北京大学 2024-2025 学年第一学期期末复习笔记

计算机系统导论（A）

一、计算机网络

1. 客户端-服务器模型：模型中所有概念的都是【进程】，客户端进程与服务器进程进行通信。二者甚至可以在同一台主机上。

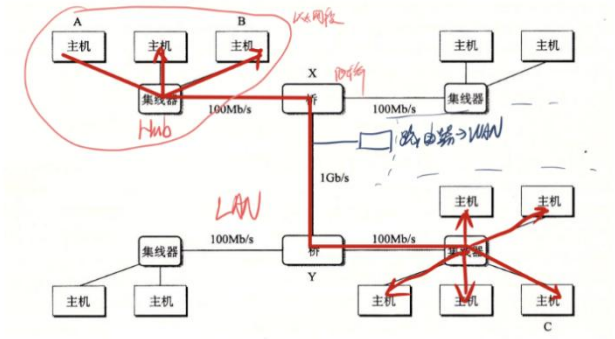
2. 网络

（1）网络是一种 I/O 设备（对主机而言），通常使用 DMA（Direct Memory Access，直接内存访问）传输；网络是一种按照地理远近组成的层次系统（物理上而言）。本质上和从磁盘/终端进行文件读写没有区别。

（2）物理上最底层是 LAN（Local Area Network，局域网），以太网（Ethernet）是一种 LAN 技术技术。以太网由：主机、集线器（Hub）——不加分辨地转发到所有端口且是全部复制（因此每个主机都能看到每个位但只有目的端口复制）组成。桥接以太网是由网桥（Bridge）——根据算法有选择性的、仅当必要时转发到端口，连接几个以太网。

我们一般认为一个 LAN 帧=LAN 帧头+互联网络包（有效载荷）组成。

（3）WAN（Wide Area Network，广域网）是由路由器连接若干 LAN 组成的。



桥接 LAN

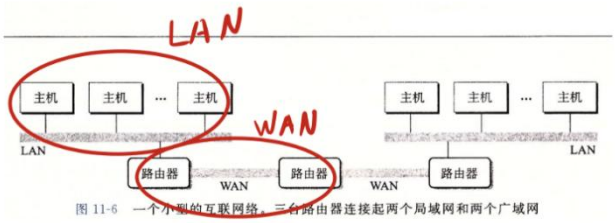


图 11-6　一个小型的互联网络，三台路由器连接起两个局域网和两个广域网

WAN

（4）进一步地，路由器将多个不兼容（跨协议）的 LAN 局域网连接，形成 internet（互联网，小写）。

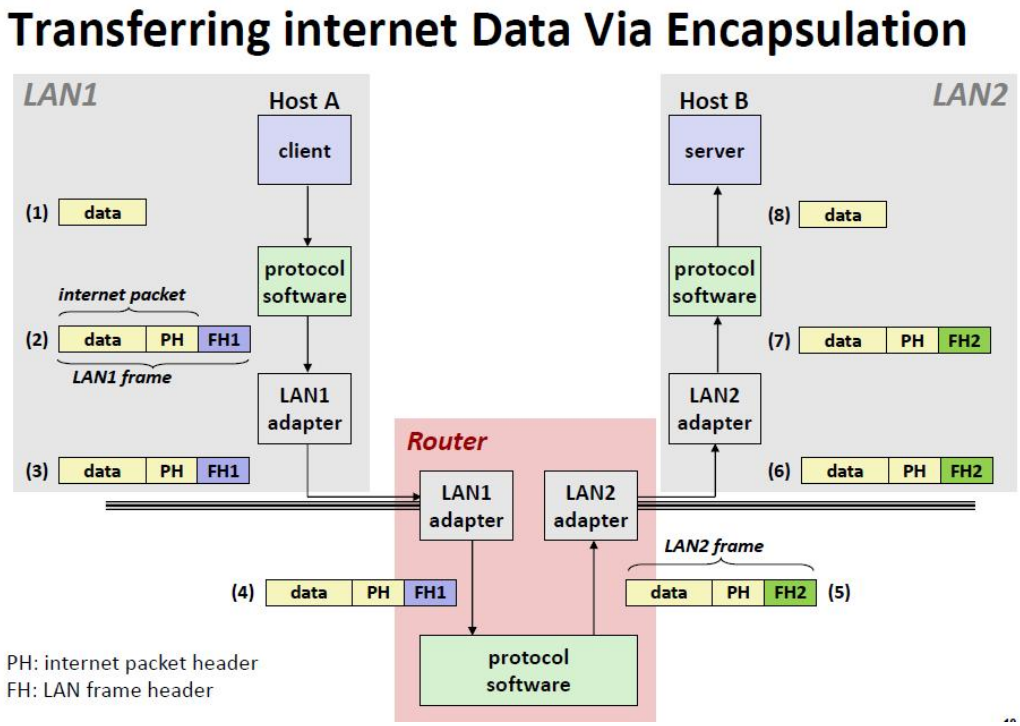
（5）Internet（全球 IP 因特网，注意是大写=特有名词=具体实现），是 internet 互联网的具体实现。

（6）网络协议：包括命名机制和传送机制，以协议软件为实现。

a. 命名机制：定义一致的主机地址格式，每台主机被分配至少一个互联网络地址，该地址唯一标识主机。

b. 传送机制：定义一种将数据位捆扎成不连续片（包）的同一方式，包：包头+有效载荷（参考前面的帧），包头含有包的 payload 大小以及源/目的主机地址，有效载荷为从源主机发出的数据位。

c. 基本封装方式：在数据前加互联网包头和 LAN 帧头（注意这里是协议封装，所以是两次）。概念区分：帧的有效载荷=互联网包，互联网包的有效载荷=数据。来源于公式：帧=LAN 帧头+互联网包；互联网包=互联网包头+数据。两个公式中后面的才是每层意义上的有效载荷。



说明：局域网、路由器中均有协议软件，局域网中的路由器进行封装/剥落（直接两层），路由器中协议软件的则负责更换 LAN 帧头，不改变互联网包头。

3. 全球 IP 因特网（Internet）

（1）Internet 是 TCP/IP 协议簇的（本部分内容的每个字都极其重要）

注意：本课程中，可靠的等价于面向连接的。

a. IP 是网络层协议，给每台电脑分配一个唯一的 32 位 IP 地址，并传送 Datagram（数据包）到正确的地址，是不可靠的（不保证数据包一定到达目的地，不重传），是主机到主机之间，无连接也非面向连接。

b. UDP 是传输层协议，建立在 IP 协议上，允许 Datagram 在不同的进程之间传送，不可靠，是进程到进程之间，无连接也非面向连接。

c. TCP 也是传输层协议，建立在 IP 协议上，使得 Stream（字节流）在不同的连接的进程之间（即进程对之间）传送，可靠，连接也面向连接。特别的还是全双工的。

d. （事先写出来，方便比对复习，且确实常考，但教材上 http 不在此处出现）

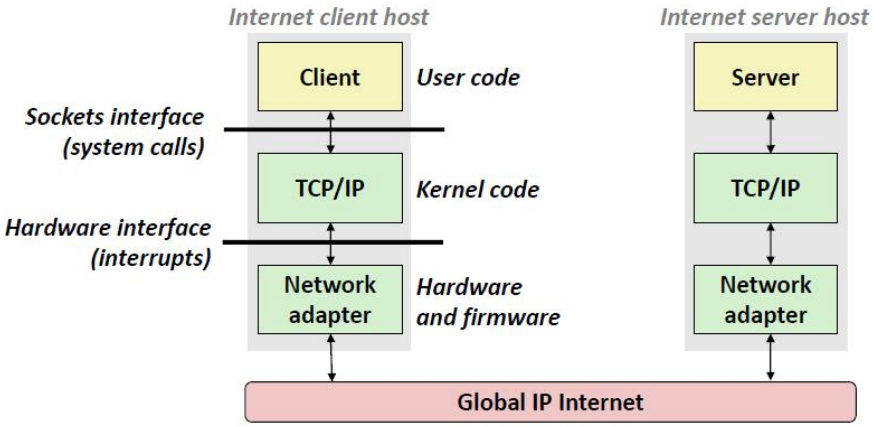
HTTP 是应用层协议，建立在 TCP 协议上，在 Web 客户端和服务器之间传输 Web Content（超文本数据，如 HTML 文档、图片、视频等），可靠，连接也面向连接。

e. 以上协议中，在应用层的 HTTP 属于 User Code，在网络层的 IP、在传输层的 TCP/UDP 属于 Kernel Code。

（2）系统通过套接字接口 Socket 和 Unix I/O 函数进行通信，TCP/IP 协议需要在内核模式下运行。

（3）主机地址被映射为一组 32 位 IP 地址；这组 IP 地址又被映射为一组因特网域名的标识符（可见 IP 与域名是多对多的关系）；因特网主机上的进程能够通过连接和任何其他因特网上的进程进行通信。

（4）协议栈



从上到下：应用层，传输层，网络层，数据链路层，物理层

应用层：http, imap, smtp, dns

传输层：tcp，udp

网络层：ip，icmp

数据链路层：ethernet，802.11，802.3，PPP

（5）关于 IP 地址

a. 顺序：永远是大端法，与机器无关

b. IPv4 使用 32 位 IP 地址，点分十进制表示；特别的 127.0.0.1 是本机地址（回送地址）

（6）关于域名

a. 是全球性大型分布式数据库，使用 DNS 协议（Domain Name System，域名系统）。

b. 域名和 IP 是多对多的关系，域名可能不对应任何 IP，反之亦然。

c. 用 hostname -i 和 nslookup 查看主机/指定域名的 IP（Linux）。

（7）基于 TCP 协议与套接字实现的因特网连接

a. TCP 的性质（三条）

点到点的连接：连接两个进程

全双工：可以同时从连接的两个方向收发数据

可靠：保证一端发送的字节流一定能以相同字节顺序在另一端被接收

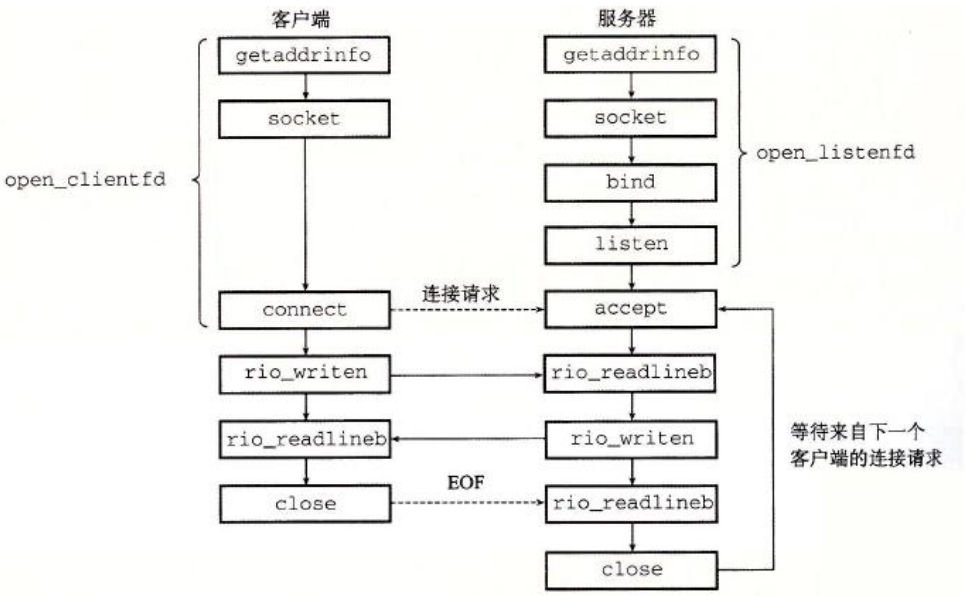
b. 套接字：是一个连接的终端，地址由 ip:port 对组成

c. TCP 套接字是一个五元组，由连接双方的 ip, port 以及协议名唯一标识

d. ip 是 32 位无符号整数，port 是 16 位短整数（常考计算题）。

e. 端口分为知名端口（well-known）以及临时端口（ephemeral），知名端口包括 http 协议的 80 端口（https 使用 443），ftp-21，ssh-22，smtp-25，echo-7

f. 知名端口供提供相应服务的服务器使用，其他进程一般不能占用（比如如果要用服务器的网页服务，此时客户端不可以用 80 端口）



本图像必须全部默写，一字不差。说明：getaddrinfo 获取地址，socket 创建套接字。

4. Web Service

（1）关于 HTTP 协议（很重要）

a. Web 服务使用 HTTP（HyperText Transfer Protocol，超文本传输协议），是建立在 TCP 之上的应用层协议，客户端与服务器间传送 Web Content，连接且面向连接，可靠。

b. HTTP 服务器使用 80 端口。

c. 关于 HTTPS：不是复数！HTTP+SSL/TLS=HTTPS。

（2）静态内容与动态内容（比较重要）

a. 静态内容：内容以磁盘文件形式存储，包括 HTML 文件，图片，音频，JavaScript 程序等。

b. 动态内容：内容需要由程序动态执行产生，文件包含可执行代码。

（3）URL 与 URI

a. URL（Universal Resource Locator，通用资源定位符）：每一个网页对象都与一个 URL 相连接。格式为：冒号（：）后面指明端口号，问号（?）分隔文件名和参数，参数使用&分隔，URL 中间没有空格，空格使用%20 转义字符或+表示，+/?%#&=等字符都需要用转义字符%+ascii 编码形式表示。URI（Universal Resource Index，统一资源标识符），为相应的 URL 后缀。

b. 确定一个内容指向静/动态内容没有标准规则；后缀开始的/表示请求内容类型的主目录，而非 Linux 中的根目录。

（4）关于 HTML（Hypertext Markup Language，超文本标记语言）

a. HTTP 请求是一个请求行+后面跟随 0 或多行请求头。

请求行：<method> <uri> <version>

HTTP1 支持 GET、POST、OPTIONS、HEAD、PUT、DELETE、TRACE 方法（可见 GET、POST、PUT 等都可以用于获取动态内容）。version 是 HTTP 版本，HTTP/1.0 或 HTTP/1.1。

请求头：<header name>: <header data>

b. HTTP 回应由一个回应行+0/多个回应头，还可能跟随着内容，以\r\n 分隔 headers 和内容。

回应行：<version> < status code> < status msg>

其中，HTTP 状态码：200 OK，301 Moved，404 Not found

回应头：<header name>: <header data>

（5）用 CGI（Common Gateway Interface，通用网关接口）来服务动态内容

a. CGI 动态文件：fork 后子进程执行请求的文件，并将运行结果打印到 buffer 中返回。

（6）Proxy（代理）

a. 代理：在原始服务器和客户端之间充当一个中介。对于客户端，代理是服务器；对于服务器，代理是客户端。使用代理来进行缓存、过滤、匿名等。

b. 两种代理：显式/透明。

显式代理，需要每条请求都包含完整的 URL。

透明代理：浏览器和客户端的行为就好像没有代理存在。

5. Socket 编程与 Echo 服务器（关注前面的各种代码实现）

（1）客户端——服务器模型的几大函数

socket：创建一个 socket 套接字，返回一个 fd。

connect：客户端专用，发起连接请求，默认是阻塞型函数。

bind：服务器专用，将创建的 socket 与一个指定地址绑定。

listen：服务器专用，将一个套接字描述符设置为监听状态。

accept：服务器专用，阻塞，当监听到 connect 请求并连接成功后返回。

rio_xxx：读写函数。

close：关闭套接字描述符连接。

（2）关于 IP 地址的获取：getaddrinfo 与 getnameinfo

二进制套接字地址结构与主机名/地址，服务名，端口号的字符串间表示的相互转化，可用于编写独立于任何特定版本 IP 协议的网络程序。addrinfo 结构体被组织为链表，每个结构体内有指向一个对应于 host 和 service 的套接字地址结构。

（3）套接字中的两种数据结构：sockaddr 与 sockaddr_in

a. sockaddr 用于各类网络连接相关的函数中。

b. sockaddr_in 是 IPv4 使用的结构体，对结构体成员赋值后需要强制类型转换后传参。

c. 前者用 char[14]，后者用 uint16_t 和 struct in_addr 存储地址。

（4）迭代 echo 服务器的实现：server 的 openlistenfd、main（极其重要，彻底理解+背）

```
#include "csapp.h"
void echo(int connfd);
int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr; /* Enough space for any address */
    char client_hostname[MAXLINE], client_port[MAXLINE];
```

```
    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        Getnameinfo((SA *)&clientaddr, clientlen, client_hostname, MAXLINE,
                    client_port, MAXLINE, 0);
        printf("Connected to (%s, %s)\n", client_hostname, client_port);
        echo(connfd);
        Close(connfd);
    }
    exit(0);
}

int open_listenfd(char *port)
{
    struct addrinfo hints, *listp, *p;
    int listenfd, optval=1;
    /* Get a list of potential server addresses */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_socktype = SOCK_STREAM;              /* Accept connections */
    hints.ai_flags = AI_PASSIVE | AI_ADDRCONFIG; /* ... on any IP address */
    hints.ai_flags |= AI_NUMERICSERV;             /* ... using port number */
    Getaddrinfo(NULL, port, &hints, &listp);
    /* Walk the list for one that we can bind to */
    for (p = listp; p; p = p->ai_next) {
        /* Create a socket descriptor */
        if ((listenfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) < 0)
            continue; /* Socket failed, try the next */
        /* Eliminates "Address already in use" error from bind */
        Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
                   (const void *)&optval , sizeof(int));
        /* Bind the descriptor to the address */
        if (bind(listenfd, p->ai_addr, p->ai_addrlen) == 0)
            break; /* Success */
        Close(listenfd); /* Bind failed, try the next */
    }
    /* Clean up */
    Freeaddrinfo(listp);
    if (!p) /* No address worked */
        return -1;
    if (listen(listenfd, LISTENQ) < 0) {
        Close(listenfd);
        return -1;}
    return listenfd;
}
```

二、链接

1. 链接的基本概念与程序的运行过程

（1）链接：将多个目标文件组合成一个可执行文件。

（2）目标文件：编译器将源代码文件编译后的产物，但还未链接形成可执行文件。

（3）可执行文件：链接后最终产物，可以复制到内存并且执行。

（4）链接在**编译时、加载时、运行时均可执行**。

（5）程序的运行过程（cpp 是 C 预处理不是 C++后缀名，gcc 驱动，cc1 编译，as 汇编，ld 链接）

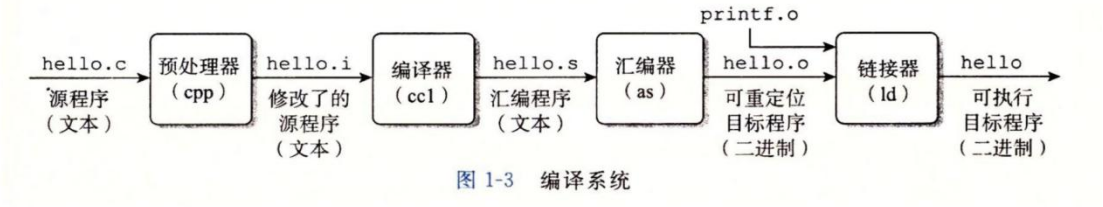


图 1-3　编译系统

$$.c \xrightarrow{cpp} .i \xrightarrow{cc1} .s \xrightarrow{as} .o \xrightarrow{ld} prog$$

- `gcc`：编译器驱动程序
- `cpp`：C 预处理（c preprocessor），将 xx.c 翻译成一个 ASCII 码的 中间文件 xx.i（intermediate file）
- `cc1`：C 编译器（c compiler），将 xx.i 翻译成一个 ASCII 汇编语言文件 xx.s（assembly language file）
- `as`：汇编器（assembler），将 xx.s 翻译成一个 可重定位目标文件 xx.o（relocatable object file）
- `ld`：链接器（linker），将 xx.o 和其他的 xx.o，以及必要的系统目标文件组合起来，创建一个可执行目标文件 prog

2. 静态链接

（1）符号解析：符号对应一个**函数、全局变量或者 static 变量**；符号解析器将每一个**符号引用**恰好与一个**符号定义**关联起来。

（2）重定位：汇编器生成的文件（.o 文件）是从 0 地址开始的，链接器需要将它们重新定位，把每一个符号定义与一个内存位置关联起来。并修改对应的符号引用。

3. 符号与符号表

（1）符号定义：int x; void f(){};这些称为符号定义。

（2）符号引用：int y=x;（x 被引用，y 是定义），f()（引用函数 f）。

（3）符号表：存储**符号定义**（由汇编器生成，是目录数组），每个条目包含符号名、大小、位置。

4. 目标文件

（1）可重定位目标文件（.o）：每个.c 文件生成一个.o 文件，可以与其他.o 文件链接后形成可执行目标文件。

（2）可执行目标文件（.out）：已完成重定位的目标文件，可以直接被拷贝进内存执行。

（3）共享目标文件（.so，s=shared）：一种特殊的可重定位目标文件，可供**动态链接**（加载/执行时），就像插件，用的时候再插。Windows 下叫做动态链接库/DDL。

5. 可重定位目标文件（ELF 文件）

（1）包含为 ELF 头（ELF Header）、节（Sections）、节头部表（Section Header Table）。

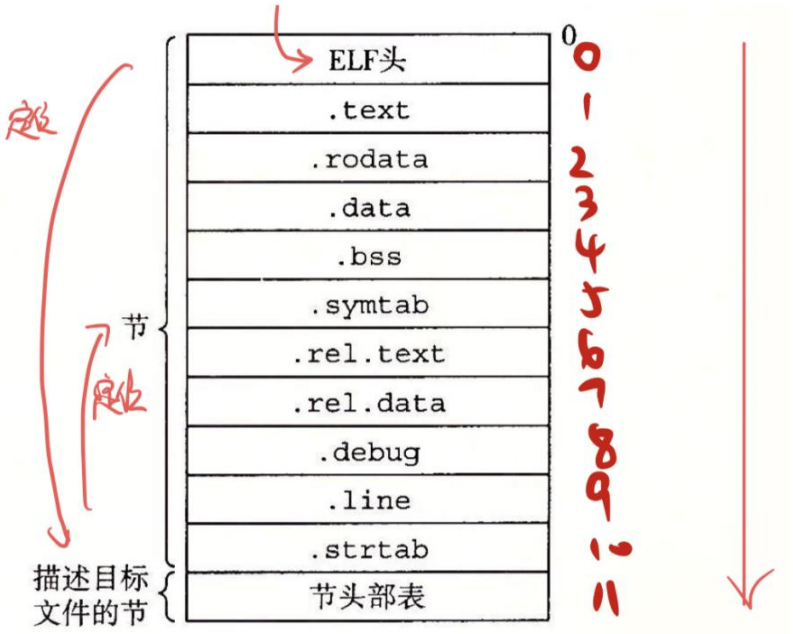


图 7-3 典型的 ELF 可重定位目标文件

（2）ELF 头存储这个文件基本信息（字节序、头长度等）。节中的内容见图片，箭头表示地址增长方向。

（3）运行时先定位到 ELF 头，再根据 ELF 头找到节头部表，最后根据节头部表找到相应的节。

（4）节的内容（只展示重要内容）

a. .text：已编译的机器代码（一般是函数及代码）。

b. .rodata：只读数据，如 printf 中的字符串、switch 的跳转表、字符串常量等。

c. .data：已（有效）初始化的全局和 static 变量。

d. .bss：未初始化的/无效初始化的全局和 static 变量（注意.bss 中有伪节，后面再谈）。该段在节头部表中有条目，但不实际占据空间，为了节省空间。

e. .symtab：符号表。

f. .rel.text/.rel.data：重定位信息（注意未初始化/初始化为 0 的信息不需要重定位）。

g. strlab：字符串表，包括.symtab 和.debug 节中的符号表以及节头部中的节名字。字符串表是以 null 结尾的字符串序列。

6. 符号和符号表

（1）伪节（在节头部表中没有条目）：经过重定位后伪节会消失。

a. ABS：不该重定位的符号。

b. UNDEF：在可重定位的目标文件里存在，例如 external 声明的符号，例如只有函数声明但是没有函数体的部分。

c. COMMON（极度重要！）：未初始化的全局变量，以及未初始化未分配空间的指针变量，例如 static int *bufp1。注意：有效初始化的一律在 data，剩下的先记住在伪节里，COMMON 里只有未初始化的全局变量，bss=初始化为 0/未初始化的 static 变量和初始化为 0 的全局变量。COMMON 中的变量由链接器分配空间，链接完成后进入.bss。

（2）符号的分类

a. 全局符号 global：定义在本模块（看清楚题目说的是哪个文件！！！）内，但可以被其他模块引用，包括非 static 的函数/全局变量。

b. 外部符号 extern：定义在别的模块，在本模块中被引用，用 extern 声明。

c. 局部符号 local：在本模块内定义的 static 全局函数和变量（不一定要求全局 static）。出现同名 static 变量时，编译器分配 x.1，x.2 等不同的名称。

d. 函数内部的非静态变量不再符号表中，由 stack 管理。

e. 牢记如下往年题（答案：A——static 优先原则，f2.c 中管理的是全局那个，f1.c 中的被解析为 static 的那个）。

7. C 源文件 f1.c 和 f2.c 的代码分别如下所示，编译链接生成可执行文件后执行，输出结果为：
A. 100　　B. 200
C. 201　　D. 链接错误

```
// f1.c
#include <stdio.h>
static int var = 100;
int main(void)
{
    extern int var ;
    extern void f() ;
    f() ;
    printf("%d\n", var) ;
    return 0;
}
```

```
//f2.c
int var = 200;

void f()
{
    var++;
}
```

7. 符号解析：多个模块间，将每个引用与一个确定的符号定义关联起来。

（1）编译器 ld 为符号标号，汇编器 as 生成符号表并计算相对地址。但是，汇编器不一定在 call 指令中填入相对地址，而是生成重定位条目等待链接器填入。

（2）静态过程变量=函数内部 static：不需要考虑同名，会标号；静态非过程变量=函数外部 static：**一个模块内只能由一个同名**，且别的模块里不能引用。

（3）**同文件内的同名全局变量在编译**时报错，**不同文件中定义的同名全局变量在链接中**报错。

（4）强弱符号

a. 强符号：函数（有函数体大括号为准，否则只是声明）、已经初始化的全局变量（包括初始化为 0！！！）。

b. 弱符号：未初始化的全局变量。

c. UNDEF 中的外部符号，以及各种带有 static 属性的变量，均属于非强非弱。

d. 强强报错，强弱选强，弱弱随机，只管定义不管类型（一个定义成 double 一个 extern int 引用会通过链接）。

（5）静态库：以.a 结尾，包含一些列模块（即可重定位目标文件的组合）。

（6）符号解析是否成功还与输入顺序有关系：链接器维护可重定位目标文件集合 E、未解析符号集合 U、一定义符号集合 D。规则如下：

a. 如果输入文件 f 是一个目标文件.o，则链接器把 f 添加到 E 并且修改 U 和 D。

b. 如果 f 是一个静态库.a，则链接器尝试匹配 U 中现有的未解析符号与静态库中定义的符号，如果成功匹配则把对应模块加入 E，修改 U 和 D，直到 U 和 D 不变，然后丢弃剩下的模块。

c. 完成扫描后若 U 非空则出错。因此库一般放在结尾，有时需要调整顺序甚至重复。

8. 重定位

（1）重定位符号：依靠重定位条目，修改节中的符号引用，使他们指向正确的运行时地址。

（2）重定位节：合并为新的聚合节，为每个节定义以及符号定义分配运行时内存地址——唯一的运行时地址。

（3）重定位的两种方式（重中之重！！！考试时先看清楚是什么寻址模式）

a. 32 位 PC 相对寻址：总公式为 addr(symbol)+addend-(addr(s)+offset)，加 offset 为了从模块起始地址（即函数体前面那坨数字）来计算当前指令的地址，addend 为了让他变换到下一条指令的地址（回忆第四章相关内容）。

b. 32 位绝对寻址：总公式为 addr(symbol)+addend（比较简单，但是记清楚是加 addend，不

要忘记）。

9. 可执行目标文件

（1）在可执行目标文件中，比 ELF 文件增加了一个**段头部表**（映射到运行时的内存段），以及一个 init 节。

（2）r：可读；w：可写；x：可执行；最后一位 s=shared，p=private（copy on write 即 COW）。

10. 运行时的内存分布（从低到高：共享代码段 init、rodata、text；读写段 data、bss；运行时堆归 malloc 管；共享库内存；运行时用户栈；内核内存）

内存布局

- **代码段**：从地址 `0x400000` = 2^{22} 开始，后面是数据段
- **堆内存**：由 `malloc` 分配，运行时堆在数据段之后
- **用户栈**：从最大合法用户地址 $2^{48} - 1$ 向下增长
- **内核区**：从地址 2^{48} 开始，用于内核代码和数据段

`_start`（入口点）→ `_libc_start_main`（定义在 libc.so 中）→ `main`

加载：将可执行目标文件的代码和数据复制到内存。

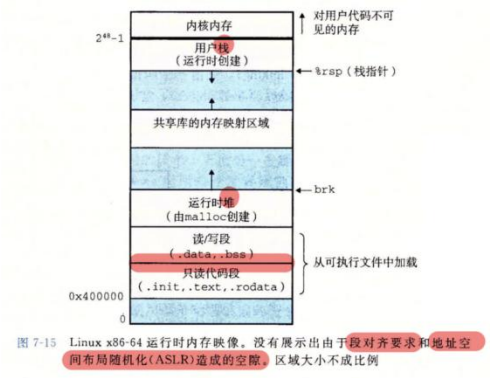


图 7-15 Linux x86-64 运行时内存映像。没有展示出由于段对齐要求和地址空间布局随机化（ASLR）造成的空隙，区域大小不成比例

11. 共享库：在运行时被动态加载和链接的文件，linux 中以.so 结尾。

（1）C 标准库是 libc.so，通常是动态链接的。无 text、data 的复制。

三、异常控制流

1. 异常

（1）是控制流中的突变，ECF 的一种形式，由硬件和操作系统实现。

（2）操作系统维护异常表，每一项指向一个异常处理函数；每类异常有个唯一的非负整数异常号。异常调用会压入内核栈中进行，异常处理程序运行在内核模式下。

2. 异常的分类（重要！！异步=来自外部控制流，不保证顺序，同步=来自当前控制流指令的直接产物，保证顺序）

类别	原因	同步 or 异步？	返回行为
中断	来自 I/O 设备的信号	异步（only）	总是返回到下一条指令
陷阱	有意的异常	同步	总是返回到下一条指令
故障	潜在可恢复的错误	同步	可能返回到当前指令
终止	不可恢复的错误	同步	不会返回

（1）中断 Interrupt：由处理器外部的事件引起的异常，因此异步（例：外部定时器，I/O 请求，收到一个网络套接字），异步发生，永远返回下一条指令。

（2）陷阱 Trap：故意设计的异常（例：系统调用，read、execve、fork、exit），同步发生，返回下一条指令。

（3）故障 Fault：非故意的异常（例：除零、缺页、机器检查），同步发生，可能返回当前指令或终止。缺页可以恢复，除法错误、一般保护错误、段错误不可恢复。

（4）终止 Abort：非故意的异常（例：硬件错误），同步发生，不返回而直接终止。

（5）关于系统调用：内核层面运行，系统调用只能用寄存器传递参数（dsd、10、8、9）。

3. 进程

（1）进程是一个运行中的程序实例。两个抽象：逻辑控制流、似有地址空间。

（2）进程上下文：既包括对应的内存状态（堆、栈、代码、数据），也包括 PC、%rsp 等重要寄存器的值。

（3）多进程：处理器时间分片，实现单处理器执行多进程。上下文切换指的是当一个进程被调度下 CPU 时，会在栈中保存它的上下文，CPU 切换为下一个进程的 PC，从内存中加载即将执行的进程上下文。

（4）并发与并行

a. 并发：进程的控制流在时间上有重叠（只看开始&结束时间！）。

b. 并行：如果两个流并发地运行在**不同的处理器核或者计算机**上，则称为并行流。

c. 并行流一定是并发流，反之不一定。

（5）私有地址空间：和这个空间中某个地址相关联的内存字节不能由其他进程读写访问。

（6）用户模式与内核模式：靠模式位区分，1=内核模式，0=用户模式。用户模式下进入内核模式的**唯一方式是系统调用**。

4. 进程控制

（1）int getpid(void)返回当前运行进程的 PID；int getppid(void)返回当前进程父进程的 PID。

（2）进程的三种状态：运行、挂起/暂停、终止。

（3）关于 pid_t fork(void)：用于创建子进程。调用一次，返回两次：子进程中返回 0，父进程中返回子进程的 pid——以此区分父子进程。父子进程 PID 不相同。

（4）父子进程获得相同但独立的虚拟地址空间，与父进程获得独立的打开文件描述符（最左边那个复制一份），文件表共享（不复制但是把左边新复制那个连上去）。变量的值对父子进程中独立。常用画进程图的方式来摸清楚 fork 的行为。

（5）进程的退出：void exit(status)。正常退出为 exit(0)，其他表示遇到错误，等价于 main 函数最后的 return 0，永不返回。exit()会进行资源回收，而_exit()不会。

（6）回收子进程：当进程终止时，仍然会占用系统资源，称为“僵尸”。因此需要父进程等待子进程终止后获得子进程的退出状态后，内核才彻底删除这个子进程。

a. 若父进程未回收子进程就终止，则子进程变成孤儿进程，托管给 init 进程由其负责回收。

b. 使用 waitpid、wait 等函数来回收

（7）关于 wait 与 waitpid（重要！）

a. pid_t waitpid(pid_t pid,int* statusp,int options)：需要掌握 pid、options 的用法。默认 options=0，挂起调用进程，直到等待集合中的一个子进程终止（若一调用就有进程终止则立刻返回）；WNOHANG=只看当前时刻有没有进程终止，不会等待，没有就返回 0；WUNTRACED 挂起直到等待集合中有一个终止/暂停的子进程；WCONTINUED 挂起直到有终止或者暂停的被 SIGCONT。以上宏必须使用｜进行组合！pid>0 表示等待该进程，-1 表示所有子进程。返回值=成功？子进程 pid：-1（WNOHANG 有可能是 0）。

b. wait(&status)：等价于 waitpid(-1,&status,0)，挂起调用等待所有子进程，放在循环中可以一个一个地回收僵尸子进程。

c. 含有 wait 的进程图需要让父进程等待子进程终止后再执行。

（8）让进程休眠——sleep 与 pause

a. unsigned sleep(unsigned secs)可以让进程暂停 secs 秒，若时间到了返回 0，否则**返回剩余秒数**。sleep 函数可以被信号中断！！！

b. int pause(void)让进程休眠直到收到一个信号，总是返回-1。

（9）加载与运行程序

int execve(const char*filename,const char* argv[],const char* envp[])加载并运行新的程序，覆盖当前进程的代码、数据、堆、栈内容，但保持 pid、打开文件及信号上下文不变。argv[0]是文件名，envp 是以 null 为结尾的指针数组，为环境变量列表，每一个元素指向 name=value 的字符串对，用于设置环境。

5. 信号

（1）关于 Shell：一种可以执行命令代表用户运行程序的程序。作业分为前、后台作业，后者以&结尾。前台作业未执行完时不可以执行下一条命令。

（2）信号：一种异步 ECF，提供用户级别的异常处理，可给单个进程/进程组/自己发信号。

a. 需要知道的信号包括：SIGINT 为来自键盘的中断、SIGTSTP 为来自中断的停止信号、SIITCHLD（注意不是 SIGCHILD！没有 I！）为子进程暂停/中止的信号、SIGALRM 为 alarm 函数的定时器信号、SIGKILL 为杀死程序、SIGSTOP 为非终端暂停信号。尤其区分 SIGTSTP 与 SIGSTOP！

b. 信号从不排队——不可以用于计数；一个待处理信号最多只能被接收一次。用位向量维护 pending 和 blocked 类型。

c. 发送信号的方式包括/bin/kill 发送信号，其中 pid 正代表进程 pid，负代表进程组所有进程；键盘输入 Ctrl+C=SIGINT 给前台进程组中的每个进程，默认终止进程，Ctrl+Z=SIGTSTP 给前台进程组中每个进程，默认暂停进程；用 kill 函数发送信号——pid=0 表示自己所在组的所有信号；用 alarm 发送 SIGALRM，secs=0 不设置任何闹钟。

（3）信号的接收

a. 内核把进程从内核模式切换到用户模式时，会强制进程接受未阻塞的待处理信号（有多个则选择 id 最小的信号类型）。

b. 三种默认行为：终止（并转储内存）、暂停直到 SIGCONT、忽略。

c. SIGSTOP 和 SIGKILL 的默认行为不可以被信号处理函数修改，故不可以被捕获、忽略。

d. 信号处理函数可以被其他信号处理函数打断，除非是同种信号的信号处理函数（自己不能打断自己）。

e. 保守的信号处理函数规则（六条），重要的是只用异步信号安全函数（唯一安全输出=w

rite，prinf、sprintf、malloc、exit 都不安全！！！）和使用 volatile 声明全局变量以禁止编译器优化把值存到寄存器中。

f. 异步信号安全函数=可重入||不能被信号处理程序中断的函数。可重入指的是函数只访问局部变量，一定异步信号安全，反之不一定。

6. 非本地跳转：从深层调用中返回。setjmp 调用一次，**返回一次或多次**；longjmp 调用一次**永不返回**，一般在 setjmp 后调用。

四、系统级 I/O

1. 文件

（1）文件的类型主要分为普通文件、目录与套接字。

a. 普通文件又分为**文本文件**（只含有 ASCII 与 Unicode）和**二进制文件**（else），对内核没有区别。

b. 目录文件是由一系列链接形成的数组，每一项是文件名到文件的链接。每个目录文件必定出现条目是‘.’（表示当前目录）和‘..’（表示上一级目录）。使用 mkdir 创建新目录文件，ls 列出当前目录下内容，rmdir 删除目录。**绝对路径以/开始**，表示从**根目录**开始的路径，**相对路径**表示从 **cwd**（current working directory）开始的路径，以文件名开始，一般是./或../。

c. 套接字的本质上也是文件描述符。

2. 打开和描述文件

（1）用 int open(char* filenames,int flags,mode_t mode)打开文件

a. open 函数返回的总是当前进程中没有打开的最小描述符。默认打开 0-stdin、1-stdout、2-stderr。

b. flags 可以用|连接，常见的有 ORDONLY、OWRONLY、ODRWR、OCREAT（若文件不存在则创建一个空文件）、OTURNC（若文件已存在则清空之）、OAPPEND（每次写操作前总是设置文件位置到结尾处）。 mode 一般是新文件的访问权限位。

（2）用 close 函数关闭打开的文件，关闭已关闭的文件会出错。

3. 读写文件（ssize_t = signed size_t）

（1）ssize_t read(fd,buffer,n)，返回**实际**传送的字节数，0=EOF，-1=error。

（2）ssize_t write(fd,buffer,n)，返回实际写入的字节数，-1=error。

（3）lseek 改变文件指针的位置。

（4）不足值：传送字节数比要求的要少。不是错误。可能原因：

a. read 遇到 EOF

b. 从终端读

c. 读写（较长的）socket

d. Unix pipe

e. 特别注意：除了 EOF 外，从磁盘读写时不会遇到不足值。

4. 用 RIO 包健壮地读写

（1）C 标准 I/O 函数中的高级 I/O 函数：在 **libc.so** 中定义，比系统级 I/O 函数更高层次的。进行了另一层抽象：将文件描述符和流缓冲区进行抽象操作（UNIX I/O 按照字节进行操作）。

（2）关于缓冲区

a. 由于引用程序经常只读写少量字符，因此每次都调用 System I/O 函数是不合算的：系统调用耗费时间很长。

b. 引入缓冲区：读时调用 read 读一大块内容填充流缓冲区，再从缓冲区中逐步读入引用程序；写时写入流缓冲区，直到进程退出、缓冲区满、遇到换行等情况调用 write 批量写入。例如：连续调用 printf 输出单个字符不一定会导致 write 调用，直到使用了 fflush(stdout)才输出。主要是为了减少 syscall 以加速提效。

（3）rio_readn&rio_writen：rio_readn 只在遇到 EOF 时返回不足值，rio_writen 不返回不足值，可以对同一个描述符交错使用。二者是不带缓冲区的（函数名没 b）。

（4）rio_readlineb, rio_readnb：线程安全，也可在同一个描述符上交错使用。rio_readlineb 读一个文本行直到到达 maxlen 长/遇到 EOF/遇到换行符。rio_readnb 读最多 n 字节，直到到达最大值长/遇到 EOF。名字带 b，有缓冲区。

（5）带缓冲区（带 b 的）的和不带缓冲区（无 b 的）的严禁混用！！！

5. 共享文件（核心重点！！！）

（1）描述打开文件的三个数据结构：描述符表、文件表、v-node 表。描述符表每个进程一个且各自独立，所有进程共享文件表和 v-node 表。

a. 描述符表中为每个进程打开的文件描述符，**默认打开** 012 为 stdin/out/err，和终端有关。

b. 打开文件表：打开文件的集合，每个文件表项含**当前文件位置**（指针位置）、**引用计数**、指向 v-node 表中对应表项的指针等。关闭一个描述符会减少相应文件表项中的引用计数，当引用计数为 0 时，内核删除这个文件表项。对应一次打开操作。

c. v-node 表：每个表项包含 stat 中大多数信息，多个描述符可通过不同文件表项引用同一个文件。对应一个文件，可以有多个打开文件表表项指向同一个 v-node 表项。

（2）关于共享文件的常见三种操作（重中之重！）

a. 使用 open 打开文件：默认把给最小的未使用 fd 连接到新创建的文件表上。特别注意：两次 open 同一个文件时会创建两个打开文件表，默认情况下（除了 OAPPEND 等特殊要求）文件位置指针独立互不干扰。

b. dup 与 dup2

dup2(oldfd, newfd) 将 oldfd 复制到/覆盖 newfd——使 newfd 指向的变成 oldfd 指向的，使后者变成前者（后变前后变前后变前！！！）newfd **可以是未打开**的 fd，若 newfd 已打开，则**复制前先关闭**。

dup(oldfd)返回一个指向 oldfd 同样的文件表项的描述符（共享文件位置指针等）。

（3）fork 创建新进程：创建一个与父进程相同但独立的 fd 表，指向相同的打开文件表（共享文件位置指针等，父位置变子也变）。

（4）大部分情况下，我们认为缓冲区对进程独立。把它当成一个 private 的数组。

6. 三种 I/O 函数的总结比较（重点掌握这个）

（1）UNIX I/O：read、wirte、open、close 等。**最底层**，可以提供例如访问元数据的操作，异步信号安全，可以在信号处理函数中使用，但作为系统调用消耗资源大，对于不足值处理较差。

（2）C Standard I/O：fopen、fclose、fread、fwrite（按字节）、fgets、fputs（按行）、fscanf、fprintf（格式化）：全双工=双方向传输，可在同一个流上输入输出，**增加了缓冲区**，自动处理不足值，但不是异步信号安全的函数，对于网络套接字的读写不一定正确。

（3）RIO I/O：增加处理不足值、缓冲区的情况。

（4）总结比较

a. 使用 UNIX I/O：信号处理函数内，对性能有特别的要求时。

b. 使用 C 标准 I/O：对磁盘或终端文件进行读写。

c. 使用 RIO：读写网络套接字。

d. 对于二进制文件，不应该使用文本导向的 I/O：fgets、scanf、rio_readlineb（不用两行+scanf），也不应该用 str 开头的字符串函数。

五、虚拟内存

第一部分 虚拟内存基本概念

1. 虚拟内存基本概念物理、虚拟寻址和地址空间

（1）虚拟内存是物理内存和磁盘结合的一种内存机制，为每个进程提供大、一致、私有的地址空间。一般会把主存看作磁盘上地址空间的 Cache，只保留最近活动的区域。

（2）CPU 可以生成虚拟地址来访问内存，VA 翻译到 PA 的过程由**内存管理单元 MMU** 执行。

（3）物理地址空间 PAS 对应于物理内存的 M 个字节，M 可以不是 2 的幂（相应的，虚拟空间必须是 2 的幂）。

2. 页与缓存

（1）虚拟内存和物流内存被分割成大小固定且相同的块：虚拟页 VP 和物理页 PP，每个块的大小为 P=2^p 字节。VP 集合中有三个不相交的子集合：unallocated（未创建的页，不占磁盘空间）、uncached（没有换存在物理内存中的 allocated 页）、cached。请注意：“页”的定义是一个基本单元大小，可以理解成一种对齐要求/一种单位，可能不存储任何实际的内容。

（2）DRAM **全相连主存**被用作物理页的缓存，已经分配的虚拟页和一个物理页关联。**DRAM 的不命中代价很高——因为访问磁盘很慢**。总是采用写回策略。

（3）页表是一个将虚拟页映射到物理页的数据结构，实际上是一个 PTE 的 vector，每个 VP 在页表的一个固定偏移量处都有一个 PTE，用以提供将 VA 转换为 PA 的信息。页表存储于主存 DRAM 中的内核部分。PTE 有效位=1 表示在 DRAM 中缓存，0 看是否为空地址——是则 unallocated，否则是指向磁盘中的地址。

（4）页命中与 Page Fault（Recall：Page Fault 属于故障）：当 valid=0 时会触发缺页，选择牺牲页进行替换，若牺牲页被修改则复制回磁盘中，**最后要修改牺牲页的 PTE**。

（5）一般而言，只有对一个页的引用产生 page fault 才会复制此页到 DRAM 中。加载器从不从磁盘到内存实际复制任何数据。

（6）内存映射：将一组连续虚拟页映射到任意文件（后续的 mmap）。

3. VM 用作内存保护

（1）内存分配：请求内存分配时，OS 分配适当数字个连续虚拟内存页面映射到物理内存中的任意物理页面（连续——任意）。

（2）用户进程的四条准则：不能修改只读段、不能读写内核中的任何代码与数据结构、不能读写其他进程的私有内存、不能修改与其他进程共享的虚拟页面（除非所有共享者都允许）。违反以上准则会触发一般保护错误，特别的，在 Linux 中直接报告为段错误（SEGV）。

4. 地址翻译（设 M=2^m、N=2^n、P=2^p 表示物理/虚拟地址空间中的地址数量和页的大小）

（1）VPN 位数=n-p，PPN 位数=m-p（可见不一定相等，一般总 VPN<=PPN），VPO=PPO=p（数值、位数都一样）。

（2）VPN=TLB=TLBT+TLBI。和 PPN 没什么关系。

（3）PPN|PPO=物理地址（拼起来）。

（4）一般翻译过程之命中（单级、无 TLB）

a. 处理器生成虚拟地址传送给 MMU。

b. MMU 生成 PTE，从高速缓存（对于虚拟内存，高速缓存是一个抽象的思想，此处代指的是 DRAM 主存）中请求得到这个 PTE。

c. 高速缓存向 MMU 返回 PTE——内容是 PPN。

d. MMU 构造物理地址并传送给高速主存。

e. 高速主存返回请求的数据。

（5）一般翻译过程之 Page Fault（单级、无 TLB）

a. 与之相同

b. 与之相同

c. 与之相同

d. 由于 valid=0，触发 Page Fault，缺页处理程序（打断）判定 PTE 需要到主存中找。

e. 确定牺牲页，如果需要复制到磁盘中。

f. 调入新页面，更新 PTE。

g. 缺页处理程序返回到原来进程，再次执行导致 Page Fault 的指令（对应返回当前指令），使得命中。后同。

5. 地址翻译的优化（对应不同的考察模式，此处没有列举：页表自映射、权限位）

（1）综合 SRAM Cache 与虚拟内存（考得少）

相当于引入两级 Cache，SRAM Cache 比 DRAM Cache 更快，SRAM Cache 有 PTE 和 PA。

注意：无论是哪个 Cache，都没有权限位。

（2）引入 TLB 加速翻译（高频考点）

a. TLB 位于 MMU 中，是虚拟寻址的、高度相连的 Cache，存储完整的 PTE。

b. TLB 命中时，地址翻译直接在 MMU 中进行，无需访问任何 Cache。

c. 计算 TLB 时，一定要拆开 TLBT 与 TLBI！特别是 TLBI 位数不是 4 的倍数时，一定记得移动位置（VPN、VPO 计算时同理）。

d. TLB 不命中时，MMU 从 L1 Cache 中取出相应的 PTE 后放在 TLB 中（即需要写回修改！！）。上述情况在多级下则是拼好 PPN 后写回并且取数据。

e. 一次翻译可能产生多次 miss：TLB miss——从页表中取 PTE，需要的 data 不在 Cache 中也是 miss。

（3）多级页表（高频考点）

a. 当页表的各个片在内存中不是连续存放时引入地址索引——页目录与多级页表机制。

b. 上一级的 PTE 剥离无关信息后+处理后永远指向的是当前级页表的基址（前提是至少有一个页面已经分配，否则为 nullptr）。

c. TLB 会将不同级的 PTE 缓存起来，保证不比单级慢太久。

d. 大页：让倒数第二级页表能一步到位（现有的大小乘上一次 2^VPN）；超大页：能让倒数第三级页表能一步到位（乘两次）。

e. 一级页表需要常驻在主存中。

第二部分 虚拟内存的相关例子

1. TLB 与上下文切换：由于 TLB 中保存的地址与进程有关，因此进程上下文切换后，原有的地址一般不使用。两个解决方案：

a. 每次切换进程上下文时刷新 TLB（默认用这个）。

b. 为 TLB 条目增加进程 ID。

2. Intel Core i7：只需注意满足四个 VPN 大小相等（一般都这样），第四级页表没有 PS 位（第 3 级中出现，定义页表大小），而是 D 位（Dirty，是否被修改，是则为 1）。

3. Linux 系统中的虚拟地址空间：分为进程虚拟内存、内核虚拟内存——其中内核代码与数据、物理内存对每个进程一样，与进程有关的数据结构则不一样。

4. Linux 中几种页相关的故障

（1）Segmentation Fault 段错误：访问不存在的页（eg 未分配的页表对应的地址空间）。

（2）普通 Page Fault：按照缺页程序重新调度即可。

（3）General Protection Fault 一般保护错误：执行与当前页面权限不符合的操作，例如试图写一个只读页。在 Linux 中报告为段错误（除非题目中明确提报告为 xx，一般写原始的）。

5. 内存映射与 COW 机制：试图写入一个 Private 地址时才会引发复制，否则一般只是先指

向区域但不复制。

6. 关于 execve 与 mmap

（1）execve 要干的几个事情：删除已经存在的用户区域、映射私有区域（为新代码、数据、bss 和 stack 区创建新的区域结构，其中 bss 会被映射为匿名文件）、映射共享区域（动态链接到程序再映射到共享区域内）、最后设置 PC 指向入口点。

（2）mmap 用于创建新的虚拟内存区域，将一个 fd 指定的连续片映射到新区域。prot=访问权限位一般用|连接，PROT_NONE 表示不可以访问。flags 表明位组成，比较特殊的是 MAP_ANON，表示被映射的对象为匿名对象。munnap 删除虚拟内存区域。注意，如果是 shared，会导致磁盘内容连带着被修改。

第三部分 动态内存分配

1. 碎片（重要）

a. 内部碎片指的是一个已分配的块比有效载荷大的碎片（例如：对齐要求、维持结构的额外字段、分配策略）。仅取决于实现方式和以前的请求模式。

b. 外部碎片指的是明明有足够的内存，但是是散开的，导致出现不能分配的空闲块。取决于实现方式，以前和将来的请求模式。

2. 隐式链表与显式链表的区别在于：显式链表使用显式的指针指向下一个未分配的块。未分配块内增加 next 和 prev 指针以构成 free list 双链表，已分配块内没有该指针。注意隐式链表中有 footer，是 header 在末尾的简单 copy，目的在于减少查看上一个块是否分配所用的时间，由 O(size_of_chunk)降低为常数时间。

3. Garbage collection

（1）Mark&Sweep 是保守的，检查所有可能为指针的内容（text 中不可能，不查）。

（2）标记-扫描回收：不移动已经分配的块。

（3）root nodes：不在堆上但是有指向 heap 指针的节点（regs、stack 上的变量、global 变量）。

六、并发

1. 并发编程的基本概念

（1）死锁：每个进程都等待它一定等不到的资源。信号处理函数中调用 printf 就可能导致死锁。

（2）饥饿：一个进程长期无法得到执行（理论可以执行但实际没执行）。

（3）迭代服务器中，client 如果不是当前被服务的，发起 connect 后会被阻塞在 read 处。

2. 三种迭代服务器模型的改进——进程、I/O 多路复用、进程

（1）基于进程的服务器

a. server 需要在 accept 返回后使用 fork 甩给子进程进行服务，fork 后再调用 read。

b. 需要在子进程中关闭 listenfd，在父进程中关闭 connfd（子进程完事也得关），避免浪费。

c. 需要有 sigchild_handler 处理程序，使用 while(waitpid(-1,0,WNOHANG)>0)死等来回收僵尸子进程。

d. 优点：编程简单直观，共享打开文件表；

缺点：进程的创建、资源切换消耗大，进程间数据共享不容易。

（2）基于 I/O 多路复用的服务器

a. 使用函数 select(fd,&rset,&wset,&errset,&timeval)来实现，第一个参数是需要监听的最大描述符+1，后三个是需要监听的可读、可写、错误 fd 集合，第五个是等待时间，超时就滚。

b. 当监听集合中至少有一个 fd 准备好，select 就立刻返回准备好的 fd 数。

c. 优点：性能高，无进/线程切换开销，单一逻辑控制流和地址空间——大型高性能 Web 常用。缺点：编码复杂，不能充分利用多核处理器。

（3）基于线程的服务器

a. 线程独立享有 TID、stack&%rsp（不受保护，可以被其他进程访问修改）、PC、通用目的的 regs、CC，与其他线程共享虚拟内存。因此，基于线程的所有线程共享代码 text、数据 data、堆 heap、共享库 libc、打开的文件 fd。线程没有自己独立的物理寄存器。

b. Posix 线程提供了很多标准函数，需要记住的是 pthread_join()等待一个线程结束（无法等待任意），pthread_exit()退出且可以让主线程安全地退出而不至于主线程一退全部玩完（相应的，exit 就直接全部玩完），pthread_cancel()取消线程操作，pthread_detach()将线程分离出来并由内核自动回收——此时为 detached 状态，不能被其他线程回收&Kill。

c. 优点：容易共享局部变量，线程创建/维护/上下文切换的资源小于进程，编程逻辑简单。缺点：无意的共享可能引发各种竞争情况。

d. 服务器中传参用内存 Malloc。

3. 多线程中的共享变量——实例被一个以上的线程引用（回忆链接中的三种变量）

（1）全局变量，定义在函数外，VM 中只含一个实例，任何线程可以引用。

（2）本地自动变量，定义在函数内无 static，每个线程的 stack 中含有所有本地自动变量的实例。

（3）本地 static 变量，尽管定义在函数内，但内存初中仍然仅有一个实例。

4. 线程的同步

（1）经典错误：我们把循环关注核心的 Load、Update、Store 三个阶段，由于 regs 共享，若线程关键区发生重叠就很可能发生错误——一个线程还没 store 到 regs 中另一个就 load。

（2）信号量

a. 信号量 s 是一个非负整数全局变量，只能使用 P 和 V 进行操作。操作系统保证 PV 操作是原子性的。

b. P(s)=[while(s==0) wait(); s--;]；V(s)=[s++;]。C 语言中对应 sem_wait(P)、sem_post(V)。

c. 使用信号量进行互斥操作——mutex 一般仅能为 0/1。

d. 一般而言，pthread_mutex_t 比 sem_t 更快一些。

5. 生产者-消费者问题

（1）引入信号量&初始化：mutex=1（互斥锁保护全局变量）、slots/empty=n（空槽位）、items/full=0（货物数量）。

（2）生产者：先给 slots/empty 上锁，确保等待有空槽位再进去，然后再给 mutex 加锁，确保全局变量修改的唯一性，随后 mutex 解锁，然后由于有了货物，对 items/full 进行 V 操作（注意：生产者给 slots/empty 上锁，给 items/full 解锁，不是同一个信号量，仔细思考一下就能想清楚）。

（3）消费者：先给 items/full 上锁，确保等待有货物再进去，mutex 同理，由于有了空槽位，对 slots/empty 进行 P 操作。

（4）当缓冲区大小为 1 时，无论有多少个生产者/消费者，都不需要互斥锁 mutex。

6. 第一类读者-写者问题（读者优先）

（1）引入信号量&初始化：mutex=1（互斥锁保护全局变量）、w=1（控制写者可写）。

（2）引入全局变量&初始化：readcnt=0，统计读者数量。其中，第一个读者到来时（readc

nt++后为 1）加锁 w 排斥写者，最后一个读者离开（readcnt--后为 0）时解锁 w 欢迎写者。

（3）可能导致写者饥饿。

（4）读者：引入一个死循环，先给 mutex 上锁，readcnt++，然后如果 readcnt==1 就给 w 上锁，此刻才给 mutex 解锁（因为要对 readcnt 读，必须用互斥锁保证），读完后给 mutex 再次上锁，readcnt--，如果 readcnt==0 就给 w 解锁，再给 mutex 解锁（后半程和前半程总体对称）。

（5）写者：引入一个死循环，等待 P(w)，然后写，最后再 V(w)，如此循环。注意读写都需要 while(1)。

7. 第二类读者-写者问题（写者优先）

```
int readcnt, writecnt;       // Initially 0
sem_t rmutex, wmutex, r, w; // Initially 1
void reader(void)
{
  while (1) {
    P(&r);
    P(&rmutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      P(&w);
    V(&rmutex);
    V(&r)

    /* Reading happens here */

    P(&rmutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      V(&w);
    V(&rmutex);
  }
}
```

```
void writer(void)
{
  while (1) {
    P(&wmutex);
    writecnt++;
    if (writecnt == 1)
        P(&r);
    V(&wmutex);

    P(&w);
    /* Writing here */
    V(&w);

    P(&wmutex);
    writecnt--;
    if (writecnt == 0);
        V(&r);
    V(&wmutex);
  }
}
```

（1）引入信号量&初始化：r=1、w=1 用于控制写者、读者能不能进行读写操作。引入 rmutex=1、wmutex=1 用于控制 readcnt、writecnt 的访问。

（2）引入全局变量&初始化：readcnt=0，writecnt=0。

（3）和第一类比麻烦的地方：读是可以很多个一起读的，写只能一个 wrietr 写，所以需要对 reader、writer 都上锁。

（4）读者：等待 r 准备好大于 0 可读的时候，执行和第一类相同的操作——rmutex、readcnt++、判断是否是第一个读者是则给 w 上锁、此后先后给 rmutex 和 r 解锁（注意给 r 解锁在这里解，不能等到最后再解锁）；开始读，读完后再修改 readcnt，操作类似，此时不需要等待 r。

（5）写者：先给 wmutex 上锁，writecnt++、特判是否为第一个写者再解锁 wmutex，然后 P

(&w)——write——V(&w)，最后再 writecnt--（wmutex 操作类似）。

8. 四类线程不安全函数

（1）不保护共享变量的函数

（2）保持跨越多个调用的状态的函数（eg rand 函数）

（3）返回指向 static 变量的函数

（4）调用线程不安全函数的函数，但这个只是一个潜在可能，不是一定不安全

9. 可重入函数：被多个线程调用时不使用任何共享数据（注意不只是全局），一定是线程安全的。

（1）显式可重入：参数都是传值传递且数据都是引用本地的自动 stack 变量（无 global/static）。

（2）隐式可重入：参数传递在以上条件下包括指向非共享数据的指针。

七、期中前内容

第一部分 数字与进制

1. 算数右移=补高位，逻辑=补 0。无符号一定是逻辑右移，大多数对有符号是算数右移。

2. 优先级：四则>位运算>移动。

3. 除法向零舍入，右移向下舍入。

4. 非规格：E=1-bias，M=0.frac；与规格的 E=exp-bias，M=1.frac 不同。

5. 浮点数不可结合、乘法不可分配。

6. int 向 float 可能是舍入，但是向 double 一定精确。高精度浮点数不一定能得到比低精度浮点数更精确或相同的结果。浮点数 f<0，则 f*f>0 不一定成立；a>b，a+b>b+b 不一定成立。

第二部分 汇编

1. 关于 gcc：-E=预处理，-S=汇编，-c=链接，啥也不加在用-o（这个是输出到 xx 的意思）就是一步到位。静态库加-l___（末尾，例如数学库就是-lm）。使用 ar rcs libmylib.a library.o 创建静态库。

2. 操作 1~2 字节的指令保持寄存器余下的位不变，而 4 字节的指令会将寄存器的高 4 字节清零。

3. PC 在汇编代码中以%rip 出现。

4. 所有算数/比较指令都是后面的对于前面的而言。

5. cltq 无操作数，只作用于%eax 和%rax 之间（符号扩展）。

6. lea 指令不改变条件码！MOV 指令也不改变条件码！并非只有算术运算和逻辑运算指令才改变条件码寄存器（例如 test 和 and，cmp 和 sub，但是前者改变后者不改变）。

7. 跳转表仅在 case 值跨度不太大时才会生成，且 case 越多，效率越高。

8. callee saved：%rbx, %rbp, %r12-%r15。caller saved：剩余的除%rsp 外的所有寄存器。

9. RISC/CISC

指令集	CISC	RISC
传参+regs	寄存器少，可以用栈传参	寄存器多，不可用栈传参
寻址方式	多样，不需专门访存指令	只能使用特定指令访存
有无 CC	使用 CC	无 CC
指令种类	多	少
执行时间	长	短
指令长度	短（便于执行）	长（为了性能）
能耗	高	低（常用于嵌入式系统、手机 ARM）

第三部分 流水线：背吧，往死里背。看清楚 SEQ/SEQ+（无 PC）/PIPE-/PIPE。

第四部分 Cache

1. 关于 RAM（一断电就完蛋）

（1）FPM(Fast Page Mode) DRAM：快页模式 DRAM，对同一行连续地访问时，第一次发送 RAS+CAS 请求，随后只发送 CAS 请求。

（2）EDO(Extended Data Out) DRAM：扩展数据输出 DRAM。

（3）S(Synchronous) DRAM：同步 DRAM。

（4）DDR(Double Data-Rate) S(Synchronous) DRAM：双倍数据速率同步 DRAM——DDR，DDR2，DDR3，DDR4（依次翻倍）。

（5）V(Video)RAM：视频 RAM。

2. 关于 ROM（非易失性存储器，本意是 read-only memory，但是大类中有的也可以写）

（1）ROM 是只读存储器的缩写，其中内容是在编程时就烧入的，传统 ROM 不能之后修改内容。ROM 也支持按地址随机访问。ROM 断电非易失，可用于 BIOS、磁盘控制器、网卡等，又称固件。

（2）P(Programmable) ROM：可编程 ROM——只可被编程一次。

（3）E(Erasable) P(Programmable) ROM：可擦除可编程 ROM——可以被紫外线、X 射线等擦除。

（4）EEP(Electronically Erasable Programmable) ROM：电子可擦除 PROM——使用电子手段擦除、编程。

（5）Flash：闪存，基于 EEPROM 技术，以块为单位进行擦除。

（6）SSD：固态硬盘——基于闪存技术。读比写更快，随机读和随机写的速率都高于机械硬盘对应操作（使用地址进行访问），顺序访问快于随机访问，修改一个页中的块时只能整页擦除后拷贝。

3. 关于单位换算

（1）在内存中，使用的 K、M、G、T 都是以 2 为底的

（2）在外存（磁盘、网络）中，以 10 为底。

（3）DRAM & SRAM: K=2^10, M=2^20, G=2^30, T=2^40（严格意义上，这是 KiB 等）。

（4）Disk & network: K=10^3, M=10^6, G=10^9, T=10^12。

4. 关于 DRAM 的组织方式

（1）DRAM 按照行和列的方式组织，一般用 d*w 的 DRAM 来表示有效存储单元为 d*w 的 DRAM。

（2）不用一维连续数组的组织方式的原因是早期发明设计时为了节省地址线宽度，所以组织成二维形式。

（3）因此，访问 DRAM 某一块时需要进行两步：RAS 和 CAS——先给出行地址选取对应的行，将对应的行拷贝到 DRAM 内部的缓冲区中，再使用列地址进行列选取，从缓冲区内选出对应的块传送数据。

5. 关于磁盘

（1）磁盘是非易失性存储设备，速度远远慢于主存。

（2）传统机械磁盘：由盘片构成，每个盘片有两个表面，表面上的磁道构成同心圆，每个磁道被划分为一组扇区，每个扇区包含相等数据位，扇区之间存在间隙，柱面指所有盘片表面上到主轴重新距离相等的磁道的集合。若磁道编号一致，则表面的数量等于一个柱面包含的磁道的数量。

（3）记录密度/磁道密度/面密度；磁盘容量计算公式

Capacity = (#bytes/sector)* (#avg. sectors/track)* (#tracks/surface)* (#surfaces/platter)* (#platters

/disk)

（4）访问时间 = 平均寻道时间（Seek） + 平均旋转时间（Rotation） + 平均传送时间（Transfer）。用寻道时间约等于旋转时间估计是合理的。传送时间很小，有时可以忽略不计。

（5）磁盘作为外存设备，通过磁盘控制器，连接在 I/O 总线上，并通过 I/O 总线与主存进行数据传递（I/O 衔接器的左侧接 CPU：系统总线；右侧接主存：内存总线；下侧接 I/O 设备：I/O 总线）。

6. 关于局部性

（1）程序会倾向于访问附近地址的数据/指令（空间局部性），或在不远的将来重复访问这一数据/指令（时间局部性）。

（2）取指：指令在内存中排布的方式天然具有空间局部性，作为循环的指令，同样具有时间局部性。

（3）数据访问：以循环访问数组求和为例，对数组而言具有空间局部性，对和的 sum 变量具有时间局部性。因此，循环语句具有良好局部性。

（4）如果取同一个东西（指令、变量、数组元素），只可能说他有时间局部性而没有空间局部性（要求是附近的不是本身）。

7. 存储器层次结构

（1）结构：寄存器，SRAM 高速缓存（可能有 L1、L2、L3），DRAM 主存，本地外存（硬盘），网络存储（服务器）其中高速缓存还可以分为不同层次。

（2）不保证上层内容一定是下层的子集。

8. 关于 Cache

（1）直接相连 E=1，全相连 S=1。问每路多少行问的就是 E。

（2）直观来看应该使用最高位作为组选择，把相邻地址的内容存在 cache 的同一个组里，但实际选择中间位，这为了避免连续访问情况下的冲突不命中：考虑极端情况，E=1，数据块 0~7 都放在 set 0 中，则连续访问会导致 8 次 miss。而使用中间位进行组选择，可以将 0~7 分别放在 set 0~set 7 之中。可见这种策略保证空间局部性良好的情况下的命中率。

（3）关于写 Cache

a. hit：写穿透 write-through 以及写回 write-back（决定什么时候写到磁盘）。

b. 当 hit 时，都会修改 cache 中对应的内容，但：write-through 会直接把新的内容同时写进内存中，而 write-back 会推迟这个写进内存中的时间——一般来说直到替换或清空缓存时才会统一写回。为了支持写回机制，需要引入一个 dirty 位，表示经过修改（可以联系第九章

学的页表机制）。

c. miss：写分配 write-allocate 以及非写分配 no-write-allocate（决定是否写回 Cache）。写分配会同时将 miss 的内存内容同时存进 cache，而非写分配直接修改内存内容却不在 cache 中读入。

d. 常见搭配策略：write-through+no-write-back 或 write-back+write-allocate。

9. 关于存储器山：注意步长为 1 随大小下降不明显——Core i7 的预取机制。

第五部分 讲座课：哥们真不知道咋复习

第六部分 未来技术（核心：摩尔定律、计算/数据密集型）

1. 摩尔定律的起源与演变——1965 年摩尔定律的核心内容：晶体管数量每年翻倍。后续修订为每两年翻倍（“摩尔预测”）。更精确的是每 18-24 个月翻倍。可能失效原因：晶体管尺寸达到原子级别后 2D 平面制造不再适用（物理原因）。还有经济成本原因、功耗原因。

2. CPU 速率在最近几十年来稳步提高（摩尔定律）（近年来采用多核技术在不提高频率情况下近一步提高处理速率），SRAM 速率提升幅度小于 CPU 速率，而 DRAM 主存速率提高不明显，磁盘（即使是 SSD）速率即使是与 DRAM 主存的速率间都有巨大差异。

3. 晶体管尺寸与芯片面积变化趋势

（1）芯片面积扩大：面积平均每 10 年翻 1 倍。

（2）晶体管尺寸缩小：平均每 10 年缩小 4 倍。

（3）数学表达式：L=sqrt(A/N)，N=设备数，A=区域大小，L=线性规模。

4. Dennard 缩放定律

（1）压缩 IC Process 的方法：横/纵尺寸缩小 k 倍，电压降低 k 倍。

（2）结果：每芯片上的设备数增加 k^2倍，时钟频率增加 k 倍，芯片功耗保持不变。

（3）局限性：电压无法降到 1V 以下（热噪声问题）。应对策略是通过增加多核芯片提高性能。

5. 高性能计算

（1）超级计算机 vs. 数据中心：

超级计算机：如 Titan，注重计算密集型任务，运行模拟与建模。

数据中心：如 Google Data Center，注重数据采集、存储与分析。

（2）编程模型：Bulk Synchronous：将问题划分为多个区域，每个节点分配一个区域计算。阶段包括节点计算和节点间通信。

（3）计算密集/数据密集

a. 计算密集型（Compute-intensive）：指需要大量 CPU/GPU 运算来完成的任务，通常涉及复杂的数学计算、科学模拟、机器学习中的大规模模型训练等。

b. 数据密集型（Data-intensive）：指主要以处理海量数据为主的任务，注重数据的存储、传输和管理，例如大数据分析、日志挖掘、海量网络服务等。

c. 特征与需求

计算密集型：对 CPU/GPU 的性能要求极高。除处理器本身外也常需要高效的并行化算法。常见应用领域包括气象预测、分子动力学模拟、计算机图形学、深度学习训练等。

数据密集型：对存储系统、数据吞吐量与网络带宽要求极高。常用分布式文件系统、分布式数据库与数据处理框架。常见应用领域包括电商推荐系统、社交网络分析、日志分析、云计算、大规模数据挖掘等。

d. 二者资源瓶颈不同

计算密集型主要受处理器的运算能力限制。

数据密集型主要受存储与网络 I/O 的限制。

e. 优化思路不同

计算密集型更注重提高硬件算力或并行算法效率。

数据密集型更注重存储与数据传输技术、采用合适的分布式计算框架。