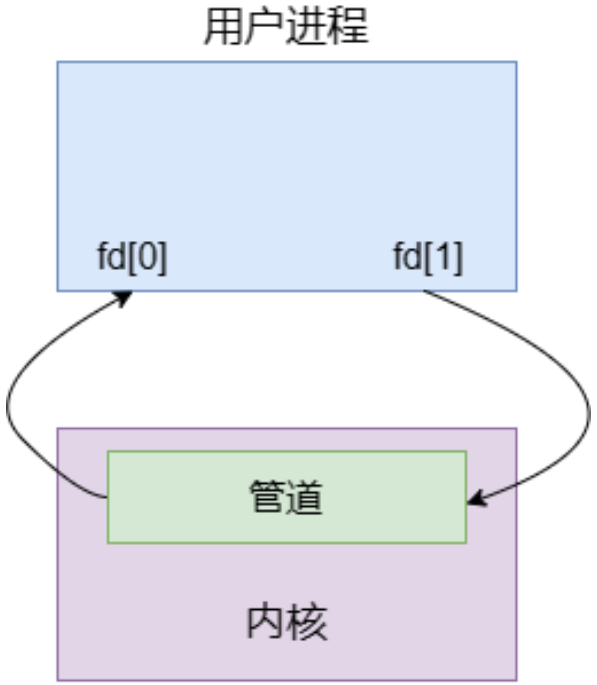
基础实验篇第3章

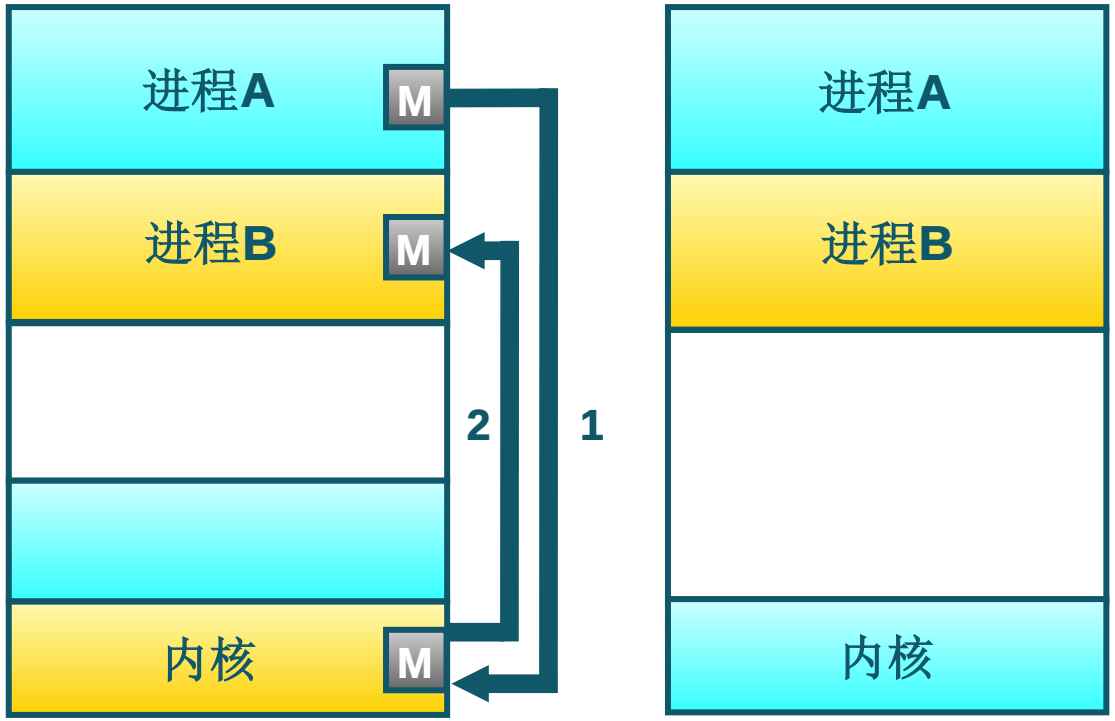
单个大规模文件的处理

研究背景

- shell编程利用管道实现从文本文件中读取特定字符串：
  - cat filename|grep [options] [pattern]
- 管道工作原理：
  - 内核缓冲区
  - fd[0]为读端，fd[1]为写端

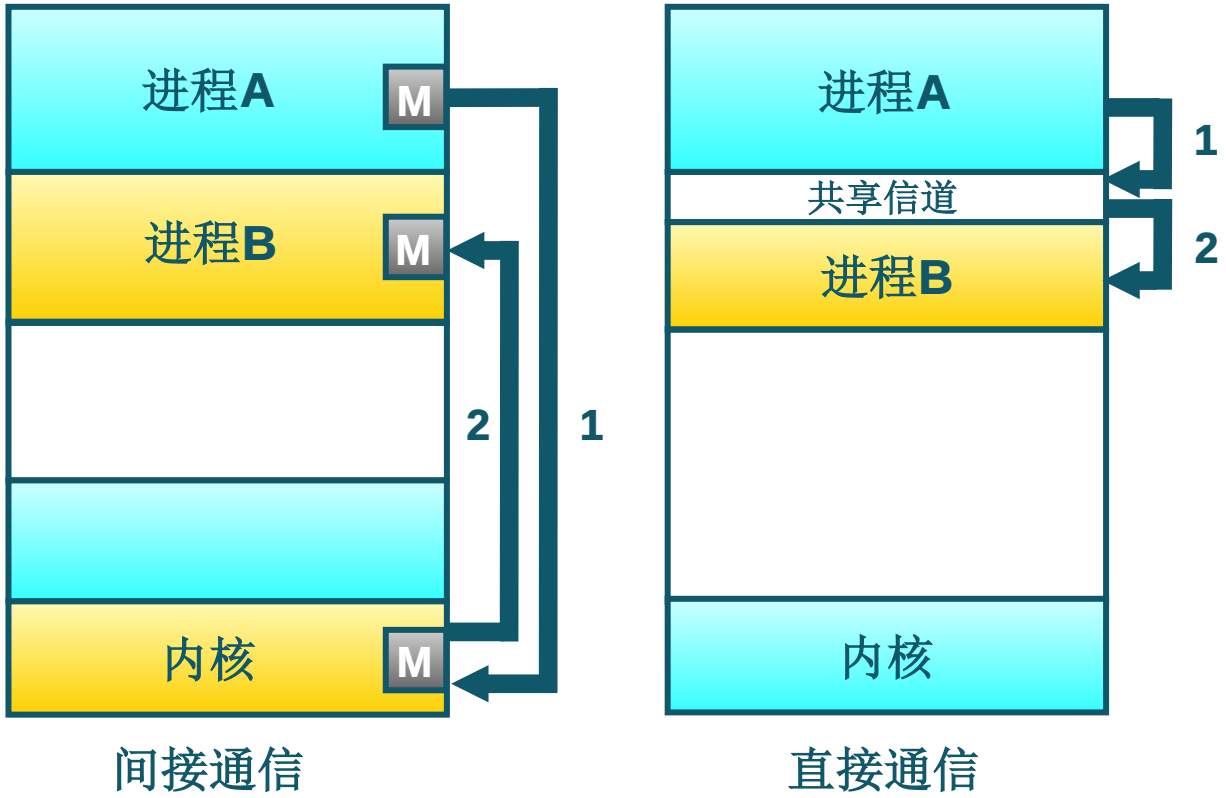


进程间的通信方式



间接通信　　直接通信

进程间的通信方式



间接通信　　直接通信

- 管道容量有限：512bytes * 8 = 4 KB

```
kjr@nkux86:~$ ulimit -a
real-time non-blocking time  (microseconds, -R) unlimited
core file size              (blocks, -c) unlimited
data seg size               (kbytes, -d) unlimited
scheduling priority                 (-e) 0
file size                   (blocks, -f) unlimited
pending signals                     (-i) 512678
max locked memory           (kbytes, -l) unlimited
max memory size             (kbytes, -m) unlimited
open files                          (-n) 1048576
pipe size                (512 bytes, -p) 8
```

- 写入管道的数据小于4KB时，Linux保证写入的原子性
- 写入管道的数据大于4KB时，写入不具有原子性，发生进程调度
  - 管道满时，写端阻塞，直至有进程读走数据
  - 工作效率低

- 使用C语言实现使用无名管道传输new.txt文件(71KB)
  - 不开启阻塞模式，每次写入4KB，使用strace统计调用write次数

```
kjr@DESKTOP-Q2L3T0M:/mnt/c/Users/lenovo/Desktop/OSbook$ strace -f -o straceout.txt ./pipe
kjr@DESKTOP-Q2L3T0M:/mnt/c/Users/lenovo/Desktop/OSbook$ grep -o "write" straceout.txt | wc -l
91
```

  - 开启阻塞模式后，当管道满时，返回errno为EAGIN的错误

```
kjr@DESKTOP-Q2L3T0M:/mnt/c/Users/lenovo/Desktop/OSbook$ gcc pipe.c -o pipe
kjr@DESKTOP-Q2L3T0M:/mnt/c/Users/lenovo/Desktop/OSbook$ ./pipe
write: Resource temporarily unavailable
```

- 使用命令拼接模式完成任务产生的弊端
  - 管道是一种昂贵的进程间通信工具
  - 管道产生额外的内存空间消耗（message同时共有3份）
  - 管道产生额外的运行时代价（大量的系统读写syscall）
  - 管道产生正确性维护的额外代价（管道会满，满时会报错）
- 因此：
  - 合并在同一个进程中可以进一步提升性能
- 改进思路：
  - 将读取、写入、正则表达式过滤都放入同一个程序中

文件读写相关的系统调用

文件读写相关的系统调用

- 文件系统相关的函数调用原型
  - int open(const char *pathname, int flags);
  - ssize_t read(int fd, void * buf, size_t count);
  - ssize_t write(int fd,const void*buf,size_t count);

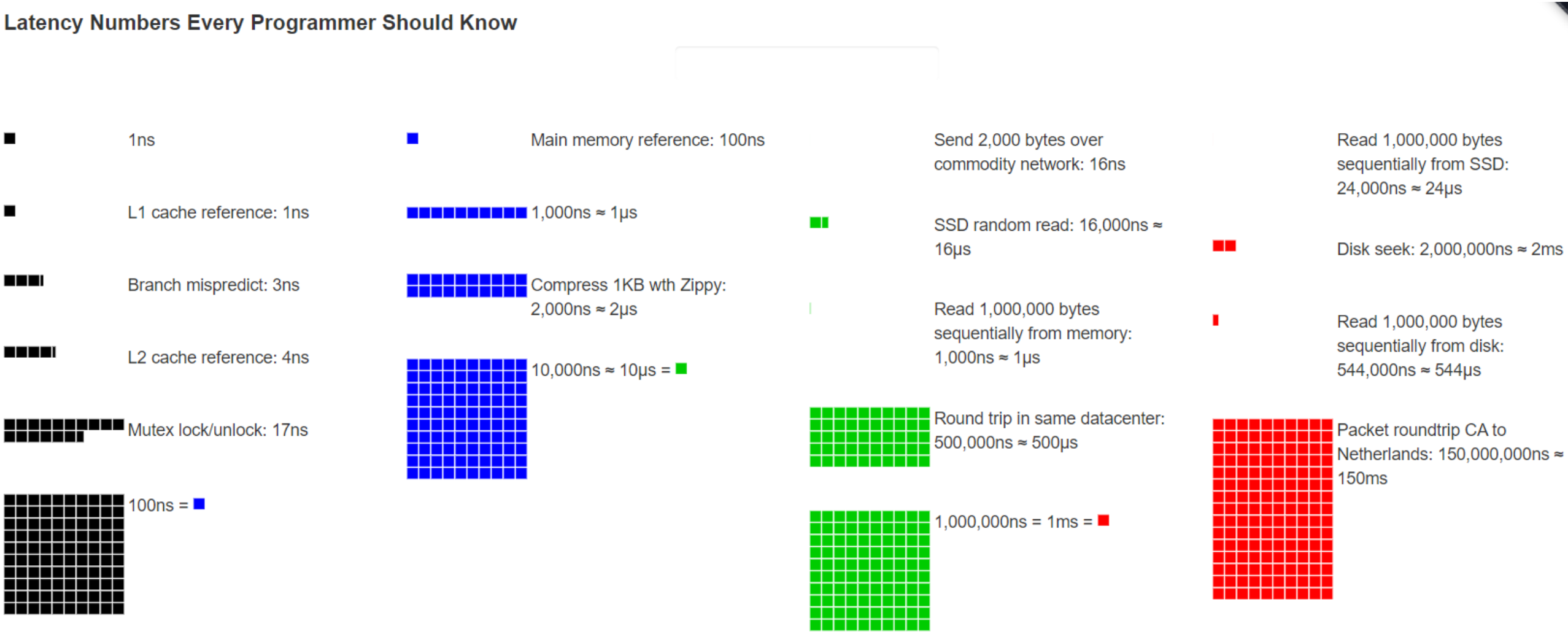
文件读写相关的系统调用

- 使用read系统调用实现从大规模文件中读取特定字符串
  - 从new.txt文件中筛选ilug-admin@linux.ie在八月份发送的邮件

```c
int main(){
    int fd=open("new.txt",O_RDONLY);//打开文件
    if(fd==-1){
        printf("can't open the file");
        return 1;
    }
    int len=1;
    int status=0;
    regmatch_t pmatch[1];
    regex_t reg;
    int count=0;//统计特定字符串的数目
    char pattern[]="^From ilug-admin@linux.ie.*Aug.*";//查找ilug-admin@linux.ie 在八月份发送的邮件

    status=regcomp(&reg,pattern,REG_EXTENDED|REG_NEWLINE);//编译正则表达式
    if(status!=0){
        printf("compile error!\n");
        return -1;
    }
    char output[1024]={"\0"};
    while(len>0){
        char buf[2048]={"\0"};
        len=read(fd,buf,2048);//每次读取 2KB
        char *temp=buf;
        while(1){
            status=regexec(&reg,temp,1,pmatch,0);//匹配正则表达式
            if(status==0){
                count++;

strncpy(output,temp+pmatch[0].rm_so,pmatch[0].rm_eo-pmatch[0].rm_so);//匹配特定字符串
                printf("matched:%s\n",output);
                temp += pmatch[0].rm_eo;
            }
            else break;
        }
    }
    regfree(&reg);//释放正则表达式
    printf("the number of matched string:%d\n",count);
    return 0;
}
```

为什么文件操作前需要open?

●常见设备的速度差异(2022)



- 常见设备的速度差异(2012)

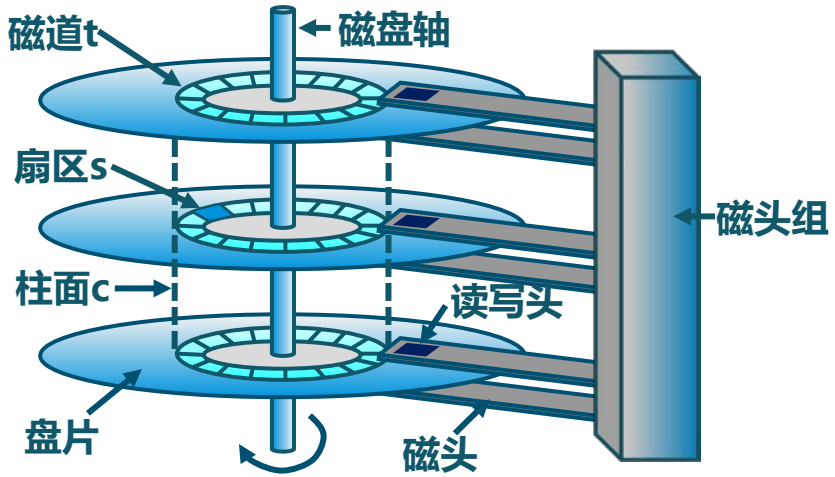
```
Latency Comparison Numbers (~2012)
----------------------------------
L1 cache reference                           0.5 ns
Branch mispredict                            5   ns
L2 cache reference                           7   ns                      14x L1 cache
Mutex lock/unlock                           25   ns
Main memory reference                      100   ns                      20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy             3,000   ns        3 us
Send 1K bytes over 1 Gbps network       10,000   ns       10 us
Read 4K randomly from SSD*             150,000   ns      150 us          ~1GB/sec SSD
Read 1 MB sequentially from memory     250,000   ns      250 us
Round trip within same datacenter      500,000   ns      500 us
Read 1 MB sequentially from SSD*     1,000,000   ns    1,000 us    1 ms  ~1GB/sec SSD, 4X memory
Disk seek                           10,000,000   ns   10,000 us   10 ms  20x datacenter roundtrip
Read 1 MB sequentially from disk    20,000,000   ns   20,000 us   20 ms  80x memory, 20X SSD
Send packet CA->Netherlands->CA    150,000,000   ns  150,000 us  150 ms
```

By Jeff Dean: http://research.google.com/people/jeff/

提问：过去的10年发生了什么？

- 硬盘为什么会这么慢?

磁盘工作机制和性能参数

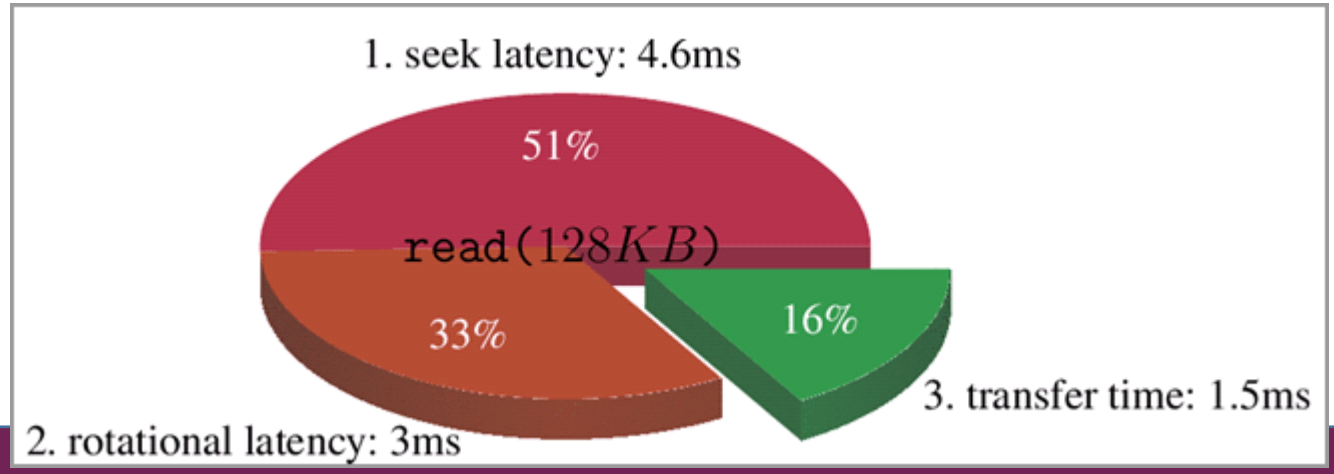


- 读取或写入时，磁头必须被定位在期望的磁道，并从所期望的柱面和扇区的开始
- 寻道时间
  - 定位到期望的磁道所花费的时间
- 旋转延迟
  - 从零扇区开始处到达目的地花费的时间

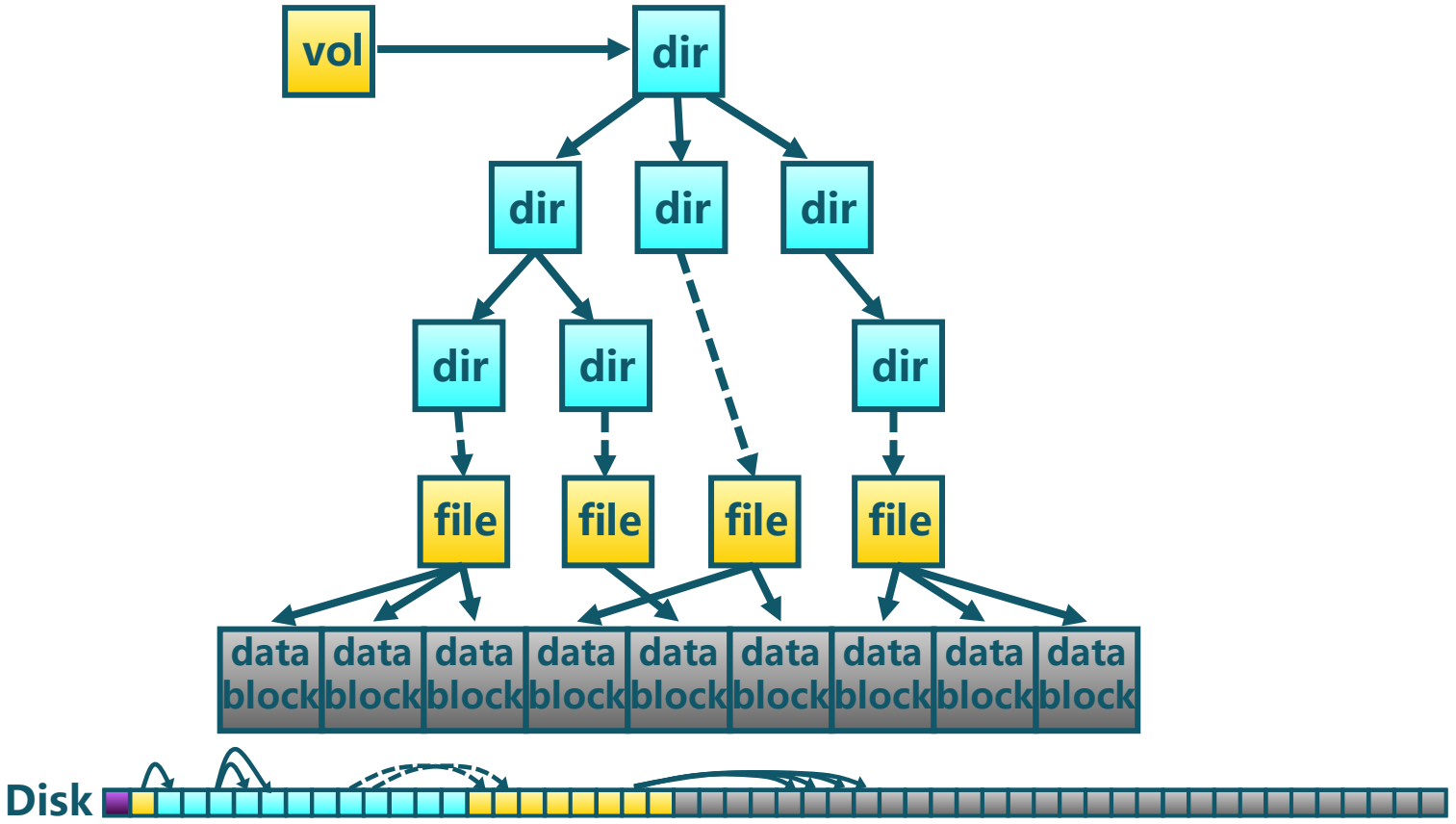
平均旋转延迟时间=磁盘旋转一周时间的一半

磁盘I/O传输时间

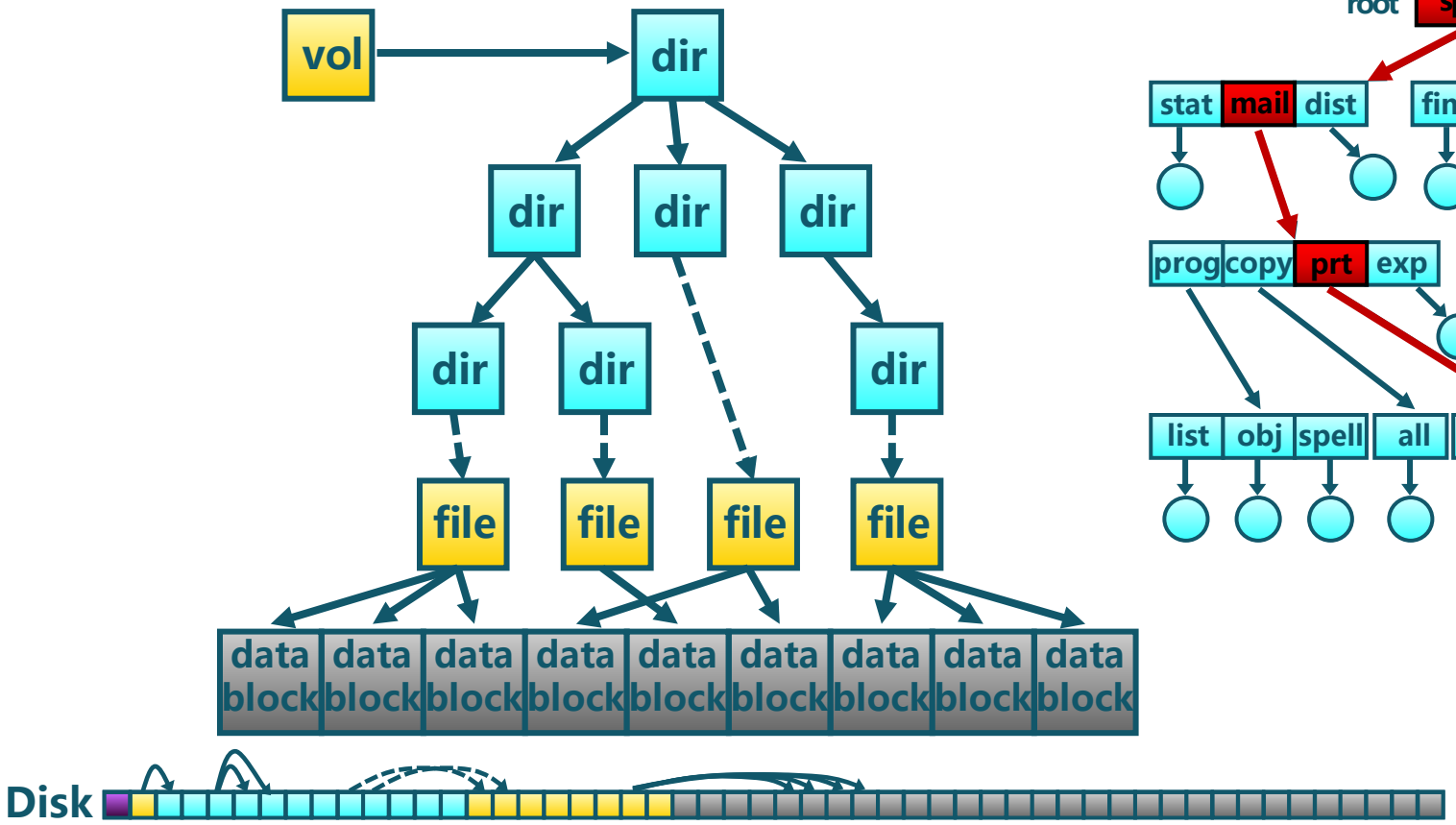
$$T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$$

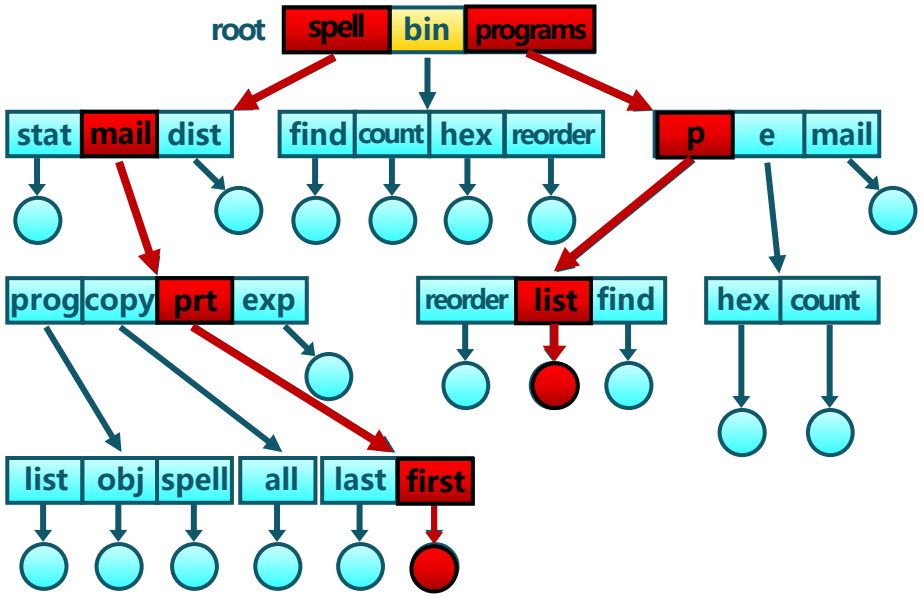


文件系统的组织形态


vol
dir
dir
dir
dir
dir
dir
dir
dir
file
file
file
file
data block
data block
data block
data block
data block
data block
data block
data block
data block
Disk

文件系统的组织形态






```
/spell/mail/prt/first

/programs/p/list
```

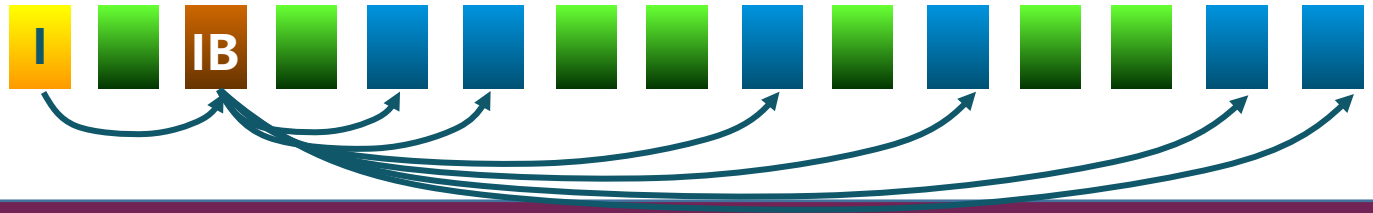
为什么需要文件描述符？记录文件的存储位置

链式分配

- 文件以数据块链表方式存储
- 文件头包含了到第一块和最后一块的指针

索引分配

- 为每个文件创建一个索引数据块
  - 指向文件数据块的指针列表
- 文件头包含了索引数据块指针



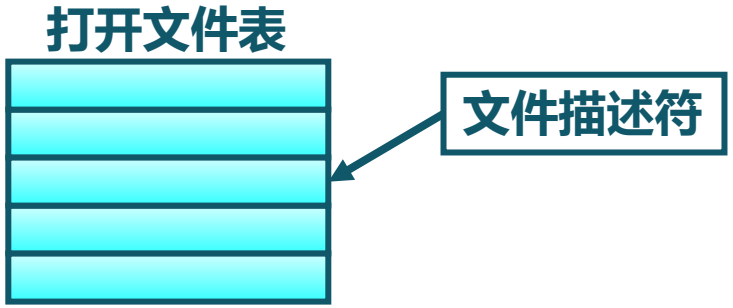
打开文件和文件描述符

- 文件访问模式
  - 进程访问文件数据前必须先“打开”文件

```
f = open(name, flag);
...
read(f, ...);
...
close(f);
```

打开文件和文件描述符

- 文件访问模式
  - 进程访问文件数据前必须先“打开”文件
- 内核跟踪进程打开的所有文件
  - 操作系统为每个进程维护一个打开文件表
  - 文件描述符是打开文件的标识



打开文件表的项数是有限的，为什么？
如何查看打开文件表？

文件描述符

- 操作系统在打开文件表中维护的打开文件状态和信息
  - 文件指针
  - 文件打开计数
  - 文件的磁盘位置
  - 访问权限

- makefile指定整个工程的编译规则，告诉make命令如何构建应用程序

```
target... : prerequisites...
command
...
```

- target目标文件依赖于prerequisites中的文件，其生成规则定义在command中。

Makefile

```
obj=lab3.o

lab3:$(obj)
    cc -o lab3 $(obj)

lab3.o:lab3.c
    cc -c lab3.c

clean:
    rm lab3 $(obj)

```

- 使用free命令观测使用read系统调用消耗的内存资源
  - new.txt文件大小约为71KB

```
kjr@ubuntu:~$ free
              total        used        free      shared  buff/cache   available
Mem:        2059820      478052     1090480       12796      491288     1335844
Swap:       2094076           0     2094076
kjr@ubuntu:~$ free
              total        used        free      shared  buff/cache   available
Mem:        2059820      478216     1090244       12796      491360     1335684
Swap:       2094076           0     2094076
```

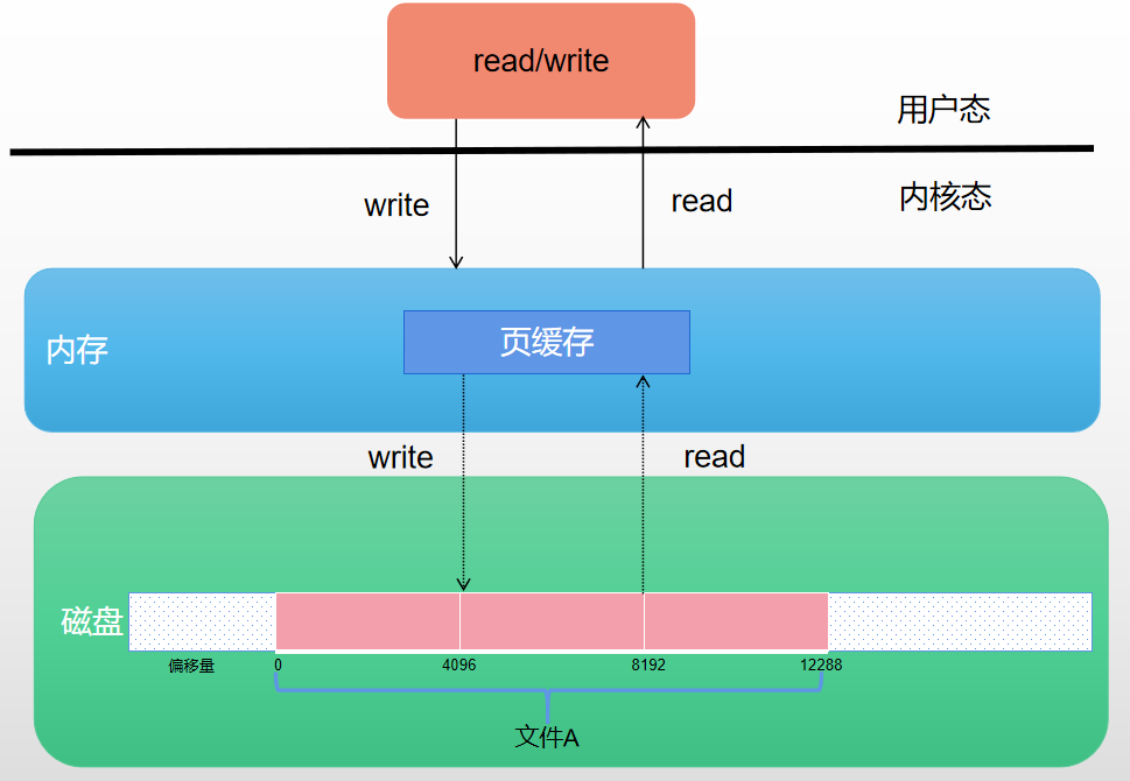
程序运行前

程序运行后

  - 消耗的buffer为72KB
  - 占用的物理内存为164KB

文件读写相关的系统调用

- 使用读写接口产生不必要的内存消耗
  - 相同的文件内容在内存中存储了两次

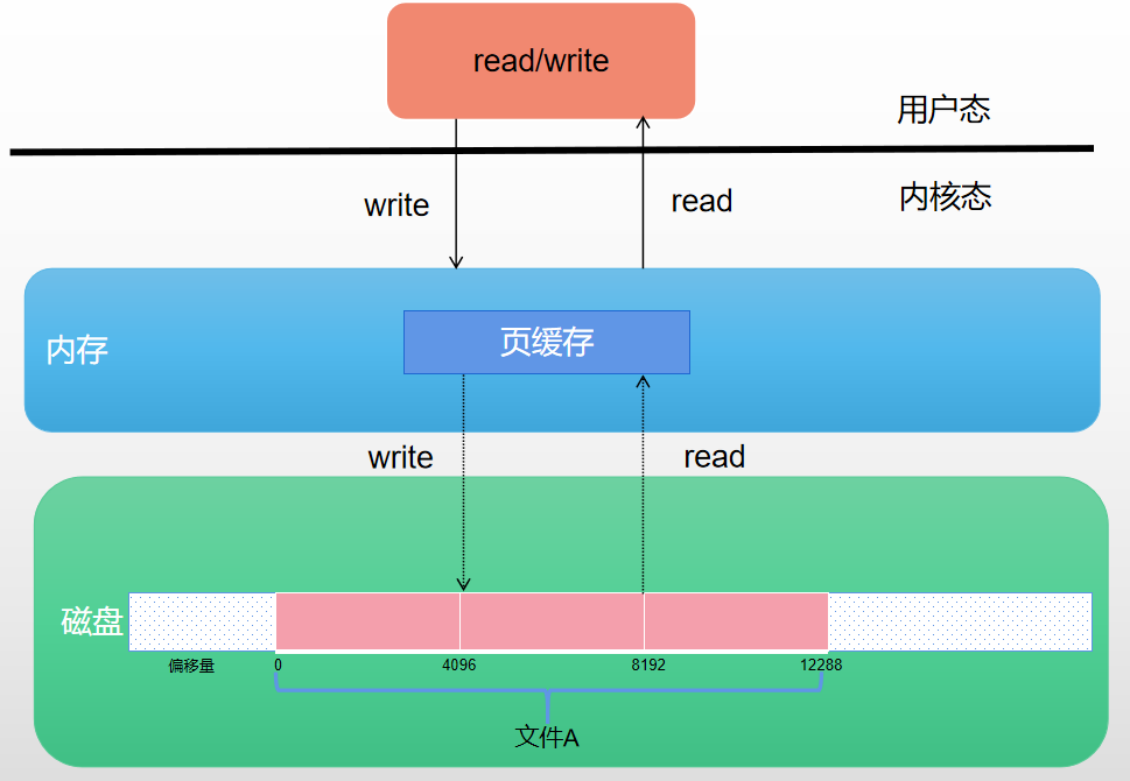


提问：借助read操作可以节省内存吗？

提问：多一份copy的价值是什么？

文件读写相关的系统调用

- 使用读写接口产生不必要的内存消耗
  - 如果不会产生数据的修改，能否省下这一份数据复制呢？

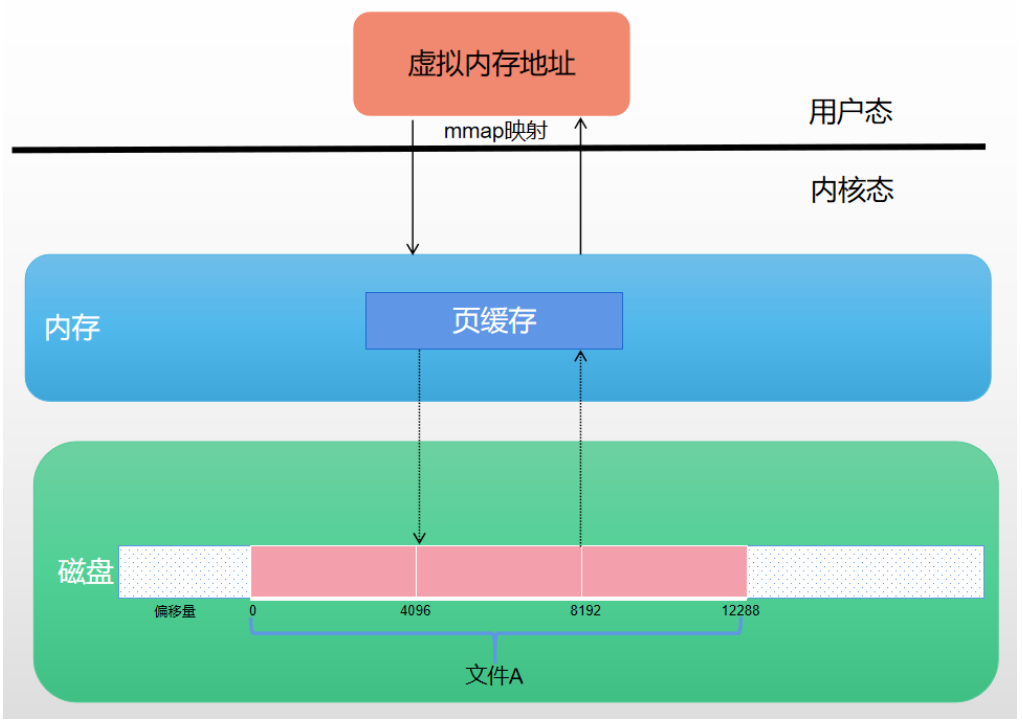


解决方法 → 内存映射相关的系统调用

内存映射相关的系统调用

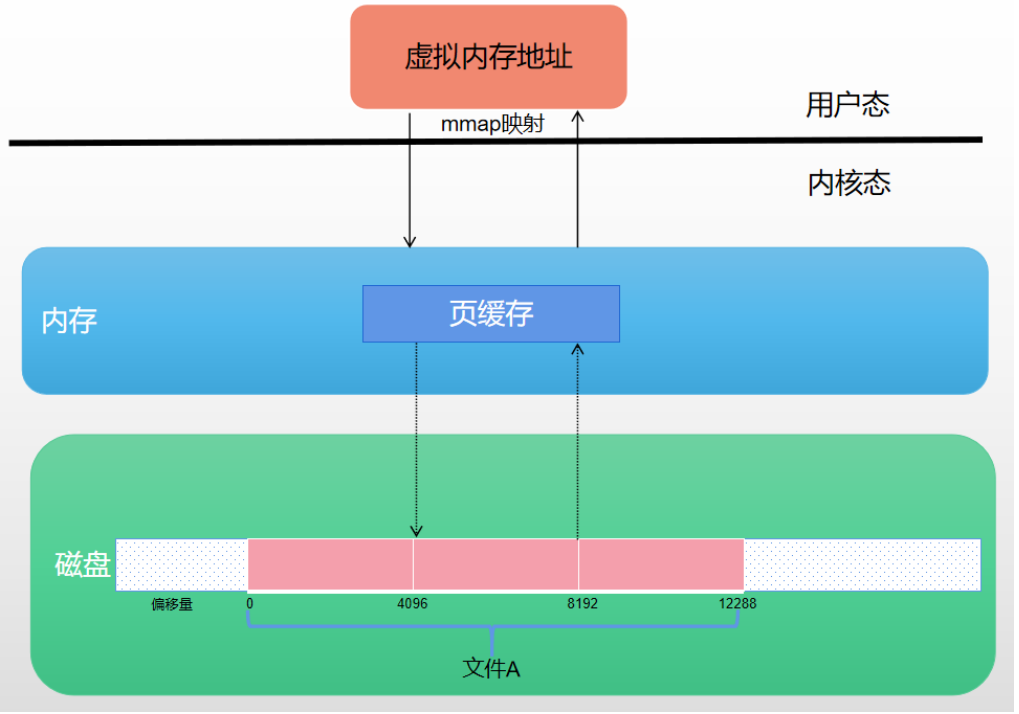
内存映射相关的系统调用

- mmap原理
  - 引入虚拟内存地址
  - 将文件内容映射到进程虚拟地址空间
  - 进程采用指针方式读写内存



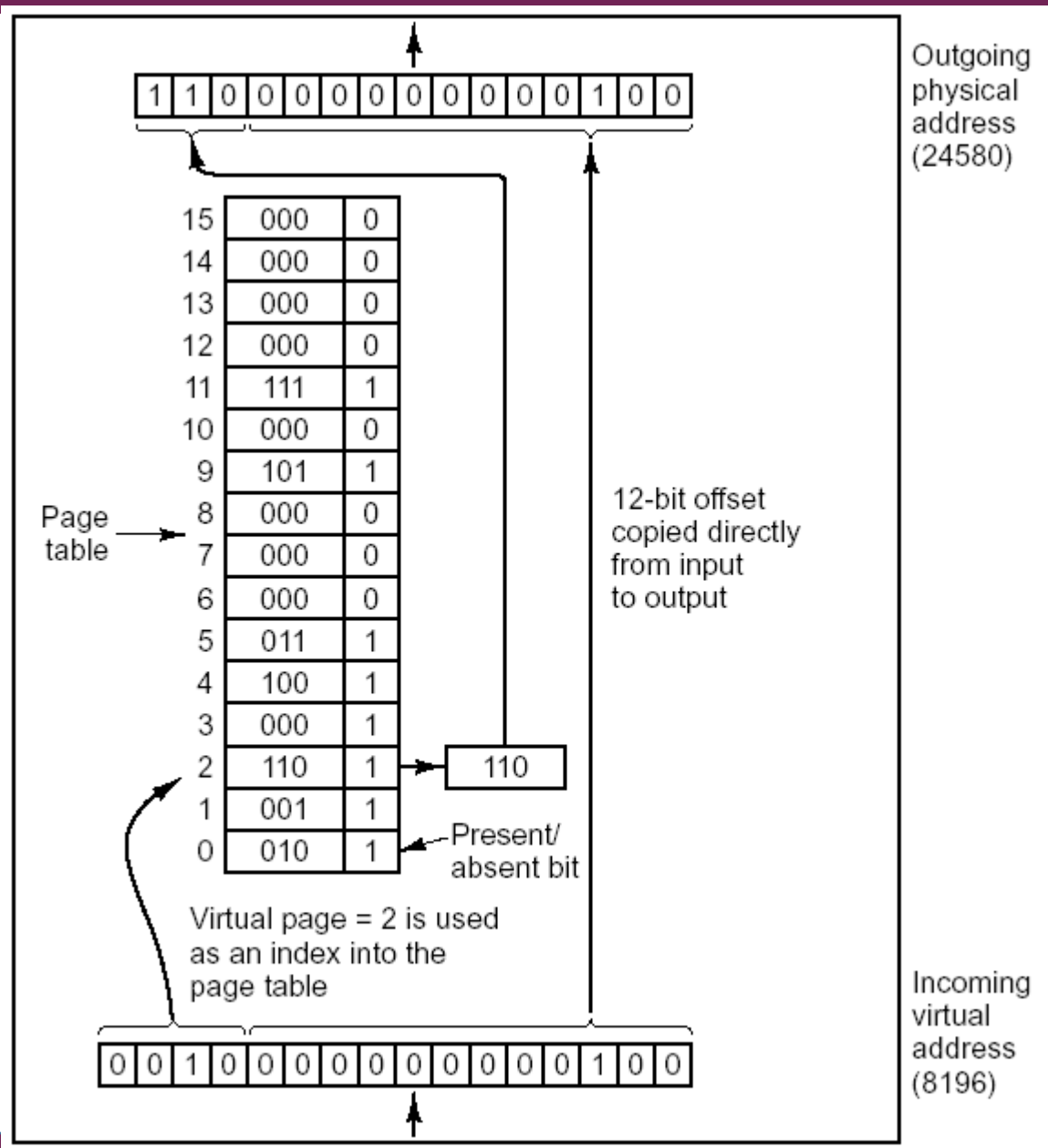
内存映射相关的系统调用

- mmap原理
  - 引入虚拟内存地址
  - 将文件内容映射到进程虚拟地址空间
  - 进程采用指针方式读写内存



- mmap系统调用函数原型：
  - void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

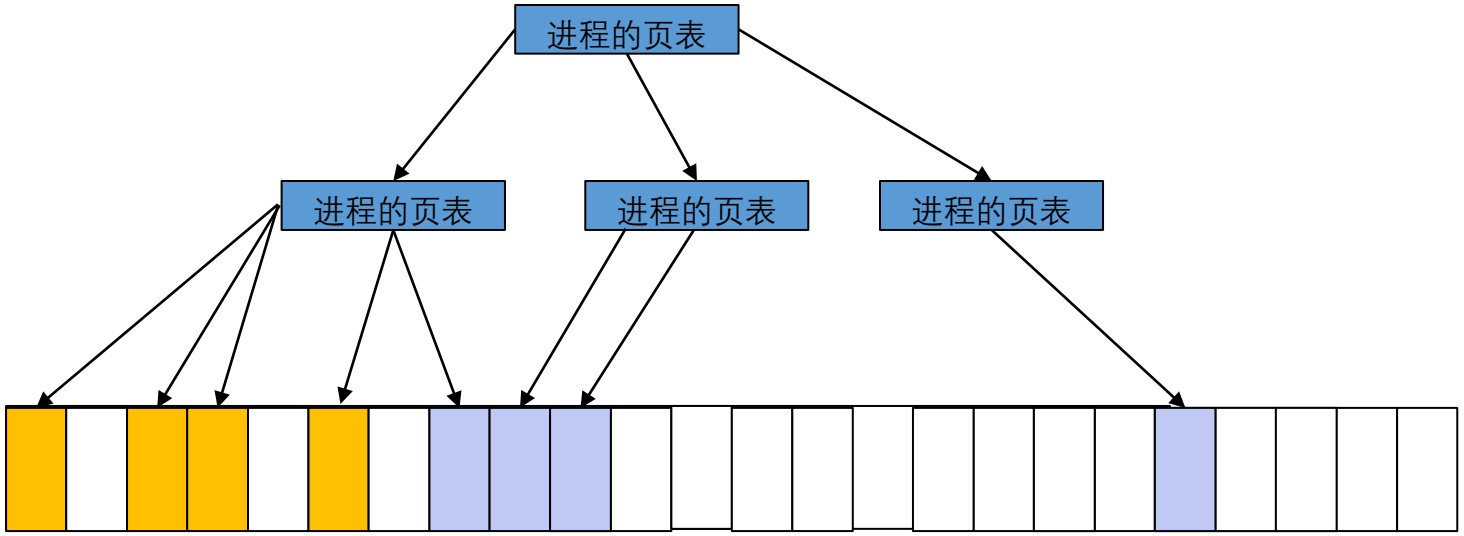
内存映射相关的系统调用



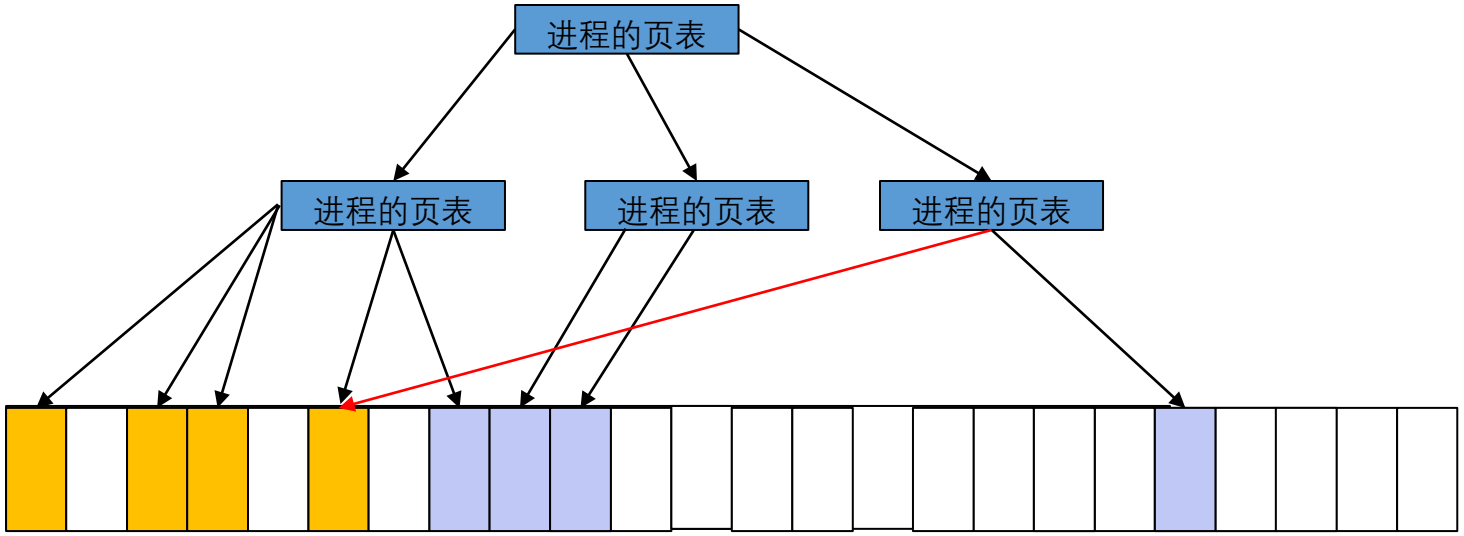
页表提供了一次物理地址到虚拟地址的转换机会

利用这个机会，可以解决相同的内容二次存放的问题

内存映射相关的系统调用

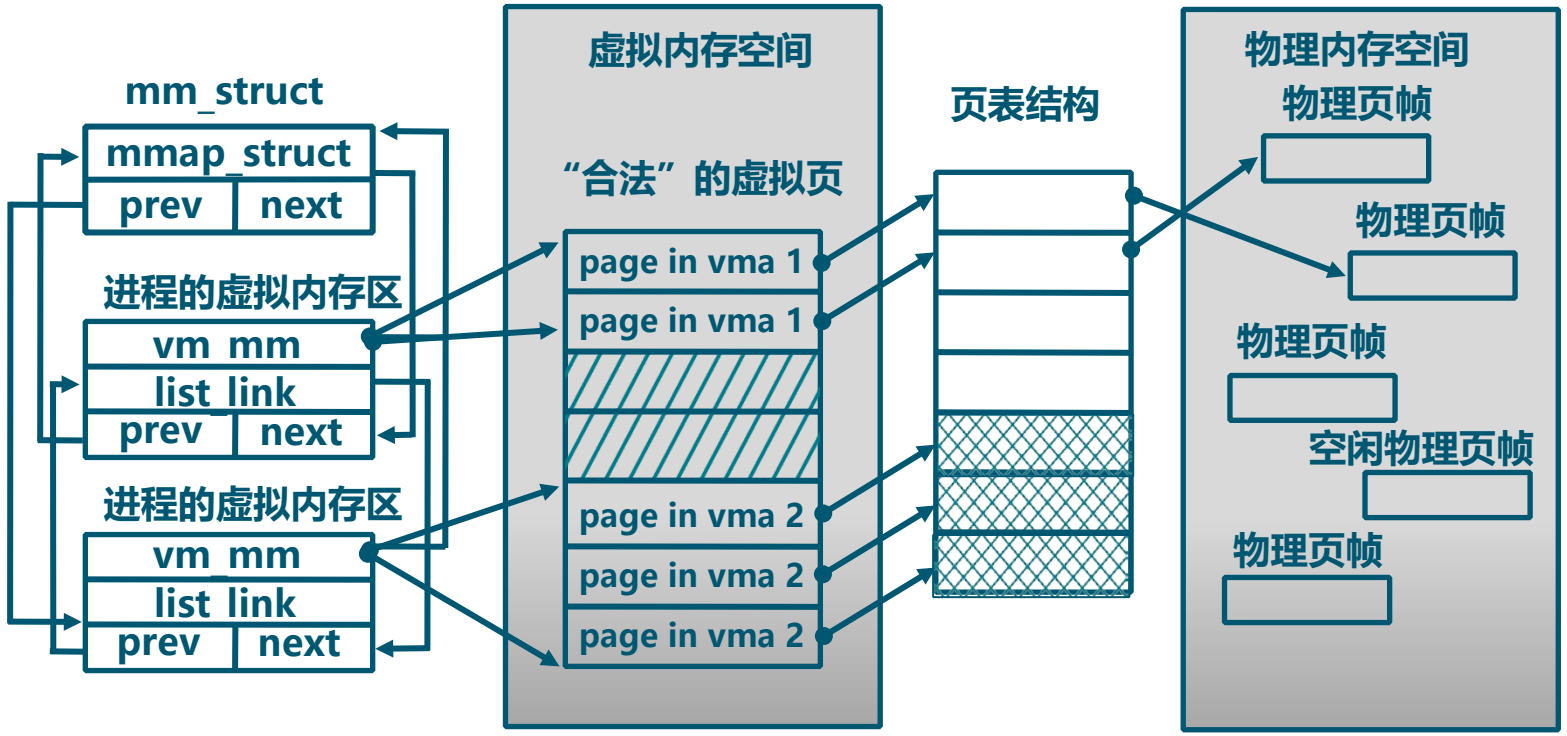


内存映射相关的系统调用



将一个物理页帧同时映射到两个虚拟页上，并没有破坏页表的结构，反而起到了节省内存的效果

内存映射相关的系统调用



mmap的功能是创建了一块“合法”但是不存在的虚拟内存地址区域，并且指明了这块区域的数据由哪个文件的内容来填充

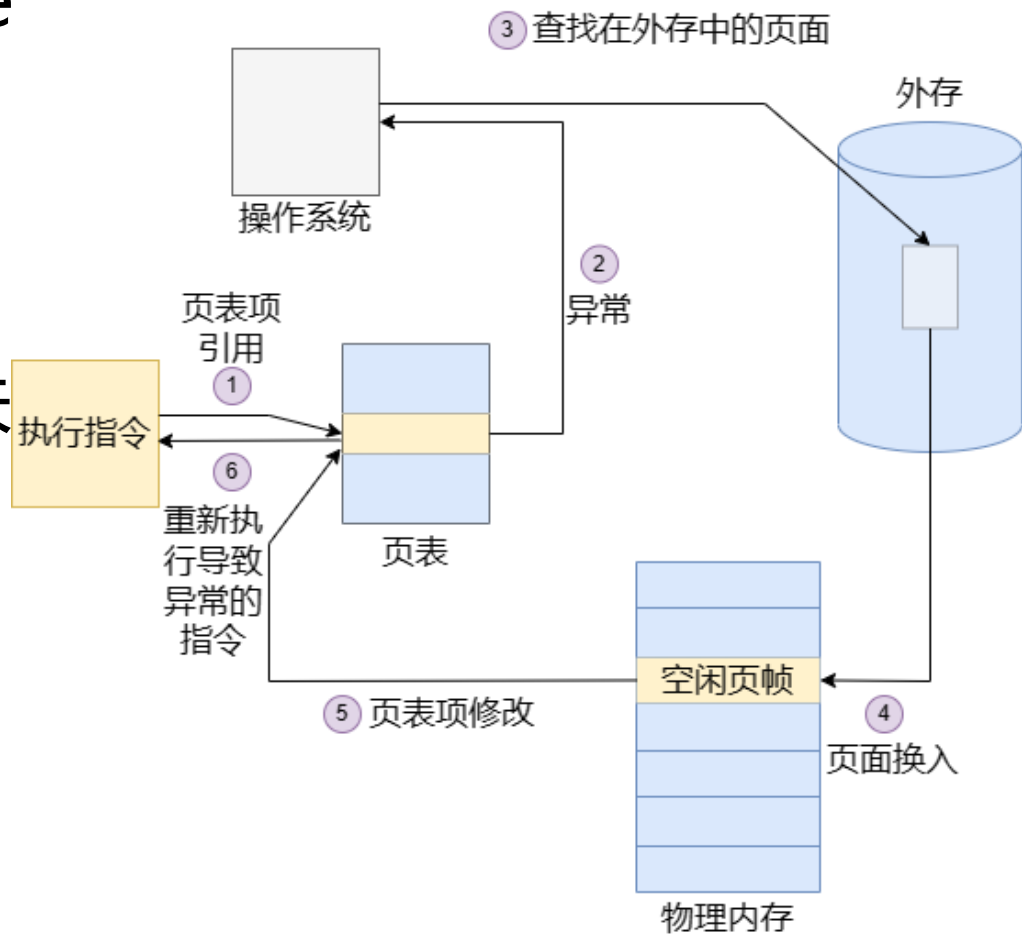
内存映射相关的系统调用

- 使用mmap系统调用实现从大规模文件中读取特定字符串
  - 将整个文件的内容以可读的方式映射到进程的虚拟内存，并得到地址

```c
int main(){
    int fd=open("new.txt",O_RDONLY);
    if(fd==-1){
        printf("can't open the file");
        return 1;
    }
    struct stat sb;
    if(fstat(fd,&sb)==-1) printf("fstat error!");
    char *mmapped;

if((mmapped=mmap(NULL,sb.st_size,PROT_READ,MAP_SHARED,fd,0))==(voi
d *)-1) printf("mmapped error!");
    close(fd);

    int status=0;
    regmatch_t pmatch[1];
    regex_t reg;
    int count=0;
    char pattern[]="^From ilug-admin@linux.ie.*Aug.*";//查找
ilug-admin@linux.ie 在八月份发送的邮件
    status=regcomp(&reg,pattern,REG_EXTENDED|REG_NEWLINE);
    if(status!=0){
        printf("compile error!\n");
        return -1;
    }

    char output[1024]={"\0"};
    while(1){
        status=regexec(&reg,mmapped,1,pmatch,0);
        if(status==0){
            count++;

strncpy(output,mmapped+pmatch[0].rm_so,pmatch[0].rm_eo-pmatch[0].r
m_so);
            printf("matched:%s\n",output);
            mmapped += pmatch[0].rm_eo;
        }
        else break;
    }
    regfree(&reg);
    printf("the number of matched string:%d\n",count);
    return 0;
}
```

内存映射相关的系统调用

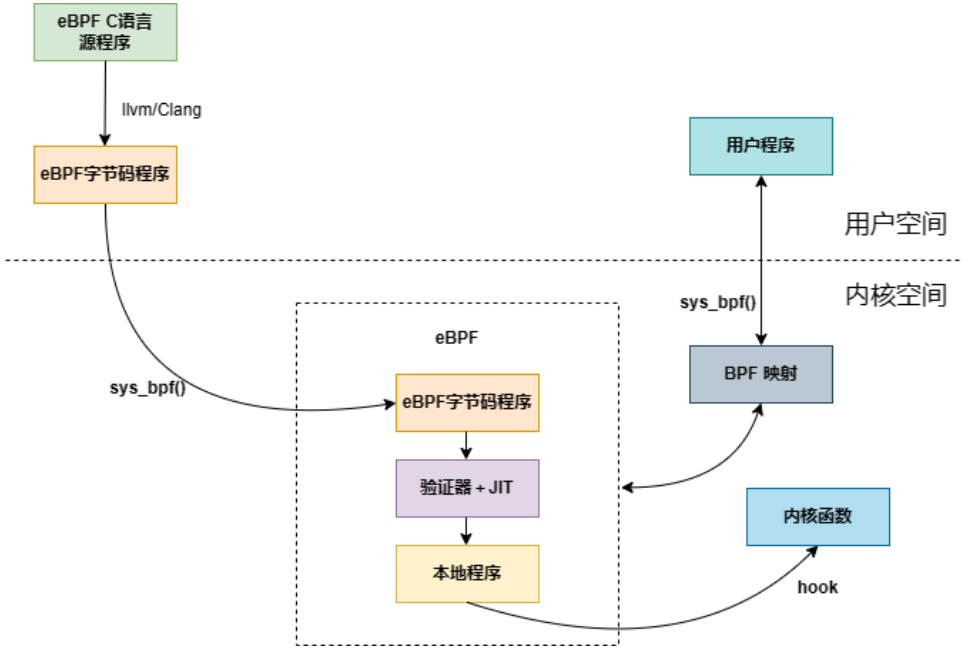
- mmap的加载过程
- 进程第一次访问虚拟地址空间时会发生page fault
- page fault过程分析
  - mmap在进程的虚拟地址空间里制造了未分配物理页的空间
  - 借助mmap调用OS记录了虚拟地址与外存的对应关系
  - 在page fault handler中完成加载



观测虚拟内存的机制

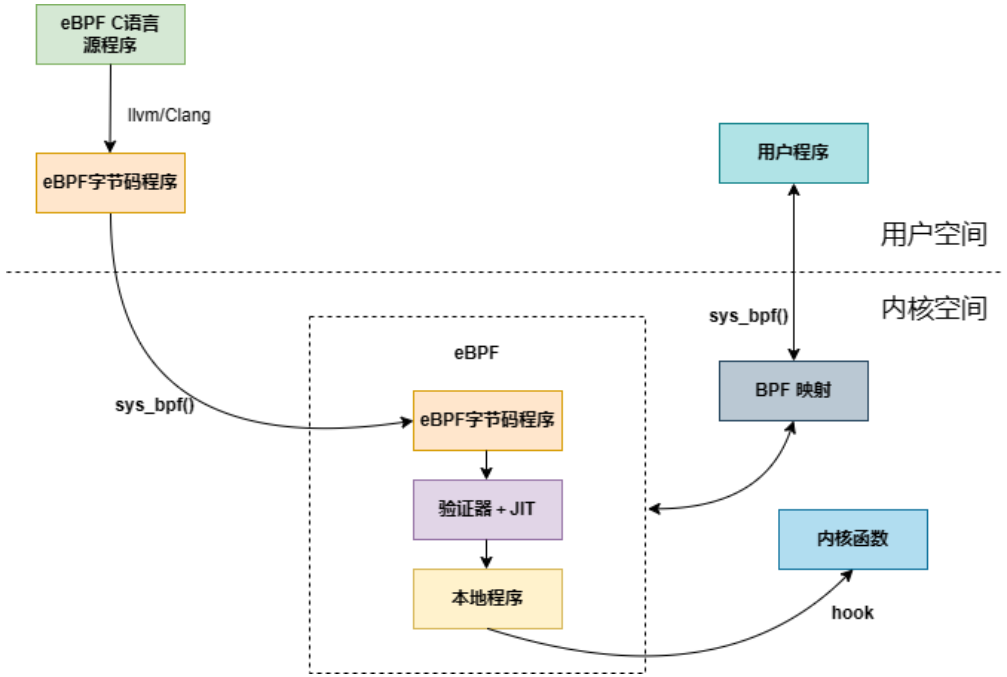
内存映射相关的系统调用

- eBPF(extended Berkeley Packet Filter)是一个可编程的内核扩展框架，允许在内核中注入自定义代码，以对特定事件进行观测和处理。
- eBPF原理
  - LLVM/Clang将编写的eBPF 程序转换为BPF字节码。
  - 通过 bpf 系统调用提交给内核执行。



内存映射相关的系统调用

- 内核在接受 BPF 字节码之前，首先通过验证器对字节码进行校验，只有校验通过的 BPF 字节码才会提交到即时编译器执行。
- BPF 程序可以利用 BPF 映射进行存储，而用户程序通常也需要通过 BPF 映射同运行在内核中的 BPF 程序进行交互。



内存映射相关的系统调用

- 使用eBPF观测使用mmap时发生的缺页事件
  - 定义mmap系统调用为hook来观测page fault

```
sudo bpftrace -l '*syscalls*mmap*'  -v
```

    - -l选项用于列出与指定的BPFtrace程序匹配的BPF程序。这里表示列出所有包含syscalls和mmap的BPF程序。
    - -v选项用于启用详细输出模式。打印更多的信息，如BPFtrace程序的解析、编译和加载过程中的详细步骤，以及其他相关信息。
  - 编写bpftrace程序，使用bpftrace的tracepoint功能来追踪syscalls:sys_exit_mmap和exceptions:page_fault_user事件。

内存映射相关的系统调用

```
55  tracepoint:syscalls:sys_exit_mmap
56  {
57      $fname = @mmap_fname[tid];
58      $mmap_start = args->ret;
59      $mmap_len = @mmap_len[tid];
60      printf("mmap(\"%s-%d\", fd=%d, fname=\"%s\",off=%d, len=%d, ret=0x%lX)\n",
61              comm, pid, @mmap_fd[tid], str($fname), @mmap_off[tid], $mmap_len, $mmap_start);
62  }
63
64  tracepoint:exceptions:page_fault_user
65  {
66      printf("page_fault_user(\"%s-%d\", address=0x%lX, ip=0x%lX)\n", comm, pid, args->address, args->ip);
67  }
```

- 当mmap系统调用完成时触发sys_exit_mmap的tracepoint。
  - 打印有关mmap调用的信息，包括进程ID、文件描述符（fd）、文件名（fname）、偏移量（off）、映射长度（len）和返回值（ret）。
- 当用户空间发生页面错误时触发page_fault_user的tracepoint。
  - 它将打印有关页面错误的信息，包括进程ID、进程名称、错误地址和指令指针。

内存映射相关的系统调用

- 使用eBPF观测使用mmap时发生的缺页事件
  - 运行bpftrace程序，将观测的结果输出到faults.txt文件中

```
sudo bpftrace mmap_pagefault_snoop.bt > faults.py
```

  - 运行map_file_pattern.c时的观测结果：

```
page_fault_user("map_file_patter-3550795", address=0x7FFFF7F8FA94, ip=0x7FFFF7E04D52)
page_fault_user("map_file_patter-3550795", address=0x7FFFF7E99AA0, ip=0x7FFFF7E99AA0)
page_fault_user("map_file_patter-3550795", address=0x7FFFF7F28140, ip=0x7FFFF7F28140)
page_fault_user("map_file_patter-3550795", address=0x5555555551C0, ip=0x5555555551C0)
page_fault_user("map_file_patter-3550795", address=0x7FFFF7DB98C0, ip=0x7FFFF7DB98C0)
page_fault_user("map_file_patter-3550795", address=0x7FFFF7E600B0, ip=0x7FFFF7E600B0)
page_fault_user("map_file_patter-3550795", address=0x7FFFF7DD4770, ip=0x7FFFF7DD4770)
page_fault_user("map_file_patter-3550795", address=0x7FFFF7DE90B0, ip=0x7FFFF7DE90B0)
page_fault_user("map_file_patter-3550795", address=0x555555556008, ip=0x7FFFF7F25722)
page_fault_user("map_file_patter-3550795", address=0x7FFFF7F8FA70, ip=0x7FFFF7DE95DD)
page_fault_user("map_file_patter-3550795", address=0x7FFFF7DFF680, ip=0x7FFFF7DFF680)
page_fault_user("map_file_patter-3550795", address=0x555555559008, ip=0x7FFFF7E17473)
page_fault_user("map_file_patter-3550795", address=0x7FFFF7DCE280, ip=0x7FFFF7DCE280)
page_fault_user("map_file_patter-3550795", address=0x7FFFF7F50BA5, ip=0x7FFFF7DCE33B)
mmap("map_file_patter-3550795", fd=3, fname="new.txt",off=0, len=72200, ret=0x7FFFF7FA9000)
page_fault_user("map_file_patter-3550795", address=0x7FFFF7E79ED0, ip=0x7FFFF7E79ED0)
```

内存映射相关的系统调用

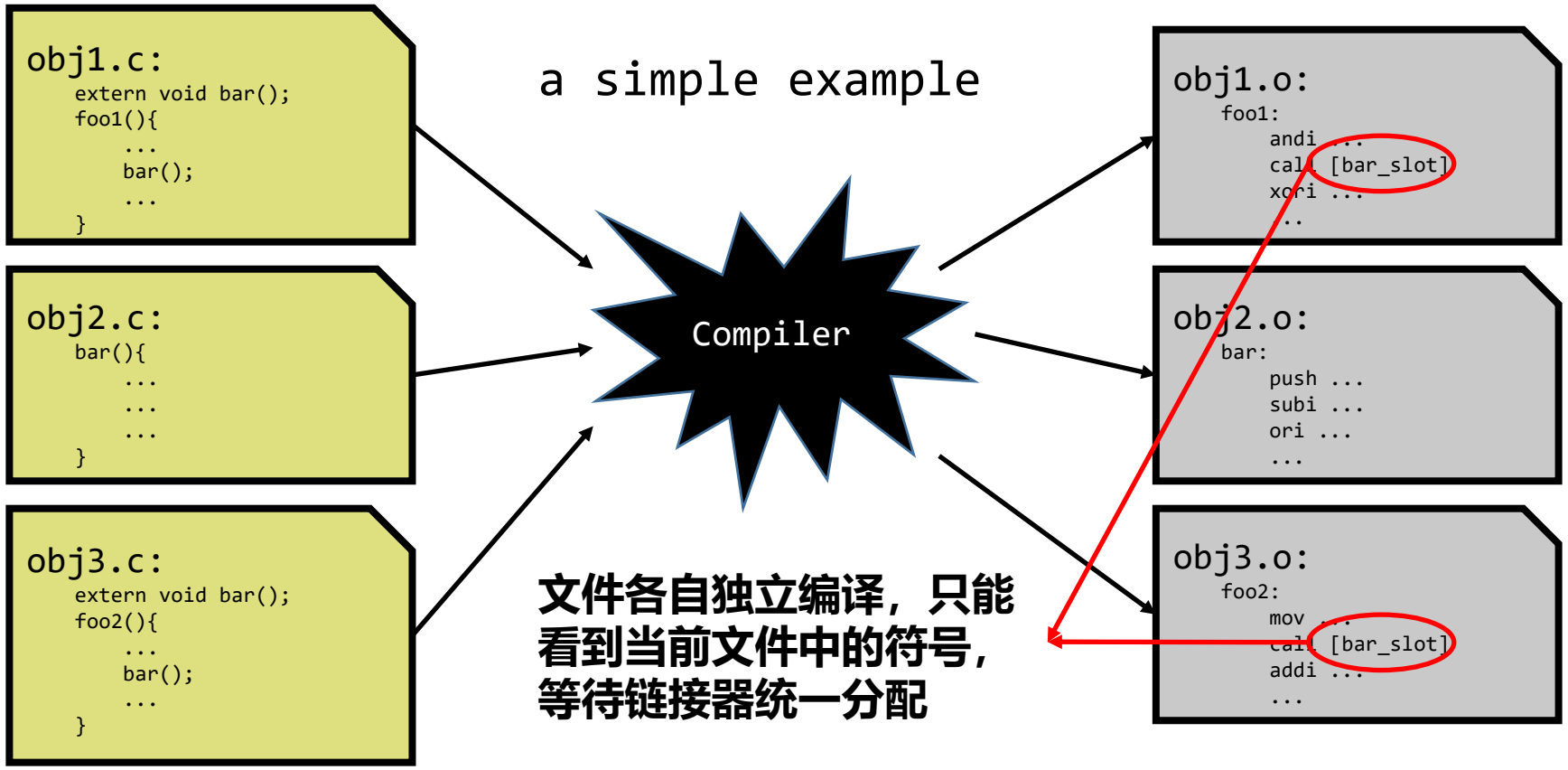
- 使用eBPF观测使用mmap时发生的缺页事件

```
page_fault_user("map_file_patter-3550795", address=0x7FFFF7F8FA94, ip=0x7FFFF7E04D52)
page_fault_user("map_file_patter-3550795", address=0x7FFFF7E99AA0, ip=0x7FFFF7E99AA0)
page_fault_user("map_file_patter-3550795", address=0x7FFFF7F28140, ip=0x7FFFF7F28140)
page_fault_user("map_file_patter-3550795", address=0x5555555551C0, ip=0x5555555551C0)
page_fault_user("map_file_patter-3550795", address=0x7FFFF7DB98C0, ip=0x7FFFF7DB98C0)
page_fault_user("map_file_patter-3550795", address=0x7FFFF7E600B0, ip=0x7FFFF7E600B0)
page_fault_user("map_file_patter-3550795", address=0x7FFFF7DD4770, ip=0x7FFFF7DD4770)
page_fault_user("map_file_patter-3550795", address=0x7FFFF7DE90B0, ip=0x7FFFF7DE90B0)
page_fault_user("map_file_patter-3550795", address=0x555555556008, ip=0x7FFFF7F25722)
page_fault_user("map_file_patter-3550795", address=0x7FFFF7F8FA70, ip=0x7FFFF7DE95DD)
page_fault_user("map_file_patter-3550795", address=0x7FFFF7DFF680, ip=0x7FFFF7DFF680)
page_fault_user("map_file_patter-3550795", address=0x555555559008, ip=0x7FFFF7E17473)
page_fault_user("map_file_patter-3550795", address=0x7FFFF7DCE280, ip=0x7FFFF7DCE280)
page_fault_user("map_file_patter-3550795", address=0x7FFFF7F50BA5, ip=0x7FFFF7DCE33B)
mmap("map_file_patter-3550795", fd=3, fname="new.txt",off=0, len=72200, ret=0x7FFFF7FA9000)
page_fault_user("map_file_patter-3550795", address=0x7FFFF7E79ED0, ip=0x7FFFF7E79ED0)
```

- 如何理解这些观测结果：
  - address是触发page fault时，程序试图读取的内存地址
  - ip是触发page fault时，读取内存的指令所在的内存地址

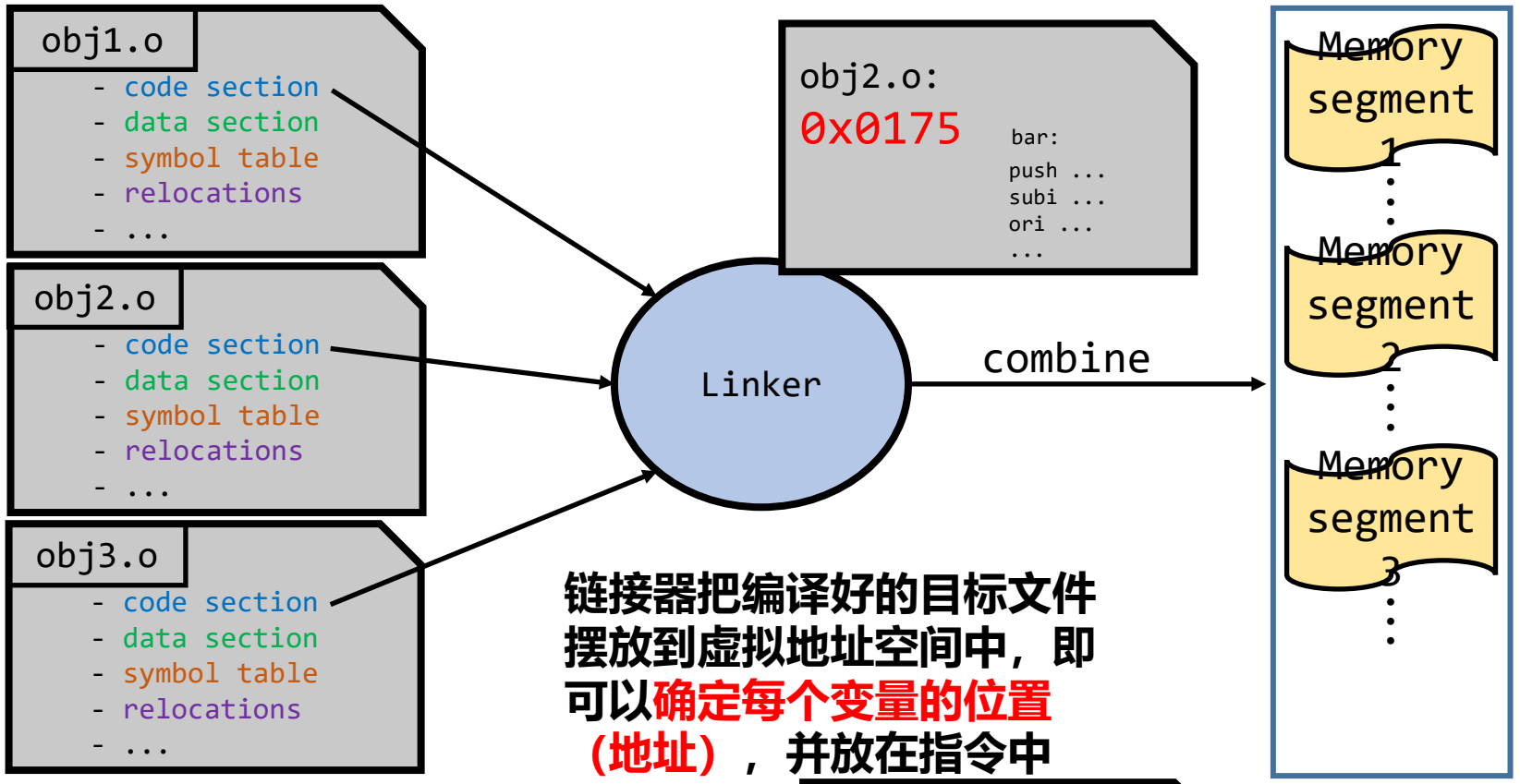
内存映射相关的系统调用

- 程序中的指令所在的内存地址，是在编译链接的时候确定的



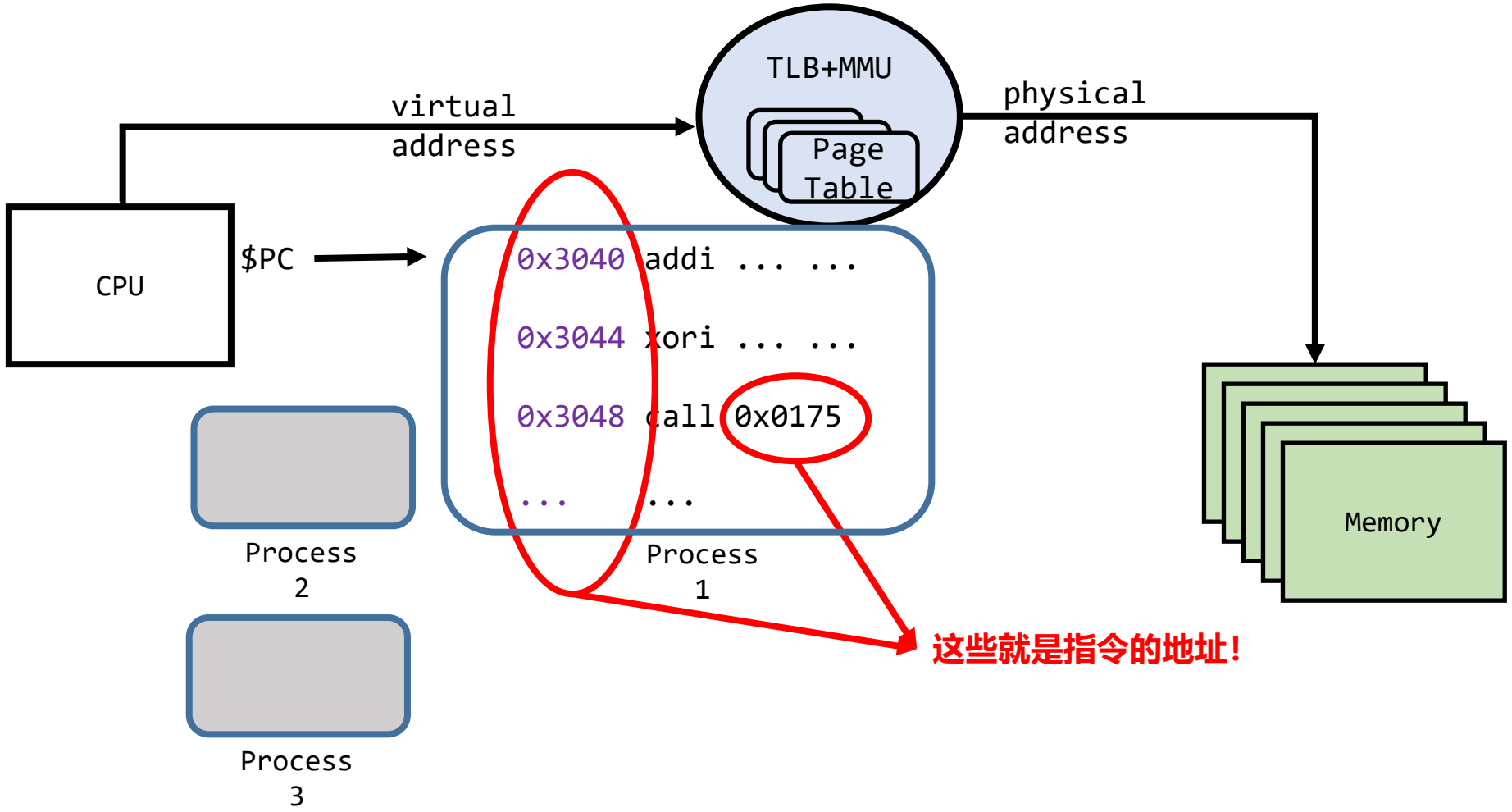
内存映射相关的系统调用

- 程序中的指令所在的内存地址[illegible]链接的时候确定的



内存映射相关的系统调用

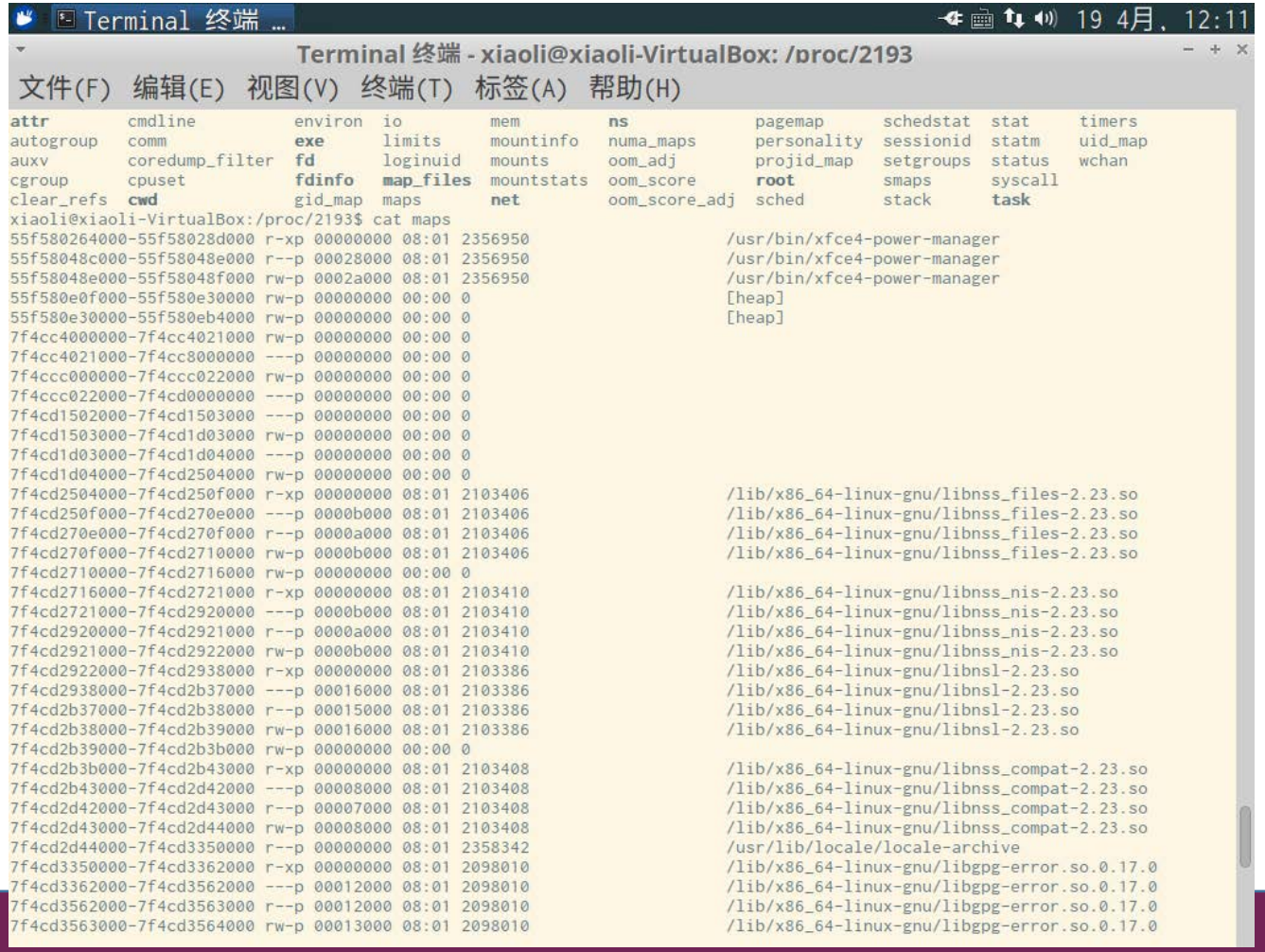
- 程序中的指令所在的内存地址，是在编译链接的时候确定的



- 虚拟地址是在编译时生成的，可以通过readelf查看
  - readelf可以用于查看Linux中二进制文件的相关信息，使用-l –s等参数可以看到段表、符号表等

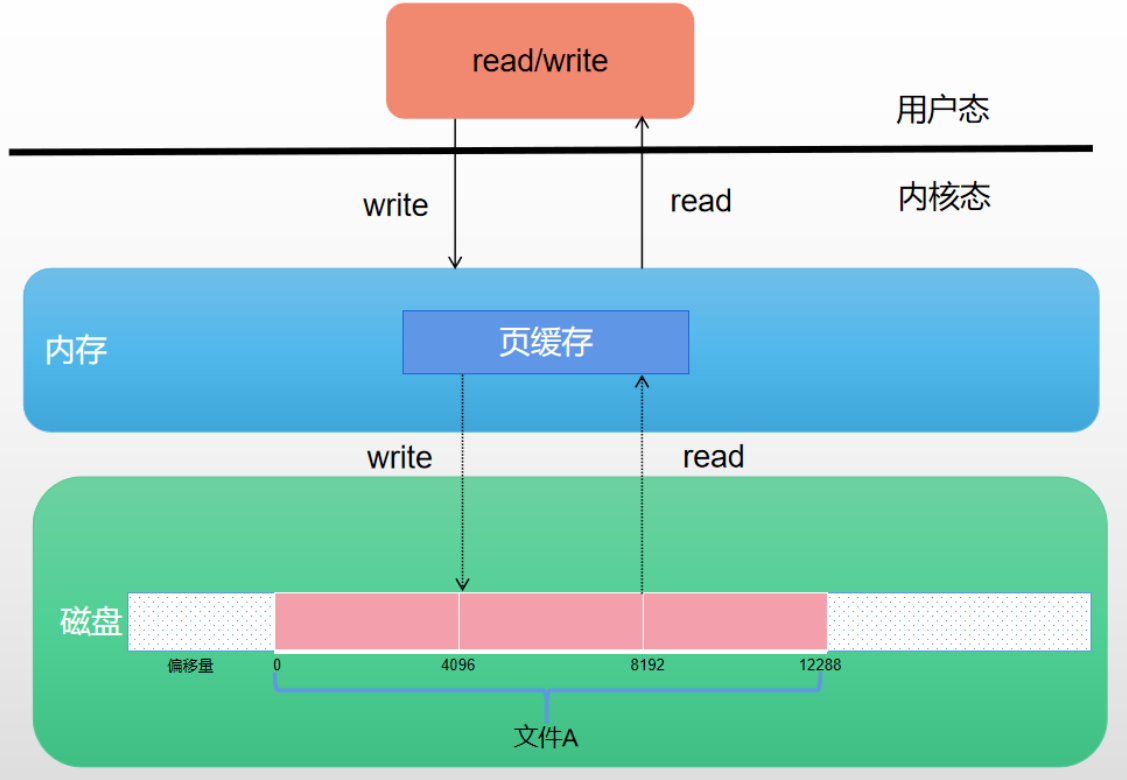
```
xiaoli@xiaoli-VirtualBox:/usr/bin$ readelf -l ./zip

Elf 文件类型为 EXEC (可执行文件)
入口点 0x408500
共有 9 个程序头，开始于偏移量 64

程序头:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R E    8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x000000000002be14 0x000000000002be14  R E    200000
  LOAD           0x000000000002ce00 0x000000000062ce00 0x000000000062ce00
                 0x00000000000019d4 0x00000000000502a8  RW     200000
  DYNAMIC        0x000000000002ce18 0x000000000062ce18 0x000000000062ce18
                 0x00000000000001e0 0x00000000000001e0  RW     8
  NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254
                 0x0000000000000044 0x0000000000000044  R      4
  GNU_EH_FRAME   0x0000000000029574 0x0000000000429574 0x0000000000429574
                 0x0000000000000554 0x0000000000000554  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x000000000002ce00 0x000000000062ce00 0x000000000062ce00
                 0x0000000000000200 0x0000000000000200  R      1

 Section to Segment mapping:
  段节...
   00
```

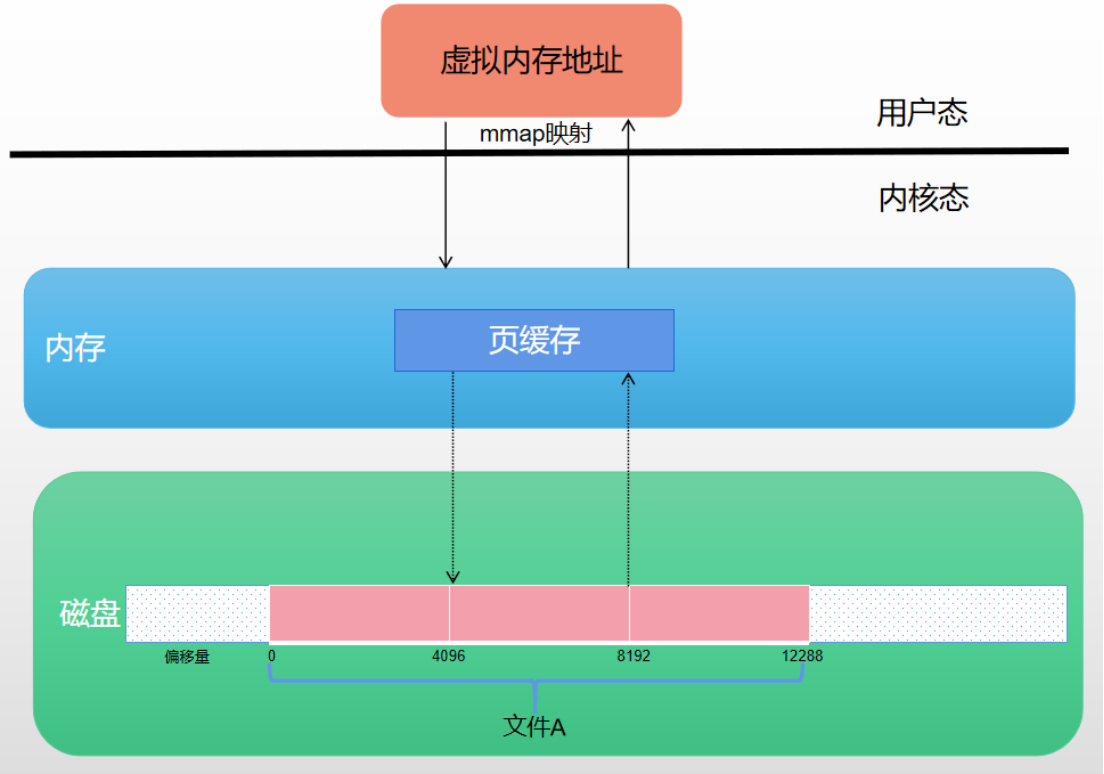
- 在操作系统运行的过程中，系统占用的虚拟内存的布局会存储在相应的数据结构中，通过/proc/PID/maps可以查看



内存映射相关的系统调用

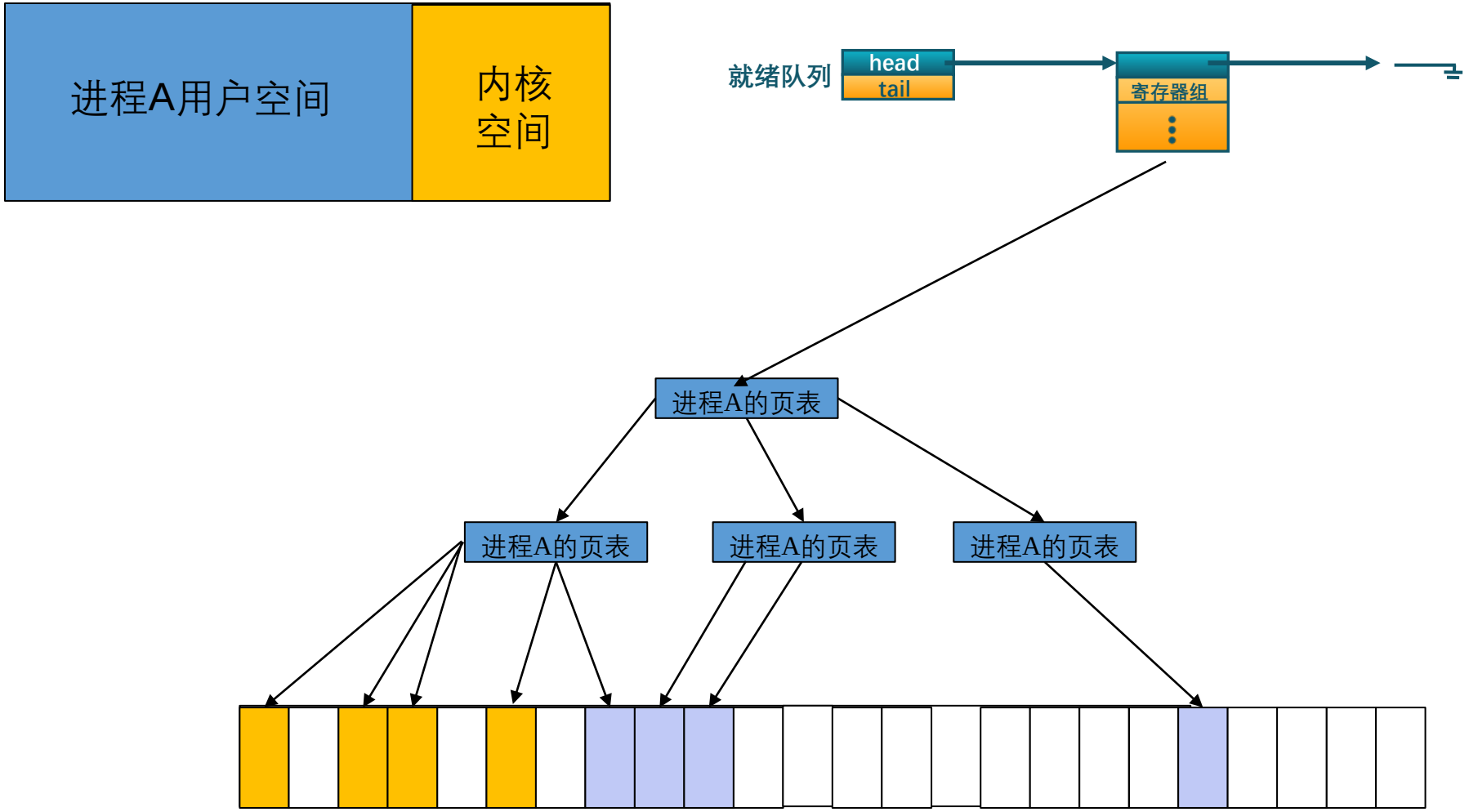
- 提问：相比于read操作，mmap为什么可以省下内存？





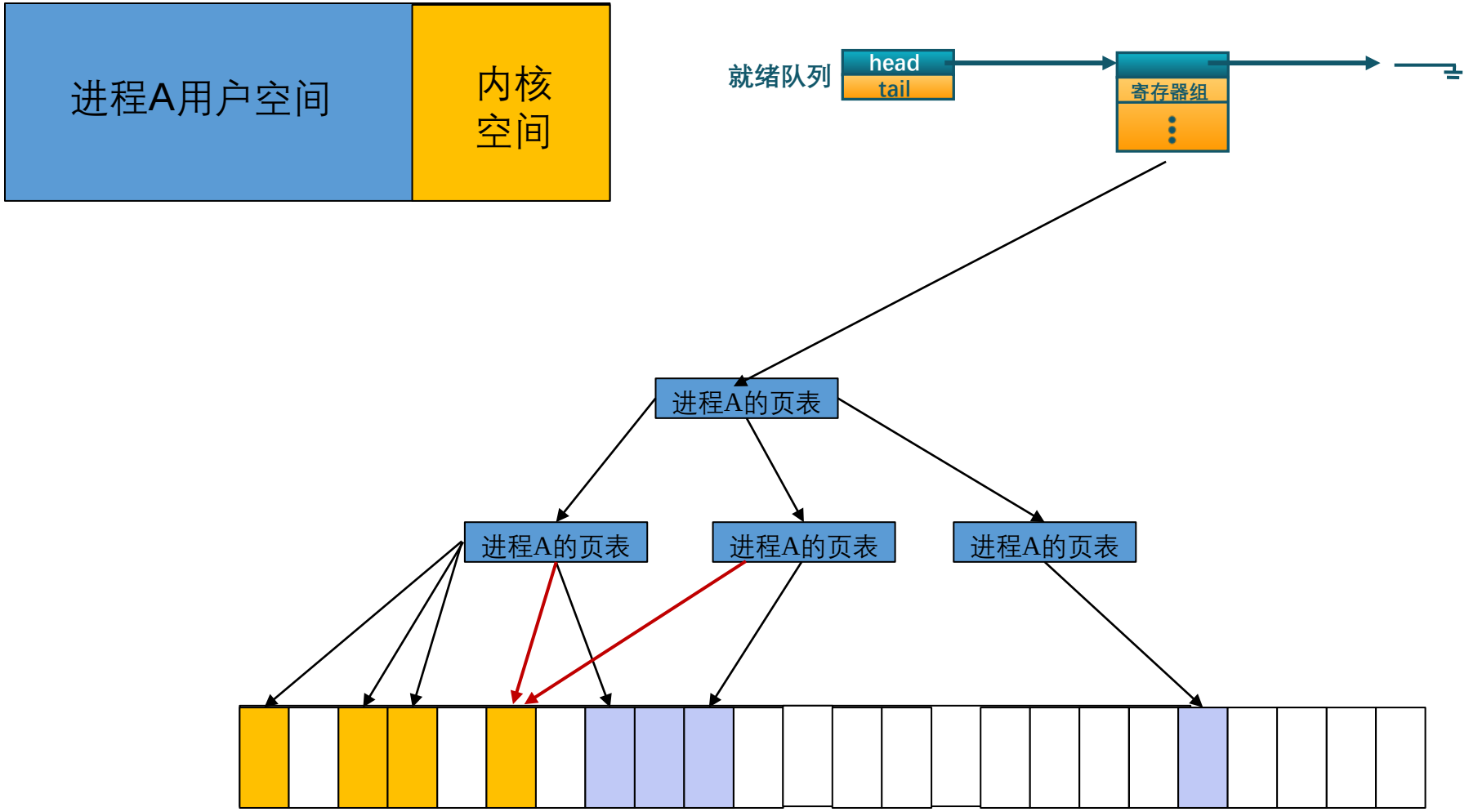
内存映射相关的系统调用

- mmap原理：mmap为什么可以省下内存



内存映射相关的系统调用

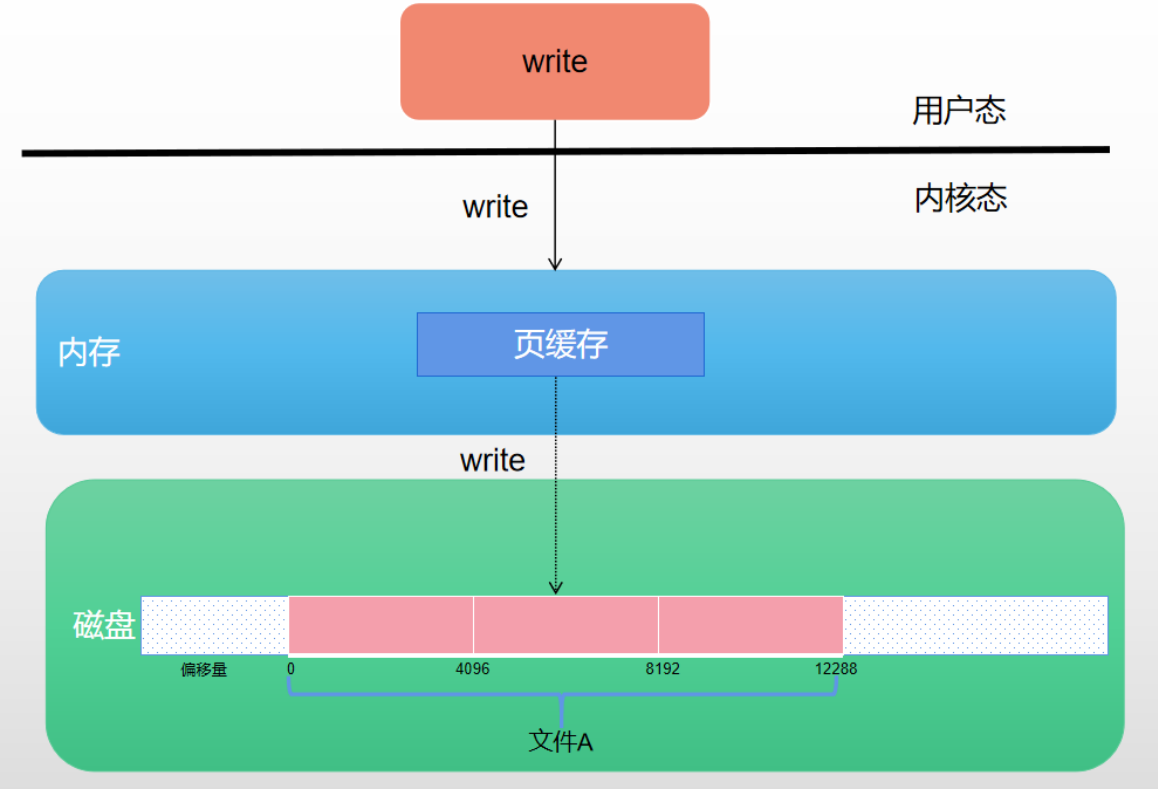
- mmap原理：mmap为什么可以省下内存



- 总结
  - 利用系统的API定制化编程，可以避免多命令（程序）配合的代价，提高程序的运行效率
  - 利用read等操作系统提供的API可以实现文件内容的读取
  - 在OS的设计中，read会附带缓存以寻求数据的复用
  - 在不存在利用的场景中，可以使用mmap减少数据的复制
  - mmap是虚拟内存提供的机制，可以借助page fault处理实现数据的按需加载，并且对用户透明

文件读写相关的系统调用

- write系统调用的函数原型：
  - ssize_t write(int fd,const void*buf,size_t count);



提问：程序员调用的写操作，会按照他们期待的顺序到达磁盘吗？

虚拟内存的管理机制

- 内存资源的观测：
  - 在Linux系统中使用free命令观测系统的内存使用情况。
    - Mem 行是内存的使用情况。
    - Swap 行是swap分区的使用情况。

```
kjr@nkux86:~$ free
               total        used        free      shared  buff/cache   available
Mem:       131451596     3822520    36955580        7112    90673496   126569372
Swap:        8388604        2328     8386276
```

- 内存的分配与回收
  - malloc/free函数用于分配和回收内存。
  - mmap/malloc分配的内存页面在内存紧缺时会被系统回收，再次访问时发生缺页事件将页面换入。
  - 使用free函数可以释放由malloc函数返回的指针指向的内存空间，释放后进程再次访问该段内存时会发生错误。

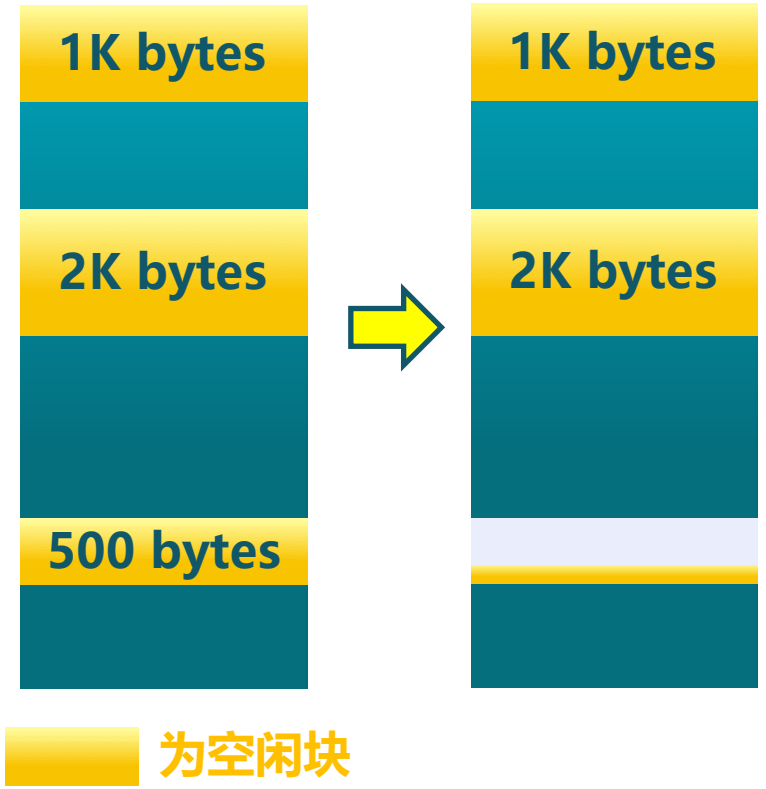
- 内存的分配与回收
  - malloc/free函数用于分配和回收内存。
  - mmap/malloc分配的内存页面在内存紧缺时会被系统回收，再次访问时发生缺页事件将页面换入。
  - 使用free函数可以释放由malloc函数返回的指针指向的内存空间，释放后进程再次访问该段内存时会发生错误。

最佳匹配

思路：

分配n字节分区时， 查找并使用不小于n的最小空闲分区

示例：
分配400字节， 使用第3个空闲块(最小)

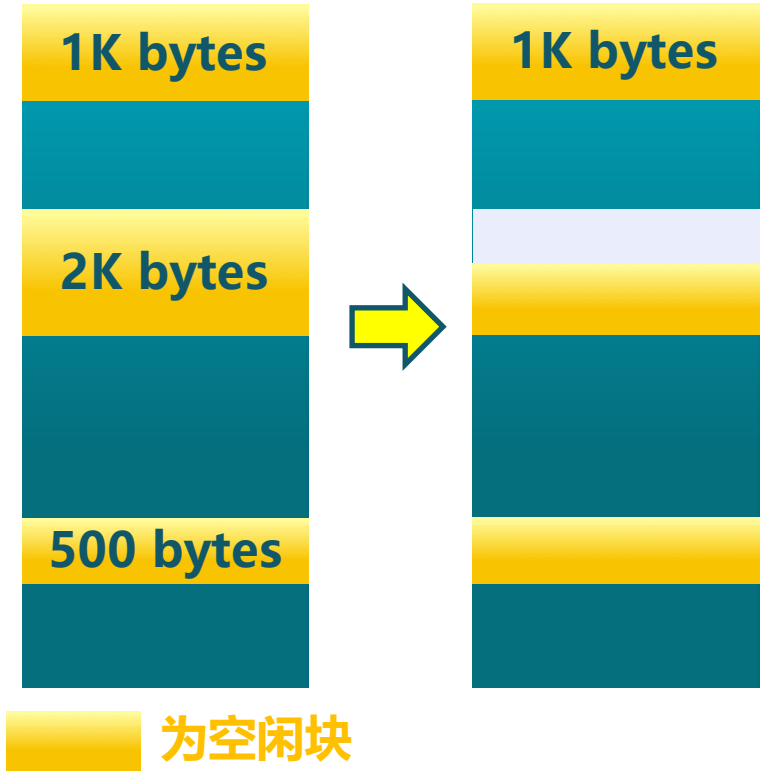


最差匹配

思路：

分配n字节，使用尺寸不小于n的最大空闲分区

示例：
分配400字节，使用第2个空闲块（最大）



扩展阅读

扩展阅读-gdb的用法

- 启用gdb调试程序有以下三种方式
  - 直接调试目标程序：gdb ./filename
  - 附加进程id：gdb attach pid
  - 调试core文件：gdb filename corename

```
kjr@nkux86:~/OSbook$ gdb ./read_file_pattern
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./read_file_pattern...
(gdb) r
Starting program: /home/kjr/OSbook/read_file_pattern
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
matched:From ilug-admin@linux.ie  Thu Aug 22 17:19:31 2002
matched:From ilug-admin@linux.ie  Thu Aug 22 17:45:53 2002
matched:From ilug-admin@linux.ie  Thu Aug 22 17:45:54 2002
matched:From ilug-admin@linux.ie  Fri Aug 23 11:07:47 2002
matched:From ilug-admin@linux.ie  Fri Aug 23 11:07:51 2002
```

扩展阅读-gdb的用法

- 添加断点

```
(gdb) b main
Breakpoint 1 at 0x555555555258: file read_file_pattern.c, line 12.
(gdb) r
Starting program: /home/kjr/OSbook/read_file_pattern
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at read_file_pattern.c:12
12      int main(){
```

- 执行step单步步入进入调试函数，执行next单步步过则跳过open函数

```
(gdb) s
13          int fd=open("new.txt",O_RDONLY);//打开文件
(gdb) s
__libc_open64 (file=0x555555556008 "new.txt", oflag=0) at ../sysdeps/unix/sysv/linux/open64.c:30
30      ../sysdeps/unix/sysv/linux/open64.c: No such file or directory.
(gdb) s
33      in ../sysdeps/unix/sysv/linux/open64.c
(gdb) s
41      in ../sysdeps/unix/sysv/linux/open64.c
(gdb) n
main () at read_file_pattern.c:14
14          if(fd==-1){
(gdb)
```

本章作业

程序设计

利用C语言编程实现对一个大文件的扫描，统计文件中特定字符模式出现的次数

1. 利用read系统调用实现对文件内容的扫描统计
2. 利用mmap系统调用实现对文件内容的扫描统计

观察并完成实验报告

1. 利用strace统计脚本方式、read方式、mmap方式的系统调用次数
2. 利用free统计三种模式的内存消耗
3. 利用eBPF追踪并统计系统的缺页数量、地址和触发的函数
4. 利用gdb实现程序的单步调试

感谢阅读