，其中使用超链接来关联不同站点上的文件。例如，在网站 A 上网页 w1 上可以通过点击一个超链接，来打开另一个网页 w2，这个网页 w2 可以来自于网站 A 也可以来自于其它网站。而 HTTP 协议能够保证这些网页之间的无缝链接，把点击链接的相应网页或其它文件发送到用户的浏览器中。

HTTP 协议是请求响应式协议，即客户端向服务器发出请求消息，服务器根据请求消息方法和自身状态来做成响应，并把响应消息发送给客户端。HTTP 协议消息使用 ASCII 编码， 其 1.1 版本具体格式按增强巴斯特范式(Augmented BNF)定义为如下2。

HTTP-message Request

= Request | Response

= Request-Line

\*(( general-header

| request-header

| entity-header ) CRLF) CRLF

[ message-body ]

; Section 5.1

; Section 4.5

; Section 5.3

; Section 7.1

; Section 4.3

1 OSI Model---https://zh.wikipedia.org/wiki/OSI ཛྷࣳ

2 https[://www.ietf.org/rfc/rfc2616.txt](http://www.ietf.org/rfc/rfc2616.txt)

Response = Status-Line ; Section 6.1

\*(( general-header ; Section 4.5

| response-header ; Section 6.2

| entity-header ) CRLF) ; Section 7.1 CRLF

[ message-body ] ; Section 7.2 Request-Line = Method SP Request-URI SP HTTP-Version CRLF Method = "OPTIONS" ; Section 9.2

| "GET" ; Section 9.3

| "HEAD" ; Section 9.4

| "POST" ; Section 9.5

| "PUT" ; Section 9.6

| "DELETE" ; Section 9.7

| "TRACE" ; Section 9.8

| "CONNECT" ; Section 9.9

| extension-method extension-method = token

Request-URI = "\*" | absoluteURI | abs\_path | authority

general-header = Cache-Control ; Section 14.9

| Connection ; Section 14.10

| Date ; Section 14.18

| Pragma ; Section 14.32

| Trailer ; Section 14.40

| Transfer-Encoding ; Section 14.41

| Upgrade ; Section 14.42

| Via ; Section 14.45

| Warning ; Section 14.46

request-header = Accept ; Section 14.1

| Accept-Charset ; Section 14.2

| Accept-Encoding ; Section 14.3

| Accept-Language ; Section 14.4

| Authorization ; Section 14.8

| Expect ; Section 14.20

| From ; Section 14.22

| Host ; Section 14.23

| If-Match ; Section 14.24 entity-header = Allow ; Section 14.7

| Content-Encoding ; Section 14.11

| Content-Language ; Section 14.12

| Content-Length ; Section 14.13

| Content-Location ; Section 14.14

| Content-MD5 ; Section 14.15

| Content-Range ; Section 14.16

| Content-Type ; Section 14.17

| Expires ; Section 14.21

| Last-Modified ; Section 14.29

| extension-header

Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF

Status-Code = "100" ; Section 10.1.1: Continue

| "101" ; Section 10.1.2: Switching Protocols

| "200" ; Section 10.2.1: OK

| "201" ; Section 10.2.2: Created

| "202" ; Section 10.2.3: Accepted

| "203" ; Section 10.2.4: Non-Authoritative Information

| "204" ; Section 10.2.5: No Content

| "205" ; Section 10.2.6: Reset Content

| "206" ; Section 10.2.7: Partial Content

| "300" ; Section 10.3.1: Multiple Choices

| "301" ; Section 10.3.2: Moved Permanently

| "302" ; Section 10.3.3: Found

| "303" ; Section 10.3.4: See Other

| "304" ; Section 10.3.5: Not Modified

| "305" ; Section 10.3.6: Use Proxy

| "307" ; Section 10.3.8: Temporary Redirect

| "400" ; Section 10.4.1: Bad Request

| "401" ; Section 10.4.2: Unauthorized

| "402" ; Section 10.4.3: Payment Required

| "403" ; Section 10.4.4: Forbidden

| "404" ; Section 10.4.5: Not Found

| "405" ; Section 10.4.6: Method Not Allowed

| "406" ; Section 10.4.7: Not Acceptable

| "407" ; Section 10.4.8: Proxy Authentication Required

| "408" ; Section 10.4.9: Request Time-out

| "409" ; Section 10.4.10: Conflict

| "410" ; Section 10.4.11: Gone

| "411" ; Section 10.4.12: Length Required

| "412" ; Section 10.4.13: Precondition Failed

| "413" ; Section 10.4.14: Request Entity Too Large

| "414" ; Section 10.4.15: Request-URI Too Large

| "415" ; Section 10.4.16: Unsupported Media Type

| "416" ; Section 10.4.17: Requested range not satisfiable

| "417" ; Section 10.4.18: Expectation Failed

| "500" ; Section 10.5.1: Internal Server Error

| "501" ; Section 10.5.2: Not Implemented

| "502" ; Section 10.5.3: Bad Gateway

| "503" ; Section 10.5.4: Service Unavailable

| "504" ; Section 10.5.5: Gateway Time-out

| "505" ; Section 10.5.6: HTTP Version not supported

| extension-code

response-header = Accept-Ranges ; Section 14.5

| Age ; Section 14.6

| ETag ; Section 14.19

| Location ; Section 14.30

| Proxy-Authenticate ; Section 14.33

根 据 上 面 的 协 议 格 式 描 述 ， 一 个 Request-Line 可 以 表 示 “ GET <http://www.w3.org/pub/WWW/TheProject.html>HTTP/1.1” 。在 RFC2616 协议中，HTTP 为区分客户端发出的请求消息方法，在 HTTP/1.1 中将请求消息分为 GET、POST、HEAD、OPTIONS、PUT、DELETE、TRACE 和 CONNECT 八种方法。

* GET 方法表示要请求获取特定的资源，比如 html 网页、jpg 图像文件等。GET 类型请求消息仅表示获取数据，并不影响数据（增加、删除、修改等）。
* HEAD 方法与GET 获得的响应一致，但要求响应消息中只包含head，不包含body。这个方法在获取元信息时非常有效。例如要获取一个文件的大小、日期等信息，

并不需要服务器的响应信息中包含这个文件，而只是将这些元信息封装到响应消 息的 head 中即可。

* POST 方法请求服务器接收封装在请求消息中的数据实体，并将其作为新的附属资

源粘贴到指定 URI 的 Web 资源中。封装在 POST 请求消息中的数据可以是邮件列表、Web 页面中需要提交的数据、公告板中的一条消息等内容。

* PUT 方法请求服务器将请求消息中的数据实体存储指定 URI 中。如果 URI 指向已经存在的资源，则将此数据实体替代已经存在的资源；如果 URI 指向的资源不存在，则将此数据实体表示为此 URI 指向的资源。
* DELETE 方法请求删除指定的资源
* TRACE 方法请求中间服务器将自身心情及对请求消息的改变添加到请求消息中， 从而使得客户端能够追踪请求消息的路由过程及消息变化情况。
* OPTIONS 方法请求服务器返回其能够支持的 HTTP 请求方法。
* CONNECT 方法将请求连接转换到透明的 TCP/IP 通道，这样做的目的是为了便于加密的 HTTPS 通过非加密的 HTTP 代理。
* PATCH 方法请求对修改一个资源的部分内容。

有关以上方法的详细说明，请见 RFC7231 和 RFC5789. 本书实验将主要关注 GET 和

HEAD 类型的请求消息处理。

请求消息的格式由下面四部分组成：

* 请求行：其用来表明请求消息类型和请求资源的 URI。例如，GET /web/index.html

表示请求获取服务器管理的虚拟路径下 web 目录中的 index.html 网页。

* 请求头域列表：在列表内部，每个请求头域描述请求消息中的一个参数及其值， 其中参数表示此请求头域的名称。具体值格式为 parameter：value。例如，Accept: text/plain 是 Accept 头域，其值 text/plain 表示响应消息中的内容格式类型为

text/plain。

* 一个空行(/r/n)
* 消息体（可选）

在请求行和请求头域每行必须以符合“ <CR><LF> ”结尾。在空行中只有符号“<CR><LF>”，不能出现空格。在 HTTP/1.1 协议中，除了 Host 头域外，其它所有请求头域都是可选的。

与请求消息相对应的是服务器给客户端的响应消息。响应消息由下面四部分组成

* 响应状态行，其内部包含状态码和原因内容。例如 “HTTP/1.1 200 OK”表示客户端请求成功
* 响应头域，例如 “Content-Type:text/html”表示
* 一个空行（/r/n）
* 一个消息体

例如，一个请求消息实例如下所示。

GET /index.html HTTP/1.1 Host: [www.example.com](http://www.example.com/)

服务器给出的响应消息为：

HTTP/1.1 200 OK

Date: Mon, 08 May 2017 22:38:34 GMT

Content-Type: text/html; charset=UTF-8 Content-Encoding: UTF-8

Content-Length: 138

Last-Modified: Sun, 08 Jan 2017 12:21:50 GMT Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux) ETag: "3f80f-1b6-3e1cb03b"

Accept-Ranges: bytes Connection: close

<html>

<head>

<title>An Example Page</title>

</head>

<body>

Hello World, this is a very simple HTML document.

</body>

</Html>

* 1. Socket **编程**

Socket 是操作系统中实现 TCP/IP 等通信协议的 API 接口。通过调用 Socket 能够实现多台计算机之间的消息传递。Socket 分为客户端和服务端两种状态。Socket 服务端状态主要用于服务器的开发。在单进程单线程的 TCP 服务器模型中，socket 接口调用顺序和状态变化如图\*\*所示。首先初始化自身，并绑定一个侦听端口；然后设置为侦听状态，并阻塞当 前运行线程；一旦有客户端的连接请求，将与客户端建立一个新的连接通道，并在这个通道

中通过读、写接口与客户端进行通信；如果处理完与客户端的通信，就可以将这个通道关闭； 然后继续阻塞当前线程，直到有新的客户端进行连接请求。

在 linux 系统中，涉及到 TCP/IP 传输的接口主要有：

* socket()函数负责初始化一个用于通信的 socket 描述符。其操作语义类似于使用 C 语言中的函数 fopen，其打开一个文件并其返回一个文件描述符，通过此描述符， 能够对文件进行读写。因此通过此函数返回的 socket 描述符，能够进行通信信息的读取和写入。

其具体函数接口如下：

int socket(int protofamily,int type,int protocol)

返回值为此操作此 socket 的描述符。

参数 protofamily 表示所使用网络地址协议，使用 AF\_INET, AF\_INET6 、AF\_LOCAL 等数值来分别表示 IPv4、IPv6、文件路径等类型通信地址。例如当使用 AF\_INET 作为此函数参数时，则在通信时需要指定 32 位的 IPv4 地址和端口号，如 127.0.0.1:8080。

参数 type 指定 socket 类型，其具体数值有 SOCK\_STREAM、SOCK\_DGRAM、SOCK\_RAW、SOCK\_PACKET、SOCK\_SEQPACKET 等。

参数 protocol 表示 socket 使用传输协议，其数值有 IPPROTO\_TCP 、IPPROTO\_UDP、IPPROTO\_SCTP、IPPROTO\_TIPC 等，分别应用 TCP 传输协议、UDP 传输协议、STCP 传输协议、TIPC 传输协议。

例如： int clientsock\_fd=socket(AF\_INET,SOCK\_STREAM,0)

* bind 函数负责将 socket 描述符与指定地址绑定。bind 函数是服务端调用的函数， 用来绑定具体的侦听端口号。因为客户端会自动创建连接和端口号，因此 bind 在客户端并不需要。其具体函数格式如下：

int bind(int sockfd, const struct sockaddr \* addr, socklen\_t addrlen)

其中参数 sockfd 为 socket 描述符（由 socket 函数产生）；addr 为地址指针，指向要为 sockfd 绑定的地址。地址数据结构要与创建 socket 描述符时参数protofamily 一致。例如，如果 protofamily 参数值为 AF\_INET，则 addr 指向一个 IPv4 的地址结构 sockadd\_in；如果 protofamily 参数值为 F\_INET6，则 addr 指向一个 IPv6 地址结构 sockaddr\_in6;如果 protofamily 参数值为 AF\_LOCAL， 则 addr 指向一个路径结构 sockadr\_un。 参数 addrlen 为地址长度。

* listen()函数主要用于服务端，使得服务器能够侦听来自于指定 socket 描述符下的消息。在调用完此函数后，指定的 socket 将变为侦听状态，用于等待用户的连接请求。其具体函数格式如下：

int listen(int sockfd, int backlog)

其参数 sockfd 表示 socket 描述符；backlog 表示此 socket 可以接受排队的连接最大个数。

* connect()函数表示为指定的 socket 文件描述符与服务器端的地址建立连接。此函数用于客户端，使得客户端能够向服务器发起连接。其具体函数格式如下：

int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen)

其参数 sockfd 表示一个已经通过 socket 函数创建的 socket 描述符；addr 为服务端的地址；addrlen 为地址长度。

* accept()函数表示使得处于侦听状态下的 socket 能够接受连接请求，同时此函数会阻塞当前线程，直到有客户端与此 socket 建立连接。其具体函数格式如下：

int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen)

其参数 sockfd 表⽰处于侦听状态下的 socket 描述符；addr ⽤于返回客户端的地址；addrlen 为客户端地址的长度。

返回值为服务端与客户端新建⽴的通信通道，即新建⽴⼀个 socket 描述符，⽤于与客户端通信。为什么会新建⽴⼀个 socket 描述符呢？这是因为服务器处在侦听状态下的 socket 只负责接收客户端的连接请求，⼀旦受到请求信号，accept函数将新建⽴⼀个 socket 与客户端 socket 进⾏通信。这样能够使得服务器能够与多个客户端同时保持通信通道（⼀个客户端，服务器就有⼀个 socket 与其对应）。

* read/write 函数，把 socket 描述符当作文件描述符，读写调用与文件操作函数一样，负责在 socket 中读取或写入信息，来实现消息的发生和接收。除此之外，socket接口函数中还包括 Recv/send 函数、sendto/recvfrom 函数和 sendmsg/recvmsg 函数。
* close 函数，负责关闭指定的 socket，并释放资源。

具体的 socket 中函数与如何支持各个协议，以及各个协议的实现细节，请查询相关书籍材料。在本实验中，将主要关注 TCP/IP 协议基础上的应⽤层协议 HTTP 的实现。下⾯例

⼦是 nweb()项⽬中的代码。其中客户端代码通过 socket 向指定服务器发⽣了⼀个 HTTP 协

议消息，其⽬的是请求⼀个⽹页 helloworld.html；然后服务器在 socket 端⼜中，读取 HTTP 协议消息，然后读取客户端指定的⽹页内容，并将此内容写⼊与客户端建⽴的 socket 中；客户端在接收到此⽹页信息后，将消息打印到控制台，并关闭此 socket。

在 TCP 客户端，socket 接口调用顺序和状态变化，如下面代码所示。其首先初始化自身，向服务器发送请求并建立连接通道；然后通过读、写接口与服务器进行通信；当通信完 毕后，关闭这个连接通道。

/\* Client Code\*/

/\* The following main code from https://github.com/ankushagarwal/nweb, but they are modified slightly

\*/

#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <string.h> #include <sys/types.h> #include <sys/socket.h> #include <netinet/in.h> #include <arpa/inet.h>

/\* IP address and port number \*/

#define PORT 8181 // Port number as an integer - web server default is 80 #define IP\_ADDRESS "192.168.0.8" //IP Address as a string

/\* Request a html file base on HTTP \*/

char \*httprequestMsg = "GET /helloworld.html HTTP/1.0 \r\n\r\n" ;

#define BUFSIZE 8196 void pexit(char \* msg)

{

perror(msg); exit(1);

}

void main()

{

int i,sockfd;

char buffer[BUFSIZE];

static struct sockaddr\_in serv\_addr;

printf("client trying to connect to %s and port %d\n",IP\_ADDRESS,PORT); if((sockfd = socket(AF\_INET, SOCK\_STREAM,0)) <0) //create a client socket

pexit("socket() error");

serv\_addr.sin\_family = AF\_INET; //Set the socket with IPv4 serv\_addr.sin\_addr.s\_addr = inet\_addr(IP\_ADDRESS);// set ip address serv\_addr.sin\_port = htons(PORT); // set ip port number

/\* Connect the socket offered by the web server \*/

if(connect(sockfd, (struct sockaddr \*)&serv\_addr, sizeof(serv\_addr)) <0) pexit("connect() error");

/\* Now the sockfd can be used to communicate to the server the GET request \*/ printf("Send bytes=%d %s\n",strlen(httprequestMsg), httprequestMsg); write(sockfd, httprequestMsg, strlen(httprequestMsg));

/\* This displays the raw HTML file (if index.html) as received by the browser \*/ while( (i=read(sockfd,buffer,BUFSIZE)) > 0)

write(1,buffer,i);

/\*close the socket\*/

close(sockfd);

}

TCP 服务端代码如下面代码所示。其中，数据结构 extensions 主要用来存放 nweb 服务器能够支持的文件类型；logger 函数主要用来给客户端返回一下服务器内部状态的响应消息

（响应代码为 403 的 Forbidden 消息和响应代码为 404 的 NOT FOUND 消息），并将相关内

容写入日志文件中；web 函数首先从 socket 中读取并解析 HTTP 消息，然后读取指定的文件内容，并合成 HTTP 的响应消息，最后将响应消息写入指定 socket。

在 TCP 服务端主函数流程中，首先对参数 argc 和 argv 进行判断和内容识别，其主要作用是从 argv 参数列表中获得端口号和网页存取路径。例如执行命令 nweb 8181

/home/newdir，在参数列表 argv[1]中保存‘8181’字符串；argv[2]中保存‘/home/newdir’ 字符串。然后创建侦听 socket，并通过 bind 函数将此 socket 绑定到指定端口(通过参数结构sockaddr\_in 实现，使用 listen 函数设置此 socket 为侦听状态，并最终阻塞在 accept 函数位置。直到有客户端与服务端建立连接，这个函数将返回与客户端建立连接的 socket 描述符。根据此 socket 描述符，使用 web 函数对用户的请求做出响应，并记录信息到日志文件。

/\*Server Code\*/

/\* webserver.c\*/

/\*The following main code from https://github.com/ankushagarwal/nweb\*, but they are modified slightly\*/

#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <errno.h> #include <string.h> #include <fcntl.h> #include <signal.h> #include <sys/types.h> #include <sys/socket.h> #include <netinet/in.h> #include <arpa/inet.h> #define VERSION 23

#define BUFSIZE 8096

#define ERROR 42

#define LOG 44

#define FORBIDDEN 403

#define NOTFOUND 404

#ifndef SIGCLD

# define SIGCLD SIGCHLD #endif

struct { char \*ext;

char \*filetype;

} extensions [] = {

{"gif", "image/gif" },

{"jpg", "image/jpg" },

{"jpeg","image/jpeg"},

{"png", "image/png" },

{"ico", "image/ico" },

{"zip", "image/zip" },

{"gz", "image/gz" },

{"tar", "image/tar" },

{"htm", "text/html" },

{"html","text/html" },

{0,0} };

void logger(int type, char \*s1, char \*s2, int socket\_fd)

{

int fd ;

char logbuffer[BUFSIZE\*2];

switch (type) {

case ERROR: (void)sprintf(logbuffer,"ERROR: %s:%s Errno=%d exiting pid=%d",s1, s2, errno,getpid()); break;

case FORBIDDEN:

(void)write(socket\_fd, "HTTP/1.1 403 Forbidden\nContent-Length: 185\nConnection: close\nContent-Type: text/html\n\n<html><head>\n<title>403 Forbidden</title>\n</head><body>\n<h1>Forbidden</h1>\nThe requested URL, file type or operation is not allowed on this simple static file webserver.\n</body></html>\n",271);

(void)sprintf(logbuffer,"FORBIDDEN: %s:%s",s1, s2); break;

case NOTFOUND:

(void)write(socket\_fd, "HTTP/1.1 404 Not Found\nContent-Length: 136\nConnection: close\nContent-Type: text/html\n\n<html><head>\n<title>404 Not Found</title>\n</head><body>\n<h1>Not Found</h1>\nThe requested URL was not found on this server.\n</body></html>\n",224);

(void)sprintf(logbuffer,"NOT FOUND: %s:%s",s1, s2); break;

case LOG: (void)sprintf(logbuffer," INFO: %s:%s:%d",s1, s2,socket\_fd); break;

}

/\* No checks here, nothing can be done with a failure anyway \*/

if((fd = open("webserver.log", O\_CREAT| O\_WRONLY | O\_APPEND,0644)) >= 0) { (void)write(fd,logbuffer,strlen(logbuffer));

(void)write(fd,"\n",1); (void)close(fd);

}

}

/\* this is a child web server process, so we can exit on errors \*/ void web(int fd, int hit)

{

int j, file\_fd, buflen; long i, ret, len; char \* fstr;

static char buffer[BUFSIZE+1]; /\* static so zero filled \*/

ret =read(fd,buffer,BUFSIZE); /\* read Web request in one go \*/ if(ret == 0 || ret == -1) { /\* read failure stop now \*/

logger(FORBIDDEN,"failed to read browser request","",fd);

}

if(ret > 0 && ret < BUFSIZE) /\* return code is valid chars \*/ buffer[ret]=0; /\* terminate the buffer \*/

else buffer[0]=0;

for(i=0;i<ret;i++) /\* remove CF and LF characters \*/ if(buffer[i] == '\r' || buffer[i] == '\n')

buffer[i]='\*'; logger(LOG,"request",buffer,hit);

if( strncmp(buffer,"GET ",4) && strncmp(buffer,"get ",4) ) { logger(FORBIDDEN,"Only simple GET operation supported",buffer,fd);

}

for(i=4;i<BUFSIZE;i++) { /\* null terminate after the second space to ignore extra stuff \*/ if(buffer[i] == ' ') { /\* string is "GET URL " +lots of other stuff \*/

buffer[i] = 0; break;

}

}

for(j=0;j<i-1;j++) /\* check for illegal parent directory use .. \*/ if(buffer[j] == '.' && buffer[j+1] == '.') {

logger(FORBIDDEN,"Parent directory (..) path names not supported",buffer,fd);

file \*/

}

if( !strncmp(&buffer[0],"GET /\0",6) || !strncmp(&buffer[0],"get /\0",6) ) /\* convert no filename to index

(void)strcpy(buffer,"GET /index.html");

/\* work out the file type and check we support it \*/ buflen=strlen(buffer);

fstr = (char \*)0; for(i=0;extensions[i].ext != 0;i++) {

len = strlen(extensions[i].ext);

if( !strncmp(&buffer[buflen-len], extensions[i].ext, len)) { fstr =extensions[i].filetype;

break;

}

}

if(fstr == 0) logger(FORBIDDEN,"file extension type not supported",buffer,fd);

if(( file\_fd = open(&buffer[5],O\_RDONLY)) == -1) { /\* open the file for reading \*/ logger(NOTFOUND, "failed to open file",&buffer[5],fd);

}

logger(LOG,"SEND",&buffer[5],hit);

len = (long)lseek(file\_fd, (off\_t)0, SEEK\_END); /\* lseek to the file end to find the length \*/ (void)lseek(file\_fd, (off\_t)0, SEEK\_SET); /\* lseek back to the file start ready for reading \*/

(void)sprintf(buffer,"HTTP/1.1 200 OK\nServer: nweb/%d.0\nContent-Length: %ld\nConnection:

close\nContent-Type: %s\n\n", VERSION, len, fstr); /\* Header + a blank line \*/ logger(LOG,"Header",buffer,hit);

(void)write(fd,buffer,strlen(buffer));

/\* send file in 8KB block - last block may be smaller \*/ while ( (ret = read(file\_fd, buffer, BUFSIZE)) > 0 ) {

(void)write(fd,buffer,ret);

}

sleep(1); /\* allow socket to drain before signalling the socket is closed \*/ close(fd);

}

int main(int argc, char \*\*argv)

{

int i, port, listenfd, socketfd, hit; socklen\_t length;

static struct sockaddr\_in cli\_addr; /\* static = initialised to zeros \*/ static struct sockaddr\_in serv\_addr; /\* static = initialised to zeros \*/

if( argc < 3 || argc > 3 || !strcmp(argv[1], "-?") ) {

(void)printf("hint: nweb Port-Number Top-Directory\t\tversion %d\n\n" "\tnweb is a small and very safe mini web server\n"

"\tnweb only servers out file/web pages with extensions named below\n" "\t and only from the named directory or its sub-directories.\n"

"\tThere is no fancy features = safe and secure.\n\n" "\tExample:webserver 8181 /home/nwebdir &\n\n" "\tOnly Supports:", VERSION);

for(i=0;extensions[i].ext != 0;i++) (void)printf(" %s",extensions[i].ext);

(void)printf("\n\tNot Supported: URLs including \"..\", Java, Javascript, CGI\n" "\tNot Supported: directories / /etc /bin /lib /tmp /usr /dev /sbin \n"

"\tNo warranty given or implied\n\tNigel Griffiths nag@uk.ibm.com\n" ); exit(0);

}

if( !strncmp(argv[2],"/" ,2 ) || !strncmp(argv[2],"/etc", 5 ) ||

!strncmp(argv[2],"/bin",5 ) || !strncmp(argv[2],"/lib", 5 ) ||

!strncmp(argv[2],"/tmp",5 ) || !strncmp(argv[2],"/usr", 5 ) ||

!strncmp(argv[2],"/dev",5 ) || !strncmp(argv[2],"/sbin",6) ){ (void)printf("ERROR: Bad top directory %s, see nweb -?\n",argv[2]); exit(3);

}

if(chdir(argv[2]) == -1){

(void)printf("ERROR: Can't Change to directory %s\n",argv[2]); exit(4);

}

/\* Become deamon + unstopable and no zombies children (= no wait()) \*/ if(fork() != 0)

return 0; /\* parent returns OK to shell \*/ (void)signal(SIGCLD, SIG\_IGN); /\* ignore child death \*/ (void)signal(SIGHUP, SIG\_IGN); /\* ignore terminal hangups \*/ for(i=0;i<32;i++)

(void)close(i); /\* close open files \*/

(void)setpgrp(); /\* break away from process group \*/ logger(LOG,"nweb starting",argv[1],getpid());

/\* setup the network socket \*/

if((listenfd = socket(AF\_INET, SOCK\_STREAM,0)) <0) logger(ERROR, "system call","socket",0);

port = atoi(argv[1]); if(port < 0 || port >60000)

logger(ERROR,"Invalid port number (try 1->60000)",argv[1],0); serv\_addr.sin\_family = AF\_INET;

serv\_addr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

serv\_addr.sin\_port = htons(port);

if(bind(listenfd, (struct sockaddr \*)&serv\_addr,sizeof(serv\_addr)) <0) logger(ERROR,"system call","bind",0);

if( listen(listenfd,64) <0) logger(ERROR,"system call","listen",0);

for(hit=1; ;hit++) {

length = sizeof(cli\_addr);

if((socketfd = accept(listenfd, (struct sockaddr \*)&cli\_addr, &length)) < 0) logger(ERROR,"system call","accept",0);

web(socketfd,hit); /\* never returns \*/

}

}

## 开发环境与测试环境

本实验代码的编写、编译和调试分别使用 vim、gcc 和 gdb 来完成。对于 VIM 程序相关命令的使用方法请查阅相关文献。编写源代码除了使用 VIM 外，还可以使用 Emacs、sublime text 和其它文本编辑器工具。

* + 1. GCC

GCC（GNU Complier Collection)是 GNU 项目下的一个编译系统，用以支持各种程序的编译。本文将主要关注与 C 语言相关的常用编译参数选项。GCC 程序在编译程序时包含预处理（Pre-Processing）、编译(Compling)、汇编（Assembling）和链接（Linking）四个阶段。每个阶段对应不同内容信息的输出。

###### 预处理阶段

该阶段执行 C 语言代码中的预处理及宏指令。根据#include 指令，在文件的相应位置插入引入的文件；根据#define 指令，将代码中相应宏替换为定义的字符串。该阶段可以使用 gcc 命令中的“-E”参数选用来完成。 例如，

gcc -E client.c -o client.i

将对 clinet.c 文件进行预处理，并将预处理结果保存为 client.i 文件。打开 client.i 文件， 如下面代码所示，将会发现在源文件中#include 指令的位置插入了相关头文件的内容，并且main 函数中的宏被替换为具体具体的定义数值。

……

……

#577 "/usr/include/sys/socket.h" 3 4 struct msghdr {

void \*msg\_name; socklen\_t msg\_namelen; struct iovec \*msg\_iov; int msg\_iovlen;

void \*msg\_control; socklen\_t msg\_controllen; int msg\_flags;

};

# 577 "/usr/include/sys/socket.h" 3 4

struct cmsghdr { socklen\_t cmsg\_len; int cmsg\_level;

int cmsg\_type;

};

# 668 "/usr/include/sys/socket.h" 3 4 struct sf\_hdtr {

struct iovec \*headers; int hdr\_cnt;

struct iovec \*trailers; int trl\_cnt;

};

int accept(int, struct sockaddr \* restrict, socklen\_t \* restrict) asm("\_" "accept" );

int bind(int, const struct sockaddr \*, socklen\_t) asm("\_" "bind" );

int connect(int, const struct sockaddr \*, socklen\_t) asm("\_" "connect" );

int getpeername(int, struct sockaddr \* restrict, socklen\_t \* restrict) asm("\_" "getpeername" );

int getsockname(int, struct sockaddr \* restrict, socklen\_t \* restrict) asm("\_" "getsockname" );

int getsockopt(int, int, int, void \* restrict, socklen\_t \* restrict);

int listen(int, int) asm("\_" "listen" );

ssize\_t recv(int, void \*, size\_t, int) asm("\_" "recv" );

ssize\_t recvfrom(int, void \*, size\_t, int, struct sockaddr \* restrict,

socklen\_t \* restrict) asm("\_" "recvfrom" );

ssize\_t recvmsg(int, struct msghdr \*, int) ssize\_t send(int, const void \*, size\_t, int)

asm("\_" "recvmsg" );

asm("\_" "send" );

ssize\_t sendmsg(int, const struct msghdr \*, int) ssize\_t sendto(int, const void \*, size\_t,

asm("\_" "sendmsg" );

int, const struct sockaddr \*, socklen\_t) asm("\_" "sendto" );

int setsockopt(int, int, int, const void \*, socklen\_t); int shutdown(int, int);

int sockatmark(int)

attribute

((availability(macosx,introduced=10.5)));

int socket(int, int, int);

int socketpair(int, int, int, int \*)

asm("\_" "socketpair" );

int sendfile(int, int, off\_t, off\_t \*, struct sf\_hdtr \*, int);

……

…… main()

{

int i,sockfd;

char buffer[8196];

static struct sockaddr\_in serv\_addr;

printf("client trying to connect to %s and port %d\n","192.168.0.8",8181); if((sockfd = socket(2, 1,0)) <0)

pexit("socket() error");

serv\_addr.sin\_family = 2;

serv\_addr.sin\_addr.s\_addr = inet\_addr("192.168.0.8");

serv\_addr.sin\_port = ((

uint16\_t)(

builtin\_constant\_p(8181) ? ((

uint16\_t)((((

uint16\_t)(8181) &

0xff00) >> 8) | ((( uint16\_t)(8181) & 0x00ff) << 8))) : \_OSSwapInt16(8181)));

if(connect(sockfd, (struct sockaddr \*)&serv\_addr, sizeof(serv\_addr)) <0) pexit("connect() error");

printf("Send bytes=%d %s\n",strlen(httprequestMsg), httprequestMsg); write(sockfd, httprequestMsg, strlen(httprequestMsg));

while( (i=read(sockfd,buffer,8196)) > 0) write(1,buffer,i);

close(sockfd)

}

###### 编译阶段

在此阶段，gcc 将检查代码的语法规范，并将 C 语言代码编译成汇编代码。该阶段可以使用 gcc 命令中的“-S”参数选用来完成。例如，

gcc -S client.i -o client.s

将生成 clinet.c 的汇编代码文件 client.s。当然也可以直接使用“gcc -S client.c -o clinet.s”来完成汇编代码的生成，这时将包括预处理和汇编两个阶段。具体 client.s 汇编代码片段如下所示。

.section

TEXT, text,regular,pure\_instructions

.macosx\_version\_min 10, 12

.globl \_pexit

.p2align 4, 0x90

\_pexit: ## @pexit

.cfi\_startproc ## BB#0:

pushq %rbp Ltmp0:

.cfi\_def\_cfa\_offset 16 Ltmp1:

.cfi\_offset %rbp, -16 movq %rsp, %rbp Ltmp2:

.cfi\_def\_cfa\_register %rbp subq $16, %rsp

movq %rdi, -8(%rbp)

movq -8(%rbp), %rdi callq \_perror

movl $1, %edi callq \_exit

.cfi\_endproc

.globl \_main

.p2align 4, 0x90

\_main:

## @main

.cfi\_startproc ## BB#0:

pushq %rbp Ltmp3:

.cfi\_def\_cfa\_offset 16 Ltmp4:

.cfi\_offset %rbp, -16 movq %rsp, %rbp Ltmp5:

….

###### 汇编阶段

在此阶段，gcc 将汇编代码转换为二进制目标代码。该阶段使用 gcc 命令中的“-c”参数选用来完成。例如，

gcc -c client.s -o client.o

同样也可以使用“gcc -c client.c -o client.o” 连续执行预处理、编译和汇编三个阶段的处理。在使用“gcc -g -o client.o client.c”后，可以使用“objdump -S client.o”命令来查看 C 源代码及其对应汇编代码的混合输出，其显示效果如下面代码片段所示。

…

; printf("client trying to connect to %s and port %d\n",IP\_ADDRESS,PORT); 100000ccc: b0 00 movb $0, %al

100000cce: e8 b9 01 00 00 callq 441

100000cd3:bf 02 00 00 00 movl $2, %edi

100000cd8:be 01 00 00 00 movl $1, %esi 100000cdd:31 d2 xorl %edx, %edx

; if((sockfd = socket(AF\_INET, SOCK\_STREAM,0)) <0) //create a client socket 100000cdf: 89 85 e4 df ff ff movl %eax, -8220(%rbp)

100000ce5: e8 ae 01 00 00 callq 430

100000cea: 89 85 e8 df ff ff movl %eax, -8216(%rbp) 100000cf0: 83 f8 00 cmpl $0, %eax

100000cf3: 0f 8d 0c 00 00 00 jge 12 <\_main+0x65>

100000cf9: 48 8d 3d 85 02 00 00 leaq 645(%rip), %rdi

; pexit("socket() error");

100000d00:

100000d05:

e8 7b ff ff ff

callq -133 <\_pexit>

48 8d 3d 6d 02 00 00

leaq 621(%rip), %rdi

; serv\_addr.sin\_family = AF\_INET; //Set the socket with IPv4 100000d0c:c6 05 66 03 00 00 02 movb $2, 870(%rip)

; serv\_addr.sin\_addr.s\_addr = inet\_addr(IP\_ADDRESS);// set ip address 100000d13: e8 68 01 00 00 callq 360

100000d18: 48 8d 3d 59 03 00 00 leaq 857(%rip), %rdi

100000d1f: ba 10 00 00 00 movl $16, %edx

100000d24:

…

89 05 52 03 00 00

movl %eax, 850(%rip)

###### 链接阶段

在此阶段，gcc 使用链接器 ld，将多个二进制目标文件和库文件链接在一起，以生成可执行格式的文件。例如，

gcc client.o -o client。

同样也可以使用“gcc client.c -o client”将上述四个阶段一起执行，并生成可执行程序。

除了涉及以上编译阶段的参数指令外，还有一些参与选项比较常用。

-include file

引入某个头函数文件，例如命令 “gcc client.c -include /usr/include/example.h”，在编译 client 文件时需要使用 example.h 的头文件。

-Idir

gcc 在遇到源代码中“#include file.h”时，将在当前文件目录查找 file.h 头文件，如果没有找到，将到缺省目录中进行查找。在此命令指定目录后，gcc 将首先在指定目录进行头文件查找，如没有找到在按上述查找顺序进行查找。

-llibrary

指定 gcc 在链接阶段所使用的库文件。

-Ldir

指定 gcc 链接阶段库文件所在路径。

例如，gcc -o webserver webserver.o -L. -ldisplay，将 webserver.o 文件与库libdisplay.so

链接在一起，并生成可执行程序 webserver.

-g

在编译过程中产生调试信息，这些信息可供 gdb 等调试器使用。

-static

用于 gcc 生成静态库文件

-shared

用于 gcc 生成动态库文件

-fPIC

表示生成与位置无关的代码。

例如，“gcc -shared -fPIC display.c -o libdisplay.so” 将生成libdisplay.so 动态链接库。

-std

表示 gcc 支持的 C 语言标准，其取值有 C89，C99，gnu99 等，以表示其支持的 C 语言版本标准。例如，“gcc -std=C99 client.c -o client”，将 client.c 文件视为使用 C 语言 1999 年版本标准进行编写。

-pedantic

当 gcc 在编译时，将不符合相关语言标准的源代码进行标注，并产生相应的警告。

-Wall

让 gcc 产生尽可能多的警告信息。

-Werror

让 gcc 将警告信息看为程序的语法错误。使用此编译选项，将使得 gcc 停止在出现警告的位置。

-O0 -O1 -O2 -O3 表示编译器生成优化代码的程度。 其中-O0 表示没有优化；-O1 为缺省值；-O3 产生的代码运行速度最快。

* + 1. 构建 makefile

Makefile 是一个包含命令集的文件，此文件中的命令能够被 make 程序所解析并执行， 以完成大型程序的编译、部署等工作。可以想象一下在一个大型工程项目里面有成千上万个

代码文件，而这些代码文件被放置在不同的目录里面。如果要将这些文件按照工程项目要求生成不同类型的可执行程序，那么该怎么按照这些要求来编译、链接和生成这些程序代码呢？ 上个世纪 70 年代贝尔实验室的 Stuart Feldman 在 unix 系统上创建了 make 工具来完成上述

任务要求。

编写能够被 make 程序解析执行的 makefile 文件，需要掌握其编写规则。其具体编写规则如下。

target: prerequisites

command1 command2

… commandn

其中，target 表示命令执行的目标。其可以是生成的目标文件，也可以是一个标签； prerequisites 表示完成 target 目标所需要的前提条件，前提条件可以是文件或标签； command1，comomand2，… ，commandn 表示要完成目标所需要执行的 shell 命令。

此规则可以被解析为要实现目标 target，需要先执行前提条件，如果前提条件已经被执行，则完成 command 中指定的命令。target 和 prerequisites 使得多个规则之间形成了偏序关系。make 程序总能够知道先执行哪个 target 和后执行哪个 target。例如下面的 makefile 文件，完成对拥有 3 个头文件和 2 个 C 文件的项目编译和程序生成。该 makefile 文件主要生成了 webserver.o libdisplay.so 和 webserver 三个文件。其中 webserver.o 是编译后的目标二进制文件；libdisplay.so 是静态库文件；webserver 是可执行程序。

webserver: webserver.o libdisplay.so

gcc -g -o webserver webserver.o -L. -ldisplay

webserver.o: webserver.c webserver.h display.h counter.h gcc -g -c webserver webserver.c

libdisplay.so: display.c display.h

gcc -g -shared -fPIC display.c -o libdisplay.so

clean:

rm webserver webserver.o libdisplay.so

* + 1. 调试代码 GDB

GDB 是 GNU 项目下的调试器，其能够调试 Ada、C、C++、Objective-C、Pascal 等许多程序语言编写的程序。GDB 是 linux 平台下被广为使用的调试器，具有跟踪程序运行、断

点调试、动态修改程序数据等特点。GDB 要进行指定程序调试的前提是该程序在使用 GCC

编译时使用参数 g。

本节将介绍其常用的命令，更为详细的命令参数请查阅其使用手册。需要注意的是，在

GDB 命令使用中为调试方便，很多命令有缩写方式。

* **GDB** 启动

GDB 调试指定程序包含以下三种启动方式：直接命令启动、恢复程序执行现场、调试指定运行程序。

“gdb program” 表示使用 gdb 启动一个指定程序，其中 program 表示此程序名。“gdb program core”表示要恢复指定程序运行的现场，其中 core 表示程序非法执行

后 core dump 产生的文件。此命令常用于分析程序运行崩溃的原因。如果让操作系统能够产生 core dump 需要使用“ulimit -c unlimited”命令来解除操作系统对生成 core 文件的限制。

“gdb program PID”表示跟踪调试目前正在运行的程序，其中 PID 表示此程序运行的进程标识符。该命令可以让 gdb 关联到正在运行的程序，并调试它。

###### list 命令

当 GDB 启动后，在调试环境中可以使用 list 命令来查看程序文件的源代码，其缩写命令为“l”。list 命令后可以跟指定的代码行和指定的函数名。例如，“list 80” 或 “l 80” 表示列出代码行 80 位置处的源代码；而“list main”或“l main”表示列出 main 函数附近的源代码。

###### break 命令

break 命令用来设置程序运行断点，其缩写命令为“b”。break 命令可以对指定函数、指定文件代码行、指定内存地址来设置断点。

“b function” 表示在指定函数入口设置断点。“b linenum” 表示在指定代码行设置断点。

“b +offset” 或 “b - offset”表示在当前代码行后面或前面 offset 行来设置断点。“b filename:function”表示在指定文件中的函数设置断点。

“b filename:linenum” 表示在指定文件中的代码行设置断点。“b \*address” 表示在程序运行的指定内存地址设置断点。

Break 命令还支持设置条件断点，其命令格式为“b … if condition” ，其中“…”表示为上述 break 命令参数，condition 为断点条件。例如，“b client.c:web if hit==1” 表示在参数变量 hit 等于 1 时位置为 client.c 文件中 web 函数的断点有效。

断点设置成功后会为此断点返回一个断点号，断点号与断点一一对应，可以作为其它断 点相关命令的参数来使用。

###### 断点操作命令

在通过 break 命令设置断点后，可以使用 info、clear、delete、disable 和 enable 等命令来操作断点。

“Info break” 命令可以查看目前设置的所有断点信息。

“clear”命令可以清除 break 命令设置的断点。例如“clear”清除所有 break 设置的断点；“clear linenum”清除指定行的断点；“clear filename:linenum”清除指定文件中代码行上的断点；“clear filename:function”清除指定文件中函数上的断点。

“ delete breakpoints” 或 “ delete range” 可以清除指定断点号的断点，其中breakpoints 表示指定的断点号，如果没有指定断点号，则清楚所有的断点；range 表示断点号范围，例如 “clear 2-6”命令表示清除断点号 2-6 的断点。

“disable breakpoints”或“disable range”将使得指定断点号的断点失效，例如“disable 2-6”将使得断点号 2-6 的断点失效。与 clear 和 delete 命令相比，disable 命令并不删除断点，其可以通过 enable 命令来恢复失效的断点。

“enable breakpoints”或“enable range” 可以恢复失效的断点。如果想让断点在执行一次后马上失效，则可以使用“enable breakpoints once”命令。

###### watch 命令

如果想让某个变量值发生变化后中断当前程序运行，可以使用 watch 相关命令。

“watch expr” 命令为变量 expr 设置一个观察点。一旦这个变量值发生变化，将中断当前程序运行。

“rwatch expr” 当 expr 变量被读取时中断当前程序。“awatch expr” 当 expr 变量被读或被写时中断当前程序。

而对于观察点操作命令与断点操作命令相同，如下面的代码图所示。

(gdb) info breakpoints

Num Type

1 breakpoint

Disp Enb Address

What

keep y 0x080483c6 in main at test.c:5

breakpoint already hit 1 time

4 hw watchpoint keep y x breakpoint already hit 1 time

(gdb) disable 4

###### 运行程序命令

如果想让被调试的程序从头开始执行可以使用 run 命令，其缩写为“r”。

如果程序执行时需要运行参数，可以使用“set args”设置程序启动运行的参数。例如，

set args 8181 “/home/nwebdir”。

在程序被中断后，如果想让程序继续运行，可以使用 continue 命令，其缩写“c”。此命令将使得程序运行到下一个断点或直到观察点变量发生变化。

###### 单步运行命令

如果想让程序单步执行，则可以使用 step 命令，其缩写为“s”。当程序单步运行到函数时，将进入函数的内部。

“next”同样使程序单步执行，其缩写为“n”。但是当程序运行到函数时，使用此命令并不会让调试器进入函数，而是执行该函数，并跳到下一行。

“finish”命令将继续运行程序，直到当前函数运行完毕并返回。 “si”和“ni”命令与“s”和“n”类似，只不过它们作用于汇编指令上。

###### backtrace 命令

用于打印当前函数调用栈的所有信息，其缩写为“bt”。

###### 帧命令

“frame”命令用于查看当前栈层的信息，其缩写为“f”。

“frame n”将程序运行栈切换到 n。其中 n 是栈中层次编号，0 表示栈顶。

###### 查看运行数据

“print expr”命令用于显示指定变量 expr 的数值，其缩写为“p”。

“print file::variable”或“print function::variable”可以用来显示指定文件或函数中的变量值。

“print address@len” 用来显示数组内指定长度内的数值，其 address 表示数组的首地址，len 表示要显示数据项的格式。例如，“print a@4”用来显示数组 a 中 4 个数据项下的数值。

“print $regiester”用来显示寄存器中数值，例如“print $pc”显示当前 pc 寄存器中数

值。

在输出变量值时，可以指定输出变量的格式参数。其中，

x 表示按十六进制显示变量；

d 表示按十进制显示变量；

u 表示按十六进制格式显示无符号整型；

o 表示按八进制显示变量；

t 表示按二进制格式显示变量；

a 表示打印一个内存地址；

c 按字符格式显示变量;

f 按浮点数格式显示变量。

例如，“p/x k” 将把 k 变量数值按 16 进制输出。

###### display 命令

“display expr”用来自动显示变量的数值。当每次程序中断或单步跟踪调试时，会自动显示变量 expr 中的数值。

###### 查看内存命令

“x address”用来查看指定内存中的数据。

###### 退出命令

“quit”命令使得 gdb 退出。

###### GDB 进程多线程调试

GDB7.0 以上版本支持多进程调试，但需要通过设置 fork 模式参数来启动对多个进程的调试。fork 模式的参数通过两个命令来体现：set follow-fork-mode [parent|child] 和 set detach-on-fork [on|off]. 其中 follow-fork-mode 表示跟踪fork 进程状态，如果设置为parent， GDB 则跟踪调试父进程，如果设置为 child，GDB 则跟踪调试子进程；detach-on-fork 表示在 fork 后是否与不跟踪的进程脱离，如果设置为 on，则脱离不跟踪的进程，如果设置为 off， 则不脱离不跟踪的进程。这两个参数的不同设置组合具有以下的意义。

表 1-1 GDB 多进程跟踪参数设置表

|  |  |  |
| --- | --- | --- |
| follow-fork- detach-on- 含义  mode fork | | |
| parent | on | 只调试主进程（GDB 默认） |
| Child | on | 只调试子进程 |
| Parent | off | 同时调试两个进程，gdb 跟主进程，子进程阻塞在 fork  位置 |
| Child | off | 同时调试两个进程，gdb 跟子进程，主进程阻塞在 fork  位置 |

“info inferiors”命令来查看正在调试的进程。“Inferior infno”命令来切换调试的进程。

“add-inferior [-copies n][-exec executable]”命令来添加新的调试进程, 其中“-copies n”表示启动 n 份进程，“-exec executable”表示要启动进程的程序文件名。

“detach inferior inno”命令终止对指定进程的跟踪，其中 inno 为 gdb 中的进程标识

号。

“kill inferior inno”命令关闭指定的进程

“Info threads”命令用来查看当前进程的线程态“thread threadno”命令用来切换调试线程。

以下面代码为例，来说明如何使用 GDB 来调试多进程多线程程序。

#多进程多线程代码#include <stdio.h> #include <unistd.h> #include <pthread.h>

void childprocess(); void threadfunc();

int main(){

pid\_t pid=fork(); if (pid == 0){

childprocess();

}

else{

pid\_t parentpid=getpid(); printf("Parent Id is %d\n", parentpid); printf("Child Id is %d \n", pid);

}

}

void childprocess(){ pid\_t pid=getpid(); pthread\_t pt;

int status=pthread\_create(&pt,NULL, (void \*)threadfunc,NULL); if (status!=0)

{

printf("Cannot create a new thread\n");

}

pthread\_t tid=pthread\_self();

printf("Current process id is %d , current thread id is %ld \n", pid, tid); sleep(10000);

}

void threadfunc(){ pid\_t pid=getpid();

pthread\_t tid=pthread\_self();

printf("Current process id is %d , current thread id is %ld \n", pid, tid); sleep(10000);

}

调试过程如下所示。

gcc -g -o multiprocessthreads multiprocessthreads.c -lpthread #编译上面的多进程和多线程程序gdb multiprocessthreads #启动 GDB

(gdb) set follow-fork-mode parent # 设置同时调试父子进程，gdb 跟主进程

(gdb) set detach-on-fork off

(gdb) b 10 #设置在代码行 10 处的断点

Breakpoint 1 at 0x4007c5: file multiprocessthreads1.c, line 10. (gdb) b childprocess #设置函数断点

Breakpoint 2 at 0x400819: file multiprocessthreads1.c, line 23. (gdb) b threadfunc #设置函数断点

Breakpoint 3 at 0x400884: file multiprocessthreads1.c, line 36. (gdb) r #从头开始运行

Starting program: /root/book-examples/multiprocessthreads1 [Thread debugging using libthread\_db enabled]

Using host libthread\_db library "/lib/x86\_64-linux-gnu/libthread\_db.so.1". Breakpoint 1, main () at multiprocessthreads1.c:10

10 pid\_t pid=fork(); #停止在断点

(gdb) info inferiors #查看进程信息，从下面信息看到目前仅有一个主进程Num Description Executable

* 1 process 20527 /root/book-examples/multiprocessthreads1 (gdb) n #单步运行

[New process 20533]...

11 if (pid == 0){

(gdb) info inferiors # 查看进程信息，可以看到目前已经启动了两个进程，并且当前跟踪进行

为父进程。

Num Description Executable

2 process 20533 /root/book-examples/multiprocessthreads1

* 1 process 20527 /root/book-examples/multiprocessthreads1 (gdb) n #单步运行

15 pid\_t parentpid=getpid();

(gdb) inferior 2 #转到子进程 2 进行跟踪调试

[Switching to inferior 2 [process 20533] (/root/book-examples/multiprocessthreads1)] [Switching to thread 2 (Thread 0x7ffff7fca740 (LWP 20533))]

...

10 pid\_t pid=fork();

Value returned is $1 = 0

(gdb) c #在子进程中继续运行，并在 childprocess 断点处停止。

Continuing.

Breakpoint 2, childprocess () at multiprocessthreads1.c:23

23 pid\_t pid=getpid();

(gdb) info threads #查看目前线程个数，下面两个线程分别为已经创建的两个进程中的线程。

Id Target Id Frame

* 2 Thread 0x7ffff7fca740 (LWP 20533) "multiprocessthr" childprocess () at multiprocessthreads1.c:23

1 Thread 0x7ffff7fca740 (LWP 20527) "multiprocessthr" main () at multiprocessthreads1.c:15

…

(gdb) n #当执行 pthread\_create 后，将创新新的线程，并且新创建的线程为 3

[New Thread 0x7ffff77f6700 (LWP 20550)] [Switching to Thread 0x7ffff77f6700 (LWP 20550)]

Breakpoint 3, threadfunc () at multiprocessthreads1.c:36

1. pid\_t pid=getpid();

(gdb) info threads #查看目前所有线程，其中标号为“\*”的表示目前正在跟踪的线程

Id Target Id Frame

* 3 Thread 0x7ffff77f6700 (LWP 20550)… threadfunc () at ultiprocessthreads1.c:36

2 Thread 0x7ffff7fca740 (LWP 20533) … childprocess () at multiprocessthreads1.c:25

1 Thread 0x7ffff7fca740 (LWP 20527) … main () at multiprocessthreads1.c:15 (gdb) n #运行线程 3 中的代码

Current process id is 20533 , current thread id is 140737353918272

1. pthread\_t tid=pthread\_self(); (gdb) thread 2 #跟踪线程 2.

…

* + 1. 服务性能测试工具

###### http\_load

http\_load 能够对 web 服务器进行性能压力测试。可以按照官方网站

（<http://acme.com/software/http_load/>）的说明来进行安装。其主要参数如下。“-parallel num” 表示并发客户端的数量。

“-fetches num” 表示所有客户端总共访问次数。“-rate num” 表示每秒访问频率。

“-seconds num”表示总共访问时间，以秒为单位。

“urls” 为保存访问网页链接的文件。在文件内部保存要访问的页面链接地址。其文件格式如下所示。

http://127.0.0.1:8088/index.html http://127.0.0.1:8088/example.html

…

例如运行测试命令“http\_load -parallel 5 -fetches 50 -seconds 20 urls”，表示启动同时

5 个客户端，并在 20 秒时间内共抓取 50 个网页。其运行结果，如下所示。

20 fetches, 5 max parallel, 5880 bytes, in 20.0022 seconds

294 mean bytes/connection

0.999891 fetches/sec, 293.968 bytes/sec

msecs/connect: 107.569 mean, 1017.36 max, 3.426 min

msecs/first-response: 4141.73 mean, 5013.75 max, 5.283 min HTTP response codes:

code 200 -- 20

执行的结果反映了如下的信息。

第一行 “20 fetches, 5 max parallel, 5880 bytes, in 20.0022 seconds”表明在 20.0022 秒时间内，最大启动了 5 个客户端，共完成 20 次抓取，总共传输了 5880 字节。可以看出在

20 秒内没有完成 50 此网页的抓取工作。

第二行“294 mean bytes/connection” 表示每次连接平均传输的数据量。

第三行“0.999891 fetches/sec, 293.968 bytes/sec”表示每秒平均完成多少次网页传输， 以及每秒传输的数据量。其中 fetches/sec 为常用的性能指标参数 QPT（每秒响应数量）。

第四行“msecs/connect: 107.569 mean, 1017.36 max, 3.426 min” 表示建立请求连接的平均时间、最大时间和最小时间（单位为毫秒）。其中 msecs/connect 为常用的性能指标参数（客户端与服务端建立连接的平均时间）

第五行“msecs/first-response: 4141.73 mean, 5013.75 max, 5.283 min”。表示每个连接

（客户端）从发出 htpp 请求消息到开始接受服务器第一个响应消息的平均时间、最大时间和最小时间。这里统计的时间是第四行参数已经建立好连接基础之上的发送请求消息到接 受响应消息之间的时间，可以看为是服务器与客户端建立连接后，响应客户请求网页的时间。

第六行“HTTP response codes: code 200 -- 20” 表示响应代码为 200 的有 20 个。通过观察上面的参数数据，能够知道 web 服务器所支持的并发访问量及响应时间、web

所支持的并发访问量和单位时间网络传输数据量等信息。通过这些信息，可以对 web 服务器性能做出分析。例如，上面的测试中，可以看出每秒才完成一个网页的数据传输，而传输

的数据量仅为约 294 字节，并且“msecs/first-response: 4141.73 mean”数值较大，可以看出每个连接都等待了很长时间才得到服务器的响应信息。

###### 性能指标

Berkeley 大学的研究人员给出了计算机系统中不同设备和组件的在 2017 年的数据传输速度3，具体如下表 1-2 所示。

在进行系统设计和性能分析时，可以参考这个数据表来发现影响系统运行性能的关键 点。例如，经过测试发现系统通过磁盘读写文件的速度在 5MB/秒，远远低于下面表格中的磁盘传输数据速度项。因此可以认为影响目前系统性能的一个因素为文件读写。通过进一步

分析，发现这个系统在进行文件读写过程中出现了大量的阻塞操作，从而导致文件读写速度 较慢。

表 1-2 计算机系统各个器件的数据传输速度表

|  |  |  |
| --- | --- | --- |
| **系统器件数据传输** | **时间** |  |
| **L1 cache reference 读取 CPU 的一级缓存** | 0.5 ns | |
| **Branch mispredict(转移、分支预测)** | 3 ns | |
| **L2 cache reference 读取 CPU 的二级缓存** | 4 ns | |
| **Mutex lock/unlock 互斥锁\解锁** | 17 ns | |
| **Main memory reference 读取内存数据** | 100 ns | |
| **Compress 1K bytes with Zippy 1k 字节压缩** | 2,000 ns | |
| **Send 1K bytes over 1 Gbps network 在 1Gbps 的网络上发送**  **1k 字节（高性能网络）** | 10,000 ns | |
| **Read 4K randomly from SSD 在 SSD 中随机读 4K 字节** | 16,000ns | |
| **Read 1 MB sequentially from SSD 在 SSD 中顺序读 1MB** | 98,000ns | |
| **Read 1 MB sequentially from memory 从内存顺序读取 1MB** | 6,000 ns | |
| **Round trip within same datacenter 从一个数据中心往返一次，**  **ping 一下** | 500,000 ns | |
| **Disk seek 磁盘寻道时间** | 10,000,000 | ns |
| **Read 1 MB sequentially from disk 从磁盘里面读出 1MB** | 20,000,000 | ns |
| **Send packet CA->Netherlands->CA 一个包的一次远程访问** | 150,000,000 | ns |

3 https://people.eecs.berkeley.edu/~rcs/research/interactive\_latency.html

* 1. **实验** 1 Web **服务器初步实现**

**题目 1**：创建 makefile 文件，将\*\*\*节中的 webserver.c 代码进行编译为 webserver 可执行程序。

**题目 2**：在指定目录内准备好 html 文件，以及这些文件中链接的图片。例如，在

/home/web 目录下有 index.html 文件和图像文件 favicon.ico、example.jpg，其代码如下所示。

<html>

<head>

<link rel="shortcut icon" href="favicon.ico" type="image/x-icon"/>

<title>The example web</title>

</head>

<body>

<H1>webserver test page</H1>

<p>

Not pretty but it should prove that webserver works :-)

<p>

<IMG SRC="example.jpg">

</body>

</html>

首先启动 webserver 程序， 例如，“webserver 8088 /home/web”命令将服务器的侦听端口设置为 8088 ，检索文件的根路径为“ /home/web ”。 然后， 在浏览器中输入“http://127.0.0.1:8088/index.html”，观察浏览器中是否能够正常显示网页。在目录中查找 webserver.log 文件，将其打开观察日志信息。

请解释为什么在浏览器中仅请求一次网页，而实际上 webserver 接收了很多次从浏览器发出的文件请求？

请查阅相关文献，说明浏览器请求网页文件时，为加快 html 网页显示的速度，都采用了什么样的技术？

**题目 3**：修改 webserver.c 文件中 logger 函数源代码，使得日志文件中每行信息的起始部分均有时间信息，以表示这行信息被写入的时间。

**题目 4：**在浏览器中多次快速点击刷新按钮后，为什么浏览器要隔很长一段时间才开始显示页面？请结合日志文件中的信息来分析具体原因。

**题目 5：**使用 http\_load 工具对此 webserver 程序进行性能测试，并记录其返回的各种参数数据。

**题目 6**：在 server.c 中增加相关计时函数，分析一下程序的哪个部分最耗时？使用 perf 工具来跟踪 webserver 程序，根据其运行报告进行程序性能分析，请指出 webserver 中比较耗费时间的函数有哪些？

# 第 2 章 Web 服务器的多进程和多线程模型

## 背景介绍

在上节实现了一个简单的单进程 Web 服务器。当此 Web 服务器正在处理一个用户请求的网页时，其他用户对网页的请求将被阻塞，直到处理完这个用户的请求后，才能响应其 他用户的请求。这使得此 Web 服务器不能满足在短时间内满足大量用户同时请求网页的要求。本节将多进程（2.2 节）和多线程（2.3 节）模型引入到此 Web 服务器中，使得其能够

并发处理大量的用户请求。

除此之外，为进一步提高 Web 服务器并发处理性能，还介绍了线程池模型（2.4 节）、业务分割模型（2.5 节）和混合模型（2.6 节）。其中，线程池模型利用“池”思想来缓存和复用线程，以避免因为线程的大量重复创建和销毁所带来的性能损失；业务分割模型通过将

业务流程分解为更小粒度的操作单元，以减少阻塞等待时间，来提高系统的并发性能；混合 模型利用进程和线程各自不同的特性，将多进程和多线程模型混合在一起，在提高系统性能 基础上保证了系统的健壮性。

## 进程模型

###### Linux 中进程创建相关函数

进程是程序的一次执行过程。程序在执行过程中，操作系统要为其分配内存空间、CPU 和 I/O 等计算机资源。操作系统为方便管理程序运行中所需的计算机资源，将与程序运行相关的计算机资源抽象为进程。因为程序一次执行对应着一个进程，而进程里面有程序运行所 需的资源，因此操作系统可以通过对进程的管理，来掌握每个程序的执行状态和运行过程。 例如程序 A 在运行时需要用内存 200MB，并且需要占用 socket 通信接口来进行通信，那么

在程序 A 运行时，操作系统会为程序 A 分配这些资源，并将资源相关信息记入这次运行的

进程中。当有多个程序需要并发运行时，操作系统为每个运行的程序分配计算机资源，并计 入它们当前运行的进程中。

当操作系统中存在多个进程时，操作系统会为它们合理地安排计算机资源，以提高它们 并发运行的效率。因此进程是操作系统资源分配和调度的基本单位。

Linux 为创建进程和使用进程提供了如下接口。

* **fork** 函数

程序在执行 fork 函数后，linux 会创建子进程，子进程和父进程共享程序 fork 函数后面执行的代码。而且在创建子进程时，linux 系统会将父进程内的几乎所有的资源复制给子进程，这样子进程就能够共享父进程已经获得的资源了。为提高效率，fork 函数采取了写时复制技术，当父子进程空间中的内容发生变化时，才将变化内容所在内存段复制一份给子进程，

否则两个进程将会共享内容空间。写时复制技术极大地减少了不必要的数据复制过程。

既然在执行 fork 函数后，父子进程会共享后面的程序代码，那么该如何让这两个进程执行不同的运行代码呢？这就需要根据 fork 函数的返回值来确定。父进程中获得的此函数值为子进程的 PID，子进程获得的应该为 0。如果创建子进程失败，父进程得到的 PID 为-1。具体运行逻辑见如下代码。

#include <stdio.h> #include <unistd.h> #include <sys/types.h>

int main(int argc,char \*argv[]){ pid\_t pid=fork();

if(pid==0)

printf("This is a child process\n"); //child process executes the line else //parent process executes the following line

printf("This is a parent process, and its child process id is %d\n", pid); return 0; //parent and child processes both execute the line

}

###### exec 系列函数

exec 系列函数包括 execl、execlp、execle、execv 和 execvp。它们仅是调用参数不同， 但具有相同的语义。它们都是用于将当前进程替换为一个新的进程，并且这个新的进程与被 替换的进程具有相同的 PID。它们的具体函数参数格式请查阅相关的函数手册，在这里仅以execl 函数来说明其用途。例如下面的代码，将子进程替换为一个“ls”程序进程。在子进程

开始执行时将打印出“entering child process”，但是当执行 execl 函数后，“existing child

process”将不会被打印，这是因为该子进程已经被执行“ls”的进程替换。

#include <stdio.h> #include <unistd.h> #include <sys/types.h>

int main(int argc,char \*argv[]){ pid\_t pid=fork();

if(pid==0){

printf("entering child process\n"); execl("/bin/ls","ls","-l",NULL); printf("existing child process\n");

}

else

printf("This is a parent process, and its child process id is %d\n", pid); return 0;

}

* **vfork 函数**

vfork 函数在创建子进程后，子进程与父进程共享空间，这样在子进程修改的变量数据， 父进程也能够看得到。但是由于两个进程共享空间，很容易导致相互破坏运行堆栈。一般会 在 vfork 子进程函数中使用 exec 函数来替换自己程中的内容。

###### Linux 中进程通信相关函数

进程间通信指的是两个或多个进程之间信息的共享，其主要方式包含管道、共享内存、 消息队列、信号量、网络通信、文件等内容。具体函数如下。

###### 管道

管道是用于进程间通信的一种特殊文件，进程间通过对这个特殊文件的读写来完成信 息的传递。管道一般是半双工的，即数据流只有一个方向。管道分为：匿名管道和命名管道。 两者的用法有所不同。其中匿名的管道主要用于亲缘性进程之间（比如，父子进程和兄弟进程之间）；而命名管道并无上述限制，可以在无关进程之间进行数据通信。

**函数：**int pipe(int fd[2])

用来创建匿名管道，其中参数 fd[2]为对此管道的读写操作描述符，其中 fd[0]为读描述符；fd[1]为写描述符。在用于父子进程通信时，因为通道为半双工的，所以在父子进程中要 关闭相关的描述符，来获得不同的数据流向。比如，在父进程中关闭 fd[0]描述符，而在子进程中关闭 fd[1]描述符，则父进程负责向管道写数据、子进程从管道读数据；如果父子进程关闭的描述符相反，则数据流向也相反。具体父进程向子进程传递消息的例子代码如下所示。

#include <stdio.h> #include <unistd.h> int main()

{

int fd[2]; // 两个文件描述符

pid\_t pid;

char readbuf[20];

char\* writebuf="pipe message"; if(pipe(fd) < 0) // 创 建 管 道

printf("create pipe error!\n");

if((pid = fork()) < 0) // 创建子进程

printf("fork error!\n"); else if(pid > 0) // 父 进 程

{

close(fd[0]); // 关闭读描述符

write(fd[1], writebuf, strlen(writebuf));

}

else //子进程

{

close(fd[1]); // 关闭写描述符read(fd[0], readbuf, 20); printf("%s", readbuf);

}

return 0;

}

**函数：**int mkfifo(const char \*pathname, mode\_t mode)

用来创建命名管道。此命名管道是文件系统中一个特殊设备文件。通过参数 pathname 来指定这个命名管道文件的名称；通过 mode 参数来制定这个文件的权限。当创建好这个文件后，可以使用文件操作函数 open 来打开这个文件，以进行读写操作。与普通文件的打开操作模式略有不同，其不包含 O\_RDWR 模式（读写模式）。其中 open 函数中的 flag 参数可以读、写、阻塞和非阻塞。具体有以下四种模式：

O\_RDONLY：open 将会调用阻塞，除非有另外一个进程以写的方式打开同一个 FIFO， 否则一直等待。

O\_WRONLY：open 将会调用阻塞，除非有另外一个进程以读的方式打开同一个 FIFO， 否则一直等待。

O\_RDONLY | O\_NONBLOCK：非阻塞方式只读方式打开文件，无论是否有其它进程以写的方式打开此管道文件，open 均会成功返回，此时 FIFO 被读打开。

O\_WRONLY|O\_NONBLOCK：非阻塞方式只写方式打开文件，如果此时没有其它进程以 读的方式打开，open 会失败打开，此时 FIFO 没有被打开，返回-1。

下面为应用命名管道来实现生产者和消费者问题的代码。

//生产者进程,produce.c #include <stdio.h> #include <stdlib.h>

#include <fcntl.h> #include <sys/stat.h>

int main(){ int fd; int n, i;

char\* buf= "fifo example"; time\_t tp;

// 以写打开一个 FIFO,如果此 FIFO 读进程不存在，则阻塞。

if((fd = open("fifo-example", O\_WRONLY)) < 0) { perror("open fifo-example failed"); exit(1);

}

printf("send message: %s to fifo-example file", buf); if(write(fd, buf, strlen(buf)) < 0) { // 将 Buf 写入到 FIFO 中

perror("write fifo failed"); close(fd);

exit(1);

}

close(fd); // 关闭此 fifo-example 文件

return 0;

}

//消费者进程 consumer.c #include <stdio.h>

#include <stdlib.h> #include <errno.h> #include <fcntl.h> #include <sys/stat.h>

int main()

{

int fd; int len;

char buf[1024];

if(mkfifo("fifo-example", 0777) < 0 ) {// 创 建 FIFO 管 道 , printf("Please check the file must not exist in the directory\n"); perror("Create FIFO Failed");

}

if((fd = open("fifo1", O\_RDONLY)) < 0) // 以 读 打 开 FIFO

{

perror("Open FIFO Failed"); exit(1);

}

while((len = read(fd, buf, 1024)) > 0) // 读取 FIFO 管道

printf("Read message: %s from the fifo-example pipe", buf);

close(fd); // 关 闭 FIFO 文 件

return 0;

}

###### 消息队列

消息队列就是操作系统内核中存放消息的队列。消息队列中的数据并不依赖于具体的 进程而存在。消息队列可以很方便地作为多个进程之间交换数据的场所。消息队列与管道相 比，其可提供格式化数据的存储，而管道只能提供流式文件存储。linux 支持 POSIX 和 System

V 接口格式的消息队列，两种差别并不是太大。本书将以 POSIX 接口来介绍消息队列的使

用。POSIX 小的队列的操作函数如下。

mqd\_t mq\_open(const char \*name, int oflag);

mqd\_t mq\_open(const char \*name, int oflag, mode\_t mode, struct mq\_attr \*attr);

为创建或获取一个消息队列，其中 name 为消息队列名字；oflag 为打开队列的模式， 包含 O\_RDONLY、O\_WRONLY、O\_RWRD、O\_CREATE、O\_EXCL、O\_NONBLOCK 等模式；

在 oflag 指定为 O\_CREATE 时，mode 参数指定消息队列的权限；attr 为消息队列的属性信息，如果其取值为 NULL，则会按默认值配置消息队列。其具体数据结构如下。

struct mq\_attr {

long mq\_flags; /\* Flags: 0 or O\_NONBLOCK \*/

long mq\_maxmsg; /\* Max. # of messages on queue \*/ long mq\_msgsize; /\* Max. message size (bytes) \*/

long mq\_curmsgs; /\* # of messages currently in queue \*/

};

int mq\_setattr(mqd\_t mqdes, const struct mq\_attr \*newattr, struct mq\_attr \*oldattr)和int mq\_getattr(mqd\_t mqdes, struct mq\_attr \*attr)，

用于设置或者获取指定消息队列的属性。

int mq\_send(mqd\_t mqdes, const char \*msg\_ptr,size\_t msg\_len, unsigned int msg\_prio); int mq\_timedsend(mqd\_t mqdes, const char \*msg\_ptr,size\_t msg\_len, unsigned int

msg\_prio,const struct timespec \*abs\_timeout);

用于发送消息到指定的消息队列。其中参数 mqdes 为消息队列描述符，为 mq\_open 所创建；msg\_ptr 和 msg\_len 分别是消息的指针和长度；msg\_prio 用于指定消息的优先级。如果消息队列满了就阻塞，新的消息必须等到消息队列中有空间才能进入。如果在创建或打开消息队列 时 oflag 中 O\_NOBLOCK 选项，则会报错。如果想让消息队列满后，只等待有限的时间，则可以使用 abs\_timeout 来进行设置等待的时间。

ssize\_t mq\_receive(mqd\_t mqdes, char \*msg\_ptr, size\_t msg\_len, unsigned int \*msg\_prio)

和

ssize\_t mq\_timedreceive(mqd\_t mqdes, char \*msg\_ptr, size\_t msg\_len, unsigned int

\*msg\_prio, const struct timespec \*abs\_timeout)； 用于从消息队列里面接受消息。

int mq\_notify(mqd\_t mqdes, const struct sigevent \*sevp)

向消息队列建立或删除消息通知事件。具体应用请查阅相关文献。

int mq\_close(mqd\_t mqdes) 关闭消息队列。

int mq\_unlink(const char \*name)

移除指定的消息队列。因为消息队列并不依附于进程而存在，因此在使用消息队列的最 后一个进程关闭消息队列后，需使用 mq\_unlink 函数以给操作系统内核明确的信号来删除此消息队列，否则将会造成系统资源的浪费。

使用消息队列的生产者和消费者问题的实现代码如下。

//头文件 book.h struct book

{

char name[36]; int id;

};

//生产者进程 produce.c #include <stdio.h> #include <stdlib.h> #include <sys/ipc.h> #include <sys/msg.h> #include <sys/types.h> #include <unistd.h> #include <errno.h> #include <mqueue.h> #include <fcntl.h> #include <sys/stat.h> int main()

{

//创建一个消息队列，并且为只写

mqd\_t mqid = mq\_open("/examplemq", O\_WRONLY|O\_CREAT, 0777, NULL); if (mqid == -1)

err\_exit("mq\_open error");

struct book bk = {"webbook", 11}; unsigned prio = 2;

//向此消息队列发送 book 实例消息

if (mq\_send(mqid, (const char \*)&bk, sizeof(bk), prio) == -1) err\_exit("mq\_send error");

//关闭这个消息队列

mq\_close(mqid); return 0;

}

//消费者进程 consumer.c int main()

{

//以只读的方式打开一个消息队列

mqd\_t mqid = mq\_open("/examplemq", O\_RDONLY); if (mqid == -1)

err\_exit("mq\_open error");

struct book bk; int itcv; unsigned prio;

struct mq\_attr attr;

//获得消息队列的属性信息

if (mq\_getattr(mqid, &attr) == -1) err\_exit("mq\_getattr error");

//从消息队列里面接受 book 信息

if ((itcv = mq\_receive(mqid, (char \*)&bk, attr.mq\_msgsize, &prio)) == -1) err\_exit("mq\_receive error");

printf("receive book message from mq %d\n", itcv);

printf("The book's id is %d and its name is %s\n", bk.id,bk.name);

//关闭消息队列mq\_close(mqid);

//删除此消息队列

mq\_unlink(mqid); return 0;

}

###### 信号量

信号量表示可用资源的数量，其通过 P、V 操作来完成可用资源量的计数更新，从而达到进程间同步和互斥的目的。同样 linux 下的信号量函数的操作函数也对应着两种：POSIX 和 System V。本文将主要对 POSIX 接口的信号量进行介绍。在 POSIX 中将信号量分为无名信号量（unname semaphore）和有名信号量（named semaphore）两种。无名信号量是基

于内存的信号量，如果其不放入进程共享内存区，则无法在进程间使用，仅能在同一进程的 多线程中使用。而有名信号量可以提供进程间的操作。

无名信号量使用 sem\_init 函数进行初始化，使用 sem\_destory 函数进行销毁；有名信号

量使用sem\_open 函数进行初始化，使用 sem\_close 函数来关闭信号量资源，使用 sem\_unlink 函数来销毁进行间信号量资源。其余信号量的操作函数为两种信息量的公用函数。

sem\_t \*sem\_open(const char \*name, int oflag);

sem\_t \*sem\_open(const char \*name, int oflag,mode\_t mode, unsigned int value);

有名信号量的打开或创建函数。用于进程间同步和互斥操作。

int sem\_close(sem\_t \*sem);

int sem\_unlink(const char \*name);

有名信号量的关闭和销毁函数。

int sem\_init(sem\_t \*sem, int shared, unsigned int value); int sem\_destroy(sem\_t \*sem);

无名信号量的初始化和销毁函数，主要用在多线程环境中。

int sem\_wait(sem\_t \*sem);

相当于信号量的 P 操作。其测试所指定信号量的值，大于 0，将它减 1 并返回，等于 0， 调用进程或线程休眠，直到该值大于 0，将它减 1，函数随后返回。

int sem\_trywait(sem\_t \*sem);

相当于信号量的 P 操作。但是其与上面的 sem\_wait 不同的是，当所指定信号量值为 0

时，其上的进程并不休眠，而是返回一个 EAGAIN 错误。

int sem\_post(sem\_t \*sem);

相当于信号量的 V 操作。

int sem\_getvalue(sem\_t \*sem, int \*valp);

此函数通过参数 valp 返回指定信号量中的当前数值

###### 共享内存

共享内存指多个进程之间共享的内存区域，这块内存区域可以被多个进程访问。在

POSIX 标准中，共享内存对象可以通过以下函数来实现。

int shm\_open(const char \*name, int oflag, mode\_t mode)； int shm\_unlink(const char \*name);

其中，shm\_open 用于创建内存或打开一个共享内存区域对象；shm\_unlink 为删除一个共享内存对象。要实现多个进程之间共享内存区域，还需要 mmap 函数来配合使用。

void \*mmap(void \*start, size\_t len, int prot, int flags, int fd, off\_t offset)

该函数的主要作用是将文件或设备映射到调用进程空间中。当文件被映射到进程空间 后，可以通过该进程的虚拟地址读写来完成文件的读写操作，能够加快文件操作的 I/O 速度。该函数提供对共享内存对象进行进程内存映射。其中，

start 为被映射内容在进程空间内的起始地址，如果为 NULL，则让内核自动为其选择起始地址；

len 映射到进程地址空间的字节数；prot 为内存映射区域的读写操作保护标志位，由PROT\_READ、PROT\_WRITE、PROT\_EXEC 和 PROT\_NONE 等值组合而成，可以分别表示该内存区域的读、写和执行权限；

flags 表示映射内存类型，如果为 MAP\_SHARD，则表示在映射内存区域内修改的数据对所有能访问该内存区域的进程可见，如果为 MAP\_PRIVATE，则该内存区域的修改数据仅能被修改该区域的进程所见，其它进程看不到被修改的数据；

fd 为文件、设备或共享内存区域对象描述符；

offset 为当前文件、设备或共享内存区域对象的偏移位置。

通过 munmap(void \*start, size\_t len)函数可以从进程地址空间中删除一个映射。

下面的例子代码通过使用共享内存和信号量来完成父子进程协作计数的功能。父子进 程互斥地对共享内存中据进行修改以达到计数目的。

//semaphoreposix.c 文件

//使用下面的命令来编译下面代码

// gcc -std=gnu99 -Wall -g -o semaphoreposix semaphoreposix.c -lrt -lpthread

#include <sys/mman.h> #include <semaphore.h> #include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <fcntl.h> #include <sys/stat.h> #include <sys/types.h> #include <signal.h>

#define NUM 100

#define SEM\_NAME "sem\_example" #define SHM\_NAME "mmap\_example"

int main(){

int count=0; sem\_t\* psem;

//创建信号量,初始信号量为 1

if((psem=sem\_open(SEM\_NAME, O\_CREAT,0666, 1))==SEM\_FAILED){

perror("create semaphore error"); exit(1);

}

int shm\_fd;

//创建共享内存对象

if((shm\_fd=shm\_open(SHM\_NAME,O\_RDWR| O\_CREAT,0666)) < 0){ perror("create shared memory object error");

exit(1);

}

/\* 配 置 共 享 内 存 段 大 小 \*/ ftruncate(shm\_fd, sizeof(int));

//将共享内存对象映射到进程

void \* memPtr = mmap(NULL, sizeof(int), PROT\_READ | PROT\_WRITE, MAP\_SHARED, shm\_fd, 0);

if(memPtr==MAP\_FAILED){ perror("create mmap error"); exit(1);

}

//为此内存区域赋值

\* (int \*) memPtr= count;

//创建子进程pid\_t pid=fork();

if (pid==0) //child process

{

for (int i = 0; i < NUM; ++i)

{ //信号量实现的临界区

sem\_wait(psem);

printf("Child Process count value: %d\n", (\*(int \*) memPtr)++); sem\_post(psem);

}

}

else if (pid > 0){ // parent process for (int i = 0; i < NUM; ++i)

{ //信号量实现的临界区

sem\_wait(psem);

printf("Parent Process count value: %d\n", (\* (int \*)memPtr)++); sem\_post(psem);

}

sleep(1);

//卸载各种资源

if (munmap(memPtr, sizeof(int)) == -1) { perror("unmap failed");

exit(1);

}

if (close(shm\_fd) == -1) { perror("close shm failed"); exit(1);

}

if (shm\_unlink(SHM\_NAME) == -1) { perror("shm\_unlink error "); exit(1);

}

if(sem\_close(psem)==-1){ perror("close sem error"); exit(1);

}

if (sem\_unlink(SEM\_NAME)==-1) { perror("sem\_unlink error"); exit(1);

}

}else{

perror("create childProcess error"); exit(1);

}

exit(0);

}

###### 网络通信

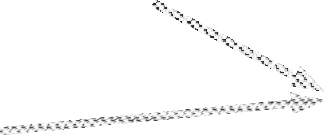
可以通过前一章节中的 socket 接口实现两个进程之间通信，这两个进程可以在同一主机上，也可以在不同主机上（不同主机之间有网络连接）

###### 多进程 Web 服务器模型

在实验 1 中，实现了一个基本的 Web 服务器。这个 Web 服务器是单进程模型的，当服务器接受到客户端请求，就建立一个网络连接，并从此连接解析请求的文件；然后从文件系 统中读取这个文件到缓存，最后通过这个网络连接将缓存中的内容发送到客户端。在 Web 服务器处理这个客户端的上述步骤内容时，如果有其它客户端也请求连接 Web 服务器，则

其它客户端的请求连接将被阻塞，直到 Web 服务器完成这个客户端的所有业务处理，才会

从其它客户端连接中选择一个再进行上述步骤处理。如下图 2-1 所示，Client B 和其它的Client 都被阻塞在 Web 服务器的 accept 函数，而当前 Web 服务器正在处理 Client A 的文件请求，也就是其它客户端要等待 Client A 的请求被处理完才能依次被处理。很明显，这样的设计使得 Web 服务器的并发处理客户请求的能力比较弱。



**Client B**

**accept**

**Client ·**

**1. read msg from connect 2.read file from filesystem**

**3. write msg to connect**

**WebServer**

**Client A**

图 2-1 Web 服务器串行处理多用户的同时请求

如果 WebServer 在处理客户端连接中请求的文件同时，也能够接受其它客户端的连接请求处理，则将会提高 Web 服务器的并发处理能力。如果采用多进程模型，则 WebServer 在接收到客户端连接请求后，就创建一个子进程，在这个子进程中进行客户端的文件请求处

理。如果多个客户端同时请求连接，则会创建多个子进程。其中，每个子进程都会处理一个 客户端的请求。这样就使得 WebServer 能够在一个时间段内能够同时处理多个客户端请求， 大大增加了其并发处理能力。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **1. read msg…** |  | **1. read msg…** |  | **1. read msg…** |
| **2.read file …** | **2.read file …** | **2.read file …** |
| **3. write msg…** | **3. write msg…** | **3. write msg…** |
| **…** | **…** | **…** |



**WebServer**

**accept/fork**



**Client A**

**Client ·**

#### Client B

图 2-2 Web 服务器多进程模型

具体过程如下图 2-2 所示，每个客户端请求连接 WebServer 指定的侦听接口（虚线箭头），在 WebServer 侦听到连接请求后，将会与客户端建立连接通道（实线箭头），同时使用fork 创建子进程。在这个子进程中处理这个连接通道，而父进程会马上返回到 accept 函数， 继续等待新的客户端的连接请求。这样就使得 WebServer 为每个客户端创建一个进程，来处理其对文件的请求。

###### 实验 2 Web 服务器的多进程模型实现

根据上节对 WebServer 多进程模型的描述，多进程相关函数，来完成如下题目。

**题目 1：**使用 fork 函数，设计并实现 WebServer 以支持多进程并发处理众多客户端的请求。

**题目 2：**使用信号量、共享内存等系统接口函数，来统计每个子进程的消耗时间以及所有子进程消耗时间之和。

**题目 3：**使用 http\_load 来测试当前设计的多进程 WebServer 服务性能，根据测试结果来分析其比单进程 Web 服务性能提高的原因。同时结合题目 2，来分析当前多进程WebServer 的性能瓶颈在何处？是否还能够继续提高此 WebServer 服务的性能？

## 线程模型

###### Linux 线程模型

线程负责具体程序逻辑的执行，是处理器调度的基本单位。与进程相比，线程不具有独 立的地址空间，可以与进程内的其它线程共享进程的资源。因此线程具有容易共享信息、调 度切换开销小等特点。在现代操作系统中将线程分为用户线程、内核线程和 LWP(Light

Weight Process-轻量级进程)三种类型。

###### 用户线程

用户线程由线程库在用户空间内创建、调度、同步和管理。由于用户线程由线程库来管 理，它们之间调度切换并不需要进行系统调用，因此切换开销小。

但是由于操作系统内核并不知道用户线程的存在，内核仅是以用户线程所在的进程为 单位来进行处理器调度，导致此线程内的所有用户线程只能共享一个处理器资源，不能充分 地利用多处理器；当一个用户线程进行系统调用而导致阻塞时，操作系统内核将阻塞其所在 的进程，因此此进程的其它用户线程也不能运行。

###### 内核线程

内核线程是由操作系统内核来创建、调度和管理的线程。内核线程是操作系统调度的基 本单位，这些内核线程在操作系统进程内竞争系统资源，如果有一个内核线程处于阻塞状态， 并不影响其它内核线程的调度和运行。由于内核线程之间切换需要进行系统调用（用户态到系统态之间相互转换），因此切换开销较大。

###### LWP

LWP（轻量级进程）是一种由内核支持的用户线程。它是基于内核线程的抽象，是用户 线程与内核线程之间的桥梁。一个 LWP 与一个内核线程相对应，因此操作系统内核是能够识别和调度 LWP。将用户线程绑定到 LWP 后，LWP 可以被看为用户线程的虚拟处理器。

如果一个用户线程与一个内核线程相对应，则为“一对一”模型；如果多个用户线程与 一个内核线程相对应，则为“多对一”模型；如果多个用户线程与多个内核线程对应，则为 “多对多”模型。每个操作系统提供了不同的线程对应模型。

在目前的 Linux 中默认的 POSIX 线程模型采用的是“一对一”模型，也就是一个用户线

程对应一个内核线程，其通过线程创建函数创建的线程是 LWP 类型的。

###### POSIX 线程库接口

Linux 提供了兼容 POSIX 标准的线程操作 API，其主要的函数如下所示。

###### int pthread\_create(thread\_t\* thread, pthread\_attr\_t\* attr,void\*

**(start\_routine)(void\*), void\* arg)**

线程创建函数，其中参数 thread 是创建好线程的指针，用于后续的线程操作；atrr 为线程属性指针，如果为 NULL，将按默认属性来创建线程；start\_routine 为完成线程逻辑功能的函数指针；arg 为向线程传递参数的指针。如果成功返回 0，失败返回-1。

###### void pthread\_exit(void\* retval)

退出当前线程函数，参数 retval 用来返回当前函数的退出值。

###### int pthread\_cancel(pthread\_t thread)

向目标线程发送请求终止（cancel）信号，其中参数 thread 为要被取消运行的线程 id。当然调用 pthread\_cancel 并不意味着目标的线程一定要被终止，而是目标线程接收到 cancel 信息后，它自己决定如何来响应这个信号：忽略这个信号、立即退出、运行至取消点

（cancelation-point）后再退出。

在此函数发出 cancel 信号后，目标线程的 cancel state 来决定是否接受此 cancel 信号， 如果 cancel state 是 PTHREAD\_CANCEL\_ENABLE （默认）则接收信号；如果是PTHREAD\_CANCEL\_DISABLE ，则不接收此信号。对 cancel state 的设置使用“ int pthread\_setcancelstate(int state, int \*oldstate) ”函数，其中参数 state 可以设置为上述两种状态之一。

当目标线程接收到 cancel state 信号，目标线程的 cancel type 来决定何时取消。如果cancel type 是 PTHREAD\_CANCEL\_DEFERRED（默认），目标线程并不会马上取消，而是在执行 下 一 条 cancellation point 的 时 候 才 会 取 消 ； 如 果 cancel type 是PTHREAD\_CANCEL\_ASYNCHRONOUS，目标线程会立即取消。对 cancel type 的设置使用“int pthread\_setcanceltype(int type, int \*oldtype) ”函数，其中参数 type 为上述两种数值之一。

而 cancellation point 为调用 POSIX 库中 pthread\_join 、 pthread\_testcancel 、pthread\_cond\_wait、pthread\_cond\_timedwait、sem\_wait、sigwait 以及 read、write 等函数的位置。

###### int pthread\_join(pthread\_t\* tid, void\*\* thread\_return)

等待线程结束函数，其中参数 tid 为被等待的线程 id 指针；thread\_return 为被等待线程的返回值，也就是 pthread\_exit 里面的参数值。如果当前线程调用此函数，将会阻塞，直到被等待线程运行结束后或者被其它线程取消运行，当前线程才会继续运行。另外需要注意的

是，一个线程不能被多个线程等待，否则除第一个等待线程外，其它等待线程均会返回错误 值 ESRCH。

下面为 linux 手册中关于 pthread 的例子代码，演示了上述函数的使用。

#include <pthread.h> #include <stdio.h> #include <errno.h> #include <stdlib.h> #include <unistd.h>

#define handle\_error\_en(en, msg) \

do { errno = en; perror(msg); exit(EXIT\_FAILURE); } while (0)

//进程函数

static void \* thread\_func(void \*ignored\_argument)

{

int s;

/\* 将线程中 cancel state 暂时改为不接收 cancel 信号，以不响应其它进程向它发出的 cancel 信号

\*/

s = pthread\_setcancelstate(PTHREAD\_CANCEL\_DISABLE, NULL); if (s != 0)

handle\_error\_en(s, "pthread\_setcancelstate"); printf("thread\_func(): started; cancellation disabled\n"); sleep(5);

printf("thread\_func(): about to enable cancellation\n");

/\* 恢复线程中接收 cancel 信号的状态\*/

s = pthread\_setcancelstate(PTHREAD\_CANCEL\_ENABLE, NULL); if (s != 0)

handle\_error\_en(s, "pthread\_setcancelstate");

/\* sleep 函数是一个 cancellation point \*/

sleep(1000); /\* Should get canceled while we sleep \*/

/\* 下面的代码在正常情况下应该不会被执行\*/ printf("thread\_func(): not canceled!\n");

return NULL;

}

int main(void)

{

pthread\_t thr; void \*res;

int s;

/\* 创建一个线程，并向它发生 cancel 信号 \*/

s = pthread\_create(&thr, NULL, &thread\_func, NULL); if (s != 0)

handle\_error\_en(s, "pthread\_create");

sleep(2); /\* 给这个线程留出足够多的时间启动 \*/

printf("main(): sending cancellation request\n"); s = pthread\_cancel(thr);

if (s != 0)

handle\_error\_en(s, "pthread\_cancel");

/\* 调用 pthread\_join 函数，并查看目标线程的退出状态\*/ s = pthread\_join(thr, &res);

if (s != 0)

handle\_error\_en(s, "pthread\_cancel");

/\* 调用 pthread\_join 函数，并查看目标线程的退出状态\*/ s = pthread\_join(thr, &res);

if (s != 0)

handle\_error\_en(s, "pthread\_join");

if (res == PTHREAD\_CANCELED)

printf("main(): thread was canceled\n"); else

printf("main(): thread wasn't canceled (shouldn't happen!)\n"); exit(EXIT\_SUCCESS);

}

这段代码正常执行将打印出如下信息。

thread\_func(): started; cancellation disabled

main(): sending cancellation request thread\_func(): about to enable cancellation main(): thread was canceled

对与 pthread 的属性设置，提供了如下的接口函数。

* + int pthread\_attr\_init (pthread\_attr\_t\* attr)

属性初始化函数，创建一个线程属性结构，并通过 attr 指向此结构。

* + int pthread\_attr\_destroy(pthread\_attr\_t \*attr);

销毁一个线程属性结构。

* + int pthread\_attr\_setscope (pthread\_attr\_t\* attr, int scope);

设置线程的作用域，在 POSIX 标准中参数 scope 可以取 PTHREAD\_SCOPE\_SYSTEM 和 PTHREAD\_SCOPE\_PROCESS。这两个值分别表示线程调度范围是在系统中还是在进程中。但是在 Linux 中仅支持 PTHREAD\_SCOPE\_SYSTEM（也就是“一对一”模型），如果 scope 设置 PTHREAD\_SCOPE\_PROCESS，则会报错，返回 ENOTSUP。

* + int pthread\_attr\_setdetachstate (pthread\_attr\_t\* attr, int detachstate)

设置分离属性。创建的线程分为分离和非分离状态。如果线程为分离状态(detachstate 取值为 PTHREAD\_CREATE\_DETACHED），则线程在运行完就自行结束并释放资源；如果线程为非分离状态（detachstate 取值为 PTHREAD\_CREATE\_JOINABLE），则此线程需要等待它的线程中 pthread\_joint 函数返回后，才终止并释放资源。线程默认为非分离状态。如果将线程状态设置为分离状态，需要注意的是创建的线程运行可能非常快，在 pthread\_create 函数没有返回时已经运行结束，这时 pthread\_create 中可能会得到操作的线程号。为了避免这个问题，可以在创建线程里调用 pthread\_cond\_timewait 函数，让线程等待一会。

* + int pthread\_attr\_setaffinity\_np(pthread\_attr\_t \*attr, size\_t cpusetsize, const

cpu\_set\_t \*cpuset)

设置线程的 CPU 亲缘性。在多 CPU 环境下，如果设置一个线程在一个指定 CPU 运行， 则需要调用此函数。如下代码将指定线程运行在 0 号 CPU。

pthread\_attr\_t attr1; pthread\_attr\_init(&attr1); cpu\_set\_t cpu\_info;

CPU\_ZERO(&cpu\_info); CPU\_SET(0, &cpu\_info);

pthread\_attr\_setaffinity\_np(&attr1, sizeof(cpu\_set\_t), &cpu\_info)

* + int pthread\_attr\_setschedpolicy(pthread\_attr\_t \*attr, int policy)

设置线程的调度策略。policy 可以取值 SCHED\_FIFO、SCHED\_RR 和 SCHED\_OTHER。其中 SCHED\_OTHER 为默认的分时调度策略，表示线程一旦开始运行，直到时间片运行完或者阻塞或者运行结束才让出 CPU 控制权，此状态下不支持线程的优先级。SCHED\_FIFO 为实时

调度，执行先来先服务的调度策略，一个线程一旦占有 CPU，则运行到阻塞或者有更高优先级的线程到来。SCHED\_RR 为实时调度，执行时间片轮转调度。

* + int pthread\_attr\_setschedparam(pthread\_attr\_t \*attr, const struct

sched\_param \*param)

设置线程的优先级。其中 sched\_param 结构中仅有属性 sched\_priority，其用来设置线程的优先级。线程优先级可以取 1-99 中任意个数字，数值越大优先级越高。

pthread\_attr\_t attr;

struct sched\_param param; pthread\_attr\_init(&attr); pthread\_attr\_setschedpolicy(&attr, SCHED\_FIFO); param.sched\_priority = 50; pthread\_attr\_setschedparam(&attr,&param);

###### Linux 线程间同步与互斥

线程除了可以使用进程间通信函数来实现同步与互斥外，在 Linux 系统中 POSIX 库中还提供一些列接口函数用于线程间的同步。

###### 信号量

线程间同步和互斥还可以使用无名信号量来实现。有关无名信号量的创建、操作和销毁 函数见进程模型。

###### 互斥量操作

互斥量可以用来实现临近区，让线程互斥地使用临界资源。在 linux 的 POSIX 库中，提供 pthread\_mutex\_init、pthread\_mutex\_lock、pthread\_mutex\_trylock、pthread\_mutex\_unlock 和 pthread\_mutex\_destroy 等函数来完成互斥量的初始化、加锁、释放、摧毁等操作。其中pthread\_mutex\_trylock 为非阻塞函数，如果互斥量没有被锁住，其对互斥量加锁，并进入临界区；如果互斥量已经加锁，则返回 EBUSY，而不会阻塞。pthread\_mutex\_lock 为阻塞函数， 如果已经有其它线程占有互斥量，则阻塞直到获得这个互斥量为止。

###### 读写锁

对于读写者问题（多个读线程能同时读取数据，只有写线程写入数据时才会阻塞其它线程），POSIX 库提供了 pthread\_rwlock\_init 和 pthread\_rwlock\_destroy 函数用来创建和销毁读写锁；pthread\_rwlock\_rdlock、pthread\_rwlock\_wrlock 和 pthread\_rwlock\_timedrdlock 函数使

用阻塞的方式来获得读锁或者写锁；pthread\_rwlock\_tryrdlock 和 pthread\_rwlock\_trywrlock

函数使用非阻塞方式来获得读锁或写锁； pthread\_rwlock\_unlock 函数释放读写锁。具体用法，如下代码所示。

# 编 译 命 令 ：gcc -std=gnu99 -o readerwriter readerwriter.c -lpthread #include <stdio.h>

#include <unistd.h>

#include <pthread.h>

pthread\_rwlock\_t rwlock; //读写锁int num=0;

//读线程函数void \* reader1(){

for (int i = 0; i < 10; ++i)

{

pthread\_rwlock\_rdlock(&rwlock);

printf("reader1 reads %d times num = %d\n", i,num); pthread\_rwlock\_unlock(&rwlock);

sleep(1);

}

}

void \* reader2(){

for (int i = 0; i < 10; ++i)

{

pthread\_rwlock\_rdlock(&rwlock);

printf("reader2 reads %d times num = %d\n", i,num); pthread\_rwlock\_unlock(&rwlock);

sleep(1);

}

}

//写线程

void \* writer1(){

for (int i = 0; i < 10; ++i)

{

pthread\_rwlock\_wrlock(&rwlock); num++;

printf("writer1 writes %d times num=%d\n",i,num); pthread\_rwlock\_unlock(&rwlock);

sleep(1);

}

}

int main(int argc, char const \*argv[])

{

pthread\_t thr1,thr2,thw1; //读线程、写线程

pthread\_rwlock\_init(&rwlock,NULL); //初始化读写锁

//rwlock=PTHREAD\_RWLOCK\_INITIALIZER; //使用宏来初始化读写锁

//创建读写线程pthread\_create(&thr1,NULL,reader1,NULL); pthread\_create(&thr2,NULL,reader2,NULL); pthread\_create(&thw1,NULL,writer1,NULL);

//等待线程结束回收资源

pthread\_join(thr1,NULL); pthread\_join(thr2,NULL); pthread\_join(thw1,NULL);

//销毁读写锁pthread\_rwlock\_destroy(&rwlock); return 0;

}

###### 条件变量

条件变量用于某个进程或线程等待某个信号条件到来时才继续运行的场景。POSIX 库中提供了 pthread\_cond\_init 和 pthread\_cond\_destroy 函数用来完成条件变量的创建和销毁； 提供 pthread\_cond\_wait 和 pthread\_cond\_timewait 函数用来完成线程等待或限时等待在某个条件量上，函数中的参数为条件变量和互斥量，以上函数将利用互斥量来完成对条件变量

状态的修改，以保证多线程状态下条件变量一致性，因此在调用这两个函数之前，一定要获 得这个互斥量的资源，即在此函数调用前一定要有互斥量的加锁操作；提供pthread\_cond\_signal 和 pthread\_cond\_broadcast 函数用来唤醒等待在条件变量上的一个线

程或所有线程。

###### Web 服务器的多线程模型

与 Web 服务器的多进程模型类似，在主线程接收到客户端连接请求信号后（accept 函数返回与客户端的连接），通过 pthread\_create 函数来创建一个线程来处理这个客户端的请求信号。Web 服务器将为每个连接客户端创建一个线程来单独处理这个客户端的请求信息， 如下图 2-3 所示。与 Web 服务器多进程模型不同的是，这里每个线程与主线程共存于 Web 服务器进程空间中，共享 Web 服务器进程资源。



**WebServer**

**accept/pthread\_create**

**1. read msg… 1. read msg… 1. read msg… 2.read file … 2.read file … 2.read file …**

**3. write msg… 3. write msg… 3. write msg…**

**… … …**

**Client A**

**Client ·**

### Client B

图 2-3Web 服务器的多线程模型

下面代码在 nweb 项目基础上，利用 POSIX 线程函数，实现了多线程模型的 Web 服务

器。

//编译代码指令 gcc -std=gnu99 -g -o multithread\_webserver multithread\_webserver.c -lphtread #include <stdio.h>

#include <stdlib.h> #include <unistd.h> #include <errno.h> #include <string.h> #include <fcntl.h> #include <signal.h> #include <sys/types.h> #include <sys/socket.h> #include <netinet/in.h> #include <arpa/inet.h> #include <pthread.h> #include <sys/stat.h>

#define VERSION 23

#define BUFSIZE 8096

#define ERROR 42

#define LOG 44

#define FORBIDDEN 403

#define NOTFOUND 404

#ifndef SIGCLD

# define SIGCLD SIGCHLD #endif

struct {

char \*ext; char \*filetype;

} extensions [] = {

{"gif", "image/gif" },

{"jpg", "image/jpg" },

{"jpeg","image/jpeg"},

{"png", "image/png" },

{"ico", "image/ico" },

{"zip", "image/zip" },

{"gz", "image/gz" },

{"tar", "image/tar" },

{"htm", "text/html" },

{"html","text/html" },

{0,0} };

typedef struct { int hit;

int fd;

} webparam;

unsigned long get\_file\_size(const char \*path)

{

unsigned long filesize = -1; struct stat statbuff; if(stat(path, &statbuff) < 0){

return filesize;

}else{

filesize = statbuff.st\_size;

}

return filesize;

}

void logger(int type, char \*s1, char \*s2, int socket\_fd)

{

int fd ;

char logbuffer[BUFSIZE\*2];

switch (type) {

case ERROR: (void)sprintf(logbuffer,"ERROR: %s:%s Errno=%d exiting pid=%d",s1, s2,

errno,getpid());

break;

case FORBIDDEN:

(void)write(socket\_fd, "HTTP/1.1 403 Forbidden\nContent-Length: 185\nConnection: close\nContent-Type: text/html\n\n<html><head>\n<title>403 Forbidden</title>\n</head><body>\n<h1>Forbidden</h1>\nThe requested URL, file type or operation is not allowed on this simple static file webserver.\n</body></html>\n",271);

(void)sprintf(logbuffer,"FORBIDDEN: %s:%s",s1, s2); break;

case NOTFOUND:

(void)write(socket\_fd, "HTTP/1.1 404 Not Found\nContent-Length: 136\nConnection: close\nContent-Type: text/html\n\n<html><head>\n<title>404 Not Found</title>\n</head><body>\n<h1>Not Found</h1>\nThe requested URL was not found on this server.\n</body></html>\n",224);

(void)sprintf(logbuffer,"NOT FOUND: %s:%s",s1, s2); break;

case LOG: (void)sprintf(logbuffer," INFO: %s:%s:%d",s1, s2,socket\_fd); break;

}

/\* No checks here, nothing can be done with a failure anyway \*/

if((fd = open("nweb.log", O\_CREAT| O\_WRONLY | O\_APPEND,0644)) >= 0) { (void)write(fd,logbuffer,strlen(logbuffer));

(void)write(fd,"\n",1); (void)close(fd);

}

//if(type == ERROR || type == NOTFOUND || type == FORBIDDEN) exit(3);

}

/\* this is a web thread, so we can exit on errors \*/ void \* web(void \* data)

{

int fd; int hit;

int j, file\_fd, buflen; long i, ret, len; char \* fstr;

char buffer[bufsize+1]; /\* static so zero filled \*/ webparam \*param=(webparam\*) data; fd=param->fd;

hit=param->hit;

ret =read(fd,buffer,bufsize); /\* read web request in one go \*/ if(ret == 0 || ret == -1) { /\* read failure stop now \*/

logger(forbidden,"failed to read browser request","",fd);

}else{

if(ret > 0 && ret < bufsize) /\* return code is valid chars \*/ buffer[ret]=0; /\* terminate the buffer \*/

else buffer[0]=0;

for(i=0;i<ret;i++) /\* remove cf and lf characters \*/ if(buffer[i] == '\r' || buffer[i] == '\n')

buffer[i]='\*'; logger(log,"request",buffer,hit);

if( strncmp(buffer,"get ",4) && strncmp(buffer,"get ",4) ) { logger(forbidden,"only simple get operation supported",buffer,fd);

}

for(i=4;i<bufsize;i++) { /\* null terminate after the second space to ignore extra stuff \*/ if(buffer[i] == ' ') { /\* string is "get url " +lots of other stuff \*/

buffer[i] = 0;

break;

}

}

for(j=0;j<i-1;j++) /\* check for illegal parent directory use .. \*/ if(buffer[j] == '.' && buffer[j+1] == '.') {

logger(forbidden,"parent directory (..) path names not supported",buffer,fd);

index file \*/

}

if( !strncmp(&buffer[0],"get /\0",6) || !strncmp(&buffer[0],"get /\0",6) ) /\* convert no filename to

(void)strcpy(buffer,"get /index.html");

/\* work out the file type and check we support it \*/ buflen=strlen(buffer);

fstr = (char \*)0; for(i=0;extensions[i].ext != 0;i++) {

len = strlen(extensions[i].ext);

if( !strncmp(&buffer[buflen-len], extensions[i].ext, len)) { fstr =extensions[i].filetype;

break;

}

}

if(fstr == 0) logger(forbidden,"file extension type not supported",buffer,fd);

if(( file\_fd = open(&buffer[5],o\_rdonly)) == -1) { /\* open the file for reading \*/ logger(notfound, "failed to open file",&buffer[5],fd);

}

logger(log,"send",&buffer[5],hit);

len = (long)lseek(file\_fd, (off\_t)0, seek\_end); /\* 使用 lseek 来获得文件长度，比较低效\*/ (void)lseek(file\_fd, (off\_t)0, seek\_set); /\* 想想还有什么方法来获取\*/

(void)sprintf(buffer,"http/1.1 200 ok\nserver: nweb/%d.0\ncontent-length: %ld\nconnection: close\ncontent-type: %s\n\n", version, len, fstr); /\* header + a blank line \*/

logger(log,"header",buffer,hit); (void)write(fd,buffer,strlen(buffer));

/\* send file in 8kb block - last block may be smaller \*/ while ( (ret = read(file\_fd, buffer, bufsize)) > 0 ) {

(void)write(fd,buffer,ret);

}

usleep(10000);/\*在 socket 通道关闭前，留出一段信息发送的时间\*/

close(file\_fd);

}

close(fd);

//释放内存

free(param);

}

int main(int argc, char \*\*argv)

{

int i, port, pid, listenfd, socketfd, hit; socklen\_t length;

static struct sockaddr\_in cli\_addr; /\* static = initialised to zeros \*/ static struct sockaddr\_in serv\_addr; /\* static = initialised to zeros \*/

if( argc < 3 || argc > 3 || !strcmp(argv[1], "-?") ) {

(void)printf("hint: nweb Port-Number Top-Directory\t\tversion %d\n\n" "\tnweb is a small and very safe mini web server\n"

"\tnweb only servers out file/web pages with extensions named below\n" "\t and only from the named directory or its sub-directories.\n"

"\tThere is no fancy features = safe and secure.\n\n" "\tExample: nweb 8181 /home/nwebdir &\n\n" "\tOnly Supports:", VERSION);

for(i=0;extensions[i].ext != 0;i++) (void)printf(" %s",extensions[i].ext);

(void)printf("\n\tNot Supported: URLs including \"..\", Java, Javascript, CGI\n" "\tNot Supported: directories / /etc /bin /lib /tmp /usr /dev /sbin \n"

"\tNo warranty given or implied\n\tNigel Griffiths nag@uk.ibm.com\n" );

exit(0);

}

if( !strncmp(argv[2],"/" ,2 ) || !strncmp(argv[2],"/etc", 5 ) ||

!strncmp(argv[2],"/bin",5 ) || !strncmp(argv[2],"/lib", 5 ) ||

!strncmp(argv[2],"/tmp",5 ) || !strncmp(argv[2],"/usr", 5 ) ||

!strncmp(argv[2],"/dev",5 ) || !strncmp(argv[2],"/sbin",6) ){ (void)printf("ERROR: Bad top directory %s, see nweb -?\n",argv[2]); exit(3);

}

if(chdir(argv[2]) == -1){

(void)printf("ERROR: Can't Change to directory %s\n",argv[2]); exit(4);

}

/\* Become deamon + unstopable and no zombies children (= no wait()) \*/ if(fork() != 0)

return 0; /\* parent returns OK to shell \*/ (void)signal(SIGCLD, SIG\_IGN); /\* ignore child death \*/ (void)signal(SIGHUP, SIG\_IGN); /\* ignore terminal hangups \*/ for(i=0;i<32;i++)

(void)close(i); /\* close open files \*/ (void)setpgrp(); /\* break away from process group \*/ logger(LOG,"nweb starting",argv[1],getpid());

/\* setup the network socket \*/

if((listenfd = socket(AF\_INET, SOCK\_STREAM,0)) <0) logger(ERROR, "system call","socket",0);

port = atoi(argv[1]); if(port < 0 || port >60000)

logger(ERROR,"Invalid port number (try 1->60000)",argv[1],0);

//初始化线程属性，为分离状态pthread\_attr\_t attr; pthread\_attr\_init(&attr);

pthread\_attr\_setdetachstate(&attr,PTHREAD\_CREATE\_DETACHED);

//

pthread\_t pth; serv\_addr.sin\_family = AF\_INET;

serv\_addr.sin\_addr.s\_addr = htonl(INADDR\_ANY); serv\_addr.sin\_port = htons(port);

if(bind(listenfd, (struct sockaddr \*)&serv\_addr,sizeof(serv\_addr)) <0) logger(ERROR,"system call","bind",0);

if( listen(listenfd,64) <0) logger(ERROR,"system call","listen",0);

for(hit=1; ;hit++) {

length = sizeof(cli\_addr);

if((socketfd = accept(listenfd, (struct sockaddr \*)&cli\_addr, &length)) < 0) logger(ERROR,"system call","accept",0);

webparam \*param=malloc(sizeof(webparam)); param->hit=hit;

param->fd=socketfd;

if(pthread\_create(&pth, &attr, &web, (void\*)param)<0){ logger(ERROR,"system call","pthread\_create",0);

}

}

}

###### Web 服务器的多线程模型

**题目 1.** 将上述多线程 Web 服务器与实验 2 中多进程模型的 Web 服务器性能进行对比，说明它俩各具有什么优缺点。具体对比的指标包括如下：

* 使⽤ http\_load 命令来测试这两个模型下 Web 服务器的性能指标。并根据这些测

试指标对⽐，分析为什么这两种模型会产⽣不同的性能结果？

* 对这两个模型中的 socket 数据读取、发送、⽹页⽂件读取和⽇志⽂件写⼊四个 I/O 操作分别计时，并打印出每个进程或线程处理各项 I/O 计时的平均时间。例如， 编写的程序应该打印出如下结果。

共用 10000ms 成功处理 100 个客户端请求，其中

平均每个客户端完成请求处理时间为 5100ms。平均每个客户端完成读 socket 时间为 500ms。平均每个客户端完成写 socket 时间为 1000ms。平均每个客户端完成读网页数据时间为 110ms。平均每个客户端完成写日志数据时间为 50ms。

* 根据上⾯的计时数据结果，分析并说明哪些多进程模型和多线程模型中哪些 I/O 操作是最消耗时间的？。
* 思考⼀下，怎么修改线程模型，才能提⾼线程的并发性能？

**题目 2.** 调整 http\_load 命令参数，增加其并发访问线程数量，会发现随着并发访问数

量在达到一定数量后，再增多会导致多线程 Web 服务进程的性能出现下降的现象。试分析产生上述现象的原因是什么？

## 线程池模型

与进程相比，虽然线程创建、销毁的代价较小，但还是需要系统内核为其分配运行堆栈。 由于在多线程模型中每个线程完成的业务逻辑基本一样，因此如果线程在完成一次客户端 请求处理后并不退出，而是等待运行客户端的下一次请求，那么将节省线程的创建、销毁所 耗费的时间。

同时如果有大量客户端同时请求 Web 服务器时，将造成 Web 服务器同时创建大量的

线程，这些线程将相互竞争 CPU 资源、I/O、进程内临界资源等计算机资源，从而导致 Web 服务进程并发吞吐量降低。对于 CPU 利用率来说，如果线程数量增多，则因为线程的读写I/O 阻塞而导致的线程上下文切换次数增多，则 CPU 的利用率会下降。对于外存 I/O，如果多个线程竞争对外存的读写权，由于外存存储数据的特性以及 I/O 传输数据带宽限制，会导

致大多数线程存在阻塞状态。如果多个线程之间存在临界资源、数据同步等使用问题，随着 线程数量的增多，也会使得线程的并发性下降。总之，通过以上分析，会发现 Web 服务进程的并发、吞吐性能并不是随着线程数量的增多而增强。一般情况下，在初始状态，Web 服

务进程的性能随线程数量增多而增强，但是线程到达一定数量后，其性能会随着线程进一步 增多而下降。

基于以上分析，会得到两个结论：1）进程中并非线程数量越多，I/O 处理能力越强；2）

在 Web 服务器中，每个线程处理的业务逻辑是相同的，而每个线程的创建、销毁都要销毁时间。

根据以上两点，如果设计一种结构，能够在初始化时就创建一定数量的线程，并且这些 线程在处理完任务后并不退出，而是等待下一次任务的到来。这样就能够同时克服以上两个 问题。目前实行这种思想的结构被称为线程池。

在设计和实现线程池时，需要考虑两个问题：1）要完成的任务如何进行封装，才能让

已经创建的线程来运行它？；2）已经创建的线程如何能够知道任务的到来，并且在运行完 后并不销毁？

要解决第一个问题，首先考虑 pthread\_create 函数中的参数 void\* (start\_routine)(void\*) 和 void\* arg 分别为要运行任务逻辑的函数指针和这个函数的参数指针。此函数在创建线程堆栈等运行体后，一定通过执行“start\_routine(arg)”代码来执行业务代码（即封装运行任 务逻辑的函数）。虽然在线程池中，表示运行任务逻辑的这两个参数不能通过创建线程来进

行传递，但是在线程执行函数内，如果能够得到这两个参数，那么可以通过执行“start\_routine(arg)”代码来完成。因此，运行任务中如果封装了这两个参数，则就能够让 线程运行它。

要解决第二问题，首先分析一下线程池中线程运行状态和场景。既然在初始化线程池时， 需要创建一定数量的线程，那么这些线程的运行函数在创建后一定要阻塞，直到有信号通知 它们，它们才能够执行任务。在执行任务完成后，线程的运行函数还要阻塞，等待下一次信 号到来。

根据上面分析，可以描绘出一幅线程池运行状态图，如图 2-4 所示。其中，task queue

为存放任务的队列；thread array 为线程数组。在初始化时，由于任务队列中没有任务，所有的线程全部阻塞在条件变量上。当向任务队列里面添加任务时，可以恢复阻塞在条件变量

上的线程运行。线程运行时从任务队列头部里面取一个任务，然后执行这个任务，当执行任 务结束后，会判断目前是否还有任务在任务队列，如果有则继续执行前面的步骤；如果没有 则阻塞在这个条件变量。

##### thread array



**running**

**blocked**

**semaphore/**

**conditional variable**

**taskn**

**…**

**…**

**…**

**task2**

**task1**

**Add task**

**task queue**

图 2-4 线程池运行状态图

通过整理上面线程运行描述，可以得到以下的线程之间同步/互斥操作。

* 如果任务队列⾥面有任务，则线程不会阻塞；没有任务线程才会阻塞。
* 线程从任务队列消费任务，增加任务函数（Add task）向任务队列生成任务。因此任务队列是生产者和消费者问题中的临界资源。
* 线程池在销毁时，需要等待线程池内所有的线程运行完毕，才能释放线程池所占 资源。

下面为线程池的数据结构和线程池接口函数。其中在线程池接口函数中，使用“…”，表示忽略的代码，但是这些忽略代码的逻辑在相关位置都有描述。除此之外，代码中并没有 给出任务队列操作的相关函数， push\_taskqueue ， take\_taskqueue ， init\_taskqueue 和

destory\_taskqueue。

/\* queue status and conditional variable\*/ typedef struct staconv {

pthread\_mutex\_t mutex;

pthread\_cond\_t cond; /\*用于阻塞和唤醒线程池中线程\*/

int status; /\*表示任务队列状态：false 为无任务；true 为有任务\*/

} staconv;

/\*Task\*/

typedef struct task{ struct task\* next;

void (\*function)(void\* arg); void\* arg;

} task;

/\* 指向下一任务 \*/

/\* 函数指针

\*/

/\* 函数参数指针 \*/

/\*Task Queue\*/

typedef struct taskqueue{ pthread\_mutex\_t mutex; task \*front;

task \*rear; staconv \*has\_jobs; int len;

/\* 用于互斥读写任务队列 \*/

/\* 指向队首

/\* 指向队尾

\*/

\*/

/\* 根据状态，阻塞线程 \*/

/\* 队列中任务个数

\*/

} taskqueue;

/\* Thread \*/

typedef struct thread{

int id; /\* 线 程 id \*/ pthread\_t pthread; /\* 封装的 POSIX 线程 \*/ struct threadpool\* pool; /\* 与线程池绑定 \*/

} thread;

/\*Thread Pool\*/

typedef struct threadpool{

thread\*\* threads; /\* 线程指针数组 \*/ volatile int num\_threads; /\* 线程池中线程数量 \*/ volatile int num\_working; /\* 目前正在工作的线程个数 \*/

pthread\_mutex\_t thcount\_lock; /\* 线程池锁用于修改上面两个变量 \*/ pthread\_cond\_t threads\_all\_idle; /\* 用于销毁线程的条件变量 \*/ taskqueue queue; /\* 任务队列 \*/

volatile bool is\_alive; /\* 表示线程池是否还存活 \*/

}threadpool;

/\*线程池初始化函数\*/

struct threadpool\* initTheadPool(int num\_threads){

//创建线程池空间threadpool\* pool;

pool=(threadpool\*)malloc(sizeof(struct threadpool)); pool->num\_threads=0;

pool->num\_working=0;

//初始化互斥量和条件变量pthread\_mutex\_init(&(thpool\_p->thcount\_lock), NULL);

pthread\_cond\_init(&thpool\_p->threads\_all\_idle, NULL);

//初始化任务队列

//\*\*\*\* 需 实 现 \*\*\*\*\* init\_taskqueue(&pool->queue);

//创建线程数组

pool->threads=(struct thread \*\*)malloc(num\_threads\*sizeof(struct\* thread));

//创建线程

for (int i = 0; i < num\_threads; ++i)

{

create\_thread(pool,pool->thread[i],i); //i 为 线 程 id,

}

//等等所有的线程创建完毕,在每个线程运行函数中将进行 pool->num\_threads++ 操作

//因此，此处为忙等待，直到所有的线程创建完毕，并马上运行阻塞代码时才返回。while(pool->num\_threads!=num\_threads) {}

return pool;

}

/\*向线程池中添加任务\*/

void addTask2ThreadPool(threadpool\* pool,task\* curtask){

//将任务加入队列

//\*\*\*\* 需 实 现 \*\*\*\*\* push\_taskqueue(&pool->queue,curtask);

}

/\*等待当前任务全部运行完\*/

void waitThreadPool(threadpool\* pool){ pthread\_mutex\_lock(&pool->thcount\_lock);

while (pool->jobqueue.len || pool->num\_threads\_working) { pthread\_cond\_wait(&pool->threads\_all\_idle, &pool->thcount\_lock);

}

pthread\_mutex\_unlock(&thpool\_p->thcount\_lock);

}

/\*销毁线程池\*/

void destoryThreadPool(threadpool\* pool){

//如果当前任务队列中有任务，需等待任务队列为空，并且运行线程执行完任务后

....

....

....

//销毁任务队列

//\*\*\*\* 需 实 现 \*\*\*\*\* destory\_taskqueue(&pool->queue);

//销毁线程指针数组,并释放所有为线程池分配的内存

....

....

....

}

/\*获得当前线程池中正在运行线程的数量\*/

int getNumofThreadWorking(threadpool\* pool){ return pool->num\_working;

}

/\*创建线程\*/

int create\_thread（struct threadpool\* pool, struct thread\*\* pthread, int id){

//为 thread 分配内存空间

\*pthread = (struct thread\*)malloc(sizeof(struct thread)); if (pthread == NULL){

error("creat\_thread(): Could not allocate memory for thread\n"); return -1;

}

//设置这个 thread 的属性(\*pthread)->pool = pool; (\*pthread)->id = id;

//创建线程

pthread\_create(&(\*pthread)->pthread, NULL, (void \*)thread\_do, (\*pthread)); pthread\_detach((\*pthread)->pthread);

return 0;

}

/\*线程运行的逻辑函数\*/

void\* thread\_do(struct thread\* pthread){

/\* 设置线程名字 \*/

char thread\_name[128] = {0};

sprintf(thread\_name, "thread-pool-%d", thread\_p->id);

prctl(PR\_SET\_NAME, thread\_name);

/\* 获得线程池\*/

threadpool\* pool = pthread->pool;

/\* 在线程池初始化时，用于已经创建线程的计数，执行 pool->num\_threads++ \*/

....

....

....

/\*线程一直循环往复运行，直到 pool->is\_alive 变为 false\*/ while(pool->is\_alive){

/\*如果任务队列中还要任务，则继续运行，否则阻塞\*/

....

....

....

if (pool->is\_alive){

/\*执行到此位置，表明线程在工作，需要对工作线程数量进行计数\*/

//pool->num\_working++

....

....

....

/\* 从任务队列的队首提取任务，并执行\*/ void (\*func)(void\*);

void\* arg;

//take\_taskqueue 从任务队列头部提取任务，并在队列中删除此任务

//\*\*\*\*需实现 take\_taskqueue\*\*\*\*\*

task\* curtask = take\_taskqueue(&pool->queue); if (curtask) {

func = curtask->function; arg = curtask->arg;

//执行任务

func(arg);

//释放任务free(curtask);

}

/\*执行到此位置，表明线程已经将任务执行完成，需更改工作线程数量\*/

//此处还需注意，当工作线程数量为 0，表示任务全部完成，要让阻塞在 waitThreadPool 函数上的线程继续运行

....

....

....

}

}

/\*运行到此位置表明，线程将要退出，需更改当前线程池中的线程数量\*/

//pool->num\_threads--

....

....

....

return NULL;

}

**2.4.1 实验 4 Web 服务器的线程池模型**

**题目 1.** 添补相应的程序代码到上面函数中“……”位置处。

**题目2.** 完成函数push\_taskqueue，take\_taskqueue，init\_taskqueue 和destory\_taskqueue。

**题目 3.** 添加必要的程序代码，以最终完成线程池。

**题目 4.** 利用实现的线程池，替换实验 3 中 Web 服务的多线程模型。

**题目 5.** 调整线程池中线程个数参数，以达到 Web 服务并发性能最优。利用 http\_load

及其它性能参数，分析和对比多线程模型与线程池模型在 Web 服务进程中的优点和缺点。

## 业务分割模型

在多进程和多线程模型中每个进程或线程完成相同的任务（由于多进程与多线程模型 在本节的描述中具有相同的含义，下面主要使用多线程来进行问题描述）。针对客户端的文 件请求处理，每个任务包含五个步骤：网络读取数据、解析数据、读取文件、向网络发送数 据和写日志文件。

这些步骤主要涉及两类 I/O 设备：外部存储器和网络。在多线程模型中，每个线程都要

竞争使用这两类 I/O 设备，并且这两类设备 I/O 速度慢，从而导致在一个线程执行任务过程

中大量时间阻塞在这两类设备上。同时，每个设备都是在处理完一个线程的 I/O 数据请求后， 由操作系统恢复另一个阻塞在此设备上的线程来使用此设备。从设备的角度上看，该设备也

并没有一直在处理数据，而是处理完一块数据后，需要等待操作系统调度另外线程后，才能 再处理数据。具体设备工作时间如下图 2-5 所示，其中 W 表示设备在工作，I 表示设备在等待。

**I**

**W**

**I**

**W**

**I**

**W**

**I**

**W**

图 2-5 设备工作时间状态图

除了设备的等待时间外，由于每个线程中与 I/O 设备相关的数据区在内存独立，从而造成 I/O 设备操作数据的离散化和碎片化，尤其是每次读写的数据内容过少，会严重影响 I/O 设备性能。

例如，可以设计以下实验来进行验证：一次性从文件中读取 1MB 数据所消耗时间；分1000 次，每次从文件读取 1KB 数据，读 1MB 数据所消耗的时间；分 1000000 次，每次从文件读 1 字节数据，读 1MB 数据所消耗的时间。具体实验代码如下。

// 编译指令 gcc -std=gnu99 -g -o readfiletimedemo readfiletimedemo.c

#include <stdio.h> #include <unistd.h> #include <errno.h> #include <sys/types.h> #include <sys/stat.h> #include <sys/time.h> #include <fcntl.h>

#define BUFSIZE 1024\*1024 #define KB 1024

int main(int argc, char const \*argv[])

{

int fd ;

char buffer[BUFSIZE];

struct timeval start; struct timeval end; unsigned long timer;

if((fd = open("nweb.log", O\_CREAT| O\_RDONLY,0644)) >= 0) { gettimeofday(&start,NULL);

read(fd,buffer,BUFSIZE);

gettimeofday(&end,NULL);

timer

start.tv\_usec;

=

1000000 \* (end.tv\_sec-start.tv\_sec)+ end.tv\_usec-

printf("read all 1MB date timer = %ld us\n",timer);

gettimeofday(&start,NULL); for (int i = 0; i < 1024; ++i)

{

read(fd,buffer,KB);

}

gettimeofday(&end,NULL);

timer

start.tv\_usec;

= 1000000

\* (end.tv\_sec-start.tv\_sec)+ end.tv\_usec-

printf("read 1024 times, each 1KB, total 1MB date timer = %ld us\n",timer);

gettimeofday(&start,NULL);

for (int i = 0; i < BUFSIZE; ++i)

{

read(fd,buffer,1);

}

gettimeofday(&end,NULL);

timer

start.tv\_usec;

= 1000000 \*

(end.tv\_sec-start.tv\_sec)+ end.tv\_usec-

printf("read 1024\*1024 times, each 1B, total 1MB date timer = %ld us\n",timer);

close(fd);

}else{

printf("cannot open the file\n");

}

return 0;

}

这个程序执⾏的结果如下所⽰。

read all 1MB date timer = 1043 us

read 1024 times, each 1KB, total 1MB date timer = 1523 us

read 1024\*1024 times, each 1B, total 1MB date timer = 1081759 us

通过上面分析，可以通过以下两个方面来提高 I/O 设备利用率和读写速度。

* 尽量减少多个线程同时互斥使用设备的情况。
* 在每次操作 I/O 设备时，尽量向 I/O 设备读取或写入更多的数据。

为达到这两方面目标，可以考虑将原来任务按逻辑步骤进行分割，每个逻辑步骤为一个 线程，步骤与步骤之间通过缓冲区进行数据传递，这就是业务分割模型，如下图 2-6 所示。



**WebServer**

**Client A**

**Client B**

**read msg threadpool**

**…**

**Client**

**send msg read file**

**l**

**msg queue**

**threadpoo**

**~~threadpool~~**

**socketfdn**

**socketfd1**

**msgn**

**msg1**

**filename queue**

**socketfd1**

**filenamen**

**socketfdn**

**…**

**filename1**

**accept**

图 2-6 Web 服务器的业务分割模型

在 WebSever 服务进程中，按照业务步骤分别创建三个线程池（readmsg threadpool， readfile threadpool 和sendmsg threadpool）和两个消息队列（filename queue 和msg queue）。其中，readmsg threadpool 中的线程主要完成从客户端socket 通道中读取消息并进行解析，然后将请求的文件名和 socket 通道加入到 filename queue；readfile threadpool 中的线程从filename queue 中提取文件名，读取文件，并将文件内容和 socket 通道发送到 msg queue； sendmsg threadpool 中的线程从 msg queue 中提取文件内容和 socket 通道，并将文件内容通过 socket 通道发送到客户端。

在上述模型中，可以将 filename queue 作为 readfile pool 中的任务队列，将 msg queue 作为 sendmsg threadpool 中的任务队列；也可以将这两个消息队列作为单独的队列，然后通过增加消息队列上的线程，完成消息的读取和加入到下一个业务步骤线程池中。

建立上述的 WebServer 业务分割模型后，可以简单分析一下，其与多线程模型的不同。首先，在多线程模型中，每个步骤之间有严格的时序关系；而在业务分割模型中，由于每个

步骤之间并没有直接的联系，而是通过消息队列而发生间接联系，从而达到步骤之间的解耦。 并且通过消息队列作为数据缓冲区，可以减少和避免因为业务处理速度不一致而导致的性 能下降现象。例如，如果消息队列长度仅为 1，并且在某个时间段，从 readmsg threadpool

中线程接收消息速度大于 readfile pool 中线程读取文件的速度，那么 readmsg threadpool

中的线程将会频繁阻塞于 filename queue 队列，等待队列有空余空间；而如果消息队列足够长，则在一定程度上可以避免此现象发生，因为这个时间段内 readmsg threadpool 中线程处理的所有数据都可以缓存在这个队列上。

其次，通过分割任务，使得每个线程执行的业务单元粒度变小。对于 I/O 密集型业务单元，可以通过极少量的线程就能够达到充分利用 I/O 设备的效果。例如，在上图的 readfile pool 中如果仅有一个线程，其几乎可以不断地读取磁盘中每个请求文件的内容，因为与 I/O

设备传递数据相比，与消息队列之间传递极为快速（内存数据传输速度为外存数据传输速度 的几倍到数百倍，一般内存速度为 SSD 速度的几倍~几十倍，为磁盘的几十倍~数百倍）。即使考虑到 I/O 设备的并行性，也仅需几个线程就可以充分利用 I/O 设备。

最后，通过分割任务，每个线程执行的业务逻辑变得简单，这样更有利于分析、调试和

优化多任务环境的程序性能。

###### 实验 5 Web 服务器的业务分割模型

**题目 1.** 实现上述的业务分割模型的 Web 服务程序。

**题目 2.** 在程序里面设置性能监测代码，通过定时打印这些性能参数，能够分析此 Web

服务程序的运行状态。例如，线程池中线程平均活跃时间及其阻塞时间，线程最高活跃数量、 最低活跃数量、平均活跃数量；消息队列中消息的长度等。除此以外还可以利用相关系统命 令来监测系统的 I/O、内存、CPU 等设备性能。

**题目 3.** 通过上述的性能参数和系统命令，对 Web 服务程序进行逻辑分析，发现当前程

序存在性能瓶颈的原因。进而通过控制各个线程池中的线程数量和消息队列长度，来改善此 程序的性能。

## 混合模型

前面讲解的模型可以分为进程模型和线程模型两大类。进程模型主要依赖创建子进程 来处理任务；线程模型主要依赖线程来处理任务。这两个模型各有优缺点。从性能角度考虑， 多线程模型占据优势。这是因为，与进程相比，线程的创建、销毁和维护所需系统资源较少， 并且线程之间数据共享方便。从安全角度考虑，多进程模型占据优势。这是因为，与线程相 比，进程独立占有系统资源（内存、I/O 设备），一个进程运行崩溃不会或很少干扰另一个进

程的运行。

如果综合考虑上述两种模型的特点，可以将多进程模型与多线程模型进行融合。其主要 实现思想是在多进程模型中，每个进程运行环境内建立多线程模型。这样做的好处就是在为Web 服务器带来安全的同时，也使得其具有较高的并发性。

实际上，这种将多进程与多线程进行混合的模型也出现在日常使用的程序内部。例如，

如下图 2-7 和图 2-8 所示，chrome 浏览器会为每个标签页面建立一个进程，当在这个标签页面输入 URL 地址来进行文件请求时，chrome 浏览器会在这个进程内默认启动三个线程来并发获取其渲染页面所需要的资源。这样做除了能提高页面的渲染显示速度外，还具有很

好的安全性，使得标签页面之间的运行并不相互干扰。当一个标签页面所对应的进程因为某 种原因崩溃，并不会影响到其它标签页面的工作。

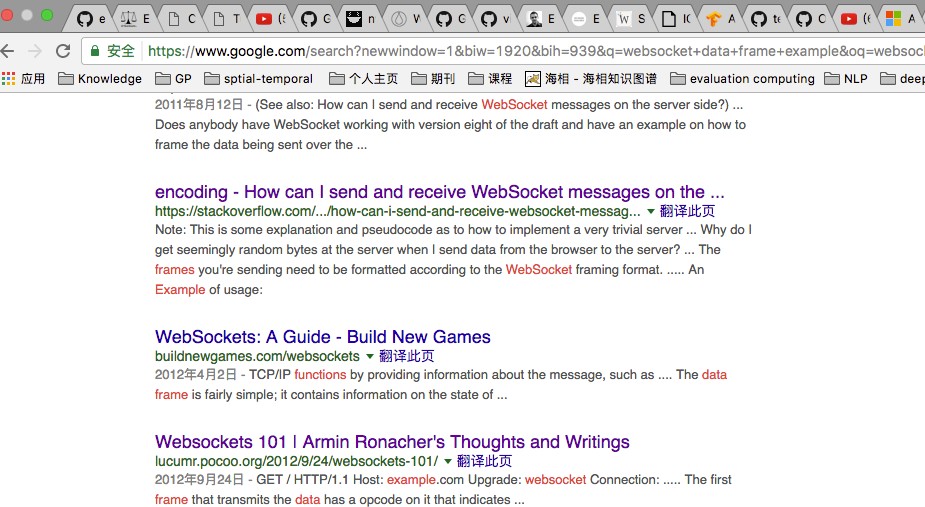


图 2-7 Chrome 浏览器多标签运行界面

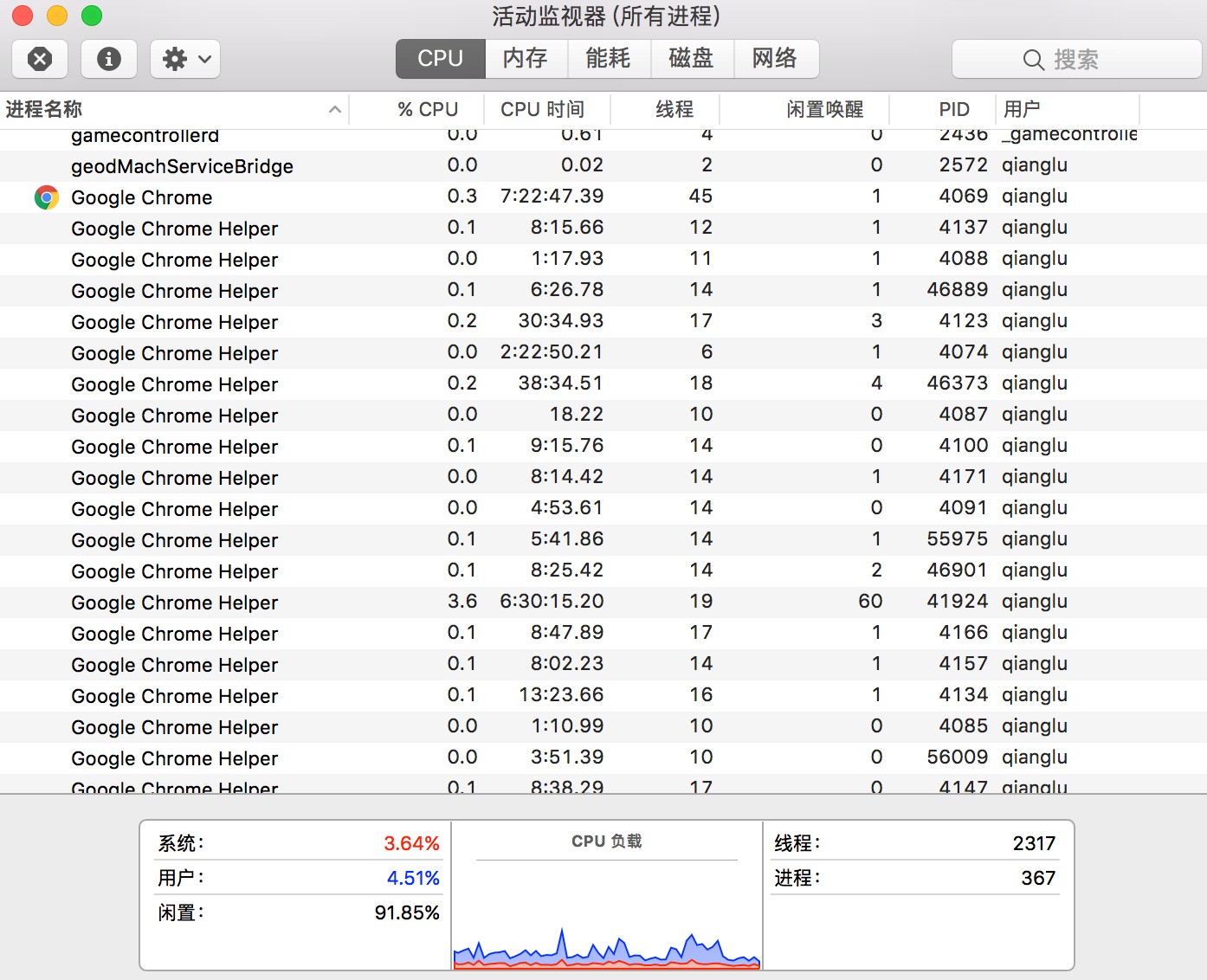


图 2-8 Chrome 浏览器启动的进程视图

与 chrome 浏览器中简单的多进程多线程混合模型相比，Web 服务器混合模型的设计会更加复杂。它需要综合考虑性能和安全前提下，动态调整此模型下进程和线程的数量。其 主要涉及以下问题：

1. 在每个进程中，是使用多线程模型还是线程池模型？
2. 在客户端并发请求数量增多后，是新建一个子进程及其多线程模型来处理新增请求， 还是在原有的进程内部增加线程数量？如果在原有的进程内部增加线程数量，应该在哪个 进程中增加线程？
3. 在客户端并发请求数量减少后，是减少原来进程中的线程数量？还是关闭进程？ 这些问题并没有统一标准的答案，需要设计人员根据业务逻辑规则以及系统运行环境

来进行权衡。设计人员在设计系统时，不能忽略的一个因素就是让系统逻辑和结构尽可能地 保持简单（chrome 浏览器中多进程多线程使用理念）。因为复杂系统会带来设计、开发和维护的难度增加，会导致系统 bug 增多、系统容易崩溃和不安全。

在下图 2-9 中，给出了一个混合模型的 Web 服务器逻辑架构设计方案，以供读者以后

进行相关服务设计时参考。此系统架构可以很容地扩展为分布式系统。



**WebServer**

**accept**

**Client A**

**fd**

**…**

**Client B**

**MGR**

**HB**

**MGR**

**HB**

**…**

**Client ·**

**TP TP**

**Child Process 1 Child Process… Child Process n**

**…**

**fd**

**socket fd queue**

**…**

**…**

**monitor**

**… …**

**statuts queue**

**manager**

**…**

图 2-9 Web 服务器的混合模型

在 WebServer 主进程中包含三个线程，其中 accept 线程主要用于针对服务端口，当建立好客户连接通道后，就把这个通道描述符发送到内存共享队列 socketfd queue 中；

manager 线程负责维护子进程，其根据状态队列（status queue）里面每个子进程运行状态、socketfd queue 的长度和当前系统性能等参数来决定是否创建或关闭任务子进程；monitor 线程为系统信息收集线程，其接收子进程心跳线程发送的心跳信号以及子进程运行状态信 息，并将这些信息保存在 status queue 中。

在每个子进程中包含三个运行组件：MGR、HB 和 TP。其中，MGR 为子进程的管理线

程，其负责从 socketfd queue 中提取客户端连接通道，然后根据本进程的执行状态，来决定是启动新的任务线程还是利用已经创建好的任务线程来处理此通道的信息；HB 为心跳线程

（heartbeat），其定时将本进程的状态信息发送到主进程的 monitor 线程侦听端口；TP 为任务线程池或任务线程，其内部包含要执行具体业务逻辑的线程，此部分也可以进一步分解为

业务分割模型。除此以外，子进程还要包含信号处理函数，以响应主进程发送过来的管理信 号。例如，主进程中 manager 线程通过检测发现目前并发请求的客户端少，系统中存在大量子进程，并且这些子进程中的任务线程未工作，这时可以向这些子进程发送关闭信号以关

闭子进程。而这些子进程中关闭信号处理函数，将执行子进程退出时的处理工作，如等待执 行完成、释放内存等。

###### 实验 6 Web 服务器的混合模型

**题目 1.** 参考本节给出的基于混合模型的 Web 服务器，请尝试设计并实现它。在考虑心跳信息基础上，请仔细设计 manager 内部的子进程运行及控制的调度算法，使得系统具有优良的自适应能力。

**题目 2.** 考虑一下，是否能将本混合模型进行扩展分布式模型，即 Web 服务的主进程部署在一台主机，能够将客户端请求信号转发到后置系统，并具有负载平衡能力；而每个子进 程部署在独立的主机上，作为后置系统来处理客户端的请求。这时，Web 服务的主进程被称为反代理服务器；而后置系统的子进程被称为 Web 服务器。

# 第 3 章 Web 服务器的内存管理

## 背景介绍

Web 服务器不断地接收客户端对 Web 相关文件的请求，这些文件随着用户关注的不同，被请求的频率也不尽相同。有些文件会被用户在短时间内大量请求，有些文件可能在很

长一段时间内都不会有用户需要。例如，对于百度、天猫和京东等网站首页，平时都会有大 量用户在同时访问，而对于天猫或京东某个商品的页面可能在一段时间内都不会有用户访 问。

通过前面的实验，已经了解到读写文件是比较耗时的。如果能够将平时用户大量访问的 文件缓存在内存中，那么在用户请求这些文件时，Web 服务器将只在内存中进行查询，并将结果返回给用户，这将极大地缩短用户得到 Web 服务器响应消息的时间，同时也能提高

Web 服务器的并发性能。例如，在下图 3-1 中，Client A 首先发送了一个 URL 请求，这个

URL 文件内容被 Web 服务器从文件系统中读出来，并缓存在内存，然后将其发送回 Client A。然后，Clinet B 也发送了相同的 URL 请求，这是 Web 服务器将从内存中找到此请求内容，并将其返回给 Client B。与外存相比，内存读取速度是外存的几十倍到数百倍，因此 Client B 会在极短的时间内得到 Web 服务器的响应消息。



[**http://example.com/index.htm**](http://example.com/index.htm)

**HTTP TCP/IP**

**Client A**

**Internet**

**HTTP TCP/IP**

**index.html File System**

**HTTP TCP/IP**

**Web Server**

[**http://example.com/index.htm**](http://example.com/index.htm)

**http://…**

**http://…**

**Client B**

图 3-1 Web 服务器处理多用户请求相同文件

大型网站服务器商会通过浏览器 Cache、Content Delivery Networks(CDN)、反向代理服务器等手段来进行 Web 相关文件的缓冲，以达到尽可能提高响应用户请求文件速度和缓解中心服务器压力的目的。这些技术手段的基本实现原理都是对 Web 相关文件进行缓冲， 即把文件内容放在内存。

将 Web 页面相关文件缓存到内存中，能够提高客户端得到页面文件响应消息的速度。但是将这些文件放入内存，面临两个问题：

1. 如何设计缓存结构，以使得在用户查询时，能够最快的检索到其所需页面（见 3.2

节）？

1. 由于内存有限，不可能把所有的文件全部缓存到内存中，该缓存什么样的页面文件 到内存中（见 3.3 节）？

在回答上述问题中还将引起更深一层对内存管理的思考：从操作系统中申请到的内存 该如何进行管理，才能够更加高效地完成 Web 服务器中相关内存操作？ （见 3.4 节）

本章将探讨缓存的设计和组织，以及Linux 中的管理内存方法，来回答上述提出的问题，

以实现在内存中对 Web 文件的高效管理和快速检索。

* 1. Web **页面的缓存逻辑结构**

对于第一个问题（也就是根据客户端提交的请求文件名，在内存中查找这个文件是否存 在）可以使用 hash table 来进行处理，其中文件名作为 key，缓存文件内容的内存块作为content，具体结构见下图 3-2。

**hash table**



**malloc**

**malloc**

**memory**

**exa.html**

**index.html**

**key content**

图 3-2 Web 文件缓存的逻辑结构

在 hash table 中，利用 malloc 函数在内存中分配空间以缓存 Web 文件内容。content 的数据结构可以按如下代码进行定义。

typedef struct content{

void\* address; //内容起始地址int length; //内容长度

} content

当有用户请求文件时，可以根据此 hash table 来判断文件是否在此 hash table 内，如果没有则将文件读入内存，并把此文件名和 content 作为 hash table 中的一项进行存入，并返回此 content；如果已经存在，则返回此 content。

根据上述描述，创建的 hash table 代码如下所示，其中在 main 函数中包括了如何使用hashtable 的测试代码。此例子中的 hashtable 支持多线程操作，特别要注意代码是通过自旋锁的方式来达到互斥地修改 hashtable 表中的项。而这个自旋锁通过 gcc 的内置函数“ sync\_lock\_test\_and\_set” 和 “ sync\_synchronize()”来实现。

// gcc -std=gnu99 -DHASHTHREAD -o hashtable hashtable.c -lpthread

// 此代码在 Jonathan Watmough 创建的 hashtable 代码（https://github.com/watmough/jwHash）基础上进行了修改

#include <stdio.h> #include <stdlib.h> #include <string.h> #include <sys/time.h>

#ifdef HASHTHREADED

#include <pthread.h> #include <semaphore.h>

#endif

typedef struct content{

int length; // 内 容 长 度 void\* address; //内容起始地址

} content;

typedef struct hashpair{

char\* key; // key 值为文件名

content \* cont; //内容项

struct hashpair\* next; //在 hash 桶中，指向下一个 hashpair

} hashpair;

typedef struct hashtable{ hashpair \*\* bucket; int num\_bucket;

#ifdef HASHTHREAD

volatile int \* locks; //对 hash 桶进行加锁

// volatile int lock; //对 hashtable 进行加锁

#endif

} hashtable;

// 字符串的 hash 编码算法-djb2

// <http://www.cse.yorku.ca/~oz/hash.html> static inline long int hashString(char \* str)

{

unsigned long hash = 5381; int c;

while (c = \*str++)

hash = ((hash << 5) + hash) + c; /\* hash \* 33 + c \*/ return hash;

}

static inline char \* copystring(char \* value)

{

char \* copy = (char \*)malloc(strlen(value)+1); if(!copy) {

printf("Unable to allocate string value %s\n",value); abort();

}

strcpy(copy,value); return copy;

}

//判断两个 content 是否相同，相同返回 1，不同返回 0

static inline int isEqualContent(content\* cont1, content\* cont2){ if(cont1->length!=cont2->length)

return 0;

if(cont1->address != cont2->address) return 0;

return 1;

}

//创建一个 hashtable

hashtable\* createHashTable(int num\_bucket){

//创建一个 hashtable

hashtable\* table=(hashtable \*) malloc(sizeof(hashtable)); if(NULL==table){

return NULL;

}

//根据 num\_bucket,创建 hash 桶指针

table->bucket=(hashpair\*\*) malloc(num\_bucket\*sizeof(void\*)); if(!table->bucket){

free(table); return NULL;

}

memset(table->bucket,0,num\_bucket\*sizeof(void\*)); table->num\_bucket=num\_bucket;

//初始化锁信号

#ifdef HASHTHREAD

table->locks = (int \*)malloc(num\_bucket \* sizeof(int)); if( !table->locks ) {

free(table); return NULL;

}

memset((int \*)&table->locks[0],0,num\_bucket\*sizeof(int)); #endif

return table;

}

//释放 hashtable 中的资源

void freeHashTable(hashtable\* table){ if(table==NULL)

return; hashpair\* next;

for (int i=0; i< table->num\_bucket; i++) {

//逐个桶释放

hashpair\* pair=table->bucket[i]; while(pair){

next=pair->next;

//删除 pair,释放资源free(pair->key);

free(pair->cont->address); free(pair->cont);

free(pair); pair=next;

}

}

//

free(table->bucket); #ifdef HASHTHREAD

free(table->locks); #endif

free(table);

}

//向 hashtable 中增加一个 Item=<key,content>

//返回 1， 表示要添加项已经在 hash 表中存在,

//返回 0，如果仅是 key 相同，而 content 不同

//返回 2， 如果不存在 key，并且正常加入到 hashtable 中

int addItem(hashtable\* table,char\* key, content\* cont){

//根据 hash 值来计算 key 在 hash 表中的位置

int hash=hashString(key)% table->num\_bucket;

//检索此项的 key 是否已经存在,如果已经存在就在 hash 桶中寻找此项值，并进行替换

hashpair\* pair=table->bucket[hash];

#ifdef HASHTHREAD

//利用 GCC 中函数，加自旋锁

while ( sync\_lock\_test\_and\_set(&table->locks[hash], 1)) {

//gcc 内部函数, 原子操作，将 table->locks[hash]中的值设置为 1，并返回原来的数值

//当第一次进入时，返回 0，而同时第二次进入则为 1，因此后面进入的线程获得此值均为 1， 导致在此处忙等待

//do nothing

}

#endif

while(pair!=0){

if(0==strcmp(pair->key,key) && isEqualContent(pair->cont, cont)) //已经存在return 1;

if(0==strcmp(pair->key,key) && !isEqualContent(pair->cont, cont)) {

//仅是 key 相同，需要进行 content 替换

free(pair->cont->address); free(pair->cont);

pair->cont=cont; return 0;

}

pair=pair->next;

}

//否则在 hashtable 中不存在，在 hashtable 中新建一个项,并插入 hash 桶首部

pair=(hashpair\*) malloc(sizeof(hashpair)); pair->key=copystring(key);

pair->cont=cont;

pair->next=table->bucket[hash]; table->bucket[hash]=pair;

#ifdef HASHTHREAD

//解锁

sync\_synchronize(); // memory barrier table->locks[hash] = 0;

#endif

return 2;

}

//从 hashtable 中删除指定 key 对应项

//如果在 hashtable 未发现此项则返回 0，

//如果发现，并成功删除返回 1

int delItem(hashtable\* table,char\* key){

//根据 hash 值来计算 key 在 hash 表中的位置

int hash=hashString(key)% table->num\_bucket;

//检索此项的 key 是否已经存在,如果已经存在就在 hash 桶中寻找此项值，并进行替换

hashpair\* pair=table->bucket[hash];

hashpair\* prev=NULL; //记录 hash 桶中前一项数值

if(pair==0) //此 key 不存在

return 0; #ifdef HASHTHREAD

//利用 GCC 中函数，加自旋锁

while ( sync\_lock\_test\_and\_set(&table->locks[hash], 1)) {

//gcc 内部函数, 原子操作，将 table->locks[hash]中的值设置为 1，并返回原来的数值

//当第一次进入时，返回 0，而同时第二次进入则为 1，因此后面进入的线程获得此值均为 1，导致在此处忙等待

//do nothing

}

#endif

//遍历 hash 桶

while(pair!=0){

if(0==strcmp(pair->key, key)){

//在桶中找到匹配 key，更改桶链表if(!prev)//在桶中第一项

table->bucket[hash]=pair->next; else //在桶中其它位置

prev->next=pair->next;

//删除 pair,释放资源free(pair->key);

free(pair->cont->address); free(pair->cont);

free(pair); return 1;

}

//运动到 hash 桶的下一项

prev=pair; pair=pair->next;

}

#ifdef HASHTHREAD

//解锁

sync\_synchronize(); // memory barrier table->locks[hash] = 0;

#endif

return 0;

}

//根据 key 值，从 hash 表中查找相应项

//如果没有找到返回 NULL

content\* getContentByKey(hashtable\* table, char\* key){

//根据 hash 值来计算 key 在 hash 表中的位置

int hash=hashString(key)% table->num\_bucket;

//检索此项的 key 是否已经存在,如果已经存在就在 hash 桶中寻找此项值

hashpair\* pair=table->bucket[hash];

while(pair){

if(0==strcmp(pair->key, key)) return pair->cont;

pair=pair->next;

}

return NULL;

}

#define NUMTHREADS 8

#define HASHCOUNT 1000000

typedef struct threadinfo {hashtable \*table; int start;} threadinfo;

void \* thread\_func(void \*arg){ threadinfo \*info = arg; char buffer[512];

int i = info->start;

hashtable \*table = info->table; free(info); for(;i<HASHCOUNT;i+=NUMTHREADS) {

sprintf(buffer,"%d",i);

content\* cont=malloc(sizeof(content)); cont->length=rand()% 2048;

cont->address=malloc(cont->length); addItem(table, buffer, cont);

}

}

int main(void){

hashtable\* table=createHashTable(HASHCOUNT); srand((unsigned)time(NULL)); // 初始化随机种子

// hash a million strings into various sizes of table

struct timeval tval\_before, tval\_done1, tval\_done2, tval\_writehash, tval\_readhash; gettimeofday(&tval\_before, NULL);

int t;

pthread\_t \* threads[NUMTHREADS]; for(t=0;t<NUMTHREADS;++t) {

pthread\_t \* pth = malloc(sizeof(pthread\_t)); threads[t] = pth;

threadinfo \*info = (threadinfo\*)malloc(sizeof(threadinfo)); info->table = table; info->start = t; pthread\_create(pth,NULL,thread\_func,info);

}

for(t=0;t<NUMTHREADS;++t) {

pthread\_join(\*threads[t], NULL);

}

gettimeofday(&tval\_done1, NULL); int i,j;

int error = 0; char buffer[512];

for(i=0;i<HASHCOUNT;++i) {

sprintf(buffer,"%d",i); getContentByKey(table,buffer);

}

gettimeofday(&tval\_done2, NULL); timersub(&tval\_done1, &tval\_before, &tval\_writehash); timersub(&tval\_done2, &tval\_done1, &tval\_readhash); printf("\n%d threads.\n",NUMTHREADS);

printf("Store %d ints by string: %ld.%06ld sec, read %d ints: %ld.%06ld sec\n",HASHCOUNT, (long int)tval\_writehash.tv\_sec, (long int)tval\_writehash.tv\_usec,HASHCOUNT, (long int)tval\_readhash.tv\_sec, (long int)tval\_readhash.tv\_usec);

freeHashTable(table); return 0;

}

* 1. Web **页面的缓存置换算法**

对于第二个问题（该怎样缓存 Web 页面及相关文件内容），是缓存内容置换问题。即需要让经常使用的文件内容尽量缓存在内存中，将不经常使用的文件内容从内存中置换出去， 以增强缓存命中率，节省 Web 服务器文件查找和文件读取时间。

对于如何置换缓存中的内存页面，常用的有最优置换（OPT）、先进先出置换（FIFO）、最近使用置换（MRU）、最近最久未使用置换（LRU）、最少使用置换（LFU）、自适应置换（ARC）和多队列置换（MQ）等算法4。

* 最优置换**算法（OPT）**

最优置换算法是指将“未来”一段时间内都不会用到内存页面内存，从缓存中置换出去。 因为无法获知未来会有哪些内容会被重复使用，因此这个算法并不能被实现。但是，如果内 存页面内容使用序列已知，那么这个算法可以用来评估其它置换算法的好坏。

* 先进先出置换**（FIFO）**

从缓存中置换出最先进入缓存的内存页面内容。这个置换策略比较简单，但是它并未考虑缓存中内存页面内容被使用的情况（有些内存页面内容被经常使用，而有些使用较少），因此在一般情况下，它的命中率指标较低。

* 最近最久未使⽤置换**（LRU）**

由于最佳置换算法无法知道未来用户请求内存页面内容的情况，该策略以“最近使用的 内容页面，未来最有可能被使用”的假设为前提，通过记录目前缓存中页面被使用的时间次 序，来决定将缓存中最长一段时间都未使用的页面内容进行替换。LRU 算法效率为 O(1)。

除了基本的 LRU，其还有很多的变体。其中，时间敏感性 LRU（TLRU）置换在 LRU 置

换策略基础上，考虑了缓存中内容时间有效性因素。这是因为在网络上许多的内容都是有时 效的。例如，一个网页文件中的内容只有 1 个小时的有效性，在下一个小时其内容很可能会被更新。因此要设计好的置换算法，要考虑缓存中的内容时效性。TLRU 置换策略在缓存中为每个内存页面保存一个有效使用的时间，在进行缓存中页面内容置换时，首先选择已经超

过有效时间的页面内容进行置换；然后根据时效时间，选择则最短时效的候选集，在此候选 集内，选择最近最久未被的页面内容进行替换。TLRU 经常被应用于网络环境下的内容缓存管理，例如在 Content Delivery Networks(CDN)、Information-centric network(ICN)和分布式

网络中经常会使用 TLRU 来进行缓冲区中内容置换。

* 最近使⽤置换**（MRU）**

置换最近使用的文件。这个策略与 LRU 相反，将最近经常使用的文件进行置换。这个置换策略经常用在请求文件序列是循环模式或者随机模式，并且在这两种请求模式下要比LRU 命中率高。这是因为此置换策略更倾向于保存在缓存中更为持久的数据。MRU 的算法效率为 O(1)

* 最少使⽤置换**（LFU）**

4 https[://www.wikiwand.com/en/Cache\_replacement\_policies#](http://www.wikiwand.com/en/Cache_replacement_policies)

通过记录缓存中每个页面内容被使用的次数，来置换缓存中使用次数最少的页面内容。 例如，在仅有三个页面的缓存中存在页面 A 被使用 5 次，页面 B 被使用 3 次和页面 C 被使用 6 次，如果有请求 D 内容，则 LFU 将选择 B 页面中的内容进行置换。由于每次使用缓存，

需要维护页面使用次数的排序队列，因此 LFU 的算法效率为 O(log2n)。

但是，使用 LFU 会造成频繁使用的页面在以后不被使用后，还会继续存在于内存中（这种现象被称为缓存污染-cache pollution）。为了克服这种现象，LFU-Age 置换策略在页面引

用次数的基础上引入平均最大引用次数阈值，当缓存中所有页面的引用次数平均值高于此 阈值后，将对所有页面的引用次数进行减少，可以是减少一个固定值，也可以是减少其原来 的一半。

###### Least-recently/frequently-used (LFRU）

LRU 使用时间作为置换的依据，能够较好地处理最近访问热点数据模型，但不能区分哪些数据页是经常访问的？哪些数据页不是经常访问的？；而 LFU 使用次数作为置换的依据，与 LRU 相比，虽然能够掌握哪些数据页是经常访问的，但是并不知道这些数据页被访问的时间。结合两者的优点，产生了 LRFU。

在 LRFU 中，缓存的每个页面具有一个数值，这个数值能够同时表示这个页面被使用的次数和时间次序。将这个数值表示为 C(x)，其中 x 表示缓存页面。在初始化时，首先将每个页面 x 的值设置为 C(x)=0。在后续使用中，如果缓存中页面被使用（命中），则 C(x)=1+2-λ C(x)；如果没有使用，则 C(x)=2-λC(x)，其中 0<λ<1。在进行页面置换时，选择 C(x)最小的页面进行置换。很明显，当λ=0 时，C(x)的数值为页面被使用的次数；当λ=1 时，C(x)强调的是页面近来被使用的时间次序。此算法的效率介于 O(1)和 O(log n)之间。

2

* 多队列置换**（MQ）**

在介绍多队列置换之前，先介绍一下 2Q 置换算法5。在 2Q 算法中存在两个缓存队列， 一个是 FIFO 队列，一个是 LRU 队列，如下图所示。当数据第一次访问时，将数据缓存在 FIFO 队列里面，当数据第二次被访问时，则将数据从 FIFO 队列移动到 LRU 队列。当 LRU 队列中的数据内容被再次访问，将此数据页面移到队列首部（LRU 置换方法）。

5 Johnson T, Shasha D. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm[J]. very large data bases, 1994: 439-450.

**新加入数据**

**淘汰数据**

**FIFO**

**第二次访问数据**

**LRU**

**再次访问数据**

**淘汰数据**

图 3-3 2Q 内存置换算法

如果 FIFO 队列中的数据一直都未被访问，则按照 FIFO 规则进行淘汰；在 LRU 队列中的数据将按 LRU 规则进行淘汰。发生 LRU 队列进行数据内容置换的时机仅在于 FIFO 队列中的数据内容被第二次访问，这时需要将此数据调入到 LRU 队列中。

通过上述描述，可以看出 2Q 通过 FIFO 队列将能够快速移除冷数据，使用 LRU 队列来维护热数据。2Q 的算法效率为 O(1)。

在 MQ6算法中，将使用多个 LRU 队列(Q ,…,Q )来存储访问不同次数的数据页，并使用 Qout 来存储数据页面 ID、访问次数和超时时间（expireTime），其中超时时间是一个抽象的数字，只是用来表示发送内存访问的时间先后。具体数据结构如下图 3-4 所示。其中， Q0 队列存储访问 1 次的数据页，Q1 队列存储访问 2~3 次的数据页，Q2 队列存储访问 4~7 次的数据页，Qm-1 队列存储访问 2m-1~2m-1 次的数据页。数据页被访问的次数，可以决定其存在于哪个 LRU 队列。在图中例子，每个队列访问次数分级按照 2n 给出的，读者也可以自己定义的访问次数分级，但需要注意的是，如果 i<j，Qj 中的数据访问次数一定要大于 Qi。Qout 是 FIFO 队列，并具有固定长度。

0 m-1

6 Yuanyuan Zhou, James Philbin, and Kai Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. USENIX, 2002

Q0 **a b …**

Q1 **f**

Q2 **c d**

Q… **…**

Qm-1 **…**

Qout

**id reference expireTime**

图 3-4 MQ 内存置换算法

**a**

**b f**

**c d**

**…**

**1**

**1**

**2**

**4**

**6**

**…**

**24**

**20**

**15**

**30**

**29**

**…**

MQ 算法具体步骤如下面的伪代码所示。

1. 如果要访问的数据页 b 在缓存中（命中），则将此数据页从当前 LRU 队列中移除；如果不在缓存中，则执行 EvictBlock 函数来从缓存中选中一个已经存在数据的数据页 victim， 如果 b 存在于 Qout 队列则将 b 项从 Qout 中删除，否则将 b 项的使用次数设置为 0，最后将 b 数据加载入 victim。

EvictBlock 函数首先按照 0~(m-1)的次序，寻找不为空的队列；然后选择这个队列首部数据页面作为 victim，并将此 victim 从队列中移除；如果此时 Qout 队列已满，则将 Qout 队列的头部数据项移除 。最后将此 victim 的id 和使用次数加入到Qout 队列尾部，并返回此 victim。

1. 将 b 的访问次数加 1，根据访问次数获得 b 所在的队列，并将 b 加入此队列尾部， 并重新计 b 的超期时间。在计算超时中，lifeTime 是 MQ 算法中的参数，用来规定一个数据内存页在一个队里中存在的时长。
2. 执行 adjust 函数，判断每个队列里面的队首内存页的 expireTime 是否超过了当前时间 currenTtime，如果超过则意味着其在这个队列(Qk)的存活时间已过，要将这个数据页面移入到下一级队列(Qk-1)的尾部，并重新计算它的 expireTime。通过这种方式，能够保证原来被

经常访问而最近不被访问的数据，在经过一段时间后能够逐步进行降级，直到其最终从缓存 中清除。具体清除的速度，决定于参数 lifeTime 的大小

//MQ 算法伪代码

/\*Procedure to be invoked upon a reference to block b\*/ if b is in cache{

i = b.queue;

remove b from queue Q[i]; } else{

victim = EvictBlock(); if b is in Qout {

remove b from Qout; }else{ b.reference = 0;

}

load b’s data into victim’s place;

}

b.reference ++;

b.queue = QueueNum(b.reference); insert b to the tail of queue Q[k]; b.expireTime = currentTime + lifeTime; Adjust();

EvictBlock(){

i = the first non-empty queue number; victim = head of Q[i];

remove victim from Q[i]; if Qout is full

remove the head from Qout; add victim’s ID to the tail of Qout;

return victim;

}

Adjust(){ currentTime ++;

for(k=1; k<m; k++){ c = head of Q[k];

if(c.expireTime < currentTime){ move c to the tail of Q[k-1];

c.expireTime = currentTime + lifeTime;

}

}

}

如果当 m=1 时，MQ 类似 LRU；m=2 时，MQ 类似于 2Q，但是它们并不相同。除了它们之间的数据结构不同：MQ 中有两个 LRU 队列，而 2Q 中有一个 LRU 队列和一个 FIFO 队列，还主要因为 MQ 有 expireTime 属性，能够将在 Q1 超期的数据页移动到 Q0 中，而 2Q 不具有这样的性质，只能将其中任一个队列中的数据进行移除。

参数 lifeTime 对于 MQ 算法的好坏也影响较大。为了克服此弱点, 在假设“连续访问相同数据页的时间间隔分布服具有一个山的形型”的前提下，lifeTime 数值可以在算法运行过程中进行动态调整7。

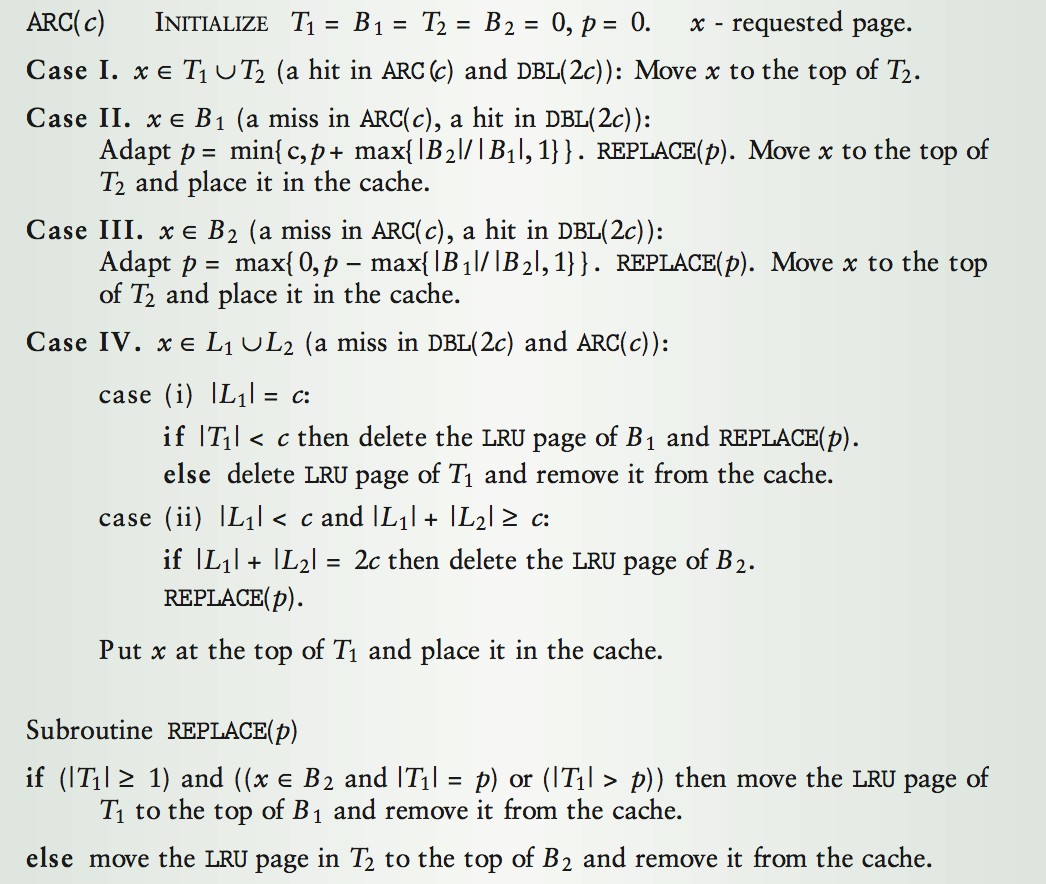
理论上 MQ 的算法复杂度与 2Q 相同，均为 O(1)。但是实际运行过程中，每次需要检查m 个队列的首部，因此其运行开销比 LRU、2Q 等要较高。

###### ⾃适应替换策略(Adaptive Replacement Cache-ARC)

ARC8维护两个 LRU 数据页面队列：L1 和 L2。于 2Q 算法类似，L1 中维护只被访问一次的数据页面，而 L2 中维护着短时间至少被访问两次的数据页面。这样可以将 L1 看为维护数据页面的时效性，而将 L2 看为维护数据页面的访问频率。将通过下面的数据结构和数据更新策略，ARC 能动态调整这两个队列的大小，以尽量来提高缓存的命中率。

7 Yuanyuan Zhou. Memory Management for Network Server(Thesis). Technical report, Princeton University, Computer Science Department, November 2000.

8 Megiddo N, Modha D S. Outperforming LRU with an adaptive replacement cache algorithm[J]. Computer, 2004, 37(4): 58-65.



###### 随机置换

图 3-5 ARC 伪代码 8

利用随机函数，随机选中缓存中一个数据页进行置换。以上算法考虑的是大小相等的数 据页面替换，而对于 Web 缓存要考虑的是大小不相等的文件，因此要从以下两个指标来衡量缓存命中率。

###### 文件命中次数比率（document hit rate）

document hit rate = 缓存中命中的文件数量/用户请求的文件数量

###### 字节命中比率（byte hit rate）

byte hit rate = 缓存中命中文件所有字节总和/用户请求文件所有字节总和

除此之外，Web 缓存与其它缓存不同的，Web 缓存可以根据 http 协议中提供的相关信息来决定其缓存的替换策略。例如，HTTP 1.1 协议在响应头设置 expires 字段以及 Cache Control 字段中设置 max-age 来通知客户端其请求资源的有效性时间，并在 ETag 字段中封装这个文件的验证码，如下图 3-6 所示；如果超出这个有效性时间，用户在检索这个文件时，在请求头中加入此 Etag，当服务器接收到此请求消息后，首先进行 Etag 校验（Etag 校

验码一般是根据文件内容生成的），如果与服务器生文件一致，则说明此文件还为修改，此时服务器将向客户端发送“not modified”的响应消息，具体过程如下图 3-7 所示。

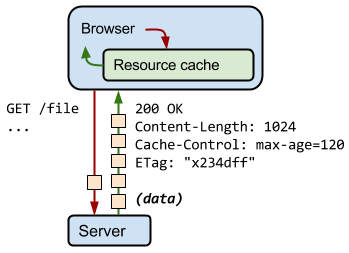


图 3-6 浏览器与 Web 服务器请求消息交互图（来自 Ilya Grigorik9）

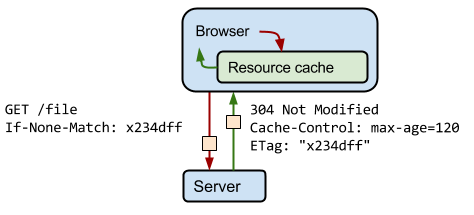


图 3-7 浏览器与 Web 服务器响应消息交互图（来自 Ilya Grigorik）

因此，在构建 Web 缓存时，将主要根据表 3-1 中的参数指标来进行缓存 Web 页面置换算法设计。

表 3-1 Web 缓存参数表

|  |  |
| --- | --- |
| **标志符 说明** | |
| **si** | 文件 i 的大小 |
| **ti** | 上次请求文件 i 的时间 |
| **Ti** | 目前距离上次请求文件 i 已经过的时间 |
| **fi** | 请求文件 i 的次数 |
| **li** | 获得文件 i 的时长 |
| **ci** | 从文件 i 所在服务器获取它的代价。ci 比 li 意义更为广泛，其即可以表示时长，也可以  表示网络节点的跳数 |

9 https://developers.google.com/web/fundamentals/performance/optimizing-content- efficiency/images/http-request.png?hl=zh-cn

* 基于 LRU 策略的 Web 置换算法

最为直接的是将数据页面的 LRU 算法应用在 Web 文件置换中。在此基础上，LRU- Threshold 算法只缓存 si 大于一定数值的文件，在替换时使用 LRU 进行替换。

LRU-Min 算法以替换最少文件数量为目标，进行文件替换算法设计。设 L 为一个 LRU 临时队列，其存储着在缓存中大小均大于或等于 T 的文件 ID。假设请求文件的大小为 S，初始时 T=S。则 LRU-Min 算法如下：1）并从缓存中寻找文件大小大于或等于 T 的文件 ID，并根据时间先后形成队列 L。2）不断地应用 LRU 算法在 L 中寻找需要替换的文件，直到 L 缓存为空或者寻找出来的替换文件空间总会至少为T 时。3) 如果替换空间总和大于或等于S， 则将这些文件所占内存替换为请求文件的内容，并退出；如果小于 S，则设 T=T/2，转到步骤 1）。

Value-Age 算法以下面的计算得到的 V 值来形成

* 基于频率策略的 Web 置换算法

此策略在 LFU 算法基础上进行改进。

LFU-DA 算法以下面的公式计算缓存每项内容的数值 K，然后选出 K 值最小的缓存进行替换。

Ki=fi+L，其中 fi 为请求文件的次数，L 为文件在缓存中的时间长度，当文件刚进入缓存时 L=0，然后每发生一次替换，就将此文件的 L 增加 1.

* 基于 LRU 和 LFU 融合策略的 Web 置换算法

此置换策略下有 LFRU、MQ、ARC 等算法。

* 基于函数策略的 Web 置换算法

GD(GreedyDual)-Size 算法按照下面公式计算，并选择数值 K 小的文件项进行替换。

Ki=Ci/Si+L，其中 Ci 为从外存加载入缓存的代价，Si 为文件 i 的大小，L 为文件在缓存中的时间因素。当文件刚进入缓存时 L=0，然后每发生一次替换，就将此文件的 L 增加 1.

GD 算法并没有考虑访问文件的次数因素，因此在 GDSF(GreedyDual-Size with Frequency)算法中，引入了访问次数因子，其计算公式为：

Ki=fi\*Ci/Si+L。

除此之外，有关其它的 Web 缓存算法，可以看文献《A Survey of Web Cache Replacement Strategies》10

10 Podlipnig S, Böszörmenyi L. A survey of web cache replacement strategies, ACM Computing Surveys (CSUR), 2003, 35(4): 374-398.

* 1. Web **服务器的内存管理模型**

在上一节中，哈希表中维护的文件内容是通过 C 语言的库函数 malloc 和 free 来进行内存的分配和释放的。那么在 C 语言库函数 malloc 和 free 是如何管理内存的呢？其管理内存的模型对于 Web 缓存内容管理是否高效？如果不高效，还有没有其它的内存管理模型来适合 Web 页面内容的缓存管理？

这些问题都将会在本节中做出解答。本节内容的主要安排为，首先介绍常用的内存管理 模型，然后介绍 linux 下“aka ptmalloc2”版本的 malloc 和 free 内存管理模型，最后以 Nginx 为例，来讲解实际 Web 服务器中的缓存模型结构。

由于 Web 缓存中存储的文件大小不一样，并且不断有文件读入缓存和从缓存中删除，

缓存管理中最容易产生“内存碎片”现象。内存碎片包括外部碎片和内部碎片。

其中，外部碎片指的是在一块内存区域存在着一些空闲内存页，虽然所有这些空闲页累 计起来能够满足用户对内存大小的要求，但是由于这些空闲内存页并不连续，从而不能满足 连续分配内存给用户的要求。如下图 3-8 所示，在 8 个页的内存块中分配了 0、1 和 4 页，

此时不能分配 4 个连续内存页。随着内存不断的申请和释放，使得连续空闲的内存块大小会

越来越小（连续的页数越来越小），从而造成外部碎片。

***0***

***3***

***4***

***5***

***6***

***7***

***2***

***1***

图 3-8 内存页外部碎片

内部碎片指的是当用户申请几十个字节时，系统也要分配一个内存页，从而在每个页内 形成有很多没法利用的空间。

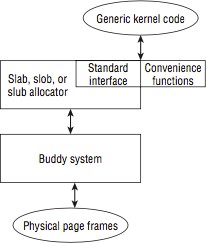


图 3-9 Linux 内核管理内存接口关系图11

11 Professional Linux Kernel Architecture, P.259 中图 3-43

为克服外部碎片现象，Linux 内核中采用伙伴（buddy）内存管理方法；为克服内部碎片，Linux 内存采用 slab 内存管理方法。具体 Linux 内核内存管理接口关系如图 3-9 所示， 在内核中，最底层的内存管理方式是以页为单位的伙伴管理方法。在此基础上，根据不同系 统（例如计算机设备、嵌入式设备）所设计的 slab、slob 或 slub 等内存管理算法，从伙伴

内存管理中申请内存，并进行精细化管理，以为克服内部碎片问题。内核上层的接口将根据 使用内存的大小来决定向 slab 内存管理方法或伙伴内存管理方法来申请内存。

###### Linux 内核内存管理模型

1. Linux **进程逻辑地址空间模型**

Linux 系统中每个进程的逻辑地址空间包括用户地址空间和内核空间。在用户空间中包含了程序代码段（text）、数据段（data）、bss 段、堆（heap）、栈（stack）、内存映射区（memory mapping）等段，如下图所示。其中，程序代码段存放程序的代码；数据段存放程序中已经 初始化的全局变量-代码中全局变量的静态内存分配；bss 段存放程序中未初始化的全局变量；堆用来存放程序运行过程中动态分配的内存（在 C 代码中通过 malloc 库函数来完成）；栈存放程序代码中创建的临时变量（在 C 语言中“{ }”之间出现的变量为临时变量）；内存映射区存放映射到内存的文件或者共享内存。

内核空间存放操作系统内核代码和相关数据结构，在 32 位 x86 架构下，内核位于

3GB~4BG 的逻辑地址空间内。内核空间主要包括物理空间映射区（physical space）、安全区

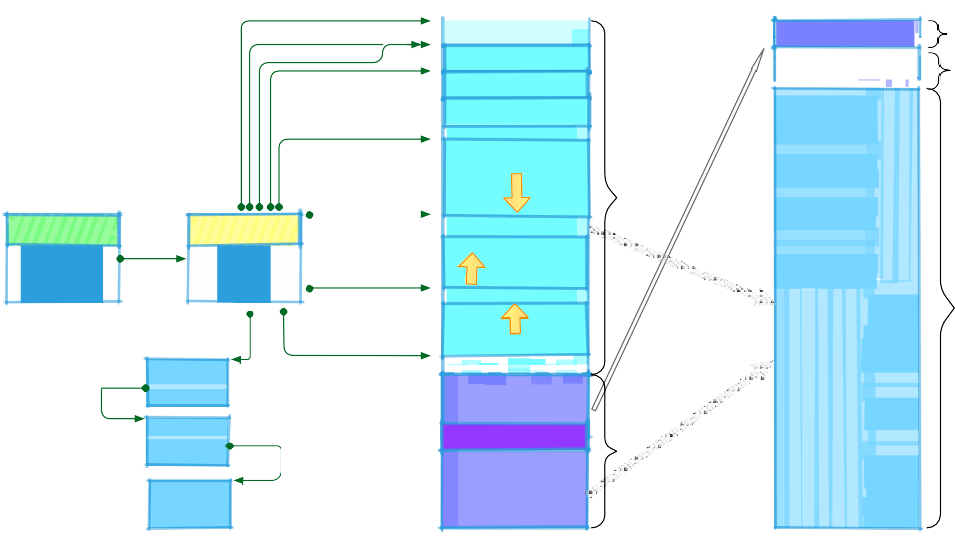
（safe space）和虚拟空间映射区（virtual space）。其中安全区大小为 8MB，主要用于隔离

物理空间映射区和虚拟空间映射区；物理空间映射区主要用于连续物理内存的分配；虚拟空 间映射区主要用于非连续物理内存分配。

在 Linux 中，每个进程对应一个进程描述符 task\_struct。在 task\_struct 中的 mm\_struct

描述了进程逻辑地址空间以及进程页表，具体如下图 3-10 所示。其中，mm\_struct 通过start\_code、end\_code、start\_data 和 end\_data 指针分别指向代码段的起始和结束位置，以及数据段的起始和结束位置；通过 start\_brk 和 brk 指针记录了“堆”段的起始和结束位置； 通过 start\_stack 指针记录了“栈”段的起始位置；通过 mmap\_base 指针指向内存映射段的基地址；通过 pgd 指针指向页目录的基地址，（建立了“页目录-页表”的二级页表结构或 “页目录-中间页目录-页表”的三级页表结构或四级页表结构）。

**Z0NE\_DMA Z0NE\_NORMAL**



start\_code end\_code start\_data

end\_data

**0**

start\_brk

kmalloc

**用户空间**

ᬳᖅฉ੘

brk

mm

mmap\_base

pgd

start\_stack

ۖாฉ੘

vmalloc

**页目录**

pte

**页表**

**3G**

**内核空间**

ۖாฉ੘

**page**

**页**

**4G**

**mm\_struct**

**task\_struct**

**virtual space**

**safe space(8M)**

**physical space**

**896M**

**stack**

**memory**

**mapping**

**heap**

**…**

**bss**

**data**

**text**

**896M~4GB**

**880M**

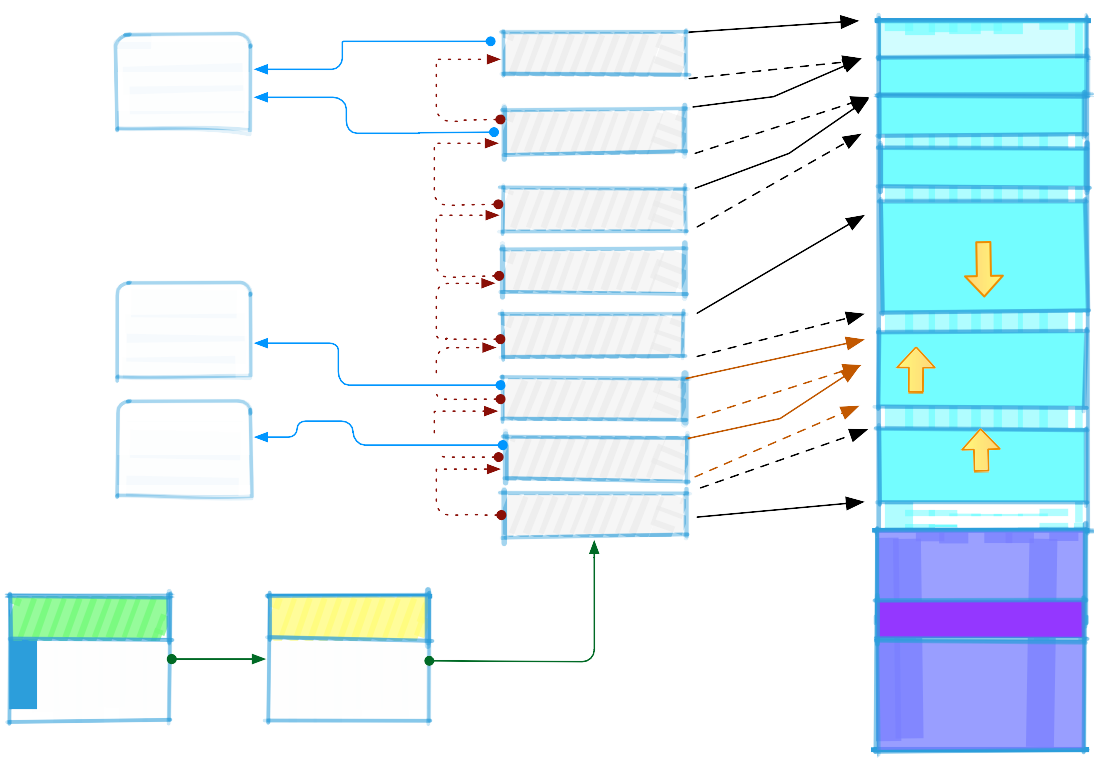
**16M**

**Z0NE\_HIGHMEM**

**进程逻辑地址空间 物理地址空间**

图 3-10 Linux 逻辑地址管理与物理内存映射

除此以外，在 mm\_struct 中还维护一个 vm\_area\_struct 链表，其中每个 vm\_area\_struct 对映进程逻辑地址空间一个段，如下图所示。mm\_struct 并通过 mmap 指针指向此链表的首地址。在 vm\_area\_struct 中 vm\_strat 为段首地址，vm\_end 为段结束地址；而 vm\_file 指向被映射到内存的文件段，例如将动态库 lib1.so 和 lib2.so 分别映射到进程逻辑空间的指定位置。



vm\_file

**struct\_file**

**/bin/demo**

vm\_next

vm\_start

**vm\_area\_struct** vm\_end

vm\_file

**vm\_area\_struct**

vm\_next

vm\_next

**vm\_area\_struct**

**text data bss**

**…**

**heap**

**struct\_file**

**/lib/lib1.so**

vm\_file

vm\_next

vm\_next

**struct\_file**

**/lib/lib2.so**

**vm\_area\_struct**

**vm\_area\_struct**

**memory**

**mapping**

vm\_file

vm\_next

vm\_next

**vm\_area\_struct**

**vm\_area\_struct**

vm\_end

vm\_start

**stack**

mm

mmap

**physical space**

**896M**

**safe space(8M)**

**virtual space**

**task\_struct**

**mm\_struct**

**进程逻辑地址空间**

图 3-11 Linux 进程逻辑地址空间与内存段管理

通过 vm\_area\_struct 和页表映射结构（pgd、pte 和 page）能够将进程逻辑地址空间中各个数据段映射到具体物理内存地址。

1. Linux **物理内存模型**

在现代 CPU 体系结构下，物理内存多按照页进行管理，并通过 MMU 来完成逻辑地址到物理地址的映射。Linux 为每个物理页创建一个 page 数据结构，其代码具有如下。每个页占用 40 个字节。在页大小为 4K、物理内存为 4G 的系统情况下，内核要分配 1M 个 page， 占用 40MB 内存空间。

//include/linux/mm\_types.h struct page {

unsigned long flags; //表示页的状态atomic\_t \_count; // 引用计数atomic\_t \_mapcount; // 映射计数unsigned long private; //权限

struct address\_space \*mapping; //指向 address\_space 对象地址

pgoff\_t index;

struct list\_head lru; //lru 链表

void \*virtual; //对应的虚拟地址（逻辑地址）

};

为支持多 CPU，内存管理模型主要分为统一内存存取模式（UMA-uniform memory access）和非统一内存存取模式（NUMA-non-uniform memory access）12。目前众多 CPU 采用 NUMA 内存管理方式，以支持对称多处理器（SMP-Symmetric multiprocessing ）架构13。 Linux 为支持多种 CPU 管理内存架构，采用 Node 来对应一个 CPU，而每个 node 的数

据结构使用 pg\_data\_t 进行描述（/include/linux/mmzone.h ），如图 3-12 所示。图 3-12 中以 NUMA 的内存管理架构为例来进行说明，如果是 UMA 架构近需使用其中一个 pg\_data\_t 来描述所有 CPU 使用内存情况即可。每个 pg\_data\_t 中 node\_mem\_map 指向 page 链表首部，也就是这个 CPU 管理的本地内存页位置。

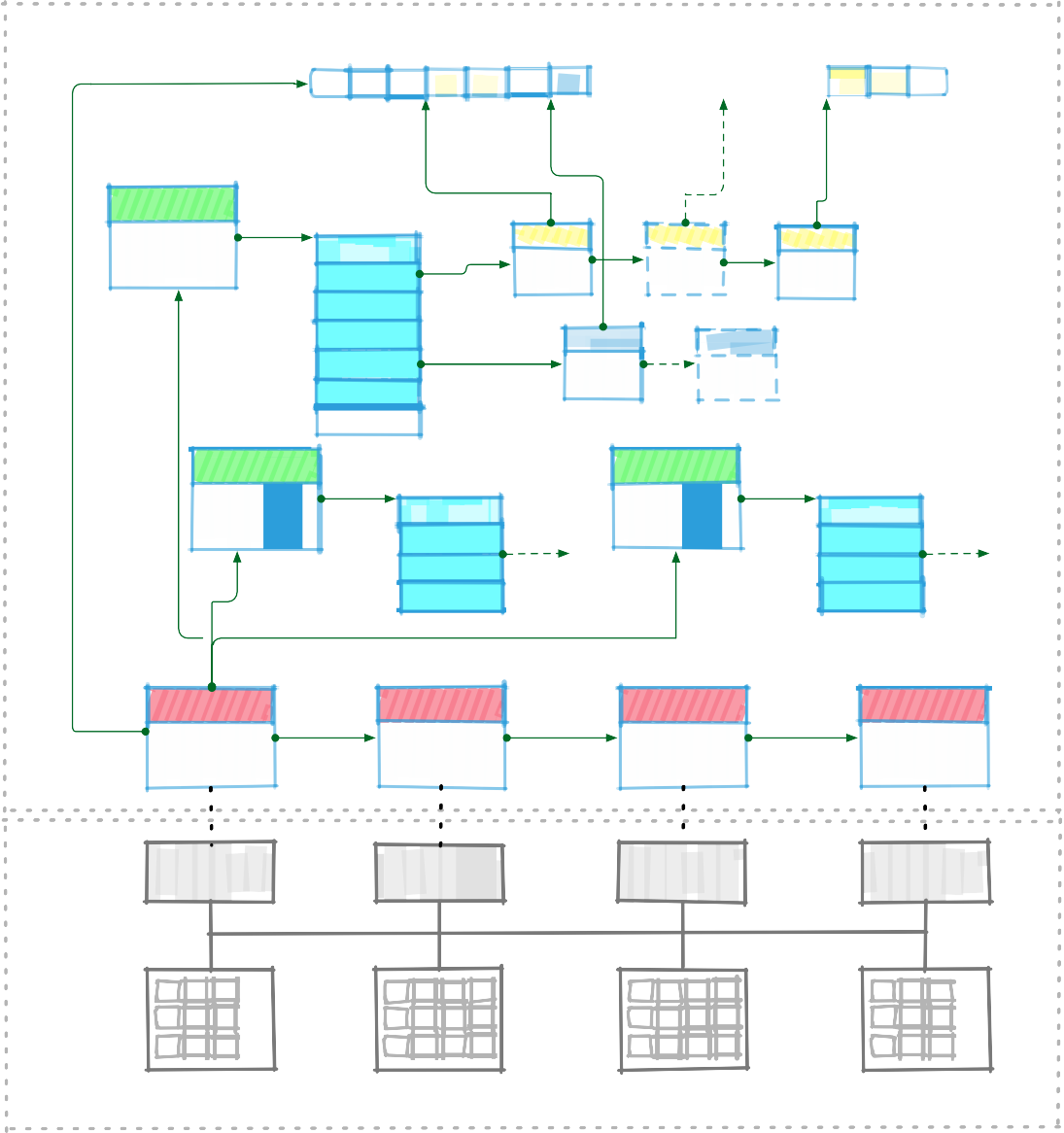
在 pg\_data\_t 中的 node\_zones 数组中分布保存着指向不同内存区域的指针。为支持不同设备或代码不使用内存数据不同的方式，Linux 系统中将多个连续的物理内存页划分为区，形成不同的内存池，主要包括 ZONE\_DMA、ZONE\_NORMAL 和 ZONE\_HIGHEM 区，如图3-10 所示。其中， ZONE\_DMA 用于 DMA 设备访问的数据区域，大小为 16MB ； ZONE\_NORMAL 为内核正常映射，大小为 880MB；ZONE\_HIGHEM 用于描述处在高端区域的物理页（32 位 x86 架构下，其范围为 896MB~4GB）。

ZONE\_DMA 和 ZONE\_NORMAL 内存池用于连续物理页的分配；而 ZONE\_HOGHEM 用于非连续的物理页分配。

12 https://en.wikipedia.org/wiki/Non-uniform\_memory\_access

13 https://en.wikipedia.org/wiki/Symmetric\_multiprocessing

Linux 系统为每个内存池创建一个 zone 数据结构。常用的 zone 有 Normal,DMA 和HighMem。在每个 zone 中，free\_area[MAX\_ORDER]描述了内存空闲和使用情况。free\_area 是为支持伙伴内存管理方法创建的数据结构，如下图 3-12 所示。free\_area 数组的下标表示连续的页数量，而 free\_area 中的 free\_list 为一个连接 page 的 LRU 链表。在这个链表上的每个页都对应连续物理内存的起始页。例如，free\_area[1]中每个 page 中的 virtual 都指向了两个连续物理内存页的起始位置；而 free\_area[k]中每个 page 的 virtual 指向了 2k 个连续物理内存页的起始位置。



**Linux Memory Data Structure** physical memory

21pages

2kpages

21pages

**…**

virtual

virtual

free\_area [ ]

free\_list

lru

lru

free\_list

virtual

**page** lru **page**

free\_area [ ]

free\_area [ ]

node\_zones [ ]

**pg\_data\_t**

**pg\_data\_t**

**pg\_data\_t**

**pg\_data\_t**

node\_mem\_map

**CPU0**

**CPU1**

**CPU2**

**CPU3**

**NUMA**

memory

memory

memory

memory

**MAX\_ORER-1**

**…**

**1**

**0**

**MAX\_ORER-1**

**…**

**1**

**0**

HighMem

**zone**

DMA

**zone**

**MAX\_ORER-1**

**…**

**k**

**…**

**2**

**1**

**0**

**page**

**page**

**page**

Normal

**zone**

**…**

**…**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **…** |  |  | **…** |  |  |  |

|  |
| --- |
|  |
|  |
|  |

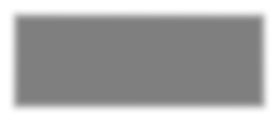
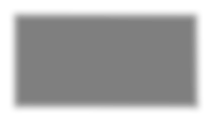
|  |
| --- |
|  |
|  |
|  |

图 3-12 Linux 内核内存数据结构

###### 逻辑地址和物理地址空间分配

操作系统为用户程序和内核程序提供了两种不同的内存分配方法，如图 3-13 所示。对于用户程序，其可以通过库函数 malloc 进行内存分配或者调用系统函数 fork、mmap 等来引起内存分配。其中 malloc 会调用系统函数 brk 来进行内存分配；fork、mmap 会调用do\_map 来进行内存分配。这两个函数都是修改 vm\_area\_struct 数据结构和相应的页表来完成虚拟逻辑内存段分配的。在完成上述操作后，操作系统仅是完成了逻辑地址分配，并

没有分配实际的物理内存。操作系统直到用户代码实际使用这些逻辑地址时，而引起缺页 中断时，才通过 get\_free\_page 函数来分配具体的物理内存。



|  |  |  |
| --- | --- | --- |
| vm\_struct |  | physical space |

|  |  |  |
| --- | --- | --- |
| **user space**  User Application  malloc/fork/mmap/… | | |
|  |  |  |
| **kernel space**  brk |  | Kernel Application  do\_map vmalloc/kmalloc    vmalloc kmalloc |
| vm\_area\_struct    **process page table**  page fault page fault  **get\_free\_page get\_free\_page get\_free\_page**    physical memory | | |

图 3-13 Linux 内存分配方法调用关系图14



|  |  |  |
| --- | --- | --- |
| **kernel page table** |  | **slab** |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |

操作系统为内核代码提供 vmalloc 和 kmalloc 两个函数来分配内存。其中 vmalloc 函数分配连续的逻辑地址空间，但这些逻辑地址对应的物理地址并不一定连续；kmalloc 函数从物理映射空间中分配连续的逻辑地址空间，并且这些逻辑地址对应的物理地址也是连续的。

14图像根据 <http://gityuan.com/2015/10/30/kernel-memory/中的图像修改而成>

Linux 内核通过 vm\_struct 来描述内核占用段，vmalloc 函数与用户库函数 malloc 函数类似，主要修改 vm\_struct 及其对应的页表，然后通过缺页中断来分配实际的物理内存。而kmalloc 函数主要完成物理映射空间中逻辑地址分配，其通过 slab 层来进行时间的内存分

配。

Linux 内核在分配和管理内存时，采用了两种不同的管理方法:buddy 管理方法和 slab 内

存管理方法。其中 get\_free\_space 是通过 buddy 管理方法来完成内存的分配；slab 可以看为是在 buddy 基础上对内存结构进一步的细化管理及其分配机制。buddy 管理方法是以页为单位进行内存的分配和回收的，如果需要的内存小于一页，则会造成内存分配的浪费。这 时 slab 方法在 buddy 分页基础上，根据不同内核对象的大小合理地划分内存页，并对这些

划分有效地管理，从而达到按照这些内核对象实际大小来分配内存空间的目的。（有关 buddy

和 slab 方法详见后面两节）。

通过上述描述可以知道，针对分配大的逻辑地址空间，适合采用 vmalloc 函数；针对小的逻辑地址空间分配，适合采用 kmalloc 函数来完成。由于 vmalloc 分配的内存物理地址不连续，在访问过程中需要修改页表，因此访问速度会比 kmalloc 分配的内存慢一些。

用户程序通过 malloc 和 free 库函数来分配和释放内存，那么在用户进程空间中，这两个库函数如何来管理进程逻辑地址空间的呢？这种管理方法是否是高效的呢？回答这两个 问题见（3.4.2 节）。

1. buddy **管理内存方法**

通过上面内容可以知道 linux 内核中底层是通过伙伴（buddy）方法来管理物理内存页分配和回收的。伙伴内存管理方法思想如下。

* 伙伴内存管理方法每次分配 2order 个连续的物理页，其中将 order 称为阶，即伙伴内存管理方方法每次分配 k 阶个内存页。
* 初始时将物理内存按照页大小进行分页，然后将所有内存看为一块整体，即为 n 阶内存页；
* 当有用户请求内容时，根据用户请求内容大小，为其分配内存页，假设需分配内存 阶数为 k。首先查找当前的系统是否已经有阶数为 k 空闲内存块，如有则直接分配；如没有，则向上查找阶数为 k+1 的内存空闲块，如有则将 k+1 阶的内存块分裂为两个 k 阶内存块，并将其中一个分配给用户，如没有则向上查找阶数为 k+2 的内存空闲块，如有则将此内存空闲块分裂为 1 个 k+1 阶内存块和 2 个 k 阶内存块，并将其中一个分配给用户，如没有，则向上查找阶数为 k+3 的内存空闲块。以此类推，直到找到能够分裂为 k 阶内存页的内存空闲块为止。
* 当用户释放内存时，也就是释放 k 阶内存页时，在系统中查找是否存在与释放 k 阶内存成对的 k 阶内存空闲块（成对指的是在分配 k 阶内存时，应该将一个 k+1 阶内

存分裂为 2 个 k 阶内存块，这两个 k 阶内存块称为内存对—相互之间称为内存伙伴），如没有则将此 k 节内存块直接插入系统；如有则合并为 k+1 阶内存块，并查找是否存在与此 k+1 阶内存块相对应的伙伴，如有则合并为 k+2 阶内存块，再重复上述内存合并步骤，直到内存块无法合并回收为止。

以图 3-14 为例，假设在系统中有 8 个内存空闲页，即存在 3 阶的内存块。当用户请求1 个内存页（即 0 阶内存页）时，经过在 free\_area 数组中查找查找，发现 free\_area[3]中有内存空闲块，则将此空闲块分裂为 1 个 2 阶内存块，1 个 1 阶内存块和 2 个 0 阶内存块，并将其中一个返回给用户。

当用户是否此内存块后，其内存块合并过程与分配时相反，最终形成 1 个 3 节内存块。

**pages** #8

**3**

**4**

**5**

**6**

**7**

free\_area [ ]

**pages** #8

**page**

|  |  |  |
| --- | --- | --- |
| **0** | **1** | **2** |

|  |
| --- |
| **0** |
| **1** |
| **2** |
| **3** |
| **…** |
| **…** |
| **MAX\_ORER-1** |

图 3-14 伙伴内存管理方法分配内存示意图



**0**

**3**

**4**

**5**

**6 7**

free\_area [ ]

**page**

**page**

free\_list

**page**

**MAX\_ORER-1**

**…**

**…**

**3**

**2**

**1**

**0**

**2**

**1**

伙伴内存管理方法针对 k 阶内存进行成对地分配和回收，能够较好地解决外部内存碎片问题。但伙伴内存管理方法也存在以下问题。

* 在一大片内存中，仅有一小片内存存没有释放，也会造成其两边的空闲内存块不能 进行回收。
* 内存分配存在浪费现象，因为其分配按照 k 阶内存块进行分配，如果用户所需内存

小于 k 阶内存块，也要按照 k 阶内存大小进行分配，从而造成浪费；另外，其在分

配内存时按照成对地递归分裂内存方法来分配和管理空闲内存块，其有时会造成严重的内存浪费。例如，假设系统中存在 1024 页内存块（10 阶），当用户申请 16 页

（4 阶）内存块后，系统剩下一个 9 阶内存块，一个 8 阶内存块，…，一个 4 阶内

存块。当用户再申请 600 页内存块时，系统将无法进行分配。

* 在分配和合并内存块中，需要频繁地修改内存链表（free\_area），开销较大。

1. slab **管理内存方法**

伙伴管理方法能够很大程度上避免内存外部碎片的产生，但是如果用户申请小于 1 页

的内存空间，使用伙伴管理方法依据会为每次用户请求分配 1 页的内存，从而造成内存浪费

（内存内部碎片）。为解决内存内部碎片问题，在伙伴内存管理方法基础上，linux 使用 slab

内存管理方法对内存进行精细化管理，以最大程度地避免内存碎片产生。

具体结构如下图 3-15 所示，所有的缓存区（kmem\_cache）放置在缓存 cache\_cache 的双向链表 cache\_chain 中。每个缓存区包含了两个重要的数据结构：array\_cache 数组和nodelist 数组。其中，array\_cache 数组为每个 CPU 分配一个 array\_cache 项，在这项里面包括对象大小、对象最大个数等 slab 属性描述和指向 slab 中可用对象的指针数组 entry。entry 中保存了指定 CPU 刚刚释放的对象指针数组，此数组采用 LIFO(last in, first out )策略来维护和替换数组中的内容。在内存中，每有一个内存 node 就在 nodelist 数组中有一项，每一项的数据类型为 kem\_list3。在 kem\_list3 中保存了三个 slab 双向链表，分别用于表示处于空闲

（free）、完全占用（full）、半空闲半占用（partial）的 slab 双向链表。

每个 slab 包括 slab 头和缓存对象的内存页帧（page frames）。其中 slab 头可以放在后面的对象内存页，也可以独自存在。在 slab 头中描述了所存对象的大小、个数限制、空闲对象指针数组（management array）、着色区（color space）等信息。对象内存页帧（Page Frames）包含 1~32 个内存页，用于保存指定结构的对象数组。

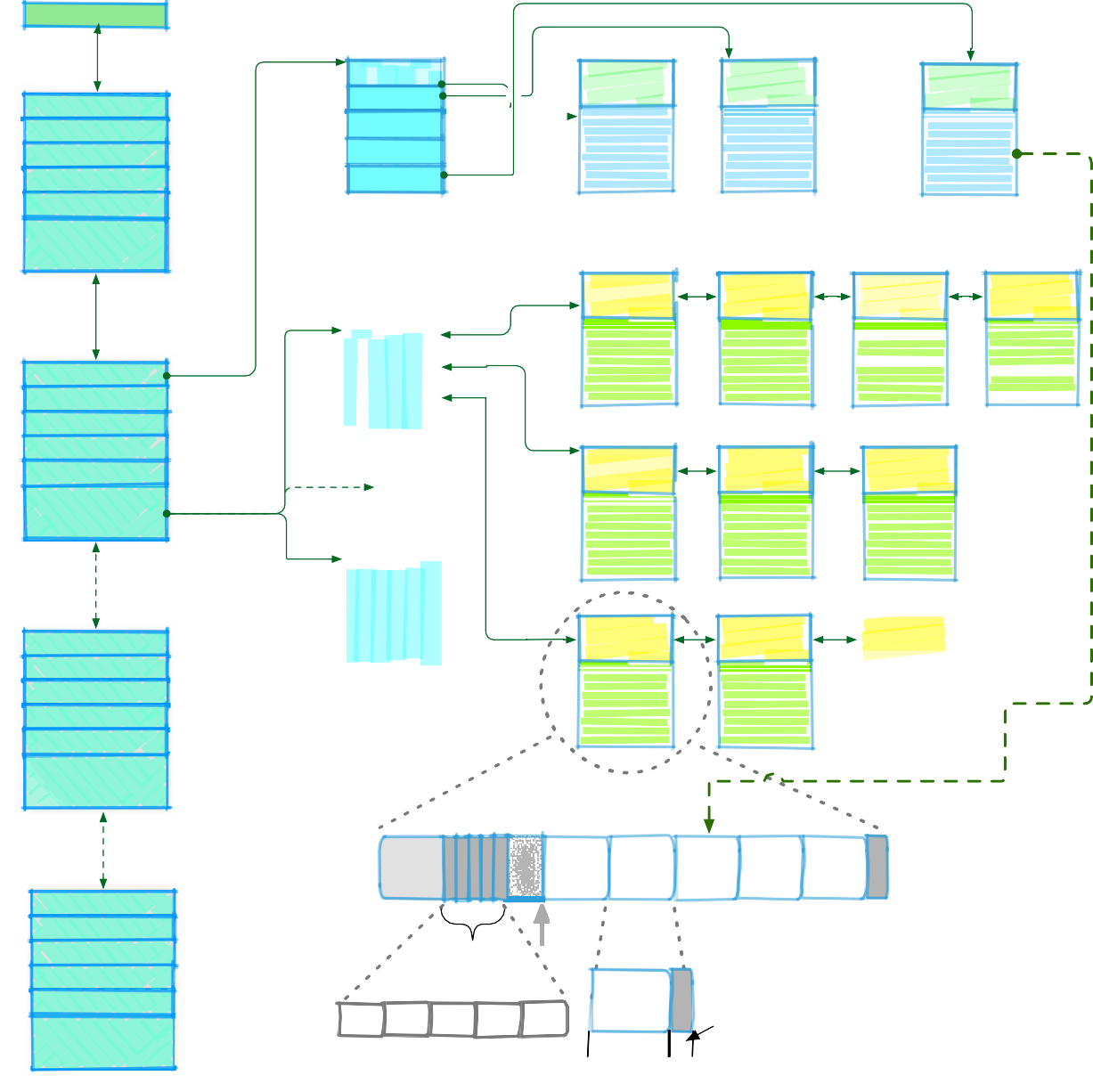
空闲对象指针数组（management array）的数据项为整数，保存下一个空闲对象在对象数组的位置。例如图中 management array 为“3□459□ ”，其中“□”表示此位置的对象已经被使用，即在对象数组中下标为 2 和 6 的对象已经被使用，而 1、3、4、5、9 位置的对象目前为空闲状态，很明显“3、4、5、9”形成了数组结构的指针链表，即每一项内容为 下一项所在数组的位置。着色区（color space）信息使得 slab 中每个对象的起始位置都按照CPU 中的 cache line 大小进行对齐并均匀放置，使得一个 slab 中的各个对象均匀地放置在各个页的不同起始位置，减少产生 cache line 冲突现象，从而提高缓存利用率并获得更好的性能。

|  |  |
| --- | --- |
| **partial** |  |
| **full** |  |
| **free** | |
| **…** | |

|  |
| --- |
| **partial** |
| **full** |
| **free** |
| **…** |

|  |  |
| --- | --- |
| **slab head** | |
|  |  |
| Page | |
| Frames | |
|  | |
|  | |

图 3-15 Linux 内核 slab 内存管理示意图



**cache\_chain**

kmem\_cache

array\_cache [ NR\_CUPS]

**colour name**

entry[ ]

**…**

arrays with

points to slab objects

next

kmem\_cache

kem\_list3

Page

Page

Frames

**objsize**

**…**

Page

next

kem\_list3

kmem\_cache

**name**

Frames

next

kmem\_cache

color

space

**manage**

**ment**

**used free …**

unused

memory

**name**

kmem\_bufctl\_t

management array

s\_mem

**3 4 5 9**

**object**

object size

fill bytes

for alignment

**nodelist [ ]**

**colour**

**objsize**

**array**

**free**

**nodelist [ ]**

**colour**

**objsize**

**array**

Frames

Page

**slab head**

Page

**slab head**

Frames

Page

**slab head**

Frames

**slab head**

Frames

Page

**slab head**

**nodelist [ ]**

**name**

**colour**

**array**

Page Frames

**slab head**

Page Frames

**slab head**

**slab head**

Frames

**slab head**

**nodelist [ ]**

**objsize**

**array**

entry[ ]

**array\_cache**

**array\_cache**

entry[ ]

**array\_cache**

**…**

**CPU3**

**CPU2**

**CPU1**

**CPU0**

**cache\_cache**

通过上述结构，建立了三层的对象内存分配体系。

* 为每个 CPU 建立了可用对象的高速缓存（array\_cache 中的 entry）
* 每个 slab 中的可用对象数组
* 使用伙伴系统分配新的 slab，以建立对象数组

当进行指定对象内存分配时，首先从 cache\_chain 链表中，根据 objsize、name 等信息找到合适的对象缓冲区 kmem\_cache，然后根据 CPU ID 在 array\_cache 中判断此 CPU 目前是否有此空闲对象，如果有则分配，并修改相应数据结构。这是因为 array\_cache 中指向的对象为 CPU 最近使用完的对象，其很可能还保存在 CPU 的 cache 中，因此重用此对象速度会很快。

然后，如果在 array\_cache 中未找到合适的对象指针，则需要在 partial 的 slab 双向链表中寻找可分配的对象，如找到则返回，并修改相应结构；如未找到，则到 free 的 slab 双向链表中寻找，如找到则返回，如未找到，则需要调用伙伴内存管理方法申请一些新的 slab。

在一般情况下，slab 中管理的对象并不释放给系统。释放的对象只是标识为空闲，以供下次重用。因此，slab 在最大程度上避免了伙伴管理方法中小对象分配引起的内部内存碎片问题；slab 分配器还支持通用对象的初始化，从而避免了为同一目的而对一个对象重复进行初始化；slab 通过三个层次的对象内存分配体系以及硬件缓存对齐和着色技术，使得它在分配和管理对象时能够充分利用 CPU 中的 cache 和已经分配的对象缓存，从而达到提高对象内存分配、初始化和回收的速度。

###### Linux 用户库函数管理内存方法

用户程序通过 malloc 和 free 等函数来分配和释放内存。为支持 malloc 和 free 等内存管理函数，目前存在多个版本的 C 语言运行库，其中比较著名的有 ptmalloc2，jemalloc15， Hoard malloc16和 Thread-caching malloc（tcmalloc）17等。

在 linux 中 malloc 和 free 存在于 glibc 库中，默认采用的是 Wolfram Gloger 和 Doug Lea 编写的 ptmalloc2 版本。本节将讲解 ptmalloc 库函数通过什么数据结构来管理从内核获得的内存，以及如何来进行内存分配和释放。

此版本的 malloc 用两种方式来获得内存空间，如图 3-16 所示。一个是通过 brk 系统调用来移动逻辑 heap 段的指针来分配或缩减地址空间；另一个是通过系统调用 mmap，在内存映射段开辟新的逻辑地址空间，如系统所属。根据前面讲解，这两个系统调用都是完成

逻辑地址分配和页表设置，然后通过缺页中断，由操作系统内核来完成实际的物理内存分配。

15 [http://jemalloc.net](http://jemalloc.net/)

16 https://en.wikipedia.org/wiki/Hoard\_memory\_allocator

17 https://github.com/gperftools/gperftools

**main\_arena**

brk

**text data bss**

**…**

**heap**

mmap

**heap(thread) heap(thread)**

**heap(thread)**

**stack**

**physical space**

**896M**

**safe space(8M)**

**virtual space**

**malloc**

**thread\_arena memory mapping**

**进程逻辑地址空间**

图 3-16 malloc 内存逻辑地址分配示意图

###### 内存空间组织方式

为描述由这两个系统调用从内核申请到的内存空间，malloc 使用了如下的数据结构：

malloc\_state，heap\_info 和 malloc\_chunk， 具体见下面的代码。

//glibc/malloc/malloc.c struct malloc\_state

{

/\* Serialize access. \*/ libc\_lock\_define (, mutex);

/\* Flags (formerly in max\_fast). \*/ int flags;

/\* Fastbins \*/

mfastbinptr fastbinsY[NFASTBINS];

/\* Base of the topmost chunk -- not otherwise kept in a bin \*/ mchunkptr top;

/\* The remainder from the most recent split of a small request \*/ mchunkptr last\_remainder;

/\* Normal bins packed as described above \*/ mchunkptr bins[NBINS \* 2 - 2];

/\* Bitmap of bins \*/

unsigned int binmap[BINMAPSIZE];

/\* Linked list \*/

struct malloc\_state \*next;

/\* Linked list for free arenas. Access to this field is serialized by free\_list\_lock in arena.c. \*/

struct malloc\_state \*next\_free;

/\* Number of threads attached to this arena. 0 if the arena is on the free list. Access to this field is serialized by

free\_list\_lock in arena.c. \*/ INTERNAL\_SIZE\_T attached\_threads;

/\* Memory allocated from the system in this arena. \*/ INTERNAL\_SIZE\_T system\_mem;

INTERNAL\_SIZE\_T max\_system\_mem;

};

#define top(ar\_ptr) ((ar\_ptr)->top)

/\* A heap is a single contiguous memory region holding (coalesceable) malloc\_chunks. It is allocated with mmap() and always starts at an address aligned to HEAP\_MAX\_SIZE. \*/

typedef struct \_heap\_info

{

mstate ar\_ptr; /\* Arena for this heap. \*/ struct \_heap\_info \*prev; /\* Previous heap. \*/ size\_t size; /\* Current size in bytes. \*/

size\_t mprotect\_size; /\* Size in bytes that has been mprotected

PROT\_READ|PROT\_WRITE. \*/

/\* Make sure the following data is properly aligned, particularly that sizeof (heap\_info) + 2 \* SIZE\_SZ is a multiple of MALLOC\_ALIGNMENT. \*/

char pad[-6 \* SIZE\_SZ & MALLOC\_ALIGN\_MASK];

}

struct malloc\_chunk {

INTERNAL\_SIZE\_T mchunk\_prev\_size; /\* Size of previous chunk (if free). \*/ INTERNAL\_SIZE\_T mchunk\_size; /\* Size in bytes, including overhead. \*/

struct malloc\_chunk\* fd; /\* double links -- used only if free. \*/ struct malloc\_chunk\* bk;

/\* Only used for large blocks: pointer to next larger size. \*/

struct malloc\_chunk\* fd\_nextsize; /\* double links -- used only if free. \*/ struct malloc\_chunk\* bk\_nextsize;

};

/\* arena\_get() acquires an arena and locks the corresponding mutex.

First, try the one last locked successfully by this thread. (This

is the common case and handled with a macro for speed.) Then, loop once over the circularly linked list of arenas. If no arena is

readily available, create a new one. In this latter case, `size'

is just a hint as to how much memory will be required immediately in the new arena. \*/

#define arena\_get(ptr, size) do { \

ptr = thread\_arena; \

arena\_lock (ptr, size); \

} while (0)

#define arena\_lock(ptr, size) do {

if (ptr && !arena\_is\_corrupt (ptr)) libc\_lock\_lock (ptr->mutex);

else

ptr = arena\_get2 ((size), NULL);

} while (0)

\

\

\

\

\

/\* find the heap and corresponding arena for a given ptr \*/

#define heap\_for\_ptr(ptr) \

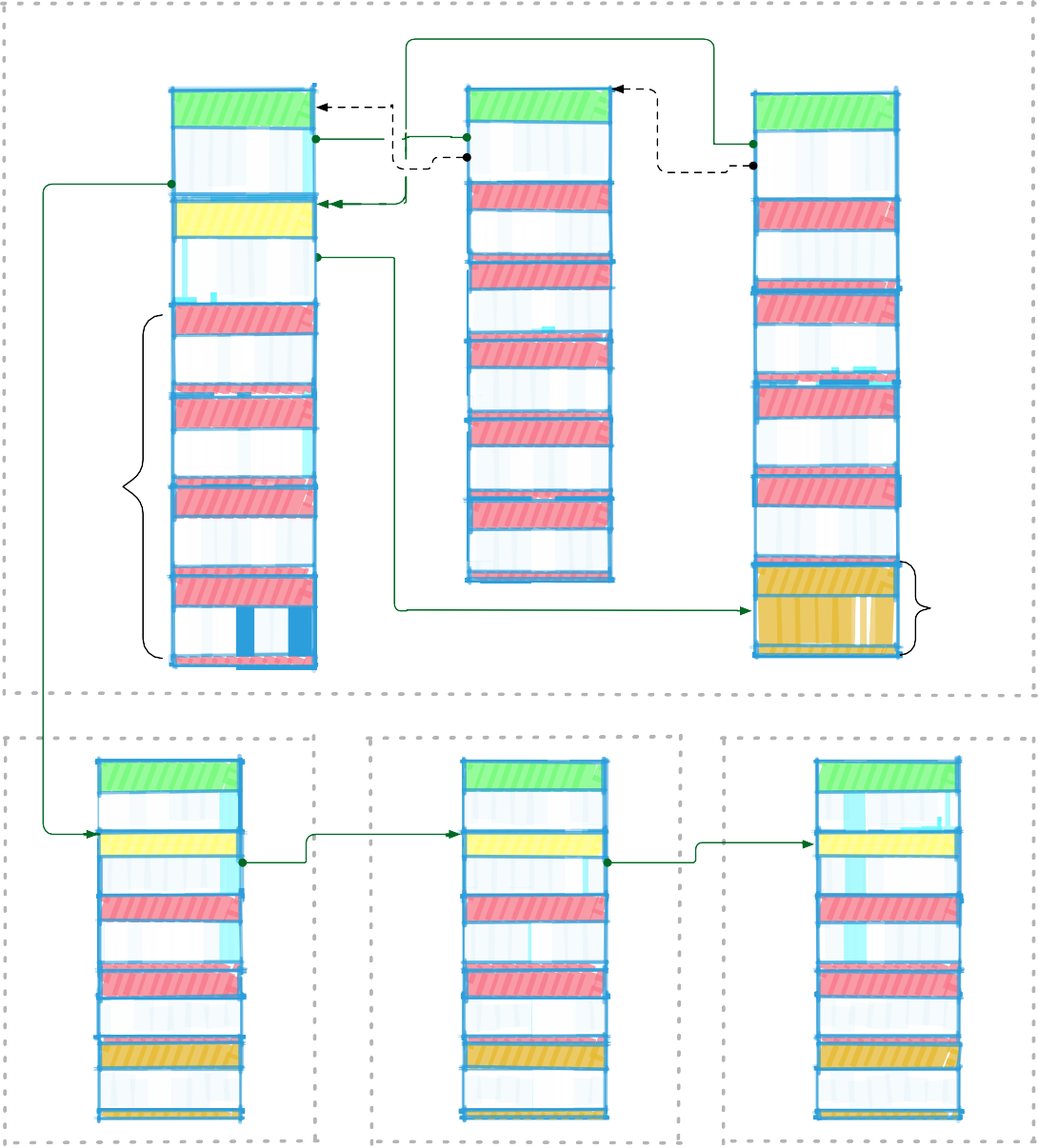
((heap\_info \*) ((unsigned long) (ptr) & ~(HEAP\_MAX\_SIZE - 1))) #define arena\_for\_chunk(ptr) \

(chunk\_main\_arena (ptr) ? &main\_arena : heap\_for\_ptr (ptr)->ar\_ptr)

这些数据结构主要辅助 malloc 完成如下的内存逻辑结构，以合理地组织和管理内存。这些内存逻辑结构包括：arena、heap 和 chunk，如图 3-17 所示。其中，arena 能够将多个heap 的组织在一起，为一个或多个线程分配内存或删除内存；heap 是一段连续的内存块（逻辑地址连续的段）；chunk 是在 heap 中分配的一块内存段

malloc\_state 描述了 area，其能够将多个 heap 组织起来，并通过 fastbinsY 和 bins 数组来指向 heap 中的 chunk。具体如下图所示。malloc\_state 的 top 指向了最新分配 heap 空间中位于最高地址的 chunk（top chunk）。通过这个 top 指针能够获得 top chunk 所在的 heap

（heap\_for\_ptr），并通过 heap\_info 的 pre 指针来遍历在这个 arena 已经创建好的 heap。每个heap 的头信息heap\_info 中ar\_ptr 指针指向了这个arena 的描述头信息结构malloc\_state。Arena 通过 malloc\_state 中的 next 指针指向下一个 arena，因此可以形成一个 arena 链表。



**thread\_arena#1**

0xb7900000

**Heap#1**

**heap\_info**

**Heap#2**

**heap\_info**

**Heap#2**

**heap\_info**

next

ar\_ptr

ar\_ptr

pre

ar\_ptr

pre

**malloc\_state**

**malloc\_chunck**

**malloc\_chunck**

**Area**

top

**malloc\_chunck**

**malloc\_chunck**

**malloc\_chunck**

**chunk malloc\_chunck**

**malloc\_chunck**

**malloc\_chunck**

**…**

**chunks**

**…**

**malloc\_chunck**

**…**

**malloc\_chunck**

**malloc\_chunck**

**Top chunk**

0xb7921000

**thread\_arena#2 Heap#1**

**heap\_info**

**thread\_arena#… Heap#1**

**heap\_info**

**thread\_arena#n H**

**p**

**malloc\_state** next

**malloc\_state**

next

**lloc**

**malloc\_chunck**

**malloc\_chunck**

**c**

**…**

**…**

**malloc\_chunck**

**malloc\_chunck**

**c\_chunck**

**lo**

**mal**

**…**

**nck**

**hu**

**\_c**

**lo**

**mal**

**te**

**sta**

**\_**

**ma**

**nfo**

**\_i**

**a**

**he**

**eap#1**

图 3-17 Malloc 内存组织结构图

Arena 包括两种形式，一种为 main\_arena，主要与用户进程空间中的 heap 段相对应； 另一种为 thread\_arena（如图 3-17 所示），主要为每个或多个线程来管理其所需的内存。 main\_arena 的结构形与 thread\_arena 结构稍有不同，如图 3-18 所示。在 main\_arena 中仅存在一个 heap，这个 heap 就是逻辑地址空间中的 heap 段。在这个 heap 中没有 heap\_info 数据结构，全部为 chunks。在 malloc 中通过一个全局变量 main\_arena 中的 top 指针来指向这个 heap 的 top chunk。

图 3-18 Main\_Arena 结构图

**Main Heap**

0x8042000

**Heap**

**malloc\_chunck**

**malloc\_chunck**

**main\_arena**

**malloc\_state**

top

**malloc\_chunck**

**malloc\_chunck**

**…**

**malloc\_chunck**

**Top chunk**

0x8067000

malloc 为每一个线程创建一个 arena（thread\_arena），这个线程将使用这个 arena 所表示的内存空间。这样组织内存最大的好处是使得每个线程尽量有自己的内存空间管理结构， 以避免因多个线程使用同一个内存空间而导致的等待（在多个线程使用同一个 arena 时，如果一个线程占用这个结构，则其它线程则需要等待，以保证 arena 中内存组织结构的一致性）。

但是，malloc 创建的 arena 数量是有限的，在 32 位平台下 arena 最大数量为“2\*cpu 内核数量”；在 64 位平台下 arena 最大数量为“2\*cpu 内核数量”，当创建的线程数量超过了 arena 最大数量限制时，将新创建的线程绑定到已经创建好 arena。如果这个已创建好的arena 正在被使用，则需要通过 arena 链表寻找目前处于空闲状态的 arena，并将这个空闲的 arena 分配给这个线程。如果寻找不到空闲的 arena 则阻塞这个线程。

chunk 描述了分配给用户进程的内存空间。chunk 使用两种数据结构来表示其是处于空闲状态还是占用状态。malloc 使用显性链表来组织这些 chunks。当 chunk 处于空闲时，其内部结构如下图 3-19 中绿色方块所示，通过 fd 和 bk 指针来组成空闲块的双向链表。其中的 pre\_size 指的是上一个 chunk 中 user data（已分配空间）或 unused space（为分配空间） 的大小，而 size 指的是本 chunk 中的空间大小。

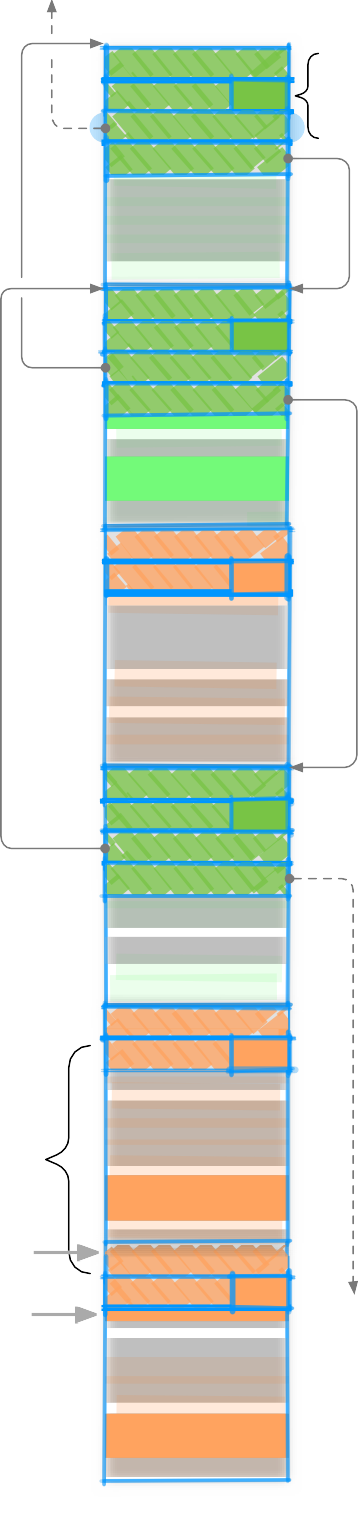
**pre\_size**

**size AMP**

**bk**

**fd**

A—1: mmap; 0: heap

M—1: ignore other bits; 0 P—1: prev\_inuse; 0: free

unused space

**pre\_size**

**size**

**000**

**fd**

**bk**

unused space

**pre\_size**

**size 000**

user data

**pre\_size**

**size 001**

**fd**

**bk**

unused space

**pre\_size**

**size**

**000**

user data

**chunk**

footer

**size**

**001**

**pre\_size**

returned by malloc

user data

图 3-19 chunks 链表结构

如果在生成 chunk 时，都将这个 chunk 可分配大小写入 size 和下一个 presize 中，则这个 presize 可以充当 footer。通过 footer-边界标记技术，使得在回收内存时，很容易找到上一个块。例如，图中倒数第二个 chunk 中的 user\_data 被释放时，通过此 chunk 的 P 标志位知道紧邻它的上一个 chunk 为空闲状态，这时根据 pre\_size 所指示的大小，经过指针偏移可

以得到上一个 chunk 的 malloc\_chunk 结构，通过修改这个结构的 size 来达到将相邻两块

chunk 合并的目的。

###### 多种长度的空闲链表组织

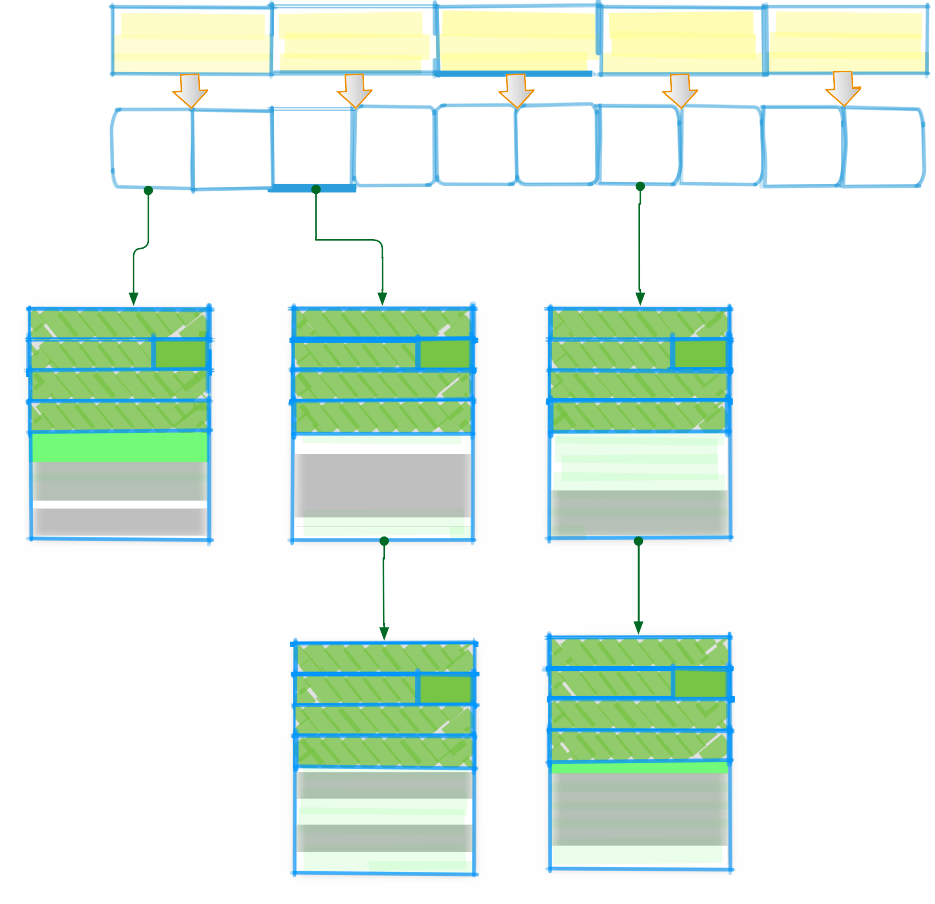
如果所有的空闲 chunk 使用一个双向链表进行组织，那么在申请一块空间时，需要遍历此链表，这样使得分配内存效率较差。为提高寻找合适 chunk 的速度，可以按照 chunk 的大小，对所有的空闲 chunk 进行组织。

在 malloc 中使用桶（bins）结构来组织空闲的 chunk。具体做法是，根据空闲的 chunksize

大小将其放入不同的桶中（单向或双向链表）。在 arena 中保存两个桶结构fastbinsY 和bins。

###### fastbin

fastbinsY 用于存放空闲的“小”chunk（size 大小在 16~80 字节之间。malloc 在 arena 的 fastbinsY 结构基础上封装了 fastbin，用于小内存的分配和释放管理。在 fastbin 中，每个桶存放的 chunk 大小相差为 8 个字节，第一个桶存放为 16 字节 chunk 的链表，第二个为 24 字节 chunk 的链表，依次类推，最多能存放 80 字节 chunk 的链表。但是在默认初始化时， fastbin 中保存的最大 chunk 为 64 字节，具体结构如下图 3-20 所示。



**fastbin**

**1**

**2**

**…**

**6**

**…**

**arena**

**.fastbinsY**

**0**

**3**

**4**

**12**

**…**

**…**

**…**

**fd=** 0x804b040

unused space

0x804b040

unused space

48 bytes

8 bytes

unused space

**bk**

**fd=** NULL

**AMP**

**24**

**pre\_size**

**bk**

**fd=** NULL

**AMP**

**64**

**pre\_size**

48 bytes

unused space

**bk**

**fd=** NULL

**AMP**

**64**

**pre\_size**

unused space

8 bytes

**bk**

**AMP**

**24**

**pre\_size**

0 bytes

**bk**

**fd=** NULL

**AMP**

**16**

**pre\_size**

**2**

**1**

图 3-20 fastbin 内存管理结构图

每个桶中存放的是 chunk 的单向链表。在分配内存和回收 chunk 时采用后进先出（LIFO） 策略，即在每个桶的链表首部删除或者增加 chunk。需要注意的是，为提高速度，在 fastbin 中不对地址相邻的空闲 chunk 进行合并。这虽然会加快外部碎片产生速度，但能够提高内存分配速度。

在进程运行初始阶段，fastbin 中各个桶中为空，当用户在申请小内存时，首先在下面的bins 桶中的 small 区域进行查找，在分配给用户合适的内存后，剩余内存所对应的 chunk 将被插入 fastbin 相应桶中的链表。

###### Bins

malloc bins 与 arena 结构（malloc\_state）中的 bins 相对应，在 malloc bins 中每一项分别对应 arena bin 中的两项，如下图 3-21 所示。与 fastbins 不同，bins 中每个桶均为双向链表（通过 malloc\_chunk 中的 fd 和 bk 指针实现）。

在 malloc bins 中包含 unsorted bins、small bins 和 large bins 三种区域，共占用了 126 个桶。其中 unsorted bin 只有一项，在 bins[1];small bins 共有 62 项，在 bins[2]~bins[63]； large bins 共有 63 项，在 bins[64]~bins[126]。

Unsorted bin 用于存放回收的 chunk。small chunk 或 large chunk 在处于空闲状态时， 将被加入 unsorted bin。这是因为考虑到这些空闲的 chunk 可能会被马上重用。

Small bin 用于存放小于 512 字节的 chunks。small bins 主要用于小内存的分配，在每个桶中保存大小相同的 chunk 双向链表。相邻桶的 chunk 大小相差 8 个字节。例如，在 bins[2] 中保存 16 字节的 chunk 双向链表；bin[3]保存为 24 字节的 chunk 双向链表。以此类推， bins[63]中保存 504 字节的 chunk 双向链表。

Large bin 用于存放大于或等于 512 字节的 chunk。每个 large bin 一个双向链表，用于存放大小在一定范围区间的空闲 chunk。与 small bin 不同的是，在 large bins 中每个桶存放着大小不等的空闲 chunk。因此对这些空闲 chunk 进行“从大到下”的排序。

63 个 large bin 存放空闲 chunk 具体分配方案如下：

32 个桶，相邻两个桶中保存的 chunk 大小范围相差 64 个字节。例如 bins[64]中包含大小在 512~568 字节的 chunk；bins[65]中包含大小在 576~632 字节的 chunk。

16 个桶，相邻两个桶中保存的 chunk 大小范围相差 512 个字节

8 个桶，相邻两个桶中保存的 chunk 大小范围相差 4096 个字节

4 个桶，相邻两个桶中保存的 chunk 大小范围相差 32768 个字节

2 个桶，相邻两个桶中保存的 chunk 大小范围相差 262144 个字节

1 个桶，保存余下大小的 chunk。

当用户请求的内存需要在 large bin 区域的桶中分配时，首先根据请求内存大小确定具体的桶，然后在这个桶所指向的双向链表尾部，按照“从小到大”的最佳适应方法查找合适

**unsorted bin**

**small bins** #62

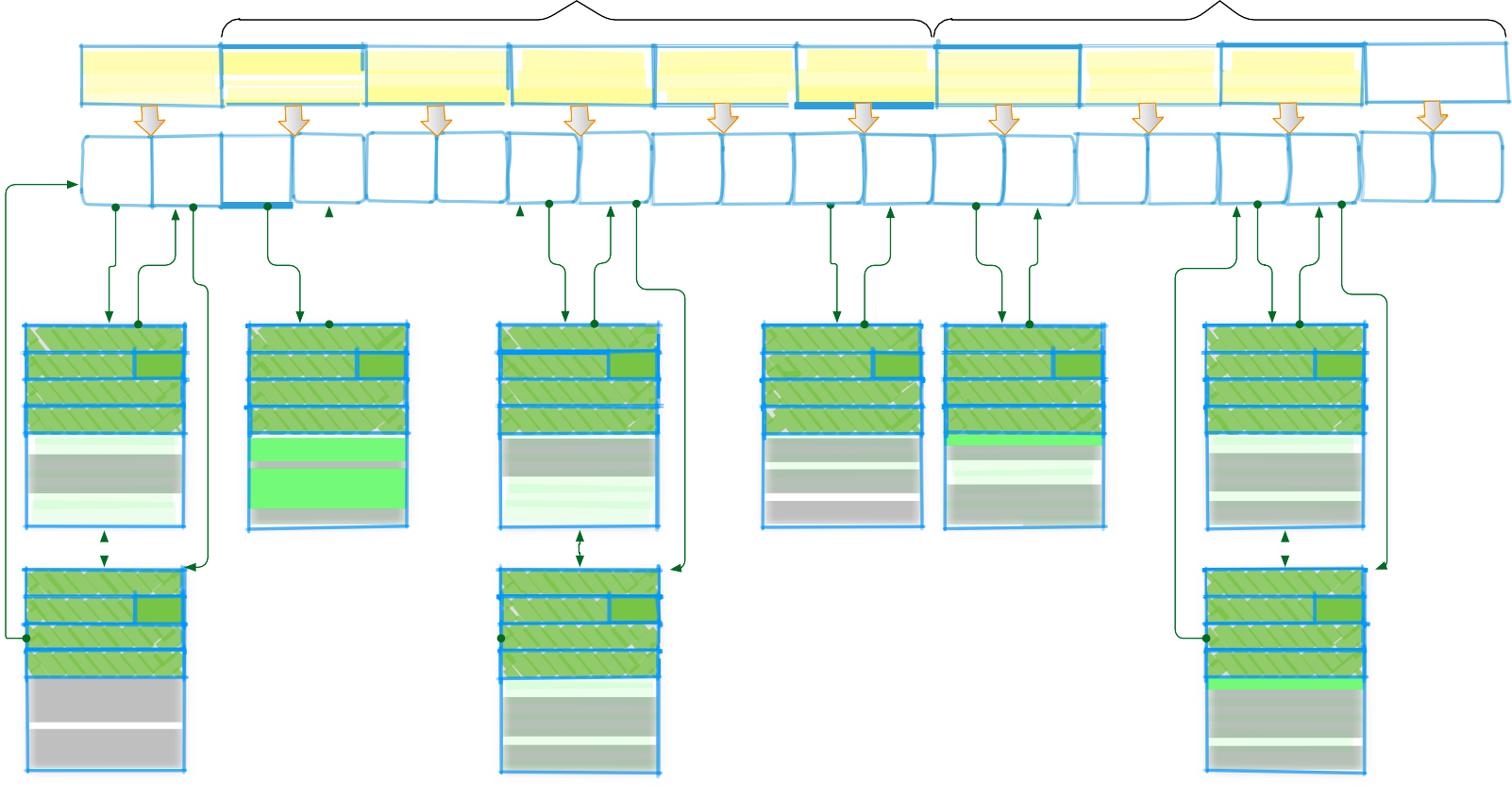
**large bins** #63

的 chunk。此 chunk

将被分为两部分，一部分用于返回给用户使用；剩余部分将加入到

unsorted bin 中。

图 3-21 bins 内存管理结构图



**bins**

**…**

**arena**

**.bins**

**0**

**fd**

**3**

**bk**

**4**

**fd**

**12**

**fd**

**13**

**bk**

**…**

**…**

**…**

**…**

**…**

**…**

**…**

**…**

**…**

**…**

**…**

**…**

**pre\_size**

**16 AMP fd**

**bk**

unused space 0 bytes

**1400 AMP**

unused space

unused space

**1344 AMP**

unused space

**bk**

unused space 1328 bytes

**fd**

**pre\_size**

32 bytes

**bk**

**fd**

**48**

**AMP**

**pre\_size**

1016 bytes

unused space

**bk**

**fd**

**AMP**

**1032**

**pre\_size**

1384 bytes

**bk**

**fd=** NULL

**pre\_size**

488 bytes

496 bytes

unused space

**bk**

**bk**

**fd**

**fd**

**AMP**

**512**

**AMP**

**504**

**pre\_size**

**pre\_size**

32 bytes

unused space

**fd**

**bk**

**AMP**

**48**

**pre\_size**

48 bytes

unused space

**bk**

**fd**

**AMP**

**64**

**pre\_size**

**2**

**fd**

**1**

**bk**

**…**

**77**

**…**

**64**

**63**

**6**

**…**

**2**

**1**

* **Top** chunk

Top chunk 位于 arena 最顶端，它即不属于 fastbin 也不属于 bin。当用户请求的内存大于上面 fastbin 和 bins 中所保存的内存空闲块，并且小于 top chunk 大小时，top chunk 将分裂为两个部分，一个用于用户请求的内存；剩余的将变为新的 top chunk。如果 top chunk 也不能满足用户所请求的内存，则调用 sbrk 或者 mmap 函数来进行内存扩展。

###### Last Remainder Chunk

Last Remainder Chunk 与 top chunk 类似，不属于上述上述两种桶结构。在分配一个small chunk 时，如果在 small bins 中找不到合适的 chunk，并且 last remainder chunk 满足要求，则将 last remainder chunk 分裂为两部分，一个用于用户请求的内存；剩余的将变为新的 last remainder chunk。此项值能够提高 malloc 连续分配小 chunk 的速度。

* + - 1. Malloc **内存分配或回收**

###### 分配算法

在进程初始状态，heap 的大小为 0。如果第一次用户请求的内存大小小于 mmap 分配阈值时，malloc 会申请 128kb+sizeof(malloc\_chunk)的内存给 heap；如果大于 mmap 分配阈值，则直接使用 mmap 分配内存。

* 如果用户请求的内存大小小于 mmap 阈值，malloc 首先在 fastbin 中进行查找，如果找到将找到的 chunk 返回给用户；
* 如果找不到合适的 chunk，则在 small bins 中进行查找。如找到则将 chunk 返回给用户；
* 若还找不到，则将 fast bins 所有的 chunk 加入到 unsorted bin 中，并进行 chunk

合并；

* 然后在 unsorted bin 中进行查找。如果找到合适的 chunk，则返回给用户；如果在unsorted bin 中没有找到合适 chunk，则将 unsorted bin 中的所有 chunk 加入到small bins 或 large bins 中，并进行 chunk 合并；
* 然后在 large bins 中进行查找。如果找到，则分裂此 chunk 为两部分，一部分返回给用户；另一部分加入到 unsorted bin 中；
* 如果在 large bins 中也未找到，malloc 则会查看 top chunk。如果 top chunk 满足用户要求，则将 top chunk 分裂为两部分，一部分返回给用户；另一部分成为新的top chunk；
* 如果 top chunk 大小也不满足要求，则根据 mmap 阈值和用户请求内存大小，来决定采用 mmap 增加内存映射区以在对应 arena 中生成新的 heap；还是应用 sbrk 进行 heap 内存扩展以增加 arena 中 top chunk 的大小。

###### 回收算法

用户通过 free 函数所释放的内存，在一般情况下并没有被操作系统回收，而是被重新打包成 chunk，并放入到 malloc 相应的内存结构中，以供重用。但是如果释放的内存紧邻top chunk，使得它们合并起来足够大时，将通过 munmap 系统调用将这些内存返回给操作系统。具体释放过程如下：

* 如果回收的 chunk 足够小，则将它放入合适的 fastbin 中；
* 如果回收的 chunk 是经过 mmap 得到的大数据块，则将它通过 munmap 释放给操作系统；
* 查看此 chunk 是否有相邻的处于空闲的 chunk，如果有则合并它们成为新的chunk；
* 如果此 chunk 为 top chunk 则根据 top chunk 的大小来决定是否将内存释放给操作系统；如果此 chunk 不是 top chunk，则将它放入 unsorted bin 中；
* 如果此 chunk 足够大，则合并 fastbin 中所有的 chunk；然后看 top chunk 是否有足够大的空间，以给操作系统返回一些内存。由于性能原因，此步可能会被推迟到malloc 函数或其它函数调用时才完成。

###### Nginx 内存管理模型

Web 服务器为分析、处理用户发出的请求及发送响应消息，需要不断分配不同长度的内存。虽然可以通过库函数 malloc/free 来管理内存，但是在高并发环境下，需要服务器在处理大量用户请求过程中不断地使用 malloc/free 来申请和释放内存，这时使用此库函数并不高效。

这是因为 malloc/free 等库函数都是针对通用目的来进行设计的，其通过内部复杂的数据结构在一定程度上避免或减缓了内存碎片产生，但是其内部数据结构维护代价较大。尤其

在短时间内大量的内存申请和释放操作，会造成其内部数据结构的维护效率变低，从而导致 内存申请和释放的速度变慢。

Nginx 是目前常用的 HTTP 服务器，其针对处理用户请求消息及响应消息的特点，设计

了内存池结构，通过“多次分配，一次释放”的原则，来达到对内存高效使用的目的。通过 内存池，使得服务器在一定阶段中频繁地从内存池中申请内存，但是这些内存并不释放给操 作系统，并在此阶段过后，通过销毁内存池的方式一次性地将内存池中所有内存释放给操作 系统。这样设计的好处是，内存池数据结构简单，维护代价较低，从而提高内存管理的效率。 例如，假设 Web 服务器在 1 秒中内要处理 1000 个用户请求，处理每个请求需要申请

30 次内存。如果使用 malloc 和 free 库函数，则需要调用 30000 次 malloc 和 free 函数，每

次调用都会修改其内部数据结构（ptmalloc2），随着其内部的 chunk 增大，这些操作所引起的修改代价会越来越大。而在 Nginx 中，只需调用类似 malloc 的内存申请函数 30000 次和

1 次内存池释放函数。并且由于内存池结构简单，在 Web 服务器环境下，其分配和管理内存效率远远高于 malloc/free 库函数。

Nginx 中的内存池具体结构如下图 3-22 所示。在内存池中维护两个链表，一个是小内存链表，另一个是大内存链表。当用户需要申请的内存大于 max 时，则通过 malloc 库在系统中申请，然后生成 ngx\_pool\_large\_t 结构，并加入到大内存链表中；如果小于 max 时，则

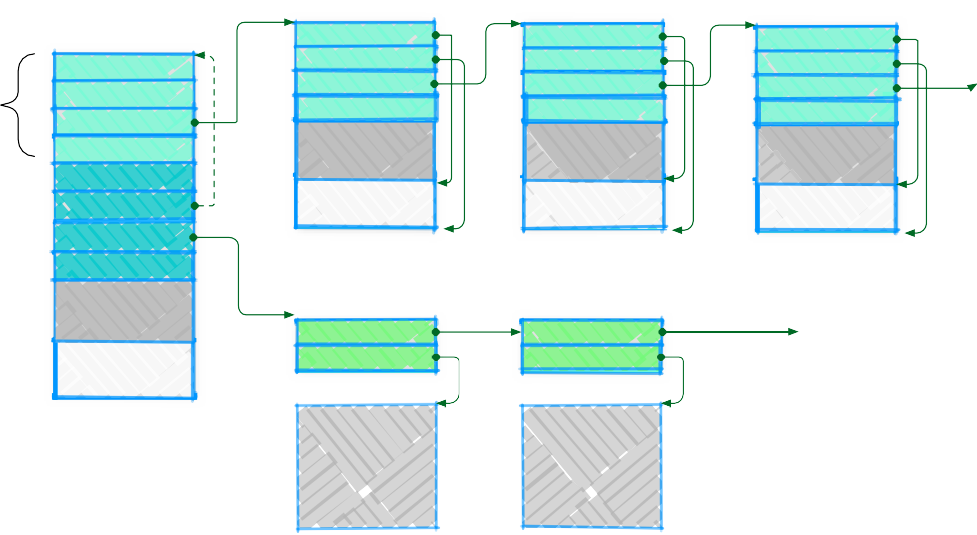
在小内存链表中寻找合适的内存区域进行分配。如果在小内存链表中没有适合用户申请大小的内存，则新建一个小内存节点（ngx\_pool\_data\_t），并插入到小内存链表，然后在给用户分配内存。

ngx\_pool\_data\_t

**failed**

**failed**

**failed**



ngx\_pool\_t

ngx\_pool\_data\_t

**last**

ngx\_pool\_data\_t

ngx\_pool\_data\_t

**last end next failed max current**

**large**

**…**

**used**

NULL

ngx\_pool\_large\_t

**next**

ngx\_pool\_large\_t

**next**

NULL

**free**

**alloc**

**alloc**

**free**

**used**

**failed**

**next**

**end**

**last**

**free**

**used**

**failed**

**next**

**end**

**last**

**free**

**used**

**failed**

**next**

**end**

**big memory**

**big memory**

图 3-22 Nginx 内存池结构图

由于此内存池在分配给用户内存后并不担心其释放内存，因此它只需记录目前内存已 经分配的情况（通过 last 和 end 指针来分别指向未分配内存的起始和结束位置）。当内存池需要销毁时，其将根据两个链表结构，依次释放从操作系统申请的内存。因此，此内存池不

会产生内存碎片。

从上述内存池数据结构及分配、释放内存方法来看，内存池使用“以空间换时间”策略。 其应用的合理假设前提为：在一段时间内，系统内存足够大，以致能够满足用户在不释放内 存情况下连续申请内存空间的要求。在应用此内存池结构时，一定要注意应用场景和环境。 其不能替代 malloc/free 等库函数作为一般应用场景下的内存管理方法来使用。

###### 实验 8 Web 服务器的内存管理

**题目 1**. 查询 tcmalloc 相关材料，写出其组织和管理内存的结构，并说明其为什么在多线程环境下管理内存（分配和释放内存）的效率比 ptmalloc 高？

**题目 2**. 根据上面对 Nginx 中内存池的描述，实现与此描述类似的内存池（可以参考

Nginx 相关源代码）。注意此内存池要支持内存对齐和多线程。

**题目 3.** 设计测试代码，在不同内存申请和释放情况下，对比 malloc/free 和内存池的内存申请和释放效率。（比如，连续分配 300000 个小内存，两种方法所需要的时间；连续分配和释放 300000 个不同大小的内存，malloc/free 函数所消耗时间，以及内存池完成连续分配300000 个不同大小的内存和一次释放这些内存所消耗的时间；…）

**题目 4**. 通过指定数量的线程或任务共享一个内存池的方式，来修改前面 Web 服务器中申请和释放内存的代码。每个任务都从指定的内存池申请内存。当使用一个内存池的任务 全部完成后，才释放这个内存池。例如，指定 k 个任务共享一个内存池，当服务器中存在 10k 个任务时，服务器就会创建 10 个内存池。当内存池中 k 个任务完成后，这个内存池就是被释放。

**题目 5**.根据前面介绍的 linux 内核内存模型和 ptmalloc2 用户内存管理模型，设计合理的内存管理结构，以支持实验 7 中缓存 Web 页面的 hash 结构对内存的使用和释放。在设计此内存管理结构时，需要考虑性能问题和碎片问题，以及两种之间的折中和权衡。

# 第 4 章 Web 服务器的文件存储系统

## 背景介绍

此 Web 服务器主要访问的是 html 页面以及与页面相关的文件（图像文件，声音文件等）。这些文件具有数量众多、长度较小（几十 KB~几百 KB）、不经常修改（一般情况下只读，或批量删除）等特点。linux 文件系统（Ext2 或 Ext3）虽然能够支持存储、读取和删除这些海量的小文件，但由于这些文件系统一般是针对通用文件及其操作来进行设计的，从而

会造成在海量小文件中查找并读取文件速度较慢、存储空间利用率较低等问题。为解决上述 问题，本章将探讨如何设计合理的文件系统来高效地对存储和查询这些小文件。

本章首先介绍 linux 系统中常用的 Ext 文件系统，通过分析它的结构特点，说明其在存

储和读取只读海量小文件时的缺点；然后通过介绍淘宝的图片文件系统结构，来说明如何来 进行海量小文件存储。最后通过布置的作业，来实现一个能够进行小文件存储和查询的文件 系统。

* 1. Linux **中的** Ext **文件系统**

Linux 下常用的文件系统为 Ext 系列（Ext2、Ext3 和 Ext4）文件系统。本节通过介绍 Ext2

的文件系统结构来分析它在存储和查询小文件时的不足。

###### Ext2 文件系统结构

Ext2 文件系统将磁盘分为大小相等的块（block）。对于磁盘上的启动扇区（boot block）内部存储的是操作系统引导程序。这段程序在开机自检后，通过 BIOS 加载入指定内存并运行。磁盘上其余的空间被分为块组（block group）。在每个块组中包含超级块（super block）、组描述符（group desc）、数据位图、inode 位图、inode 表和数据块等内容，如下图 4-1 所示。

其中，超级块用于存储此文件系统自身的属性信息，包括空闲块和使用块数目、块大小、 各种时间戳、版本等信息。超级块描述了文件系统的核心信息，为了防止此块数据被破坏， 超级块的备份将稀疏地保存在不同块组中，例如出现在以块组 0、1 和其它以 3、5、7 为幂

的租客中。

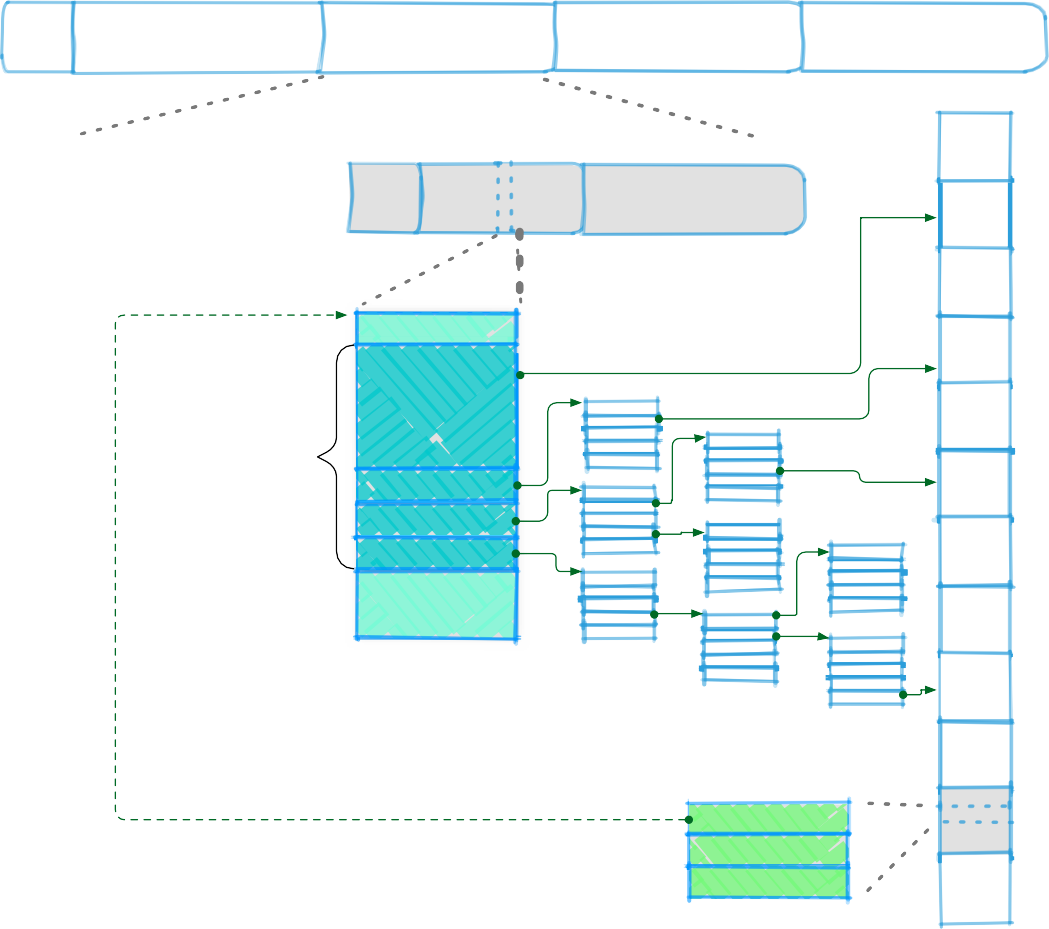
组描述反映了各个组块状态的信息，包括组块中的空闲块和 inode 数目。需要注意的是每个组块的组描述都包含了文件系统中所有组块的状态信息。组描述占据着 k 个块。

数据位图和 inode 位图分别占用一个块，它们内部为一个比特串，通过不同位上的 0 或

1 来表示后面每个数据块和 inode 节点使用情况。

在 inode 表中保存此文件系统中所有的 inode 信息，此使用情况可以通过前面的 inode

位图来进行管理。数据块中保存文件或目录的数据。



**boot**

**block**

**block group 1**

**block group 2**

**block group n**

block # 1

**super block**

blocks # k block # 1 block # 1 blocks # m

blocks # n

**..**

**inode table**

**data blocks**

data block

i\_node

**…**

**…**

**i\_block[15]**

…

…

…

…

ext2\_dir\_entry\_2

**…**

**…**

**…**

**name #255**

**inode**

**…**

**treble indirect**

**double indirect**

**single indirect**

**12**

**direct**

**…**

|  |  |  |
| --- | --- | --- |
| **group desc** | **data bitmap** | **inode bitmap** |

图 4-1 Ext2 文件系统结构

在 Ext 文件系统中，将文件和目录统一看为文件，每个文件都至少有一个 inode 来描述它的元信息（长度、访问时间、创建时间、修改时间、文件属性等）和数据所在块的位置。Ext2 文件系统为了支持不同长度的文件，在 inode 节点中使用了混合文件索引结构。其中， 数组 i\_block 的前 12 项保存的是数据所在的块号，后三项分别指向了 1、2 和 3 级的文件索

引结构所在块。

如果 inode 表示的是一个目录，则在块中存储的是目录项（ext2\_dir\_entry\_2），此目录项中包含文件名（长度为 255 的字符数组）、文件所对应的 inode 编号、名称长度、目录项长度和文件类型。

###### Ext2 文件系统分析

* + - * **存储空间效率**

Ext2 文件系统以块为最小的存储单元，与内存页管理一样，会产生内部碎片和外部碎片。内部碎片指的是块内部空间并没有被一个文件完全利用。外部碎片指的是连续空闲块长 度较小，不能满足文件连续分配块的要求。

针对海量的小文件（网页及其图片），使用 Ext2 文件来进行存储将会造成极大的浪费。假设每个块的大小为 1KB，1 个 1.5KB 大小的网页文件将占用 2 个块。如果一个文件系统存储的全部是这样的小文件，将浪费 25%的存储空间。

除此之外，由于文件比较小而且为只读，在 inode 节点中并不需要 2 级和 3 级文件索引块指针以及文件修改时间等一些属性信息，因此 inode 数据结构较为庞大，也造成了存储空间的浪费。

###### 文件读取效率

首先考虑所有的小文件全部保存在一个目录中的情况。由于小文件数量众多，从而导致 目录项过多，此目录文件大。因此，此目录文件所占块可能需要 inode 节点的 2 级或 3 级索引结构来进行管理。例如，一个块大小为 1KB，一个目录项为 256 字节，如果一个目录里面

有 1M 个小文件，那么此目录文件大小为 256MB，需要 3 级的索引结构。那么每次读取此

目录文件中不同块内容时，都需要读取 4 次块（3 次为读取索引结构块）。由于块所在设备为磁盘或者其它外部存储设备，其读取速度远远低于内存，因此从目录文件中读取目录项的 速度较慢，检索文件的速度也较慢。可以通过将 inode 索引结构缓存到内存来部分缓解此问题。另外，如果目录项过多，那么检索此目录中文件会消耗较多的时间，从而造成文件检索

速度变慢。

然后考虑小文件分散存放在不同目录的情况。在这种情况下，每个目录中存放的文件较 少，但是目录会较多。例如，每个目录存放 1K 个文件，那么 1M 个小文件将需要 1K 个目录。而这些目录的存放结构将影响文件的检索性能。假设使用 4 级目录结构来组织这些目录和文件，那么对其中一个文件的索引将多次读取目录数据块，这也会造成文件块检索速度下降。例如读取“/2017/8/1/web1.html”，将首先找到根目录 inode，然后索引到根目录下数据块，从中找到名字为“2017”的目录项，在此目录项中找到对应的 inode，然后在此 inode

所对应的数据块中寻找名字为“8”的目录项，…，重复以上过程直到找到 web1.html 所对

应的 inode 节点，在这个过程中，花费了大量时间来检索和读取目录 inode 和及其数据块。此问题也可以通过将 inode 目录节点及其数据放入到缓存中来得到部分缓解，但是随着

小文件的增多，由于缓存中不能存放所有目录节点的信息，以及由于访问文件的随机性导致 的缓存命中率下降，文件检索效率将逐步下降。

* 1. TFS **文件系统**

TFS(Taobao File System)是一个高扩展、高可用、高性能、面向互联网服务的分布式文件系统。它最初的设计目标是为淘宝提供海量小文件（不超过 1MB）的存储和快速地数据检索和读取服务。为实现此目标，TFS 将它的服务部署在计算机集群的各个节点上，以分摊海量用户并发读取数据的“压力”（系统负荷）。

* + 1. **TFS 文件系统架构**

TFS 主要包含两类服务：命名服务和数据服务。命名服务主要负责建立 block（文件块） 与数据服务之间的映射关系，以及维护各个数据服务的状态。数据服务主要负责完成对文件 的存储和读取功能。通常将命名服务部署在集群的两个节点上，此两个节点被称为

NameServer；将数据服务部署在集群中多个节点上，这些节点被称为 DataServer。根据用户对系统存储容量和读取数据速度的要求来决定 DataServer 节点的数量。TFS 文件系统的架构如图 4-2 所示。

###### 本地文件系统逻辑结构

TFS 构建在 linux 文件系统 Ext 之上，TFS 中每一个物理块对应于 LInux 文件系统下的一个文件。物理块主要用于存储用户上传的文件数据，多个用户上传的文件数据被存储到一个 物理块中。物理块分为两种：主块和扩展块。物理块的逻辑结构信息如上图中 main block 所示（具体见源代码中的 BasicPhysicalBlock 类和 PhysicalBlock 类）。

逻辑块由一个主块和多个扩展块组成。主块主要用于存储用户上传的文件数据；扩展块 用于存储文件更新时文件大小发送变化的文件数据。因此，在一个逻辑块中，主块是存储服务的“主力军”，一般情况下，其长度远大于扩展块的长度。逻辑块中每个物理块保存文件数据的情况还被记录在索引文件（index）中

索引文件（index）记录了一个逻辑中每个物理块保存文件数据的情况，如文件 id、文

件长度、偏移位置等，具体见图示中 FileInfo 结构。每个索引文件都与一个逻辑块向对应。超级块（super block）作用类似于 ext 文件系统中的超级块，但是在 TFS 中它以文件的

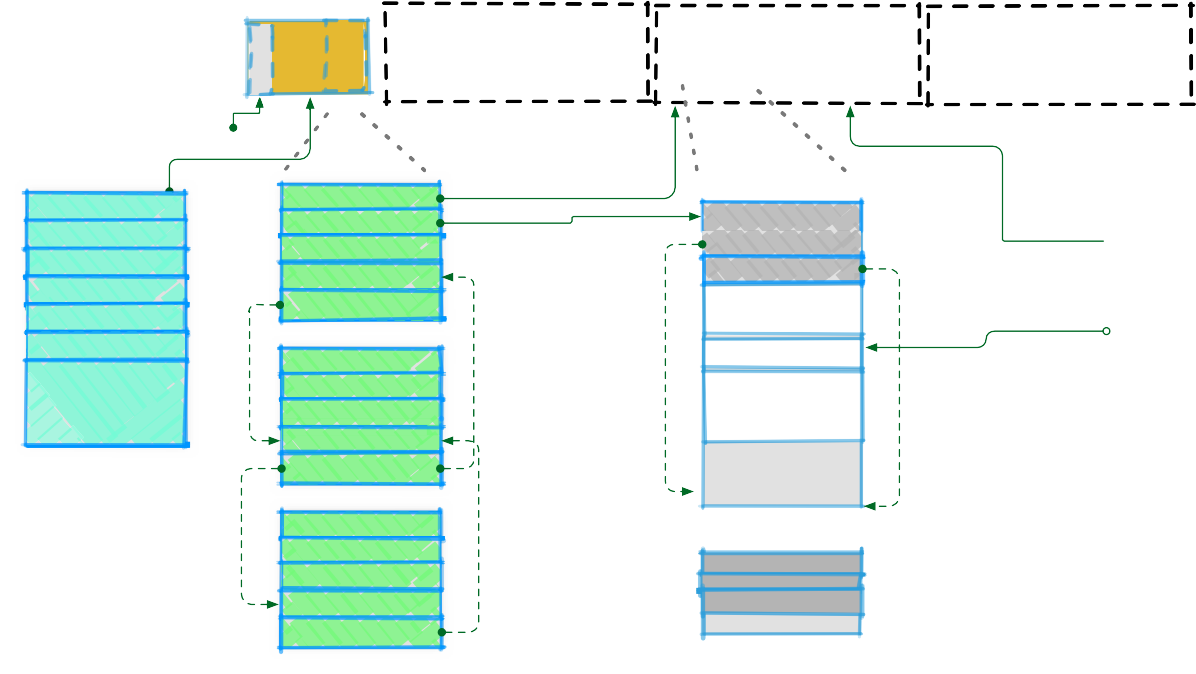
形式存在。其内部结构如上图所示。在超级块的首部预留一些字节，然后保存了此文件系统 的属性信息，例如版本、挂载位置、主块最大尺寸、扩展块最大尺寸等信息，具体见源代码 的 super\_block\_info 类。超级块的最后一部分保存了一组 blockindex 结构。这组 blockindex

组成了双向链表，其中的每一项有指向逻辑块的 id、物理块的 id 等内容。

super block info

**version**

reserve

next\_index

**phy\_file\_name\_id**

**phy\_block\_id**

**logic\_block\_id**

**index**

**phy\_file\_name\_id**

**phy\_block\_id**

**logic\_block\_id**

**…**

**max\_extand…\_size**

**max\_main…\_size**

**block\_index\_offset**

**…**

**mount\_point**

next\_index

**index**

block index

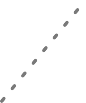
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| main | ext | · | ext |  | index |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| main | ext |  | · | ext |  | index |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| main | ext | · | ext |  | index |

**…**

**…**

pre\_index

pre\_index

**logic\_block\_id**

main block

**physical\_block\_id**

**start**

**file1\_data**

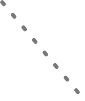
**file2\_data**

**file…\_data**

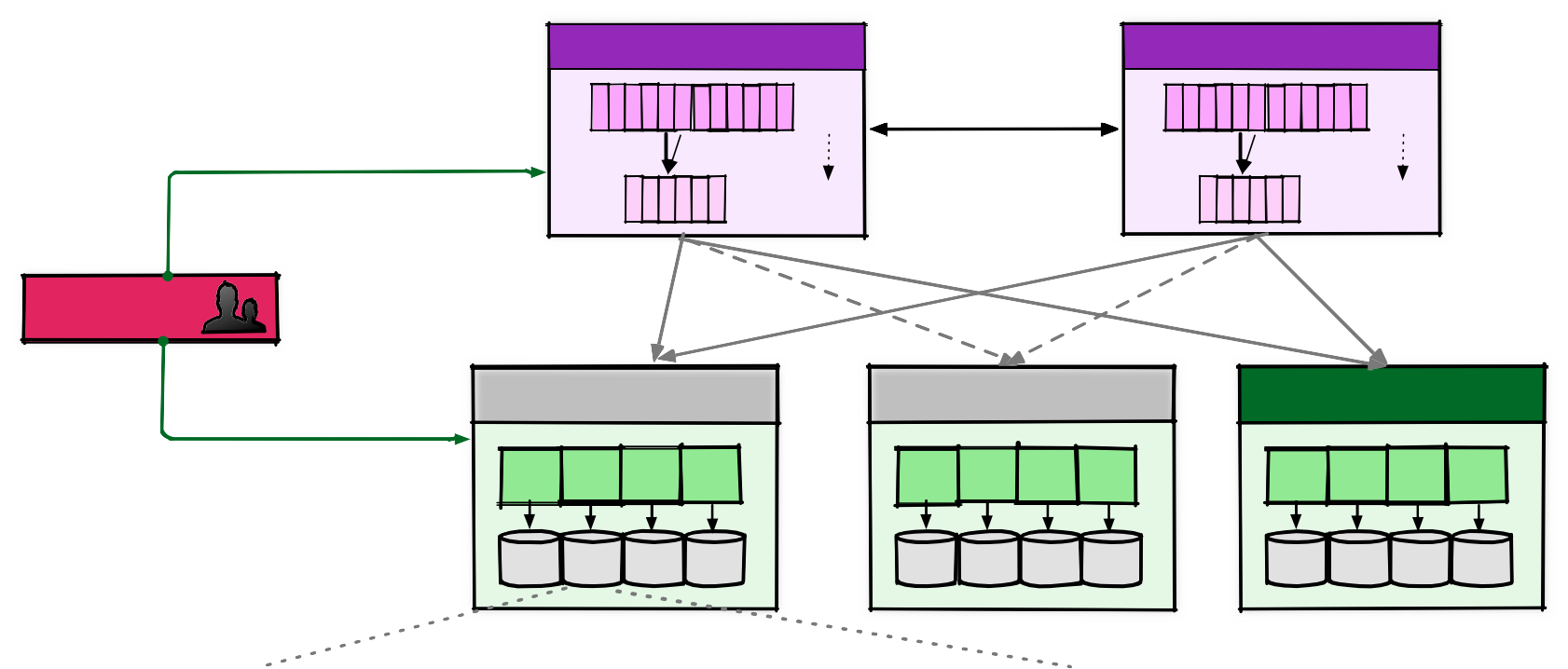
ext\_block

**end**

index



|  |
| --- |
| **index header** |
| **file\_info #1** |
| **file\_info #2** |
| **file\_info #…** |
| **file\_info #n** |



sync data

block id, file id

allocate

dataserver id block id, file id

super block

logic block #1

logic block #…

logic block #1

hda hdb … hdn

dsp1 dsp2 … dspn

DataServer # n

hda hdb … hdn

dsp1 dsp2 … dspn

DataServer # …

hda hdb … hdn

dsp1 dsp2 … dspn

DataServer # 1

data server

block id

NameServer-slave

data server

block id

NameServer

**Application/client**

图 4-2 FTS 文件系统结构

**phy\_block\_id**

**phy\_file\_name\_id**

**index**

uint64\_t id\_; // file id

int32\_t offset\_; // offset in block file int32\_t size\_; // file size

int32\_t usize\_; // hold space

int32\_t modify\_time\_; // modify time int32\_t create\_time\_; // create time int32\_t flag\_; // deleta flag

uint32\_t crc\_; // crc value;

**FileInfo**

**…**

…… ……

###### 文件命名

TFS 文件命名使用了对象存储的概念，用户存取的每个文件名字包含了此文件所在的集群代码、block id 和文件 id 等内容。文件名的编码方式如下图所示。

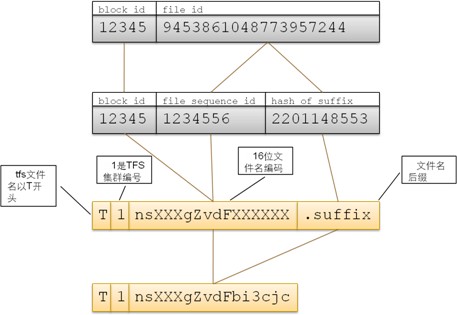


图 4-3 TFS 文件命名编码18

TFS 文件名最大长度为 18 个字节，文件名第一个字节为 T，第二个字节为集群编号（1~9），余下的字节由 block id 和 file id 通过一定的编码方式来获得。因此，在已知文件名情况下， 根据编码规则可以很容易得到此文件的 id 和所在 block 的 id。

###### NameServer

NameServer 完成分布式系统中的命名服务功能，在 TFS 中主要包括维护 blockid 与DataServer 之间的映射表，并通过心跳服务来维护各个 DataServer 的运行状态。NameServer 在运行过程中，一直接收 DataServer 上传的 blockid 或对一些 blockid 更新地数据。

同时，为增强系统健壮性， NameServer 将自己的所有数据定时备份到另外一台NameServer 节点上。如果主 NameServer 遇到异常，系统将能够切换到备份 NameServer 中运行。

当 NameServer 接收到用户发送的对指定文件的读取请求时，其能够根据文件名（由blockid、fileid 和其它编码组成）来判断此文件所在的 block 在哪个 DataServer 服务器中， 然后将此 DataServer 的地址返回给用户。

18图像来自于 <http://code.taobao.org/p/tfs/file/309/filename.png>

当 NameServer 接收到用户发送的写文件请求时，其首先判断此文件名是否存在，如果存在将返回给客户其需要写入文件数据的 DataServer 地址。如果不存在，则根据负载平衡原则，从 DataServer 服务器列表中选出一台返回给用户。

* + - * **DataServer**

DataServer 负责具体的数据存取服务。为提高数据并发读写性能，在 DataServer 中每个物理磁盘对应一个数据处理服务（dsp）。为了加快文件检索速度，DataServer 在启动时将把所有超级块中的 BlockIndex 和每个逻辑块的索引数据加载入内存，同时把其管理的blockid 发生到 NameServer。

当 DataServer 接收到用户读文件请求时，首先在内存中根据文件名查找此文件所在的物理块、文件大小和偏移位置，然后利用这些信息来读取文件数据并返回给用户。

当 DataServer 接收到用户写入文件数据的请求时，其首先判断此文件是否已经存在， 如存在则根据文件大小来完成文件的覆盖写或者追加到相应物理块的尾部；如果不存在则

将数据追加到文件尾部，如果超出当前文件块的限制将新建块来存储此文件内容。然后更新 索引信息并发送相关变化的 block id 到 NameServer 中。

* + 1. **TFS 文件系统性能分析**

TFS 文件系统采取了扁平化管理模式，取消了目录结构，直接将数据存储在文件（块） 中，并根据 blockid 建立对物理块的 hash 索引以加快文件信息的定位。在不考虑分布式系统网络通信代价的情况下，当用户向 DataServer 提交文件数据读取请求时，DataServer 将根据此文件名所对应的 blockid 和 fileid，通过内存中的 hash 索引直接定位到此文件具体的存储位置，然后进行数据读取。与 linux 文件系统 Ext 中的逐层目录检索来定位文件位置的方式相比，TFS 的文件检索和文件数据定位速度快。

针对小文件，TFS 将多个小文件合并到一个块中来进行存储。除了在每个块头部存储一些属性信息，一个物理的其余空间全部用来存储文件数据，因此 TFS 的空间利用率较高。

TFS 是针对海量小文件和其不经常更新的特点来进行设计的。如果在此文件系统中经常进行大量的文件数据更新，则会导致扩展块增多，索引时间变长，从而引起文件读取速度的 下降。

* 1. **实验** 9 Web **服务器的文件系统**

**题目 1**. 使用网络爬虫工具，下载足够多的网页及其图片文件，数据量在几十 GB 左右。

**题目 2**. 根据 TFS 的设计思想，设计并实现一个适合小文件存储和快速读取的本地文件系统。

**题目 3**. 设计并实现支持上面文件系统的 api 接口（读文件、写文件、删除文件）。这个

接口能够保存原有文件名和此文件系统中文件名之间的映射关系；能够支持多线程对同一 个或多个文件的并发操作；能够提前预读或缓冲部分文件数据以加快文件读取速度。利用这 个 API 接口，使得用户能够通过原有文件名从此文件系统中获取相关的数据，以达到最大限

度地保证与原有系统文件名称的兼容。考虑到这个映射关系的数量巨大，不能完全放在内存

中存储，因此要设计并实现保存此映射关系的文件结构和读取、更新方法（请参考 B+-tree 和 hash 索引的文件结构）。通过实现的文件系统 api 接口，将作业 1 中下载的文件导入到此文件系统中。

**题目 4**. 设计相关测试方法和程序，以比较本地 linux 文件系统和前面设计的文件系统在

存储和读取上面文件时的差别（特别要注意空间利用率和检索、读取文件速度之间的差别）。 如果本地文件系统的数据检索和文件读取速度较慢，则分析导致速度慢的原因：是文件系统 结构设计问题？还是算法问题？还是编程中的代码没有优化？。在分析这些原因过程中，要 参考 Linux 文件系统中的相关设计思想和源代码。

**题目 5**. 将上面实现的文件系统及其 api 接口集成入前面章节实现的 Web 服务器中，并

测试其性能，特别是在处理用户高并发请求时的文件 i/o 读取速度。