- Operating Systems
  - Four Fundamental OS Concepts
  - Abstraction
    - Threads
      - Concurrency
    - Files and I/O
      - High-level File API: Streams
      - Low-level File API: File Descriptors
      - How and Why of High-level File I/O
      - Pitfalls with OS Abstractions
    - IPC, Pipes and Sockets
      - Pipe
      - Socket
  - Synchroization
    - Producer-Consumer with a Bounded Buffer
    - Too Much Milk
    - Lock Implementation
    - Lock Implementation with Atomic Operations
    - Monitor
    - Readers/Writers
    - Construct Monitor from Semaphores
  - Scheduling
    - Multi-Core Scheduling
    - Real-time Scheduling
    - Ensuring Progress
    - Case Study
    - Choosing the Right Scheduler
    - Deadlock
    - Scheduling in Modern Computer Systems
      - ZygOS
      - Tiresias
      - DRF
      - FairRide
  - Memory
    - Address Translation and Virtual Memory
      - Segmentation
      - Paging
    - Caching
    - Demand Paging
    - Replacement Policy
    - Memory Management in Modern Computer Systems
      - FaRM: Fast Remote Memory
      - vLLM
      - InfiniSwap
      - AIFM
      - PipeSwitch
      - TGS
  - I/O
    - Hard Disk Devices (HDDs)
    - Solid State Drives (SSDs)
    - I/O Performance
      - Queuing Theory
  - File Systems
    - File System Design
    - Case Study: File Allocation Table (FAT)
    - Case Study: Unix File System
    - Case Study: New Technology File System (NTFS)
    - Buffer Cache
    - Durable File Systems
    - Reliable File Systems

Operating Systems

操作系统：为应用程序提供硬件资源的 special layer

- 为复杂硬件设备提供一层方便的抽象
- 对共享资源的访问提供保护
- Security and authentication
- 逻辑实体的沟通

操作系统是裁判、幻术师、胶水。

Four Fundamental OS Concepts

Four Fundamental OS Concepts:

- Thread
- Address space (with translation)
- Process
- Dual mode operation / Protection

从用户态切换到内核态：

- 系统调用
- 外部中断
- 内部中断

Abstraction

Threads

Motivation: Multiple Thing At Once (MTAO)

- Multiprocessing: 多 CPU
- Multiprogramming: 多进程
- Multithreading: 多线程

Concurrency（并发）不是 parallelism（并行）, 每个任务并非 simultaneously 进行。

线程有三种状态：RUNNING, READY, BLOCKED (**正在等待 I/O**)

程序开始后，可以通过系统调用创建线程。

```c
#include <pthread.h>

// 创建线程执行 start_routine(arg)
// 返回：若成功则为 0，若出错则非零
int pthread_create(pthread_t* tid, const pthread_attr_t* attr, void* (*start_routine)(void*), void* arg);

// 返回：调用线程的 TID
pthread_t pthread_self(void);

// 无返回值
void pthread_exit(void* thread_return);

// 返回：若成功则为 0，若出错则非零
int pthread_join(pthread_t tid, void** thread_return);
```

当顶层的线程例程返回时，线程会隐式地调用 `pthread_exit` 而终止。

通过调用 `pthread_exit` ，线程可以显式地终止。如果主线程调用 `pthread_exit` ，它会等待所有对等线程终止，然后再终止主线程和整个进程。

可选参数 `thread_return` 指定了线程例程的返回值。

`pthread_join` 会阻塞，直到线程 `tid` 终止。线程例程的返回值被存放在 `*thread_return` 中。此后，线程 `tid` 的资源被回收。

线程状态：

- 所有同地址空间的线程共享的
  - 全局变量、堆
  - I/O 状态（文件描述符、网络连接）
- 线程的私有状态
  - 保存在 Thread Control Block (TCB) 中
  - CPU 寄存器（包括 PC）
  - 执行栈（execution stack）
    - 包含局部变量、参数、返回地址

Concurrency

Mutual exclusion: 保证同一时间点只有一个线程访问共享资源。

Critical sectoin: Mutually exclusive 的代码片段。

```c
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t* mutex, const pthread_mutexattr_t* attr);
int pthread_mutex_lock(pthread_mutex_t* mutex);
int pthread_mutex_unlock(pthread_mutex_t* mutex);

int common = 162;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* thread_fn(void* arg) {
    pthread_mutex_lock(&mutex);
    common++;
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

信号量： `P` 和 `V`

- 可以用于实现 mutual exclusion
- 也可以用于 threads 之间的 signaling（ `join` 为 `P` ， `exit` 为 `V` ）

多进程编程： `fork` 和 `exec`

- 为什么线程 API 只有 `pthread_create` ，进程却有 `fork` 和 `exec` ？
  - 可以只 `fork` 不 `exec` ，从而将父子进程的代码都放在同一个可执行文件
  - 便于控制子进程的状态（在 `exec` 前设定子进程状态）

线程和进程的选择：

- 线程性能更强：上下文切换开销小、内存占用小、创建销毁/开销小
- 进程保护性强：互相隔离且独立的地址空间
- 线程间共享数据和通信更方便

Files and I/O

Unix/POSIX: 一切皆“文件”

- 磁盘上的文件
- 设备 (terminals, printers, etc.)
- 网络 sockets
- 本地的进程间通信 (pipes)

文件系统的抽象:

文件:

- Named collection of data
- 字节序列
- Metadata

目录:

- 包含文件和目录
- 路径唯一确定一个文件或目录（整个文件系统内可以有重名文件，只要它们的路径不一样）

每个进程有一个 current working directory，可以通过 `chdir` 系统调用改变。相对路径与 CWD 有关。

High-level File API: Streams

```c
#include <stdio.h>

// 流：FILE* 类型
FILE* fopen(const char* path, const char* mode);
// mode: r, w, a, r+, w+, a+; b for binary mode
int fclose(FILE* fp);

int fseek(FILE* fp, long offset, int whence);
// whence: SEEK_SET, SEEK_CUR, SEEK_END
long ftell(FILE* fp);
void rewind(FILE* fp);    // fseek(fp, 0, SEEK_SET)

// C 程序执行时，三个标准流是隐式打开的。它们可以重定向
FILE* stdin;  // 0
FILE* stdout; // 1
FILE* stderr; // 2

// 字符
int fputc(int c, FILE* fp);
int fputs(const char* s, FILE* fp);
int fgetc(FILE* fp);
char* fgets(char* s, int size, FILE* fp);
// 块
int fread(void* ptr, size_t size, size_t nmemb, FILE* fp);    // 读取 nmemb 个 size 大小的块到 ptr
int fwrite(const void* ptr, size_t size, size_t nmemb, FILE* fp);
// 格式化
int fprintf(FILE* fp, const char* format, ...);
int fscanf(FILE* fp, const char* format, ...);

// 记得错误处理
FILE* input = fopen("input.txt", "r");
if (input == NULL) {
    // Prints our string and error msg.
    perror(“Failed to open input file”);
}
```

流：字节序列及当前位置

Low-level File API: File Descriptors

Unix I/O 设计思想：

- 一切皆文件
- Open before use: 访问控制
- 面向字节
- kernel buffered reads and writes
  - 统一化流式和块式设备
  - 提高效率
- 显式 close

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>

// 系统调用
// open 系统调用返回 open-file descriptor，或 < 0 的错误码
int open(const char* path, int flags, mode_t mode);
int creat(const char* path, mode_t mode);
int close(int fd);

STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO    // macros of values 0, 1, 2
int fileno(FILE* fp);    // 从 FILE* 得到 file descriptor
FILE* fdopen(int fd, const char* mode);    // 从 file descriptor 到 FILE*

// 系统调用
ssize_t read(int fd, void* buf, size_t count);    // 返回读取的字节数，或 < 0 的错误码
ssize_t write(int fd, const void* buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);    // 返回新的文件偏移量，或 < 0 的错误码
// lseek 和 high-level API 的文件位置无关
```

Kerner buffering: read/write 系统调用昂贵，kernel 会缓存数据，减少系统调用次数。

返回文件描述符，而非文件描述条目的指针，是为了隔离用户和内核、提供抽象、增强安全性。

How and Why of High-level File I/O

```
// High level
size_t fread(void* ptr, size_t size, size_t nmemb, FILE* fp) {
    // Do some work like a normal fn

    // asm code ... syscall # into %eax
    // put args into regs
    // special trap instruction

    // Kernel: get args from regs; dispatch to sys fn; read data; return value

    // get return value from regs
    // Do some more work like a normal fn
}

// Low level
ssize_t read(int fd, void* buf, size_t count) {
    // asm code ... syscall # into %eax
    // put args into regs
    // special trap instruction

    // Kernel: get args from regs; dispatch to sys fn; read data; return value

    // get return value from regs
}
```

`FILE*` 包括：

- 文件描述符
- 缓冲区
- 锁
- ...

当调用 `fwrite` 时，数据先被写入 `FILE` 的缓冲区，缓冲区满或遇到特定字符时会 flush（即将数据写入文件描述符）。

```
char x = 'c';
FILE* f1 = fopen("file.txt", "wb");
fwrite("b", sizeof(char), 1, f1);
// fflush(f1);
FILE* f2 = fopen("file.txt", "rb");
fread(&x, sizeof(char), 1, f2);
// 没有 fflush 时，f1 的缓冲区可能还没写入文件，因此 f2 读取立即遇到 EOF，x 保持 'c'
```

使用 high-level API 时需要注意缓冲区 flush 的问题，但 low-level API 不需要（`write` 系统调用会直接写入文件描述符）。

Buffer in userspace:

- 系统调用开销很大：byte by byte 的读写，吞吐量仅为 10 MB/s，但 `fgetc` 可以匹配 SSD 的速度
- 系统调用功能简单
    - 没有“读直到换行”

Pitfalls with OS Abstractions

多线程进程的 `fork`：子进程**只继承调用 `fork` 时的线程，其他线程消失**。如果其他线程有锁或正在写入，可能导致麻烦。

在子进程调用 `exec` 是安全的，因为 `exec` 会覆写整个地址空间。

不要混用 high-level 和 low-level API，否则可能导致缓冲区不一致。

```
char x[10];
char y[10];
FILE* f = fopen("foo.txt", "rb");
int fd = fileno(f);
fread(x, 10, 1, f); // read 10 bytes from f
read(fd, y, 10); // assumes that this returns data starting at offset 10

// 由于 fread 会读入一大块数据（可能是整个文件），y 读取的数据不会是文件的第 10 个到第 19 个字节
```

IPC, Pipes and Sockets

Pipe

进程间通信：如果用文件，性能会太差。

Unix Pipe: **存储在内存中的固定大小队列**

- 对单向队列的抽象
- 本地 IPC
- 文件描述符通过继承获得

```
#include <unistd.h>

int pipe(int pipefd[2]);
// 分配两个文件描述符，用于 IPC
// pipefd[0]: read end
// pipefd[1]: write end
// 如果 pipe 满了，write 会阻塞，直到有空间
// 如果 pipe 空了，read 会阻塞，直到有数据
```

```c
#include <unistd.h>

int pipefd[2];
if (pipe(pipefd) == -1) {
    fprintf(stderr, "Pipe failed\n");
    return EXIT_FAILURE;
}

// 自己给自己发消息
ssize_t writelen = write(pipefd[1], msg, strlen(msg) + 1);
ssize_t readlen = read(pipefd[0], buf, BUF_SIZE);
// 省略错误处理

// 父子进程通信：父进程写，子进程读
pid_t pid = fork();
if (pid < 0) {
    fprintf(stderr, "Fork failed\n");
    return EXIT_FAILURE;
} else if (pid != 0) {
    // parent
    ssize_t writelen = write(pipefd[1], msg, strlen(msg) + 1);
    close(pipefd[0]);
} else {
    // child
    ssize_t readlen = read(pipefd[0], buf, BUF_SIZE);
    close(pipefd[1]);
}
```

当所有写端文件描述符关闭时，读端会检测到 EOF。

当所有读端文件描述符关闭时，写端调用 `write` 会生成 SIGPIPE 信号（如果忽略此信号， `write` 会 `EPIPE` 报错）。

Pipe 是单向的，父子进程必须关闭不用的文件描述符（**否则读取进程检测不到 EOF**），双向通信需要创建两个 pipe。

Socket

沟通需要协议：

- Syntax：信息的组织结构
- Semantics：信息的意义

客户端-服务器通信：

- 服务器总是开机，处理来自多个客户端的请求
- 客户端有时开机，有时关机

（TCP）网络连接：**两个（不同机器的）进程间的双向字节流**。

- 包括一个从 Alice 到 Bob 的队列和一个从 Bob 到 Alice 的队列。
- `write` 向输出队列写入数据， `read` 从输入队列读取数据。

Socket：对于网络连接的一个 endpoint 的抽象，就像一个文件描述符。

- 对双向队列的抽象
- 远程 IPC
- 文件描述符通过 `socket` 、 `bind` 、 `listen` 、 `connect` 、 `accept` 获得

```
// Echo client-server
void client(int sockfd) {
    int n;
    char sndbuf[MAXIN];
    char rcvbuf[MAXOUT];
    while (true) {
        fgets(sndbuf, MAXIN, stdin);
        write(sockfd, sndbuf, strlen(sndbuf));
        memset(rcvbuf, 0, MAXOUT);
        n = read(sockfd, rcvbuf, MAXOUT);
        printf("Echo: %s", rcvbuf);
    }
}

void server(int sockfd) {
    int n;
    char buf[MAXLINE];
    while (true) {
        n = read(sockfd, buf, MAXLINE);
        if (n <= 0) return;
        write(sockfd, buf, n);
    }
}
```

以上代码预设了：

- Reliable: 没有数据丢失
- In-order: 数据按顺序到达

`read` 预设已经有数据可读，当没有数据可读时会阻塞。

Server socket：

- 有文件描述符
- 不能读或写
- 两种操作
  - `listen`：开始允许客户端连接
  - `accept`：接受连接请求，返回一个新的 socket（可以读写）

一个连接由一个 5-tuple 表示：(源 IP, 源端口, 目的 IP, 目的端口, 协议)

客户端协议：

```
char* host_name;
char* port_name;

struct addrinfo* server = lookup(host_name, port_name);
int sockfd = socket(server->ai_family, server->ai_socktype, server->ai_protocol);

connect(sockfd, server->ai_addr, server->ai_addrlen);

run_client(sockfd);

close(sockfd);
```

服务器协议 v1：

```
char* port_name;
struct addrinfo* server = setup_addrinfo(port_name);

int server_socket = socket(server->ai_family, server->ai_socktype, server->ai_protocol);

bind(server_socket, server->ai_addr, server->ai_addrlen);
listen(server_socket, MAX_QUEUE);

while (true) {
  int conn_socket = accept(server_socket, NULL, NULL);
  serve_client(conn_socket);
  close(conn_socket);
}
close(server_socket);
```

服务器协议 v2：为了保护服务器，为每个连接创建一个子进程。

```
// ...
while (true) {
  int conn_socket = accept(server_socket, NULL, NULL);
  pid_t pid = fork();
  if (pid == 0) {
    // child process
    close(server_socket);
    serve_client(conn_socket);
    close(conn_socket);
    exit(0);
  } else if (pid > 0) {
    // parent process
    close(conn_socket);
    // wait(NULL);    // 不用等待子进程
  } else {
    // fork failed
    perror("fork failed");
  }
}
```

Synchroization

Process Control Block: 内核将每个进程视为一个进程控制块（PCB），包含：

- 进程状态：RUNNING, READY, BLOCKED
- 寄存器
- PID、用户、优先级、...
- 执行时间
- 内存空间、内存翻译

内核的 scheduler 会根据进程状态和优先级决定下一个运行的进程（在 PCB 之间分配 CPU 资源），如果 ready queue 为空，则会选择 idle 进程。

Context switch: 由 interrupt 或系统调用触发，包括将状态保存到 PCB、...、从 PCB 恢复状态。

PCB 在各种队列之间移动。当它们没有运行时，它们便在 scheduler 的 ready 队列中。当它们请求 I/O 操作时，它们便在 I/O 设备的队列中。

一个进程的线程之间共享地址空间（堆、全局变量、代码段），每个线程有独立的 TCB。

The dispatch loop:

```
Loop {
    RunThread();
    ChooseNextThread();
    SaveStateOfCPU(curTCB);
    LoadStateOfCPU(nextTCB);
}
```

`RunThread`：

- 加载线程的状态（寄存器、PC、栈指针）
- 加载环境（虚拟内存空间）
- 跳转到线程的入口点。

Dispatcher 拿回控制权：

- 内部事件：线程主动放弃 CPU
  - I/O blocking（ `read` 系统调用 -> trap 到 OS -> 内核发射读取命令 -> switch）
  - 等待其他线程的“信号”（调用 `wait` ）
  - 调用 `yield`
- 外部事件：中断使得 OS 拿回控制权
  - Interrupts
    - Timer：保证在没有内部事件时，每个线程也都有机会运行

`yield` 会 trap 到 OS，调度器会选择下一个线程。

进程对比线程：

- Switch overhead：线程小，进程大
- Protection：同进程弱（共享地址空间）
- Sharing overhead：同进程小，不同进程大
- Parallelism：进程不能多核并行，线程可以

TCB 和线程栈的初始化：

- 初始化 TCB
  - 栈指针：指向栈顶
  - PC 返回地址：OS 汇编 routine `ThreadRoot`
  - 寄存器 `a0` 、 `a1` 初始化为 `fcnPtr` 和 `fcnArgPtr`

线程的开始：

- `switch` 会选择新线程的 TCB，返回到其 `ThreadRoot` 的起始位置，开始运行新线程
- `ThreadRoot` 会：
  - 做一些初始化工作，如记录线程起始时间
  - 切换到用户模式
  - 调用 `fcnPtr(fcnArgPtr)`
  - 清理线程资源，唤醒睡眠线程

Shinjuku：

- 多核、小任务，进程切换的 OS 开销太大。
- 此问题的常见解决方案：
  - OS bypass
  - Polling (无中断，避免中断开销)
  - Run-to-completion (无调度，避免上下文切换)
  - d-FSFS (distributed queues + First Come First Serve scheduling)：网卡通过 Receive Side Scaling 把请求随机发给多个 CPU 中的某一个，每个 CPU 有自己的队列，队列中的任务按 FCFS 运行
- d-FCFS 的问题：Queue imbalance，某几个有难，其他的围观（non work-conserving）
- 解决方案：中心化的队列，c-FCFS
- 问题：如果所有核都在忙长请求，新的短请求会被这些长请求阻塞，造成 long tail latency。
- Shinjuku：使用类似 Linux 的抢占式调度
  - 如果采用 1ms 的时间片，对于给定的 Bimodal 请求来说永远不会触发调度，表现和 c-FCFS 完全一样
  - 如果采用 5us 的时间片，表现接近最优
  - Shinjuku 的贡献：实现了 us 级别的上下文切换
- 方法：
  - Dedicated core for scheduling and queue management
  - Leverage hardware support for virtualization for fast preemption
  - Very fast context switching in user space
  - Match scheduling policy to task distribution and target latency

相比于 Event Driven 编程，多线程可以更简单地实现 **I/O 和计算的 overlap**（前者需要 deconstruct code，代码复杂度高）。

原子操作：大部分机器中，从内存读取和赋值是原子的。锁可以用于保证原子性。

Producer-Consumer with a Bounded Buffer

问题：生产者和消费者共享一个有限大小的缓冲区。生产者往缓冲区中放入数据，消费者从缓冲区中取出数据。

```
Producer(item) {
    acquire(&buf_lock);
    while (buffer_full) {}
    enqueue(item);
    release(&buf_lock);
}

Consumer() {
    acquire(&buf_lock);
    while (buffer_empty) {}
    item = dequeue();
    release(&buf_lock);
}
```

问题：若缓冲区满， `Producer` 会陷入 `while` 死循环，而 `Consumer` 会在 `acquire` 时被阻塞，导致死锁。缓冲区空时也是如此。

简单粗暴的解决方案：

```
Producer(item) {
    acquire(&buf_lock);
    while (buffer_full) {
        release(&buf_lock);
        acquire(&buf_lock);
    }
    enqueue(item);
    release(&buf_lock);
}

Consumer() {
    acquire(&buf_lock);
    while (buffer_empty) {
        release(&buf_lock);
        acquire(&buf_lock);
    }
    item = dequeue();
    release(&buf_lock);
}
```

问题：性能低下。若缓冲区满，需要恰好在 `while` 循环中 `release` 和 `acquire` 之间切换线程才能跳出死锁。缓冲区空时也是如此。

信号量（semaphore）：支持 `P` 和 `V` 操作的非负整数。

问题的正确性约束：

1. 若空，则消费者等待
2. 若满，则生产者等待
3. 生产者和消费者不能同时操作缓冲区（mutual exclusion）

通常，**每个约束对应一个信号量，信号量的值代表资源的可用数量**。

```
Semaphore mutex = 1;     // 互斥锁
Semaphore empty = N;     // 生产者消耗一个空槽，释放一个满槽
Semaphore full = 0;      // 消费者消耗一个满槽，释放一个空槽

Producer(item) {
    P(&empty);
    P(&mutex);
    enqueue(item);
    V(&mutex);
    V(&full);
}

Consumer(&) {
    P(&full);
    P(&mutex);
    item = dequeue();
    V(&mutex);
    V(&empty);
}
```

`P(&full)` 必须在 `P(&mutex)` 之前，否则可能导致死锁。

`V(&empty)` 则不必在 `V(&mutex)` 之后（不过先 `V(&mutex)` 性能稍高）。

即使我们有多个生产者和多个消费者，以上代码也能正确工作。

Too Much Milk

问题：多个舍友共享冰箱，冰箱空时每个人可以买牛奶放入。如果 Alice 正在买牛奶，Bob 查看冰箱为空，也去买牛奶，会导致牛奶买多。

可以在查看前加锁，放入牛奶后解锁。问题：如果 Bob 只是想拿橙汁，Alice 正在买牛奶，Bob 会被阻塞。

现在假设我们不会实现锁：

Solution 1

```
if (noMilk) {
    if (noNote) {
        leave note;
        buy milk;
        remove note;
    }
}
```

问题：假如在 `if (noNote)` 和 `leave note` 之间切换，仍然会导致牛奶买多。

如果我们将 note 范围扩大：

```
leave note;
if (noMilk) {
    if (noNote) {
        buy milk;
    }
}
remove note;
```

没人会买牛奶：会被自己的 note 阻塞。

Solution 2

```
// Thread A
leave note A;
if (noNote B) {
    if (noMilk) {
        buy milk;
    }
}
remove note A;

// Thread B
leave note B;
if (noNote A) {
    if (noMilk) {
        buy milk;
    }
}
remove note B;
```

问题：**活锁**

- 线程 A 留下 note A 后切换
- 线程 B 留下 note B 后切换
- 则两个线程都不会买牛奶

Solution 3

```
// Thread A
leave note A;
while (Note B) {}    // spin-wait
if (noMilk) {
    buy milk;
}
remove note A;

// Thread B
leave note B;
if (noNote A) {
    if (noMilk) {
        buy milk;
    }
}
remove note B;
```

线程 B 的实现没有变，线程 A 的逻辑从“如果有 note B 就不买牛奶”变成了“如果有 note B 就一直等”，因此**最终线程 A 一定会执行 `if` 语句**，解决了活锁的问题。

可以 work，但复杂，每个线程的代码都不同，难以支持多个线程。且包括 busy waiting。

需要硬件提供比 atomic load/store 更强的原子操作：通过锁。

Solution 4

```
acquire(lock);
if (noMilk) {
    buy milk;
}
release(lock);
```

Lock Implementation

Naive 锁实现：加锁禁用中断，解锁开启中断。

- dispatcher 通过内部事件和外部事件两种方式拿回控制权
- 对单核系统来说，只要不引发内部事件，并通过禁用中断避免外部事件，就可以防止切换
- 问题：
    - 不能为用户所使用：用户加锁后进入死循环，系统死机

- 多核下无效：多核情况下，一个核禁用中断，另一个核无法禁用中断
- 期间不能响应外部中断，破坏系统实时性
- 不支持嵌套锁

更好的锁实现：

```
int value = FREE;

acquire() {
    disable interrupts;
    if (value == BUSY) {
        // 如果在此处 enable interrupts
        // 则若在此处切换到 release 线程，则此时 wait queue 没有当前线程
        // 如果 ready queue 为空
        // 就会浪费一次调度
        put thread on wait queue
        // 如果在此处 enable interrupts
        // 则若在此处切换到 release 线程，当前线程会平白 sleep 一次
        go to sleep
        // 我们希望在此处 enable interrupts
        // 但这里已经进入 sleep 了
        // 我们只能令下一个切换到的线程 enable interrupts
    } else
        value = BUSY;
    enable interrupts;
}

release() {
    disable interrupts;
    if (wait queue is not empty) {
        remove thread from queue
        place thread on ready queue
    } else
        value = FREE;
    enable interrupts;
}
```

以上代码中包含检查并修改多个共享变量，因此需要禁用中断。（否则如果 `value` 值空闲，则两个线程会同时认为它们持有锁）

关键区域比 naive 实现短。

在 `sleep` 后重新开启中断：

- 每个线程在睡眠前禁用中断
- 每个线程在睡眠返回时首先开启中断

Lock Implementation with Atomic Operations

禁用中断锁实现的问题：

- 不适用于用户模式
- 不适用于多核：多核禁用中断需要 message passing，耗时

原子的 read-modify-write 指令：

```
// 多数架构都支持
test&set(&addr) {
    res = *addr;
    *addr = 1;
    return res;
}

// x86
swap(&addr, reg) {
    res = *addr;
    *addr = reg;
    return res;
}

// 68000
// 如果 *addr 的值和期望的旧值相等，就将其更改为新值并返回 success
// 否则返回 failure
compare&swap(&addr, reg1, reg2) {
    if (reg1 == *addr) {
        *addr = reg2;
        return success;
    } else {
        return failure;
    }
}
```

naive 的利用 `test&set` 的锁实现：

```
int val = 0;    // 0: free, 1: busy

acquire() {
    while (test&set(val)) {}
    // 如果 val == 0, 则 test&set 返回 0, val 被设为 1, 不会进入循环
    // 如果 val == 1, 则 test&set 返回 1, val 被设为 1, busy waiting
}

release() {
    val = 0;
}
```

问题：

- **busy waiting**。对多核系统来说，每一次 `test&set` 都是一次写，值会在 cache 间 ping-pong，性能很差。
- Priority inversion：低优先级线程持有锁，高优先级线程无法获得锁（不会被抢占）。

优点：

- 不用禁用中断
- 用户模式可用
- 多核可用

改进：

```
int guard = 0;
int val = FREE;

acquire() {
    while (test&set(guard)) {}
    if (val == BUSY) {
        put thread on wait queue
        go to sleep & guard = 0
    } else {
        val = BUSY;
        guard = 0;
    }
}

release() {
    while (test&set(guard)) {}
    if (wait queue is not empty) {
        remove thread from queue
        place thread on ready queue
    } else {
        val = FREE;
    }
    guard = 0;
}

// guard 就是之前的 naive 锁，我们用小锁保护了大锁
```

优点：

- 小锁保护大锁，显著减少了临界区的范围（仅包括检查 `val` 和操作等待队列）
- 主锁会挂起并让出 CPU

相比于之前禁用中断的锁实现，我们把：

- 禁用中断替换为 `while (test&set(guard)) {}`
- `enable interrupts` 替换为 `guard = 0`

Monitor

信号量有双重用途：互斥和调度约束。调转 `P` 的顺序可能导致死锁，这很不显然。

更干净的方法：用 Lock 实现 mutual exclusion，用 Condition Variable 实现 scheduling constraints。

Monitor：一个 Lock 以及零个或多个 Condition Variables。

Conditional Variable：临界区内等待某个特定条件的线程的队列。

Operations: 必须在获得锁的情况下调用

- `wait(&lock)`：原子地释放锁并 sleep。在被唤醒后返回前，重新获得锁。
- `signal()`：唤醒一个等待的线程。
- `broadcast()`：唤醒所有等待的线程。

信号量不能在临界区内 sleep，而 Condition Variable 可以。

无限 buffer 的生产者消费者问题：

```
lock buf_lock;
cond buf_CV;
queue queue;

Producer(item) {
    acquire(&buf_lock);
    enqueue(item);
    signal(&buf_CV);
    release(&buf_lock);
}

Consumer() {
    acquire(&buf_lock);
    // 在 Hoare scheduling 中，此处应为 if
    // 因为 signal 会将锁和 CPU 交给等待的线程，等待的线程立即开始运行
    // 出了临界区/又 wait 之后，锁和 CPU 还给 signal 的线程
    // 这不容易实现，而且性能一般
    while (queue is empty) {
        wait(&buf_CV, &buf_lock);
    }
    // 大部分 OS 采用 Mesa scheduling，因此此处需要用 while
    // signal 只会将等待的线程加入 ready queue，而不是立即执行上下文切换
    // 因此等真的切换到等待的线程时，可能条件已经不满足：需要重新检查
    // 例如：消费者 A 发现队列为空，开始等待。生产者 B 生产了一个 item，唤醒了 A
    // 但由于 A 不是立即被调度的，如果 A 被调度前队列又变成空的了，则 A 就必须再次等待
    item = dequeue();
    release(&buf_lock);
    return item;
}
```

有限 buffer 的生产者消费者问题：

```
lock buf_lock;
cond consumer_CV, producer_CV;

Producer(item) {
    acquire(&buf_lock);
    while (queue is full) {
        wait(&consumer_CV, &buf_lock);    // 唤醒一个消费者
    }
    enqueue(item);
    signal(&producer_CV);
    release(&buf_lock);
}

Consumer() {
    acquire(&buf_lock);
    while (queue is empty) {
        wait(&producer_CV, &buf_lock);    // 唤醒一个生产者
    }
    item = dequeue();
    signal(&consumer_CV);
    release(&buf_lock);
    return item;
}
```

Readers/Writers

正确性约束：

- 多个读者可以同时读
- 一个写者写时，不能有读者读，也不能有其他写者写
- 每次只能有一个线程操作共享数据
- （写者优先）

基本实现：

状态变量（被 lock 保护）：

- AR：活跃读者数量
- WR：等待读者数量
- AW：活跃写者数量
- WW：等待写者数量
- okToRead
- okToWrite

```
Reader() {
    // 等待写者完成
    acquire(&lock);
    while (AW + WW > 0) {          // 有写者（非活跃写者也要等），不需要等读者，因为多个读者可以同时读
        WR++;    // 我们开始等待
        wait(&okToRead, &lock);
        WR--;    // 我们等完了
    }
    AR++;        // 我们活跃了!
    release(&lock);    // 这里要释放锁，从而其他读者可以同时读

    read data

    acquire(&lock);
    AR--;       // 我们不活跃了
    if (AR == 0 && WW > 0)   // 注意：删去这个判断不影响正确性 只是会变慢
        signal(&okToWrite);    // 改成 broadcast 不影响正确性，但会变慢
    release(&lock);
}

Writer() {
    // 等待读者完成
    acquire(&lock);
    while (AR > 0 || AW > 0) {    // 有活跃读者或写者
        WW++;    // 我们开始等待
        wait(&okToWrite, &lock);
        WW--;    // 我们等完了
    }
    AW++;        // 我们活跃了!
    release(&lock);

    write data

    acquire(&lock);
    AW--;       // 我们不活跃了
    if (WW > 0) {
        signal(&okToWrite);
    } else if (WR > 0) {
        broadcast(&okToRead);    // 唤醒所有读者（多个读者可以同时读）
        // 这里不能用 signal
        // 因为读者只会唤醒写者，不会唤醒其他读者
        // 如果有一个写者和一百个读者，那换成 signal，就会导致 99 个读者饿死
    }
    release(&lock);
}
```

这是写者优先的实现，读者可能饥饿（一直有新写者），写者不会饥饿。

> 考虑读写者序列 R1, R2, W1, R3，假设读写操作用时很长，则我们的实现会出现 R1, R2 同时读，W1 在等 R1, R2 完成，R3 在等 W1 完成（而不是 R1, R2, R3 都在同时读）。

可以只用一个条件变量 okContinue，此时必须用 broadcast，而不是 signal。

```
Reader() {
    acquire(&lock);
    while (AW + WW > 0) {
        WR++;
        wait(&okContinue, &lock);
        WR--;
    }
    AR++;
    release(&lock);

    read data

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
        broadcast(&okContinue);   // 唯一区别是 signal 变为 broadcast
    release(&lock);
}

Writer() {
    acquire(&lock);
    while (AR > 0 || AW > 0) {
        WW++;
        wait(&okContinue, &lock);
        WW--;
    }
    AW++;
    release(&lock);

    write data

    acquire(&lock);
    AW--;
    if (WW > 0 || WR > 0)
        broadcast(&okContinue);
    release(&lock);
}
```

假如直接把 `okToRead` 和 `okToWrite` 替换为 `okContinue`，则会导致死锁：

- 读写者序列 R1, W1, R2
- R1 正在读，W1 和 R2 等待
- R1 读完，唤醒了 R2
- 则 R2 在等 W1，W1 又在等 R2 的 signal，死锁

在完整的实现中，正确性可以保证：

- 我们有 `lock`，所以大家还是“一个一个出来”的
- 如果有在等待的写者，因为这是写者优先实现， `broadcast` 后某一个写者将变成活跃状态，且只有此写者活跃
- 否则 `broadcast` 后一个读者变成活跃状态，随即所有读者都变为活跃状态

读者优先？FIFO（且相邻读者并发读）？用锁的实现？信号量？

Construct Monitor from Semaphores

```
// naive
wait(Lock* lock, Sema* sem) {
    release(lock);
    P(sem);
    acquire(lock);
}

signal(Lock* lock, Sema* sem) {
    V(sem);
}
```

问题：条件变量没有历史，信号量有历史。

- 先 `signal`，再 `wait`，则 `wait` 应当阻塞
- 先 `V`，后 `P`，则 `P` 不应当阻塞
- 条件变量和信号量语义不同

如果修改 `signal` 的实现：

```
signal(Lock* lock, Sema* sem) {
    if semaphore queue is not empty
        V(sem);
}
```

不再有上述问题，但有竞态条件：在 `wait` 内 `release` 之后切换至 `signal`，则这个 `signal` 会被丢弃，而不是唤醒 `wait` 的线程（尽管它本该唤醒），导致 `wait` 阻塞。

此外，检查信号量队列是否为空本身是非法操作。

(OSDI 06) The Chubby lock service for loosely-coupled distributed systems

Scheduling

任务：

- 一个系统被不同 user 使用，每个 user 有若干 program，每个 program 有若干 thread。
- 如何调度？
  - 以某些特定指标为目标，优化 CPU 时间的分配
- 如何保证公平性？
  - 在 user 层面公平还是在 program 层面公平？
  - 我开 1 个程序，你开 100 个程序，多数系统中后者 CPU 时间更多

执行模型：程序在 CPU burst 和 I/O burst 之间不断切换。调度器需要将 CPU 时间分配给即将 CPU burst 的线程，从而最大化 CPU 利用率。

调度器的目标：

- 最小化**完成时间（completion time）**
  - 定义：**任务下达到结束的延迟**
  - 快速响应
  - **等待时间（waiting time）**：任务**在 ready queue 中的时间**
- 最大化**吞吐量（throughput）**
  - 减少开销（如上下文切换）
  - 高效利用资源（CPU、硬盘、内存）
- 公平
- （对于实时系统）可预测性

First-Come, First-Served (FCFS) Scheduling：

- 每个进程按照到达顺序排队
- 优点：简单。缓存友好。
- 问题：Head-of-line blocking，先到的长任务会阻塞后到的短任务。对短任务的响应时间经常很差。
- 性能与任务到达顺序有关。短任务先到的话，响应时间表现更好。

Round Robin (RR) Scheduling：

- 每个进程分配一个时间片，时间片用完后被**抢占**，放入 ready queue 的末尾
- 时间片（time quantum）
  - 如果太大，退化为 FCFS，**等待时间增加**
  - 如果太小，上下文切换频繁，overhead 大，**吞吐量减小**。
    - **即使上下文切换开销为零，也会导致完成时间增加**
    - 相比集中完成，雨露均沾会导致大多数任务都在整个过程的后半段完成
  - 必须相对上下文切换开销来说足够大
- 所有任务的都不会等待超过 $(n-1)q$ 的时间
- 优点：在等待时间方面公平，**对短任务友好**
- 问题：对长任务来说，上下文切换开销累积。缓存不友好。
- 典型的时间片大小：10ms 到 100ms

如果任务长度均匀，且都比较长，FCFS 更好（无上下文切换开销）。反之，如果任务长度不均匀，RR 更好。

严格优先级调度（Strict Priority Scheduling）：

- 总是先执行优先级高的任务
- 每个优先级队列是按照 RR 执行的
- 问题：
  - Starvation：低优先级的任务被阻塞
    - 不公平（公平与平均完成时间的 trade-off）
  - Priority inversion：低优先级的任务持有锁，高优先级的任务请求此锁被阻塞，而中优先级的任务妨碍低优先级任务释放锁
    - 解决方法：优先级捐献
- 公平性的实现：
  - 每个 queue 分享 CPU 时间的特定份额
    - 高速通道人太多，反而比低速通道慢
  - 提高未被执行的任务的优先级
    - 所有任务都提升 => 互动性任务响应时间变差

假如我们能预测未来，就可以仿照最优 FCFS 调度：

- **Shortest Job First (SJF)**：
  - 每次都选择下一个 CPU burst 最短的任务
  - 也叫 **Shortest Time to Completion First (STCF)**。
  - 对**平均完成时间**来说，是**最优的非抢占式调度**
- **Shortest Remaining Time First (SRTF)**：
  - SJF 的**抢占式版本**
  - **如果新任务的完成时间比当前任务的剩余完成时间短，抢占当前任务**
  - 又叫 **Shortest Remaining Time to Completion First (SRTCF)**。
  - 对**平均完成时间**来说，是**最优的抢占式调度**。比 SJF 更好。
- 相比 RR，SRTF 可以在保证最小化平均完成时间的同时减少上下文切换次数。
- 优点：**最优的平均完成时间**
- 缺点：
  - 如果短任务太多，可能饿死长任务（**不公平**）
  - 需要预测未来：**我怎么知道这个任务要跑多久？**
    - 适应性：根据历史数据来决定政策。$\hat{t}_n = f(t_{n-1}, t_{n-2}, \ldots)$

Lottery Scheduling：

- 每个任务分配一定数量的 lottery tickets
  - 短的更多，长的更少，从而模拟 SRTF
  - 每个任务至少获得一个 ticket，以防止饥饿
- 每个时间片随机选择一个 ticket，执行中奖的任务
- 平均而言，任务获得的 CPU 时间将和其分得的彩票数量成正比
- 相比严格优先级调度的优点：对负载变化的反应更柔和

Multi-Level Feedback Scheduling：

- 多个队列，每个队列有不同的优先级
- 队列之间的调度：
  - Fixed priority：先高优先级队列，后低优先级队列
  - Time slice：每个队列分享特定份额的 CPU 时间（例如最高优先级 70%，次高优先级 20%，最低优先级 10%）
- 每个队列内的调度：
  - 前台 RR，后台 FCFS，
  - 前台短时间片 RR，后台长时间片 RR
- 调整每个人物的优先级：**模仿 SRTF**
  - 一开始最高优先级
  - 如果用完时间片还没完成，降低优先级
  - 如果时间片还没用完就完成了，提高优先级
- 算法的结果和 SRTF 相似：
  - 用时长的 CPU bound 任务会被很快降到低优先级
  - 用时短的 I/O bound 任务会留在高优先级队列
- 应用对算法的反制措施：插入短的无意义 I/O 以保持高优先级

许多调度器的 assumption：

- 经常睡眠，短 bursts => interactive 应用 => 高优先级
- 计算密集 => 低优先级

Multi-Core Scheduling

Affinity scheduling：OS 最好总把线程调度到某个固定的 CPU 上运行（cache reuse）。

Spinlock:

```
int val = 0;

Acquire() {
    while (test&set(val)) {}  // spin while busy
}

Release() {
    val = 0;    // atomic store
}
```

优点：

- **不需要上下文切换**，如果锁持有时间很短，性能比互斥锁好（睡眠并唤醒的开销过大）
- 适用于多个线程在 barrier 处等待的情况

但是每次 `test&set` 都是一次写，一个核的 `test&set` 会导致所有其他核的 cache 失效，导致 ping-pong。我们更希望 `test&test&set`：

```
Acquire() {
    do {
        while (val) {}  // wait until might be free
    } while (test&set(val) == 1);  // exit if acquired lock
}
```

Gang Scheduling：多个线程完成同一个任务，将它们一起调度。

- 使得 spin-waiting 更有效率

Alternative: OS 通知并行应用其线程被调度到了多少个核上

- 应用适应其分配到的核数
- 核数增加对性能的提升是 sublinear 的，多应用 **space sharing** 更好

Real-time Scheduling

Real-time scheduling（实时调度）：

- 目标：性能的**可预测性**
  - 实时系统中，性能是任务/类别中心的，被**先验**地保证
  - 常规系统中，性能是面向系统/吞吐量，是事后统计出来的
  - 实时系统需要可信地保证系统的最坏反应时间
- 硬实时：时间关键的安全导向系统
  - 必须满足所有 deadline（若可能）
  - 理想情况下，提前确定可行性（admission control）
  - **Earliest Deadline First (EDF)**
- 软实时：多媒体
  - 以高概率满足 deadline
  - Constant Bandwidth Server (CBS)

Earliest Deadline First (EDF)：

- 周期性任务 i，周期 P_i，执行时间 C_i，deadline $D_i^{t+1} = D_i^t + P_i$
- 每次都选择绝对 deadline 最急迫的任务执行
  - 抢占式调度：如果新任务的 deadline 比当前任务的 deadline 更早，则抢占当前任务
- **EDF feasibility testing**
  - 调度可行的充分条件：$sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$

Ensuring Progress

Starvation：线程在一段不定时间内没有进展。

我们考察哪些调度算法会导致 starvation：

- Non-work-conserving 调度器
  - 即使有任务在 ready queue 中，调度器也可能让 CPU 空闲。这是导致 starvation 的调度算法的一个平凡解
- **非抢占式调度**都有 starvation 的问题
  - **Last-Come, First-Served**
    - 如果 arrival rate 大于 service rate，早来的任务就会饿死
  - **First-Come, First-Served**
    - 如果当前任务一直不 yield（如死循环），其他任务会饿死
- **Round Robin**：
  - 每个任务都确定地在至多 $(n-1)q$ 的时间内被调度
  - 无 starvation 问题
  - 从等待时间角度，RR 调度是公平的
- **优先级调度**容易导致低优先级任务的 starvation
  - 不过比这更重要的是优先级反转，它使得高优先级的任务也有可能饿死
    - 低优先级任务持有锁，高优先级任务请求此锁被阻塞，而中优先级任务阻塞低优先级任务释放锁
    - 低优先级任务持有锁，高优先级任务 `while (try_acquire(lock)) {}`
  - 解决方法：优先级捐献
    - 高优先级任务将其优先级捐献给它依赖的低优先级任务
  - SRTF 也是一种优先级调度，长任务可能会被饿死
  - MLFQ 是 SRTF 的近似，自然也有相同问题

Case Study

Linux $O(1)$ scheduler：

- **Nice**: -20 ~ 19，nice 越小，优先级越高
- **优先级**：140 个优先级，值越小越优先
  - 0 ~ 99 是内核/实时任务
  - 100 ~ 139 是用户任务（priority = nice + 120）
- **所有算法都是 $O(1)$ 的**
  - 有一个 140 位的 bitmap 表示每个优先级是否有任务
- 两个优先级队列：active queue 和 expired queue
- active queue 中的任务用完时间片后会被放入 expired queue，所有任务都 expire 后两个队列交换
- 不同优先级的时间片大小也不同
- Heuristics:
  - 用户任务如果睡眠时间相比运行时间很长，说明它是 I/O bound 的，提升优先级
  - Interactive Credit: 睡眠很长则得到，运行很长则失去。
    - 作为一种滞后机制，防止突增突减触发不必要的切换
- 实时任务：
  - 总是抢占非实时任务
  - 优先级不会动态变化

Proportional-Share Scheduling：每个任务按优先级分配 CPU 份额（Lottery Scheduling）

- Lottery 调度的简单版机制：
  - 每个任务分得 N_i 个彩票
  - 选取一个彩票编号 $d \in 1, \dots, \sum_i N_i$
  - 将 N_i 排序，第一个满足 $\sum_i^j N_i > d$ 的 j 号任务被调度

Linux Completely Fair Scheduler (CFS)：

- 基本思想：追踪每个线程的 CPU 时间，调度 CPU 时间少的线程以使它们追上平均 CPU 时间
- 任何时刻总选择 CPU 时间最少的任务执行，直到其不再是 CPU 时间最少的任务
- 使用一个 heap-like scheduling queue
  - 插入和删除 $O(\log n)$
- 睡眠中的线程，CPU 时间不会增长，因此它们被唤醒时会自动地被 boost
  - 自动实现了 interactivatity
- 目标：**Responsiveness/Starvation freedom**
  - 短的等待时间，并确保每个进程都得到进展
  - 约束：**Target Latency**（所有进程都得到服务的时间）
  - **时间片长度 = Target Latency / number of tasks**
- 目标：**Throughput**

    - 避免过多开销
    - 约束：**Minimum Granularity**（最短时间片长度）
- **Proportional shares**
    - Basic equal share：$Q_i = TargetLatency \cdot 1/N$
    - Weighted share: $Q_i = TargetLatency(w_i/\sum_j w_j)$
    - 利用 nice 值：$w = 1024/(1.25)^{nice}$

Choosing the Right Scheduler

I Care About	Then Choose:
CPU throughput	FCFS
Avg. Completion Time	SRTF Approximation
I/O Throughput	SRTF Approximation
Fairness (CPU Time)	Linux CFS
Fairness (Wait Time to Get CPU)	Round Robin
Meeting Deadlines	EDF
Favoring Important Tasks	Priority

解释：

- 吞吐量：FCFS 没有上下文切换开销，吞吐量最大
- 平均完成时间：SRTF 是平均完成时间最优的调度算法
- I/O 吞吐量：I/O 任务通常剩余时间很短，会被 SRTF 算法优先调度
- CPU 时间公平性：Linux CFS 使每个进程拥有大致相同的 CPU 时间
- 等待时间公平性：RR 确保每个进程都在 $(n-1)q$ 的时间内被调度

Deadlock

死锁：对资源的循环等待。

两个 non-deterministic deadlock 的例子：

```
// Thread A
x.acquire();
// 在这里切换到 Thread B
y.acquire();
y.release();
x.release();

// Thread B
y.acquire();
// 在这里死锁!
x.acquire();
x.release();
y.release();

// 由于内存空间有限的死锁
// Thread A
allocate_or_wait(1 MB);
// 在这里切换到 Thread B
allocate_or_wait(1 MB);
free(1 MB);
free(1 MB);

// Thread B
allocate_or_wait(1 MB);
// 在这里死锁!
allocate_or_wait(1 MB);
free(1 MB);
free(1 MB);
```

Dining Lawyers 问题：

- 五根筷子，五个律师
- 每个律师需要两根筷子才能吃饭
- 如果每个律师同时抓住一根筷子，则没有律师可以吃饭 => 死锁！
- 解决方法：
  - 让某个律师放弃一根筷子，则另一个律师可以开始吃饭
  - 等他吃完后，就不会再有死锁了
- 避免死锁：
  - 如果拿走最后一根筷子会导致此后没有人能够持有两根筷子吃饭，则不允许拿走最后一根筷子

发生死锁的四个条件：

1. Mutual exclusion：资源只能被一个线程同时使用
2. Hold and wait：持有资源的线程正在等待获取其他被其他线程持有的资源
3. No preemption：资源只能被持有的线程在用完后主动释放
4. Circular wait：存在一个线程的循环等待链 $\{T_1, T_2, \ldots, T_n\}$，其中 T_i 等待 T_{i+1} 持有的资源，T_n 等待 T_1 持有的资源

Resource-Allocation Graph：

- 系统模型：
  - 线程 $T_1, T_2, \ldots, T_n$
  - 资源种类 $R_1, R_2, \ldots, R_m$
  - 每种资源有 W_i 个实例
  - 每个线程以 `request()`、`use()`、`release()` 的方式使用资源
- Resource-Allocation Graph：
  - 结点：T_i 和 R_j
  - 请求边：$T_i \to R_j$，表示 T_i 请求 R_j 的资源
  - 分配边：$R_j \to T_i$，表示 R_j 被 T_i 持有
- 图中有死锁则一定有环，但是有环不一定有死锁
- 死锁检测算法：
  - 思路：可以轻易地得知一个线程是否能就绪
    - 只要它请求的资源都空闲
  - 从图中删除所有这样的就绪线程，如果还剩下线程，则说明存在死锁

```
// Deadlock Detection Algorithm
Array<int> avail;    // Free resource counts for each resource type
Set<Thread> threads = all_threads;

do {
    done = true;
    for (Thread t : threads) {
        // t.request is an array of m,
        // representing number of each resource type t needs
        if (t.request <= avail) {
            // Thread t can finish
            avail += t.request;    // Release resources held by t
            threads.remove(t);
            done = false;
        }
    }
} while (!done);

bool is_deadlocked = (threads.size() > 0);
```

系统解决死锁的方法：

1. Deadlock prevention：
   - 一开始就不写出来会死锁的代码
2. Deadlock recovery
   - 让死锁发生，并设法从中恢复
3. Deadlock avoidance
   - 动态地推迟资源请求，从而避免死锁发生
4. Deadlock denial
   - 掩耳盗铃，忽略死锁

- 反正出问题了重启就完了

现代操作系统确保系统中没有死锁（deadlock prevention），忽略应用程序中的死锁（deadlock denial）。

预防死锁的方法：

- 无限资源
  - 虚拟内存
- 不允许共享资源
- 不允许 wait
  - 电话公司
  - 计算机网络（if collision, back off and retry）
- 令所有线程一次性请求所有需要的资源
  - 原子的 `acquire_both(x, y)`
- 强迫所有线程都按某个特定顺序请求资源
  - 释放的顺序无所谓

从死锁中恢复的方法：

- 终止线程，强迫其放弃资源
- 抢占资源
  - 虚拟内存的机制也可以视为抢占内存资源
  - 操作系统将暂时不用的内存 page out 到磁盘，就是抢占了这块内存资源
- 回滚死锁了的线程

避免死锁的方法：

- Naive 方法：当线程请求资源时，OS 检查这次请求是否会导致死锁
  - 一次请求可能不会直接导致死锁，但可能导致未来无可避免地陷入死锁
- 三种状态
  - Safe 状态：系统可以推迟资源获取以预防死锁
    - **系统存在一个线程执行顺序，使得此顺序下不会发生死锁**
  - Unsafe 状态：尚未死锁，但线程的请求可能会导致未来无可避免地陷入死锁
  - Deadlocked 状态：系统已经死锁了（deadlcoked state 也是 unsafe state）
- 理念：当线程请求资源时，OS 检查这次请求是否会导致系统进入 unsafe 状态
  - 如果会，则使线程等待其他线程释放资源

```
// 之前的例子
// Thread A
x.acquire();
// 在这里切换到 Thread B
y.acquire();
y.release();
x.release();

// Thread B
// 这次资源获取会导致 unsafe 状态!
// 在此处等待
y.acquire();
x.acquire();
x.release();
y.release();
```

银行家算法：

- 线程事先声明它的最大资源需求量
- 当线程请求资源时，银行家算法先假设此次请求被批准，然后运行死锁检测算法，若不会发生死锁，则批准
- 系统会一直处于 safe 状态

```
Map<Thread, Array<int>> max;    // Maximum resource counts for each thread
Map<Thread, Array<int>> alloc;    // Allocated resource counts for each thread
Array<int> avail;    // Free resource counts for each resource type

if (t.request > (max[t] - alloc[t]) || t.request > avail)
    return false;


// If thread t requests resources
alloc_sim = alloc.copy();
alloc_sim[t] += t.request;
avail_sim = avail - t.request;
Map<Thread, Array<int>> need_sim = max - alloc_sim;

// Check if the system is in a safe state
return !detect_deadlock(need, avail_sim);
```

对律师就餐问题，银行家算法给出的解决方案：

- 如果拿的不是最后一根筷子，则批准拿走
- 如果拿的是最后一根筷子，但拿走后仍然有其他人能吃饭，则批准拿走
- 假如律师有 k 只手
  - 如果拿的是最后一根筷子，且拿走后没有人能拿够 k 根筷子，则不批准拿走
  - 如果拿的是倒数第二根筷子，且拿走后没有人能拿够 $k-1$ 根筷子，则不批准拿走
  - ...

Scheduling in Modern Computer Systems

ZygOS

ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks

场景：serve us-scale RPCs

- 应用：KV-stores、In-memory DB
- 数据中心环境：fan-out/fan-in（一个人给很多人发消息/一个人收到很多人的消息）
- Tail-at-scale 问题：
  - 一个请求分成若干子请求，大多数子请求延迟很低，少数请求的延迟很高，拉高了总延迟（总延迟 = max(子请求延迟)）
- 目标：在一个激进的尾延迟 service-level objectives (SLO) 下提高吞吐量
- 方法：对于叶结点
  - 减少系统开销
  - 调度

Queueing theory:

- Processor
  - FCFS
  - Processor sharing (RR)
- 多队列/单队列
  - 每个核一个队列还是所有核共享一个队列
- Inter-arrival 分布：泊松分布
- Service time 分布：
  - Fixed
  - 指数
  - Bimodal
- 无系统开销，服务时间独立，性能上界

Baseline：

- Linux：
  - Partitioned connection delegation
    - 每个核一个队列
    - 非 work-conserving：某个核空闲时不会从其他核的队列中取任务
  - Floating connection delegation
    - 每个核一个队列

    - Work-conserving
- Dataplanes:
  - 与 Linux (partitioned conn) 不同，许多工作在用户空间完成，没有内核-用户上下文切换开销
- ZygOS 的目标：Dataplanes + Linux (floating conn)

执行模型：

- Shuffle layer
  - 每个核有自己的 shuffle queue，当队列空时，可以从其他核的 shuffle queue 中偷取任务
  - 偷取完的任务通过 shuffle layer 归还任务原主人，由原主人还给网络层
- 通过 shuffle layer 使得多队列表现收敛到单队列表现

Tiresias

挑战：

- 调度：不可预测的训练时间
- 任务放置：过于激进的 job consolidation 会造成 GPU 碎片化和较长的 queueing delay

方法：

- Discretized 2D Age-Based Scheduler
  - 每次调度选择 GPU 时间最少的任务执行
  - GPU 时间 = 执行时间 * 占用 GPU 核数
  - 以时间片为单位，避免频繁上下文切换
  - 本质上 MLFQ 变种
- Model profile-based placement
  - 如果模型的 tensor size 是 highly skewed 的，则需要 consolidation

实验结果媲美 SRTF。

DRF

Fair-sharing：

- 每个用户获得 $1/n$ 的资源
- 泛化：max-min fairness
  - 每个用户获得 $1/n$ 的资源，除非它不需要这么多
- 再泛化：weighted max-min fairness
  - 每个用户获得 $w_i/\sum_j w_j$ 的资源，除非它不需要这么多

Fairness 的定义：

- Share guarantee
  - 每个用户至少获得 $1/n$ 的资源，除非它不需要这么多
- Strategy-proof
  - 用户没有动机谎报更多需求
- Pareto efficiency

问题：如果**公平地**为**不同的需求**分配**多种资源**?

模型：需求向量 $< 2, 3, 1 >$

Natural policy:

- Asset fairness: 每个用户所有种类的资源的简单加和是相等的
  - 不满足 share guarantee
- Dominant resource fairness
  - dominant resource：使得 $\frac{\text{资源占有量}}{\text{资源总量}}$ 最大的资源种类
  - dominant share：用户占优资源的 $\frac{\text{资源占有量}}{\text{资源总量}}$
- 在 dominant share 上应用 max-min fairness
  - 不同用户的 dominant share 相等，除非它们不需要这么多
  - 可以证明此策略满足 share guarantee、strategy-proof 和 Pareto efficiency

Competitive Equilibrium from Equal Incomes (CEEI)：

- 每个用户相同的初始禀赋

- 他们会通过交易达到均衡
- 但这不是 strategy-proof 的（不如 DRF 公平）

FairRide

模型：

- 用户按照固定速率访问相等大小的文件
  - r_{ij}: 用户 i 访问文件 j 的速率
- Allocation policy 决定选择哪些文件放入缓存
  - p_j：文件 j 被缓存的比重
- 用户关心其缓存命中率 $HR_i = \frac{total\_hits}{total\_accesses} = \frac{\sum_j p_j r_{ij}}{\sum_j r_{ij}}$

性质：

- Isolation guarantee (share guarantee)
  - 没有用户的状况比 static allocation 更差
- Strategy-proofness
  - 用户没有动机谎报访问速率
- Pareto efficiency

定理：没有分配策略能同时满足这三个性质

FairRide:

- 满足 isolation guarantee 和 strategy-proofness
- 达到近似最优的 Pareto efficiency
- 方法：
  - 为每个用户分配 $1/n$ 的禀赋
  - 多个用户分享同一个文件，则它们平分访问的 cost
  - 阻塞不付费的用户访问（实现为 delaying）
  - 以 $p(n_j) = \frac{1}{n_j} + 1$ 的概率阻塞用户访问，其中 n_j 是缓存文件 j 的用户数
    - 例如 $p(1) = 0.5$
- 作弊总会得到更坏的结果

Memory

Address Translation and Virtual Memory

不同的进程/线程共享相同的硬件资源（CPU、内存、硬盘、I/O 设备），因此，我们需要虚拟化。

进程虚拟地址空间：可访问地址及其状态的集合。

- 当读写某个地址时，可能发生
  - 正常内存读写
  - I/O 操作（I/O mapped memory）
  - 程序中止（segmentation fault）
  - 与其他程序的通信

Memory multiplexing：

- Protection：
  - 禁止访问其他进程的私有内存
- Translation：
  - 处理器访问虚拟地址
  - 避免重叠
  - 为程序提供统一的内存视图
- Controlled overlap：
  - 不同线程的私有状态不能占据同一块物理内存
  - 需要重叠时可以实现重叠（通信）

另一种视角：介入进程行为

- OS 介入进程的 I/O 操作：所有 I/O 操作通过系统调用实现
- OS 介入进程的 CPU 使用：中断使 OS 可以抢占线程

- OS 介入进程的内存访问：
  - 每次内存访问都经过 OS 太慢了
  - 地址翻译：硬件支持的常规访问介入
  - 缺页（page fault）：非常规访问，trap 到 OS 处理

Segmentation

Uniprogramming：

- 无翻译
- 无保护
- 同一时刻仅有一个应用程序运行，独占 CPU 和所有内存

Primitive multiprogramming：

- 无翻译
- 无保护
- Loader/Linker 调整程序内各指令包含的地址（load、store、jump）
- 一个程序的 bug 可能影响其他程序，甚至 OS

Multiprogramming with protection: **Base and Bound**

- Base：进程在物理内存中的起始地址
- Bound：进程虚拟地址的最大范围
- **加载时重定位**：
  - 加载时由**加载器**直接将程序中的虚拟地址修改为物理地址
    - 若基址为 `0x1000`，则程序中的虚拟地址 `0x100` 被改写为 `0x1100`
  - 特点：
    - **静态绑定**：加载时完成地址转换，运行时基址不变
    - **灵活性差**：一旦加载无法移动
    - **无需硬件支持**
- **运行时重定位**：
  - 加载时保留虚拟地址，**运行时通过基址寄存器动态转换**
  - 特点
    - **动态绑定**：运行时完成地址转换，基址可在进程切换时动态调整
    - **灵活性高**：进程可加载到任意空闲内存区域，减少碎片
    - **需要硬件支持**：基址寄存器和地址转换电路
- 优点：
  - **简单**
  - **保护**：**进程间隔离、进程和 OS 隔离**
    -缺点：
  - **碎片化**
    - 内部碎片化：每个进程堆和栈之间的内存浪费
    - 外部碎片化：进程之间的内存浪费
  - **不支持分段（稀疏的地址空间）**
    - 地址空间是单一的连续块，不支持代码、数据、堆、栈分段
  - **进程间难以共享内存**
  - **进程地址空间大小受限，无法扩展**

带 **segmentation** 的 base and bound：

- 虚拟地址划分为两部分：**段号**（segment number）和**段偏移**（offset）
- 处理器内部存一个**段表**（segment map）
  - 段号为索引
  - 段表项为 **(base, limit, valid) 三元组**
- 内存访问时，根据虚拟地址的段号字段，在段表中索引对应的段表项
  - 如果**有效且没有越界**，则计算物理地址
  - **物理地址 = 段表项中的基址 + 虚拟地址的段偏移**

分段的若干观察：

- **每条内存访问指令都触发地址翻译**
- 高效地支持了**稀疏的虚拟地址空间**
- 栈触发 fault 时，系统会自动扩展栈空间

- 段表需要**保护模式**
  - 代码段应只读
- 段表存放在 CPU 中，是**全局**的，**上下文切换时无需保存和恢复**
- 当段无法装进内存时，会换出到硬盘
- 问题
  - 物理内存需要容纳**变长的块**（段），导致外部碎片化
    - **如果段定长（页），则可消灭外部碎片化**
  - 为了装得下所有段，可能需要多次移动整个进程的地址空间
  - 换出到磁盘的粒度太大：**整个进程**！
  - 碎片化：
    - 外部碎片化：已分配的段之间的空隙
    - 内部碎片化：已分配的段中未使用的空间

Paging

Paging：

- 虚拟地址空间被划分为**固定大小的页**（page）
  - **不存在外部碎片化！**
  - 页大小应小于段大小，以减轻内部碎片化
  - 虚拟地址被分割为**虚拟页号**（Virtual Page Number, VPN）和**虚拟页偏移**（Virtual Page Offset, VPO）
  - 物理地址被分割为**物理页号**（Physical Page Number, PPN）和**物理页偏移**（Physical Page Offset, PPO）
  - 其中 VPO 和 PPO 相同
- **页表**（Page Table）
  - **每个进程都拥有一个页表，存放在物理内存中**
  - 索引：VPN
  - 页表项（Page Table Entry, PTE）：(PPN, 权限位)

进程之间的共享页

- 共享页的 PPN 出现在**所有共享该页的进程的页表**中
- 也就是说，**不同的进程用不同的虚拟地址访问同一个物理地址**
- **每个进程地址空间的内核区域是共享的**
  - 进程在用户模式无法访问内核区域，但在用户 -> 内核切换时，内核既可以访问内核区域，也可以访问用户区域
- 不同进程运行同一份代码时，代码段是共享的（只执行）
- 用户级别的系统库（只执行）
- 共享内存段（shared memory segment）

页表讨论：

- 因为页表是 per-process 的，**上下文切换时，页表指针和页表界限需要保存和恢复**
- 保护是如何实现的?
  - Per process 的地址翻译
  - 双模式
  - 进程不能修改自己的页表
- 优点：
  - 简单的内存分配
  - 容易实现共享
- 缺点：
  - 单级页表，**页表项太多了**（且**大部分是空的**），存不下

两级页表：

- 单级页表中 20 位的 VPN 进一步划分为 10 位的 VPN1 和 VPN2
- 二级页表基址存放在 PageTablePtr 寄存器（CR3）
- **第一级页表的 PTE 存放第二级页表基址**
- **第二级页表的 PTE 存放 PPN**
- 有效位为零：
  - **Segfault（权限错误）**
  - **缺页（页未缓存到内存）**

基于页表的虚拟内存系统的功能：

- **Demand paging**：**内存中只存放活跃的页。不活跃的页放在硬盘上**（PTE 有效位置零），访问时触发缺页异常，由 OS 将其换入到内存中。

- **Copy-on-write**：
  - Unix `fork` 系统调用**复制父进程的页表**，并**将两份页表的所有 PTE 都标记为只读**
  - 写操作触发缺页异常，OS 实际复制对应的页
- **Zero-fill-on-demand**：
  - 新的数据页应当被清零（be zeroed），防止敏感信息泄露
  - 将 PTE 标记为无效，使用时触发缺页异常，OS 清零对应的页
- **内存共享**：
  - 两个进程的二级页表的 PTE 指向同一个物理页（共享一页物理内存）
  - 两个进程的一级页表的 PTE 指向同一个二级页表（共享一大块物理内存）

多级翻译：段 + 页

- 虚拟地址分为 (VSN, VPN, VPO)
- 段表
  - 索引：VSN
  - 段表项：(base, limit, valid) 三元组，其中 base 为页表基址
- 页表
  - 索引：VPN
  - 页表项：(PPN, 权限位)
- **上下文切换时，最高级段寄存器、最高级页表基址需要保存和恢复**

x86-64：四级页表

- 48 位虚拟地址：(9, 9, 9, 9, 12)
- **一页 4 KB，因此 VPO 占 12 位**
- **PTE 大小为 8 字节，一页可以存放 512 个 PTE，因此每级 VPN 占 9 位**

IA64：六级页表

- 64 位虚拟地址：(7, 9, 9, 9, 9, 9, 12)
- 很慢
- 太多 almost-empty 页表（高级页表管很大一块物理内存，大部分都用不到）

多级页表分析：

- 优点：
  - **按需创建 PTE**（单级页表必须创建所有 PTE），**节约内存**，**对稀疏地址空间友好**
  - 容易的内存分配
  - 容易的共享（段级别和页级别）
- 缺点
  - 每页都需要一个指针
  - 页表需要是连续的（10b-10b-12b 地址组织使得每个页表都在同一页，解决了此问题）
  - 查表的时间开销（k 级查表需要 k 次内存访问以获得 PPN）
    - 加上实际的内存访问，总共需要访问 $k+1$ 次

Dual-Mode 操作：

- **进程不能修改自己的页表**
  - 否则它就能访问整个物理内存了
- 硬件提供至少两个模式
  - 用户模式：进程只能访问自己的页表
  - 内核模式：进程可以访问所有页表
  - 通过设置仅内核模式可见的控制寄存器可以切换模式
  - 内核可以切换到用户模式，用户程序必须调用特殊异常来切换到内核模式

Inverted Page Table:

- **传统多级页表，每个页表必须存放所有的 PTE，浪费内存空间**
- 倒置页表通过一个**全局的 hash 表**，将 (PID, VPN) 映射到 PPN
- 倒置页表的大小和虚拟地址空间无关，只和物理内存大小有关（对于 64 位机器很有吸引力，因为 64 位机器前者远大于后者）
- 用硬件实现 hash chain，比较复杂
- **页表缓存局部性不好**：相邻的虚拟页不一定在一起

地址翻译比较

	优势	劣势
Simple Segmentation	**快速的上下文切换**（CPU 存储全局段表）	**内部/外部碎片化**
Paging (Single-Level)	**无外部碎片化**、快速简单的内存分配	**过大的页表**（接近虚拟内存大小，大部分是空的）、内部碎片化
Paged Segmentation	**页表大小接近虚拟内存中页的数量**，快速简单的内存分配	页访问需要多次内存访问
Multi-Level Paging	页表大小接近虚拟内存中页的数量，快速简单的内存分配	页访问需要多次内存访问
Inverted Page Table	页表大小接近物理内存中页的数量	复杂的哈希实现，**页表缓存局部性差**

Caching

TLB（Translation Look-aside Buffer）：处理器用虚拟地址向 MMU（Memory Management Unit）发出请求，MMU 在 TLB 中找到对应物理地址回应给 CPU（或触发异常）。

衡量缓存性能的指标：平均访问时间

$$\begin{aligned}\text{Average Access Time} &= \text{Hit Rate} \cdot \text{Hit Time} + \text{Miss Rate} \cdot \text{Miss Penalty} \\ &= \text{Hit Time} + \text{Miss Rate} \cdot \text{Miss Penalty}\end{aligned}$$

如果没有缓存，每次实际 DRAM 访问需要 页表级数 + 1 次 DRAM 访问，开销极大，且如果页表在硬盘中，还需要磁盘 I/O。

缓存不命中的原因

- **Compulsory**：冷启动
- **Capacity**：缓存不够大
- **Conflict**：多个内存位置被映射到同一个缓存位置
  - 解决方法：增大缓存、提高 associativity
- **Coherence**：其他进程（I/O）更新了内存

Cache 回顾：

- 地址分为 (Cache Tag, Cache Index, Byte Offset)
- 直接映射：
  - 每组只有一行
- N-way set associative：
  - 每组有 N 行
  - Cache Index 决定组号，比较每行的 Cache Tag 确定对应行（或确定 Cache Miss）
    - 用 Cache Index 而不是 Cache Tag 作为组索引，目的在于**避免相邻地址被分配到同一组，造成冲突性不命中**
- Fully associative：
  - Cache Index 不存在，所有行都可以存放任何数据
  - 需要比较所有行的 Cache Tag 确定对应行（或确定 Cache Miss）
- 缓存替换策略：
  - LRU：Least Recently Used
  - Random：随机选择一行替换
  - 直接映射不存在非平凡替换策略，因为每个组只有一行
- 写入策略：
  - Write-through：每次写入都更新下层存储器（DRAM）和 cache
    - 写延迟高
    - 总线带宽压力大
  - Write-back：只更新 cache，直到 cache 被替换时才更新 DRAM
    - 频繁的写场景下性能更好
    - 复杂

物理索引高速缓存 vs 虚拟索引高速缓存：

- 物理索引高速缓存：
  - **CPU 通过虚拟地址访问 TLB，TLB 翻译得到物理地址访问高速缓存**
  - 页表中存放物理地址
  - 更常见

    - 好处：
        - **同一个数据块在高速缓存中只存一份**
        - **上下文切换无需 flush 高速缓存**
    - 坏处：
        - **TLB 处于内存访问的关键路径**
- 虚拟索引高速缓存：
    - CPU 通过虚拟地址访问高速缓存和 TLB（**高速缓存访问和内存访问可并行**）
    - 页表中存放虚拟地址
    - 坏处：
        - **同义词问题**：**同一个数据块在高速缓存中可能存多份**（因为不同进程的不同虚拟地址可能对应同一个物理地址）
        - **异义词问题**：**上下文切换需要 flush 高速缓存**（因为不同进程的相同虚拟地址可能对应不同的物理地址）

TLB 组织：

- Miss time 极高（多级页表遍历）
- 如果用低位作为 TLB 的组索引，则 code、data、stack 段可能会映射到同一组
    - 至少需要 3 路组相联
- 如果用高位作为 TLB 的组索引，则小程序可能只会用到一组
- TLB 一般很小，只包含 128-512 个条目（现在更大）
- 小 TLB 一般全相联
- 更大的则在全相联 TLB 之前放一个直接映射 TLB（4-16 个条目），称为 TLB slice

TLB 查找和高速缓存查找并行：

- (VPN, VPO) => (VPN, cache index, byte offset)
- **VPO 恰好又被划分为 cache index 和 byte offset，且 VPO = PPO**
- 从而 TLB 查找（使用 VPN）和高速缓存查找的**组索引过程**（使用 PPO）可以**并行**进行：
    - TLB 使用 VPN 查找对应的 PPN，同时高速缓存使用 PPO 查找对应的 cache set
    - TLB 取出 PPN，高速缓存根据 PPN 比对找到对应的 cache line
- VPO 为 12 位，这限制了高速缓存的大小，更大的高速缓存难以完全并行化，需要其他设计
- 虚拟地址索引的高速缓存能更完全地并行

上下文切换时

- 因为虚拟地址空间也被切换了，TLB 条目都无效了
- 选项：
    - **无效化 TLB 条目**：**简单但开销大**（两个进程来回切换）
    - **在 TLB 中包含 PID**：需要硬件支持
- 如果页表改变了，也需要无效化 TLB 条目
    - 比如**页被换出**
    - TLB Consistency
- 虚拟索引的高速缓存，还需要 flush

Demand Paging

缺页:

- 发生在地址翻译失败时
    - 原因：
        - **PTE 无效**：页不在物理内存中，需要换入
        - **特权级违规**（Privilege level violation，用户执行内核指令/访问内核空间地址）
        - **访问违规**（Access violation，写只读页、用户访问内核页等）
    - 这会导致 fault/trap
        - interrupt 指外部中断
    - 可能在指令取指或数据访问时发生
- Protection violation 通常终止指令执行
- 其他缺页会使 OS 处理并重试该指令，有可能是以下几种情况：
    - **分配新的栈页**
    - 使得页可访问（**Copy-on-write**）
    - 从下级存储器中读取页（**demand paging**）

Demand Paging：

- 现代程序用许多物理内存，但它们将 90% 的时间花在执行 10% 的代码上

- 将主存作为硬盘的缓存
- 工作过程：
  - 进程访问一个页，但页表中该页的 PTE 无效
  - MMU traps 到 OS（触发缺页）
  - OS 中的缺页处理程序
    - 选择一个空闲的物理页
      - OS 维护一个空闲物理页列表
      - 当物理内存占用过高时，OS 会运行 reaper：写回脏页、清零冷页
      - 如果没有空闲物理页：根据替换算法选择要替换的页。如果该页被修改过，将其写回到硬盘，并在 TLB 中无效化该页的 PTE
    - OS 定位新页在交换空间的位置，将其加载到内存
    - 更新页表 PTE
    - 将用户进程标记为 ready
  - 调度器未来会调度该进程，重试该指令
  - 在等待页读写的过程中，OS 可以调度其他进程
- Demand paging 作为 caching：
  - 块大小：4 KB
  - 全相联（交换空间中的任何页都可以映射到物理内存中的任何位置）
  - 替换策略
  - 未命中：从低级存储器中填充
  - 写：写回（需要 dirty bit）
- 提供了无限内存的幻觉：核心是**透明的间接寻址层（页表）**
  - 用户不需要知道数据的真实存储位置

Demand Paging 应用：

- 栈扩展
  - 分配并清零一个页
- 堆扩展
- 进程 fork
  - 创建页表的拷贝
  - 将所有 PTE 标记为非写的
  - 共享的只读页仍然是共享且只读的
  - 写时复制（Copy-on-write）
- `exec`
  - 懒加载：只在实际访问时将二进制文件加载到内存中
- `mmap`
  - 显式共享内存区域
  - 将文件映射到内存

将可执行文件加载到内存：

- OS 初始化寄存器，设置堆栈
- OS 为进程创建一个完整的虚拟地址空间（VAS）映射：
  - 通过页表记录所有虚拟页的状态：
    - 驻留页：已加载到物理内存
    - 非驻留页：仍存储在磁盘的交换文件（Swap File）中
- 硬件访问的页一定要驻留在物理内存中——通过缺页异常来实现
- **OS 必须记录非驻留页在交换文件中的位置**，以便在缺页异常时加载到内存中
  - `find_block(pid, page_num) -> disk_block`
  - 通常也会想将驻留页备份一份到 swap file
  - 可以将代码段直接映射到硬盘映像，从而节省 swap file 空间。如果程序有多份运行实例，可以共享代码段

Replacement Policy

工作集（working set）模型：

- 每个进程的工作集是它在过去一段时间内访问的页的集合
- 随着时间推移，高速缓存被越来越多进程的工作集填满

Demand paging cost:

$$\begin{aligned}\text{Effective Access Time} &= \text{Hit Rate} \cdot \text{Hit Time} + \text{Miss Rate} \cdot \text{Miss Penalty} \\ &= \text{Hit Time} + \text{Miss Rate} \cdot \text{Miss Penalty}\end{aligned}$$

Demand Paging 不命中分析：

- Compulsory：冷启动，页尚未加载到物理内存
  - 预取（prefetch），提前加载页到内存。**需要预测未来的访问模式**！
- Capacity：物理内存不够大
  - 加物理内存
  - 调整每个进程分配的内存比例
- Conflict：虚拟内存是“全相联”的，**原理上不存在 conflict miss**
  - 任意虚拟页可以映射到任意物理页，不存在固定位置的竞争
- **Policy**：**页本来在物理内存中，然而被替换政策过早地踢出了**

替换策略:

- Demand paging 的 **miss penalty 极高：硬盘 I/O！**
- First In, First Out (FIFO)：
  - 最老的页被替换
  - 不好——**FIFO 考虑的是资历，而非使用频率**
- RANDOM
  - 硬件上简单，TLB 的典型做法
  - **不可预测**
- **MIN（Minimum）**
  - **替换未来最长时间不使用的页**
  - **理论最优**，但我们**无法预测未来**
  - 过去是未来的良好指示
- **Least Recently Used (LRU)**：
  - **替换最近最少使用的页**
  - **局部性**：最近用得少，未来可能也用得少
  - 似乎是 **MIN 的优秀近似**
  - 实现：
    - 链表：**每次访问都将对应页移到链表头部，替换时删除链表尾部的页**
    - **每次页访问时需要立即更新链表**
    - 性能差：链表操作需要很多指令
  - 实践中使用 LRU 的近似实现

Stack property：**当加物理内存时，不命中率不会增加**

- LRU 和 MIN 算法可以保证此性质
  - 它们向前/向后看 X 个不同的物理页
  - 当物理内存增加到 $X+1$ 个页时，它们仍然会向前/向后看到原来的 X 个页，加上一个新的不同的页
  - **更大的内存始终包含原内存的所有页**
- FIFO 不保证此性质
  - 加内存后，FIFO 包含的页可能与原内存完全不同

Clock 算法：

- 将物理页排成一个环，一个指针指向当前页
- 硬件每次访问某个物理页时，会将该页的 use bit（accessed bit）置 1
- 每次缺页时，
  - 指针前进一页
  - 检查 use bit：
    - 如果为 0，则替换该页
    - 如果为 1，则将其清零，继续前进一页
- 指针最多前进 N 次，N 是物理页数量（当所有 use bit 都被设置）
  - 如果指针一边清空 use bit，OS 一边上下文切换重置 use bit，会不会永远找不到 victim?
  - 触发缺页的那个线程的帧不会被访问，因此至少总能找到它们作为 victim
- **指针前进很慢是好事**
  - **缺页少，或很快能找到 use bit 为 0 的页**
- 简单粗暴将物理页二分成两组

Clock 算法的变种：Nth Chance

- OS 为每帧维护一个计数器
- 每次缺页时，
  - 指针前进一页

    - 检查 use bit：
        - 如果为 0，则**增加计数器**，若计数器达到 N，则替换该页
        - 如果为 1，则将其和计数器清零，继续前进一页
- **N 的取值：越大越接近 LRU，越小效率越高**
- **替换脏页的开销更大，可以给脏页更多机会**

不需要硬件支持的脏位：利用只读位 W

- 初始，标记所有页为只读，清除所有软件脏位
- 写触发缺页：
    - 若可写，OS 置软件脏位并置 `W = 1`
    - 否则结束用户进程
- 当页被换出，清除脏位，标记为只读

不需要硬件支持的 use bit：

- 初始，标记所有页无效，清除所有软件 use 位（和软件脏位）
- 读写触发缺页
- OS 置软件 use 位：
    - 若为读，标记为只读
    - 若为写且可写，置脏位，标记为可写
- 时针扫过时清空软件 use 位，并标记为无效。**不清空脏位**，留待换出时使用

如果允许利用缺页，能否实现比 Clock 更优的替换策略？

Second Chance List:

- 内存分为两部分：FIFO 的 Active list 和 LRU 的 Second Chance list
- Active list 中的页有效，SC list 中的页无效
- 若访问的页在 Active list 中：正常访问
- 若访问的页不在 Active list 中：
    - **Active list 中溢出的页总是放到 SC list 队首，并标记为无效**
    - 若在 SC list 中：将它移动到 Active list 队首，标记为可读写
    - 若不在 SC list 中：page in 到 Active list 队首，标记为可读写；page out SC list 队尾的页
- 如果 SC list 中没有页，算法退化为 FIFO
- 如果 SC list 中包含所有页，算法等价于 LRU，但每次访问都触发缺页
- 取一个中间值，相比于 FIFO：
    - 更少的磁盘访问——近似 LRU，长时间未用才会换出
    - 缺页开销更大

Free list:

- OS 维护一个空闲物理页列表
    - 在后台用 clock 算法填充
    - 脏页在进入 free list 前需要开始写回
- 类似 VAX second-chance list：如果物理页在被回收前又被使用了，就直接中止换出
- **缺页处理更迅速：可立即获取可用物理页**

Reverse Page Mapping (Coremap):

- **当驱逐物理页时，需要无效化对应的页表项——需要物理页到虚拟页的映射**
- 注意一个物理页可能被多个页表共享——多个页表项映射到同一物理页
- 方法一：每个 page descriptor (Linux 中的物理页对象) 维护一个 PTE 链表。
    - 太昂贵
- 方法二：每个 page descriptor 维护一个指向 VMA (Virtual Memory Area) 的指针，OS 从 VMA 中找对应的页表项

物理页分配

- **每个进程得到相等还是不等份额的内存？**
    - Equal allocation: 每个进程份额相同
    - Proportional allocation: 每个进程份额与其 size 成正比
    - Priority allocation: 每个进程份额与其优先级成正比
    - **Page-Fault Frequency allocation: 若进程缺页频率过高，则增加其份额，否则减少其份额**
        - 难以确定 upper bound 和 lower bound
        - 不用 upper bound 和 lower bound，**将进程的缺页频率排序，将缺页率低的进程的页给缺页率高的进程**

- 每个进程至少需要一定数量的页，以确保可以运行
- Replacement Scope：
  - 进程可以驱逐所有物理页
  - 进程只能驱逐自己的物理页

Thrashing:

- 进程没有足够的页，不断触发缺页换入换出，实际进展很慢
- 程序内存访问有时间局部性和空间局部性，一段时间内（如最近一万条指令）访问的一组页称为工作集
- 如果分配的内存小于工作集，进程就会 thrashing
- 所有进程的工作集之和就是当前的物理页需求，如果物理页需求大于物理内存大小，就会导致 thrashing
  - 策略：此时**挂起一些进程或换出一些进程**，避免 thrashing，提高效率

Compulsory miss：

- 访问从未访问过的页（lazy loaded 或刚刚被换入）
- Clustering：缺页时一次性加载 faulting page 周围的多个页
  - 磁盘访问连续多个页的开销相比访问单个页增加很小
- 工作集追踪：追踪进程的工作集，换入进程时直接换入整个工作集

Memory Management in Modern Computer Systems

- Memory Abstraction
  - FaRM
  - vLLM
- Demand Paging: remote memory over network
  - InfiniSwap
  - AIFM
- Demand Paging: memory swapping between GPU memory and host memory
  - PipeSwitch
  - TGS

FaRM: Fast Remote Memory

硬件趋势：

- 内存变得廉价：服务器内存上 TB，小集群数十 TB
- 数据中心网络：40 Gbps 吞吐量，1-3 us 延迟，RDMA primitives

RDMA: **Remote Direct Memory Access**

- 机器 A 想要读机器 B 的内存：
  - A 的 CPU 向 A 的 NIC 发送一个 RDMA 请求，A 的 NIC 将请求通过网络发送给 B
  - B 的 NIC 收到请求，通过 DMA 将 B 的内存拷贝到自己内部的缓冲区，通过网络发给 A 的 NIC
  - A 的 NIC 收到数据，通过 DMA 将数据拷贝到 A 的内存
  - 整个过程绕过了 B 的 CPU 和内核
- 相比 TCP 有更优的吞吐量和延迟

数据中心应用访问模式不规则，对延迟敏感。

Setup:

- 我们有
  - TB 级的 DRAM
  - 数百 CPU 核心
  - RDMA 网络
- 目标
  - **数据存放在内存中，用 RDMA 访问**
  - **数据和计算放在一起**
    - 传统模型中服务器存储数据，客户端执行应用
    - 对称模型：服务器既存储数据，也执行应用

共享地址空间：

- 所有机器的内存属于同一个地址空间
- 位置（属于哪台机器）、并发、故障处理都是透明的，编程者无需关心

优化：

- Locality awareness
  - 将一起被访问的数据放在一起
  - **计算以 RPC 形式被发送的数据所在机器上被执行**
  - 完成后再把数据发回去

Transactions:

- 执行（Execution）阶段
  - 利用 RDMA 读取数据，写 buffer 在本地
- 提交（Commit）阶段
  - 将所有数据上锁
  - Validate 数据（是否最新），失败则回滚重试
  - 利用 RDMA 更新其他服务器的数据，解锁

vLLM

背景：服务 LLM 很慢且成本高昂

- Auto-Regressive 架构的 GPU 利用率不高：**batch 多个请求，并行化**
- 然而 batchsize 被 KV cache 的低效内存管理所限制
  - **先前系统的 KV cache 存储在一个按照最大长度 pre-allocated 的连续内存块中，导致严重的内部碎片化**
  - **而不同请求的 max length 可能不同，导致外部碎片化**

PagedAttention:

- 请求就像 OS 中的进程
- 内存被分为若干固定大小、连续的 KV blocks（每个可以存 4 个 token），就像 OS 中的页
- Block table 将逻辑块号映射到物理块号
- Allocate on demand
- 额外的重定向带来 10-15% 的开销
- 内部碎片化：只有每个 sequence 的最后一个 block 存在内部碎片
- 无外部碎片化：固定大小的连续分块
- 页表机制还可以轻易地实现共享和 copy-on-write（parallel sampling：一个输入产生多个输出；beam search）

当内存不够用时:

- 选项 1：Swapping to CPU
- 选项 2：Preempt and Recover (i.e. delete and recompute)
- **两种选项都需要作用在整个请求上，因为请求的每一步都用到所有先前的 tokens**
- 块大小较小时 swapping 开销较大
- **Recomputation 可以并行，较快**
- 最终选择：Request Preemption & Recovery

和 OS paging 的异同：

- 相同点：
  - OS 的页 <=> KV blocks：减轻碎片化
  - 进程间共享页 <=> 采样间共享 KV blocks：减少内存浪费
- 不同点
  - **单级块表：和数据相比，块表占用空间很小**
  - **Preemption & Recovery：抢占请求，通过 recomputation 恢复**

InfiniSwap

背景：

- 应用的工作集如果不能完全 fit 到内存中，性能就会显著下降
- **集群中已分配的内存占 80%，但实际使用的只占 50%**

想法：

- **当机器 1 内存不够时，从其他机器的内存中拿**
- 不需要添加新的硬件，不需要修改现有应用
- 可以容忍失败
- 能够 scale

方法：

- **在虚拟内存系统之下做了一个 InfiniSwap Block Device，作为 swap space 和 request router**
- 本地磁盘作为 InfiniSwap block device 的异步 backup
- **通过单边 RDMA 以及运行在远程机器上的 InfiniSwap Daemon，绕过远程机器的 CPU**
- 以 slab 为单位，用分布式的分配算法（Power of Two Choices）来分配内存

特点：

- 支持一对多（一个机器请求多个机器的空闲内存）和多对多

AIFM

Memory 是非弹性的，被物理内存容量所限制

先前基于 OS paging 的方案（InfiniSwap）性能不好：

- 语义 gap
  - 以页为单位导致 R/W amplification：只读一个字节，但要加载一整页
  - OS 缺乏应用的知识，无法预取：应用遍历链表，对 OS 而言就是随机访问
- 高内核开销
  - 缺页时从远程 swap in，浪费很多 CPU 时钟周期

方法：

- Remotable Data Structure Library
  - **提供数据结构 API，底层封装了 prefetcher**
  - 解决了语义 gap
- Userspace Runtime
  - **在应用中 yield 以避免陷入内核态**
  - 解决了高内核开销
- Pauseless Evacuator
  - **无暂停地将本地的对象转移到远程机器**
  - 解决了内存回收问题
- Remote Agent
  - **将计算转移到远程机器**
  - 解决了网络带宽小于 DRAM 带宽的问题

和 InfiniSwap 的区别：

- **InfiniSwap 在 kernel 中以 page 为单位，不需要修改应用实现**
- **AIFM 的 userspace runtime 中，以对象为单位，需要改一些应用实现**

PipeSwitch

目前深度学习的训练任务和推理任务常常在不同集群上运行。因为**推理任务有明显的时间周期性**（白天多凌晨少），如果能**在相同的集群上同时运行训练任务和推理任务**，就能更高效。

目标：

- **多个深度学习任务的细粒度 multiplexing，要求 GPU 高效**
- **毫秒级上下文切换延迟和高吞吐量**

方法：

- 上下文切换：
  - 停止当前任务，准备下一个任务
  - 通过 pipelined model transmission 执行任务
  - 清理上一个任务的环境

上下文切换的开销来源：

- 模型传输 model transmission
- 内存分配
- 任务初始化
- 任务清理

Pipelined model transmission and execution：

- **深度学习模型是分层的**
- **将模型传输和执行并行：传输第 i 层的同时可以执行第 $i-1$ 层**
- 开销：
  - 模型传输被切分成若干次，需要多次调用 PCIe——粒度不能太细，否则 PCIe 调用开销大
  - 传输和执行的同步开销
- 不以层为单位，而是**以 group（多层）为单位**，从而**摊平前面的两个开销**
- 大大降低了模型传输的开销

Unified memory management：

- **通过一个 Memory Daemon 统一管理显存**

Active-standby worker switching：

- **将每个任务的初始化切成两段，第一段不需要显存，可以在一开始就完成**
- **初始化的第二段在可以在调度后立即开始，和先前任务的清理并行，因为它只标记而不实际使用显存**

TGS

核心思想：**分享 GPU 核心以增强 GPU 利用率**

深度学习训练任务有两种：

- **Production job**：全速运行，不能接受性能损失
- **Opportunistic job**：可以接受性能损失，利用空闲 GPU 核心

先前工作：

- 应用层：AntMan
  - 修改深度学习框架（TensorFlow/PyTorch）
  - 支持 GPU 共享和显存 oversubscription
  - 透明性差，需要用户使用特定的框架，需要维护特定的框架
- OS 层：NVIDIA MPS
  - GPU 利用率低，不支持显存 oversubscription
  - 需要应用知识以设置合适的资源限制
  - 错误隔离差，一个任务出错影响其他任务
- OS 层：NVIDIA MIG
  - 性能 isolation：不能任意分区 GPU，不能动态调整 GPU 资源
  - 兼容性差，不支持 multi-GPU instance 的共享

共享 CPU 资源：

- Strawman 方法：优先级调度
  - 根据 GPU 内核队列控制 opportunistic job
  - 但队列的状态不能很好反映剩余 GPU 资源，GPU 利用率低

TGS：**适应性的速率控制**

- **监控 production job 的 GPU 发送速率，保证其不受限制**
- **根据 production job 的速率动态调整 opportunistic job 的速率**

Transparent unified memory of TGS:

- 核心思想：利用 CUDA unified memory，透明地统一显存和 host memory
- 在第一次访问时才分配实际物理显存，GPU 利用率高
- 显存 oversubscribed 时，TGS 改变虚拟显存映射，从而驱逐 opportunistic job 的显存到 host memory

TGS 的特点：

- **透明性**：不需要修改应用
- **Performance isolation**：opportunistic job 的性能不会影响 production job
- 高 GPU 利用率
- **Fault isolation**：opportunistic job 的错误不会影响 production job

I/O

总线（bus）：数据传输通路 + 协议

协议：发起方（initiator）发起请求、仲裁机制（arbitration）授权、识别接收方、地址/控制/数据信号传输握手

靠近 CPU 的地方带宽高、延迟低，但不灵活。靠近 I/O 子系统的地方带宽低、延迟高，但灵活。

用一个总线就能连接很多设备，但这使得一次只能进行一个事务（transaction），因此需要仲裁机制。

PCI（Peripheral Component Interconnect）：

- 一开始是总线
- 并行总线有许多局限性：必须兼容最慢的设备，拖慢整体时钟频率

PCIe：

- 不再是并行总线，而是一组串行链路（lane）
- 速度不再受限，慢速设备不再拖累整体

CPU 通过 device controller（可视为一个嵌入式的计算机）与 I/O 设备通信，有 port-mapped I/O 和 memory-mapped I/O 两种方式。

CPU 上也有很多 I/O 相关的部件，例如 Sky Lake 的 PCH (Platform Controller Hub)，它包含了 PCIe、USB、SATA 等控制器。

I/O 的动作参数：

- 数据粒度：**字节 vs. 块**
- 访问模式：**顺序 vs. 随机**
- 传输机制：**programmed I/O vs. DMA**

Programmed I/O：

- **CPU 直接控制 I/O 设备和内存之间传输**
- 硬件实现简单，容易编程
- 传输消耗 CPU 时钟周期

DMA（Direct Memory Access）：

- **控制器直接控制 I/O 设备和内存之间传输**，绕过 CPU
- CPU 只需设置 DMA 控制器
- DMA 控制器可以在 CPU 执行其他任务时传输数据
- DMA 控制器可以在传输完成后中断 CPU，通知传输完成

I/O 设备通知 OS 的方法：

- **I/O 中断**
  - 设备生成中断来通知 OS
  - 容易处理不可预测的事件
  - 中断处理开销大
- **Polling**
  - OS 定期轮询设备的状态寄存器
  - 开销小
  - 轮询不频繁/不可预测的 I/O 事件浪费很多 CPU 时钟周期
- 实际设备可能结合两种方式，例如网卡用中断处理第一个到达的包，然后用轮询处理后续包。

Device driver：

- 内核中设备相关的代码，负责直接与设备通信
- 可以分成两部分
  - Top half：实现 `open`、`close`、`read`、`write` 等系统调用，开始 I/O 操作，可能使线程休眠
  - Bottom half：作为中断处理程序运行，处理传输，I/O 完成时可能唤醒线程

I/O subsystem：

- 提供 I/O 设备访问的统一接口
- 块设备：`read`、`write`、`open`、`close`
- 字符设备：`get`、`put`
- 网络设备：socket API
- Timing：
  - Blocking 接口：调用后线程阻塞，直到操作完成
  - Non-blocking 接口：立即返回成功读或写的字节数，不阻塞线程
  - Asynchronous 接口：调用后线程继续执行，I/O 完成时通知用户

Hard Disk Devices (HDDs)

磁盘：

- 非易失性存储设备
- 大容量，低价格
- 块级别随机访问
- 随机访问性能差，顺序访问性能好

硬盘结构：

- 传输单元：**扇区**（sector）
  - **盘片**（surface）上连续的一圈扇区：**磁道**（track）
  - 纵向堆叠的磁道：**柱面**（cylinder）
  - **磁头**（head）在柱面上移动，读取或写入数据
- 扇区之间被不使用的 guard 区域分隔，减少写操作污染相邻磁道的概率
- 只有最外一圈的磁道被使用

Shingled Magnetic Recording (SMR)：

- 磁道重叠写入，增加存储密度

读写操作需要三个阶段：

- **寻道时间**（seek time）：磁头移动到目标磁道
- **旋转延迟**（rotational latency）：等待目标扇区转到磁头下方
- **传送时间**（transfer time）：传输扇区数据

$$\text{Disk Latency} = \text{Queueing Time} + \text{Controller time} + \text{Seek Time} + \text{Rotational Latency} + \text{Transfer Time}$$

磁盘性能算例：

- 忽略 queueing time 和 controller time
- 平均寻道时间 $5ms$
- 转速 $7200rpm$（即一秒转 120 圈）。
  - 转一圈的旋转延迟 $1/120 \times 1000 = 8.33\text{ms}$
  - 平均旋转延迟 $8.33/2 = 4.17\text{ms}$
- 传送速率 $50MB/s$，每个扇区 4KB
  - 一个扇区的传输时间 $4/50000 = 0.08\text{ms}$
- 读磁盘上一个随机扇区的延迟：$5 + 4.17 + 0.08 = 9.25\text{ms}$
- **若在同一个柱面上读，则不需要寻道时间**
- **若读下一个相邻扇区，则只需要传送时间**
- **寻道时间和旋转延迟是随机访问的主要开销**

磁盘控制器有许多精巧设计：

- 扇区有精密的纠错机制
- Sector sparing：透明地将坏扇区重新映射到同一个盘面上的空闲扇区
- Slip sparing：重新映射所有扇区，以便保持顺序访问的行为
- Track skewing：不同磁道的扇区编号不同，顺序访问时将**寻道与旋转重叠**，消除访问下一个磁道时的旋转延迟

Solid State Drives (SSDs)

闪存：

- 非易失性存储设备
- **块级别随机访问**
- **读性能好，随机写性能差**
- **只能整块擦除**
- **写入次数越多，寿命越短**

SSD：

- 没有寻道和旋转延迟
- 没有运动的部件：轻、能耗低、静音、抗冲击
- 寿命有限
- 读写性能不一致

- **顺序读和随机读的带宽都很高**

写操作比较复杂

- **只能写块中的空页**
- 控制器维护一个空块的资源池，预留一定空间
- **写比读慢十倍，擦除比写慢十倍**
- 可以一次性读写一个 chunk（4 KB）
- 但一次性只能覆写整个 256 KB 的块
  - 擦除操作很慢
  - 每个块寿命有限，只能擦除约一万次
  - 我们不想覆写或擦除整个块

解决方法：

- Layer of Indirection：
  - 维护一个 **Flash Translation Layer** (FTL)，将逻辑块号（OS 看到的）映射到物理块号（flash memory controller 看到的）
  - 可以自由地重新映射数据，而不影响 OS
- Copy on Write:
  - **OS 更新数据时，不重写已有的页，而是将新数据写入空页**
  - **更新 FTL 映射，将逻辑块号映射到新物理块号**
- 从而：
  - **小的写操作不需要擦除和重写整个块**
  - **SSD 控制器可以在块之间分散负载，延长寿命**
  - **旧版本的页被 GC，擦除后加入空闲块池**

I/O Performance

性能指标：

- **响应时间**（response time）/ **延迟**（latency）：完成一个 operation 的时间
- **带宽**（bandwidth）/ **吞吐量**（throughput）：operation 完成的速率

I/O 性能的影响因素：

- 软件路径（队列）
- 硬件控制器
- I/O 设备服务时间

Queuing Theory

假设 arrival time 固定为 T_A，service time 固定为 T_S。则 arrival rate 为 $\lambda = 1/T_A$，service rate 为 $\mu = 1/T_S$。

利用率 utilization $U = \lambda/\mu = T_S/T_A$, 其中 $\lambda \leq \mu$。

随着 offered load T_S/T_A 的增加，吞吐量也线性增加，而 queuing delay 始终为零（没有形成队列）。直到 T_S/T_A 达到 1，此时到达时间恰好等于服务时间，队列恰好开始形成，吞吐量饱和。此后 T_S/T_A 增加，吞吐量不再增加，queuing delay 线性增加（没有上界）。

如果请求以 burst 的形式到达，尽管平均 arrival time 不变，平均利用率很低，但 queuing delay 也可能很高。

指数分布：

- PDF: $f(x) = \lambda e^{-\lambda x}$
- CDF: $F(x) = 1 - e^{-\lambda x}$
- 指数分布刻画的是一个发生概率关于时间均匀分布的随机事件，连续两次发生的时间间隔
- 指数分布具有无记忆性：
  - $P(X > s + t | X > s) = P(X > t)$
  - 即过去的时间不会影响未来的概率分布
- 数学期望：
  - $E(X) = 1/\lambda$
  - 即单位时间内期望发生 λ 次事件

随机分布的数字特征：

- 均值：$E(X) = \sum p(x) \cdot x$
- 方差：$Var(X) = E(X^2) - E^2(X)$
- Squared Coefficient of Variation (**SCV**)：$SCV = Var(X)/E^2(X)$

    - SCV = 1 时，分布是指数分布
    - SCV = 0 时，分布是常数分布
    - 磁盘响应时间的 SCV 约为 1.5，大多数响应时间小于均值

Queuing theory：

- 假设：
    - **arrival rate 等于 departure rate**（系统处于均衡）
        - departure rate 不可能超过 arrival rate
        - 如果 arrival rate 大于 departure rate，则系统过载，队列无限增长
    - Arrival 和 departure 都可以用概率分布表示
    - 队列长度无限制
    - 不同的到达独立、无记忆
- 参变量：
    - λ：平均 arrival rate
        - T_{arr}：平均 arrival time，$\lambda = 1/T_{arr}$
    - T_{ser}：平均 service time
        - μ：service rate，$\mu = 1/T_{ser}$
    - C：squared coefficient of variation (SCV)，即服务时间的方差除以服务时间的期望的平方
    - u：利用率，$u = \frac{T_{ser}}{T_{arr}} = \lambda \cdot T_{ser}, 0 \leq u \leq 1$
- 我们希望计算的变量
    - T_q：queuing delay
        - 总延迟 $T = T_{ser} + T_q$
    - L_q：队列长度, $L_q = \lambda \cdot T_q$（Little's Law）
        - **水库水量等于进水速率乘水停留的时间**
        - 假设系统运行 T 时间，则 $T_q = \frac{L_q T}{\lambda T}$（平均 queuing delay 等于所有顾客在队列中等待的时间除以顾客数）
- **结论**
    - M/M/1 队列（**到达和服务时间都服从指数分布，单个服务器**）：
        - $T_q = T_{ser} \cdot \frac{u}{1-u}$
    - M/G/1 队列（**到达时间服从指数分布，服务时间服从任意分布，单个服务器**）：
        - $T_q = T_{ser} \cdot \frac{1}{2}(1 + C) \cdot \frac{u}{1-u}$
        - 代入 $C = 1$ 就得到上面的结果

算例：磁盘 I/O

- 用户发射请求的速率：$\lambda = 10 \times 8\mathrm{KB/s}$
- 请求和服务时间服从指数分布（$C = 1$）
- 平均服务时间 $T_{ser} = 20\mathrm{ms}$
- 磁盘利用率为 $u = \frac{T_{ser}}{T_{arr}} = 20\mathrm{ms} \times 10/\mathrm{s} = 20$
- 平均 queuing delay $T_q = T_{ser} \cdot \frac{u}{1-u} = 20\mathrm{ms} \cdot \frac{0.2}{0.8} = 5\mathrm{ms}$
- 队列长度 $L_q = \lambda \cdot T_q = 10/\mathrm{s} \cdot 5\mathrm{ms} = 0.05$ 个请求
- 总延迟 $T = T_{ser} + T_q = 20\mathrm{ms} + 5\mathrm{ms} = 25\mathrm{ms}$

如何提高 I/O 性能?

- Speed：make everything faster
- 并行：更解耦的系统（多条独立的总线/控制器）
- 优化性能瓶颈
- 利用队列：
    - 队列可以吸收突发负载，平滑化流
    - admission control：**有限长度队列**
        - **限制了延迟，但引入了不公平和活锁**

磁盘性能最高的时候：

- **大的顺序访问**
- 很多的访问以至于它们可以被 piggybacked（**reorder 队列，使同一个磁道上的请求连续**）
- bursts 既是挑战（队列变长、延迟增加）也是机遇（piggyback 和 batching（一次上下文切换处理多个请求））

Disk scheduling：

- FIFO: 对请求方公平，但不能很好地 piggyback
- **SSTF** (Shortest Seek Time First):
    - **优先处理离磁头最近的请求**

    - 虽然叫 SSTF，但旋转时间也需要考虑在内
    - 可能导致 **starvation**
- **SCAN**：
    - 类似 Elevator algorithm
    - **在磁头移动方向上的最近请求优先**
    - 从外侧扫描到内侧，然后再从内侧扫描到外侧
    - 没有 starvation
- **C-SCAN**（Circular SCAN）：
    - **磁头从外侧扫描到内侧，然后直接跳到外侧继续扫描**
    - 保证了每个请求都能被处理
    - **比 SCAN 更公平，不偏心中间的磁道**

Network I/O：

- 和 disk I/O 类似
- 提高 network I/O 性能
    - 分布式应用
    - 优化 TCP/IP 协议栈
    - kernel bypass
        - 用户空间的网络协议栈
        - offload to NIC

I/O 系统的栈结构：

- 应用程序
- High-level I/O
- Low-level I/O：文件描述符
- System Calls
- File System
- I/O Drivers
- Hardwares

File Systems

I/O 系统的栈结构中，文件系统是中间的支柱，为 I/O API 和系统调用提供了硬件设备的一层抽象。

- I/O API 和系统调用
    - Variable-size buffer
    - 内存地址索引
- 文件系统
    - 块（block）
    - 逻辑索引，典型粒度为 4 KB
- 硬件设备
    - HDD：
        - 512 B 或 4 KB 大小的扇区
    - SSD：
        - Flash translation layer（FTL）
        - Physical block，通常 4 KB 大小
        - Erasure page

文件系统：OS 中用文件、目录等接口包装硬盘等设备的块接口的 layer

- 经典 OS 情形：受限的硬件接口（array of blocks）被转换为有如下性质的接口
    - Naming：可以通过名字查找文件
    - Organization：文件可以组织成目录树，并被映射到物理块
    - Protection：权限控制
    - Reliability：在发生系统崩溃或硬件失败时，保持文件完整性

文件的不同视角：

- 用户视角：durable 的数据结构
- 系统视角（系统调用）：字节集合（UNIX）
- 系统视角（OS 内）：块集合

- 块是逻辑传输单元，不同于扇区（物理传输单元）
- 块大小大于或等于扇区大小

文件：一组顺序存放在逻辑空间中的用户可见的块

- 大多数文件很小
- 大多数字节被包含在大文件中

目录：

- 目录是特殊的文件，它包含其下**文件名到文件号（file number）的映射**
  - 文件号对应的可以是文件，也可以是另一个目录
  - **目录项**（directory entry）：每一条文件名到文件号的映射
- **进程不能直接访问目录的 raw bytes**，`read` 系统调用在目录上不 work。`readdir` 可以遍历目录项，但不向进程暴露其 raw bytes
  - 因为**不能让进程修改目录中的映射**
- `open`、`creat` 会遍历目录结构体，`mkdir`、`rmdir` 会添加或删除目录项。
- 解析 `/my/book` 的过程：
  - 读根目录的文件头（它在磁盘上的固定位置）
  - 读根目录的第一个数据块，线性地搜索 `my` 目录项
  - 找到 `my` 目录项后，读 `my` 目录的文件头
  - 读 `my` 目录的第一个数据块，线性地搜索 `book` 目录项
  - 找到 `book` 目录项后，读 `book` 目录的文件头
- 当前工作目录（current working directory）：**Per-address-space** 的指向当前工作目录的指针，用于文件名解析，使用户可以使用**相对路径名**

磁盘管理：

- 磁盘就是扇区的数组
- Logical Block Addressing (LBA)
  - 每个扇区有一个整数的逻辑块地址（LBA）
  - 控制器将 LBA 翻译为物理位置

文件系统需要：

- 知道哪些块包含哪些文件的数据
- 知道目录中有哪些文件
- 知道哪些块是空闲的
- 这些信息都存在磁盘上

File System Design

文件系统设计的重要因素：

- 磁盘性能（重要）
  - **最大化顺序访问**，减少寻道
- read/write 前 open
  - 权限检查，提前查找文件资源
- 文件使用时，大小固定
  - write 可以扩展文件大小
- 组织为目录树
- 块的 allocation 和 free 保证高性能

文件系统的组件：

- 通过文件路径在目录结构体中查找文件 inumber
- 通过 inumber 对应的文件头结构体（inode）找到文件对应的块
- 进程的 open fild descriptor 表其实就是把 fd 映射到 inumber
- `open` 负责 name resolution，将路径名翻译成 inumber
- `read` 和 `write` 都利用 inumber 进行
- 四个组件：
  - 目录
  - index 结构体
  - 数据块
  - free space map

Linux 中的两个 open-file table：

- **Per-process open-file table**：每个进程独有，位于 PCB 中，将文件描述符（fd）映射到 system-wide file table 的条目
  - 不同进程的相同 fd 无关，不一定指向同一个文件
  - 0、1、2 分别指向标准输入、标准输出、标准错误
- **System-wide open-file table**：所有进程共享的内核全局数据结构，包含当前文件偏移、访问模式（O_RDONLY、O_WRONLY、O_RDWR）、状态标志（O_APPEND、O_NONBLOCK）、引用计数和**指向 inode/vnode 的指针**
  - **每个 `open` 系统调用都会在 system-wide open-file table 中创建一个条目**
- 例子
  - 进程先打开文件，然后调用 `fork` 创建子进程，则
    - 子进程获得父进程 open-file table 的完整副本，父子进程的对应 fd 指向同一个 system-wide open-file table 条目，该条目的引用计数增加
    - **父子进程共享文件偏移量**
  - 进程先 `fork` 创建子进程，然后分别打开同一文件，则
    - 两个独立的 system-wide open-file table 条目被创建，父子进程的对应 fd 指向不同的条目
    - **父子进程的文件偏移量独立**
    - 写入结果取决于内核调度，可能交替写入，若未用 `O_APPEND`，则可能互相覆盖
  - `int dup(int oldfd)`：在 per-process open-file table 中新增条目，指向 oldfd 对应的 system-wide open-file table 条目
    - 返回的新 fd 是 oldfd 的副本
    - 对应的 system-wide open-file table 条目的引用计数增加
    - **新 fd 和 oldfd 共享文件偏移量**
  - `int dup2(int oldfd, int newfd)`：将 newfd 指向 oldfd 对应的 system-wide open-file table 条目，若 newfd 已经打开，则先关闭它
    - oldfd 对应的 system-wide open-file table 条目的引用计数增加
    - newfd 对应的 system-wide open-file table 条目的引用计数减少（若 newfd 已经打开）

Case Study: File Allocation Table (FAT)

FAT：

- 假设已经有目录结构体（可以将路径名解析为文件号）
- Disk storage 被组织为一个**有 N 个块的数组**
- FAT 有 N 个表项
  - 每个表项负责**文件号到块的映射**
- **一个文件的所有块对应的 FAT 表项被组织成一个链表**
  - 链表的首元素（root block）的索引就是文件号
  - 一个文件不一定被映射到连续的块
- file_read $31, \langle 2, x \rangle$：FAT 中索引为 31 的表项为链表头，此链表的索引为 2 的块中的索引为 x 的字节
- 文件偏移：**文件块号**（即该文件对应的块链表中对应块的索引）和**块内偏移量**
- 未使用的块被标记为空闲（free）
  - 扫描空闲块，或者维护 free list
- **FAT 被存储在硬盘上**（需要被持久性地保存，不能存在 RAM 里）
- 格式化：**清零所有块，将所有 FAT 表项标记为 free**
- 快速格式化：**只将所有 FAT 表项标记为 free**

FAT 目录

- 目录是一个特殊的文件，它被组织成一个目录项（文件名到文件号的映射）的链表
- 目录中有空闲的空间用于添加新的目录项
- FAT 中，**文件的 attribute 被存储在目录中，而不是在文件自己里**
- 根目录被存在硬盘上一个指定位置（FAT 中为 block 2，FAT 没有 block 0 和 block 1）

讨论：

- 给定文件号找到对应的块的时间：找第一个块 $O(1)$，找其他的块 $O(N)$，其中 N 是文件块的个数
- 不对文件的 block layout 做保证，即**不一定连续存储**
- 顺序访问性能：**由于文件不一定连续存储，性能没有保证**
- 随机访问性能：**需要遍历 FAT 链表才能找到对应块，性能较差**
- 碎片化：**没有碎片**
- **小文件友好**
- **大文件不友好**：不一定连续存储，随机访问性能差

Case Study: Unix File System

Inode：

- **File number（inumber）是 inode 数组的索引**

- **每个 inode 对应一个文件，包含其元数据**
  - **文件元数据和文件本身关联**，而不是像 FAT 那样存储在目录中
- Inode 通过一个多级树结构组织文件对应的块：
  - 12 个直接指针，每个直接指向一个 4 KB 块，共可索引 48 KB 数据
  - 1 个一级间接指针，指向一个包含 1024 个指针的块（一个指针大小 4 字节），每个指针指向一个 4 KB 块，共可索引 4 MB 数据
  - 1 个二级间接指针，指向一个包含 1024 个一级间接指针的块，共可索引 4 GB 数据
  - 1 个三级间接指针，指向一个包含 1024 个二级间接指针的块，共可索引 4 TB 数据
- 访问 block 23 需要两次磁盘访问：
  - 第一次通过一级间接指针访问指针块
  - 第二次通过指针块访问 block 23

Berkeley Fast File System (FFS)：

- 早期 Unix 和 DOS/Windows FAT 文件系统中，**文件头**（inode）**被存放在最外层柱面**（cylinder），且大小固定。这带来许多问题
  - 所有 inode 都**在一个篮子**，一次 head crush 可能损坏所有文件
  - **inode 和对应的文件距离很远**，导致不必要的寻道
  - 创建文件时不知道它将会有多大（unix 大多数写入都是追加），无法确定分配多少连续空间
- 解决方法：**Block Groups**
  - 盘面被分为多个 block group，每个 block group 包含
    - **特定目录的数据块**
    - **free space bitmap**
    - **inode 数组**
  - 为文件分配新块时采用 **first-fit 策略**
    - 小文件会填补 free space bitmap 前部的小洞
    - 而大文件能在 bitmap 后部找到大块连续空间
  - 每个 block group 保持至少 10% 的空余空间
- FFS inode 布局的优点：
  - **小目录的数据和文件头可以在同一个柱面，减少寻道**
  - **文件头比块小很多（几百字节），一次性能取出多个文件头**
  - **可靠性：磁盘不会一损俱损**

旋转延迟问题：

- 读一个扇区 -> 处理 -> 读下一个扇区
  - 在处理的同时，磁盘旋转过了下一个扇区，导致连续读取时一直错位，旋转延迟很高
- 解决方法一：**skip sector positioning**
  - **将文件隔块存储**
  - 可以由 OS 或现代磁盘控制器实现
- 解决方法二：**read-ahead**
  - **预读下一个扇区**
  - 可以由 OS 或现代磁盘控制器（带 RAM 作为缓冲区）实现

Unix 4.2 BSD FFS：

- 优点
  - **大文件和小文件都可以高效存储**
  - 大文件和小文件都有较好的**局部性**（**连续存储**）
  - **文件头和文件数据之间也有较好的局部性**
  - **不需要去碎片化**
- 缺点
  - 很小的文件效率不高（**单字节的文件也需要 inode 和一个数据块，一共占用两个块**）
  - **对连续存储的文件来说，inode 编码不够高效**（**每个数据块都需要一个指针，但一组连续块实际上只需要一个指针**）
  - 需要保留 10~20% 的空闲空间以防止碎片化

硬链接（hard link）：

- 将文件名映射到目录结构体中的文件号
- 当文件被创建时，第一个硬链接同时被创建
- `link` 和 `unlink` 系统调用可以创建和删除硬链接
- inode 维护引用计数，追踪文件被多少硬链接引用。当引用计数为 0 时，文件被删除

软链接 / 符号链接（soft link, symbolic link）：

- 普通目录项将文件名映射到 file number

- 符号链接目录项将文件名映射到另一个文件名
- 如果目标路径不存在，则为悬空链接（dangling link）
- 通过 `symlink` 系统调用创建符号链接

目录查找：打开 `/home/pkuos/stuff.txt`

- 根目录的 inumber 在内核中写死，假设为 2
- 读 inode 2 的数据块，根据其直接和间接指针找到根目录所在的数据块
- 读入根目录数据块，线性搜索 `home` 目录项，找到其 inumber
- 最终找到 `stuff.txt` 的 inumber，读入对应 inode 的数据块
- 设置 inode 对应的 file description（per-process open-file table），使 read/write 可以访问文件数据
- **检查最后一个 inode 以及所有目录 inode 的权限**

大目录：

- 早期文件系统将目录组织成链表或数组
  - 线性查找目录项，效率低
  - 找一个文件需要遍历整个目录
- B 树
  - **文件名的哈希作为 B 树的键**

Case Study: New Technology File System (NTFS)

NTFS：

- 现代 Windows 的默认文件系统
- **变长的 extent，而不是固定大小的 block**
- **Master File Table**（MFT）：
  - 对应 FAT 或 inode array
  - 每个表项最大 1 KB，对应一个文件或目录

MFT 表项：

- 四个部分：
  - **Standard Information**（SI）：文件的元数据，如创建时间、修改时间、权限
  - **File Name**（FN）
  - **Data Attribute**（DA）：文件数据，以键值对的形式存储
  - **空闲空间**
- 小文件：**数据直接存放在 data attribute 中**
- 中等大小的文件：**存 extent 的指针**（start 和 length）
- 大文件：**存指向 MFT 表项的指针**（相当于间接指针）和指向 extent 的指针
- 超大文件：**存装有 MFT 表项的 extent 的指针**（相当于二级间接指针）和其他指针

NTFS 目录：

- 实现为 B 树
- **文件号指明了文件在 MFT 中的表项索引**
- MFT 表项包含文件名
- 硬链接：MFT 表项中包含多个文件名

Buffer Cache

Buffer cache：

- **System-wide**
- 内核需要将硬盘块拷贝到主存中，才能读写其内容
- 被缓存的是文件系统中的四个组件
  - 数据块
  - inode
  - 目录的数据块
  - free space map
- `open`：
  - 先在目录中查找文件名，找到对应的 inode（这些目录数据块就顺便被缓存了）
  - 读 inode，在 PCB 中创建一个 file descriptor，指向 inode（inode 块也被缓存）
- `read`

    - 在 PCB 中根据 fd 找到 inode
    - 根据 inode 中的直接和间接指针找到数据块
    - 读入数据块到对应 buffer（数据块顺便被缓存到 buffer cache）
- `write`
    - 在 PCB 中根据 fd 找到 inode
    - 根据 inode 中的直接和间接指针找到数据块
    - 将数据块缓存到 buffer cache 并写入，这个过程也可能更新 free space map 和 inode（分配新块）
        - 被修改的块会被标记为 dirty
- buffer cache 完全由 OS 软件实现
- 块换入换出缓存不是原子操作
- 替换策略
    - LRU：**完整 LRU 实现的开销是可承担的**
        - 优点：只要内存装得下工作集，综合性能很好
        - 缺点：**当应用扫描整个文件系统时，不断缓存一次性文件**
    - 有的系统可以让应用选择替换策略
    - **Use Once 策略：块被访问之后就被丢弃**
- **Cache size**：
    - 物理内存既要给 buffer cache 用，也要给虚拟内存用
    - buffer cache 太大影响多任务能力
    - 太小又导致磁盘 I/O 过多（装不下工作集，频繁换入换出）
    - 解决方法：**动态调整 buffer cache 大小以求平衡**

文件系统预取

- **Read Ahead Prefetching**：
    - 大多数文件访问都是顺序的：预取当前读请求的块后续的块
    - **电梯算法可以高效地交错并发应用的预取**
- 预取太多：其他应用的磁盘请求的延迟增加
- 预取太少：并发文件请求中，**磁盘寻道和旋转延迟增加**

Delayed Writes：Buffer cache 是一个写回（write-back）缓存，它会将写操作推迟到数据块被换出时：

- Buffer cache 满，数据块被驱逐
- Buffer cache **周期性 flush**（**预防崩溃数据丢失**）
- delayed writes 的优点：
    - `write` 系统调用**快速返回用户**，不需要等待磁盘 I/O 完成
    - 磁盘调度器可以**对写请求重新排序**，以提高性能（**电梯算法**）
    - **推迟块分配，一次性分配多个块，更容易使块连续**，减少碎片化
    - **如果只是临时文件，可以避免不必要的磁盘 I/O**

对比 buffer cache 与 demand paging：

- LRU 开销对 demand paging 来说太大，只能用近似算法替代；但 **buffer cache 可以使用完整 LRU**
- Demand paging 在内存接近满时驱逐，buffer cache 除此之外还会**周期性 flush，以最小化崩溃时的数据丢失**

Durable File Systems

系统崩溃时，buffer cache 中的脏块会丢失。如果它恰好是目录的数据块，则会导致该目录下文件的 inode 指针丢失：文件系统状态错误，且内存泄漏！

重要的 "ilities"：

- **Availability**（可用性）：系统可以接受并处理请求的概率（系统有多少时间可用）
    - 以 9 来衡量
    - 99.9% 概率是“3-nines of availability”
    - 独立于 failure
- **Durability**（持久性）：系统从 fault 中恢复数据的能力
    - fault tolerance
    - 先前存储的数据能够被重新取得，无论 failure 是否发生
    - 磁盘挂了不可用，但数据没损坏：没有 availability，但有 durability
- **Reliability**（可靠性）：系统或组件能在给定条件下持续按要求工作指定时间的能力
    - 一般比 availability 更强，也包括 security、fault tolerance/durability

让文件系统 durable：

- 磁盘块包含 **Reed-Solomon 纠错码**（Error Correction Code, **ECC**），从小的磁盘缺陷中恢复数据
- **确保写操作短期内有效**
  - 放弃 delayed writes
  - 为 buffer cache 中的脏块采用 battery-backed RAM（non-volatile RAM or NVRAM）
- 确保写操作长期有效
  - 备份在磁盘其他位置/其他磁盘/其他服务器/其他大陆/...

RAID（Redundant Array of Inexpensive Disks）：

- 目标：可靠、性能、容量
- **虚拟化存储**：**多个物理磁盘被抽象为一个逻辑磁盘**
- RAID 1：Disk mirroring/shadowing
  - 每个磁盘都有一个“影子”
  - **写入时同时写入两个磁盘：写带宽减半**
  - **读时可以从任意一个磁盘读取**
  - 数据恢复：
    - 换用影子磁盘，将数据复制到另一块磁盘
    - hot spare：预留备胎磁盘以供替换
  - 适合高 I/O、高可用性环境
  - 最昂贵：100% 容量开销
- RAID 5：High I/O Rate Parity
  - **数据被分割成 4 个块，分布在 5 个磁盘上**（D_0 在磁盘 0，D_1 在磁盘 1，...，P_0 在磁盘 4）
  - 奇偶校验块：$P_0 = D_0 \oplus D_1 \oplus D_2 \oplus ...$
    - 因为异或就是模 2 的加法
  - 摧毁任一磁盘，仍可恢复完整数据
  - 可以将磁盘放在一起，也可以放在不同地方，用 Internet 连接（防止整个屋子发洪水把磁盘泡了）
- RAID 6 and other Erasure Codes
  - RAIDX 是一个 erasure code
    - 必须知道哪个磁盘坏了
  - 现在磁盘太大，以至于恢复磁盘的过程中可能又会有磁盘损坏
    - RAID 6 可容忍两个磁盘损坏
  - 更通用方法：Reed-Solomin codes
    - m 个数据块，$n-m$ 个校验块
    - 可以容忍 $n-m$ 个磁盘损坏
  - 例如，将数据分为 $m=4$ 块，总共生成 $n=16$ 块，分布在世界各处。则任意 4 块都可以恢复数据，非常 durable
- 非常 durable
- 读的可用性好
- 写的可用性差：必须写到所有备份

Reliable File Systems

RAID 可以防护磁盘物理损坏，但不能防护软件错误：

- 写入错误数据
- 部分磁盘未写入
- 静默数据损坏（bit rot）

Reliability 问题

- 单个文件操作可能涉及多个物理磁盘块：
- inode、间接指针的磁盘块、free space map、数据块
- 扇区重映射
- 但在物理层面，操作是原子的
- **一系列物理操作过程中间的崩溃或断电可能使系统处于不一致状态**
- 例如，银行转账时，在取出钱和存入钱之间发生了断电，导致钱丢失

两种解决方法：

- **Careful Ordering and Recovery**
- **Versioning and Copy-on-Write**

Careful Ordering

- **以特定顺序执行文件系统操作，使得操作序列可以被安全地打断**

    - 分配数据块 => 令 inode 指向数据块 => 更新 free space map => 更新目录
    - **先写数据，再更新目录项**，否则无法发现是否被打断
- 崩溃后恢复：
    - 读取数据结构以检查是否有未完成的操作
    - 清理/完成
- 应用
    - FAT 和 FFS (fsck)
    - Word, emacs 自动保存
- Berkeley FFS: 创建文件
    i. 分配数据块
    ii. 写数据块
    iii. 分配 inode
    iv. 写 inode 块
    v. 更新 free block bitmap（数据块和 inode 块从空闲变为在使用）
    vi. 更新目录项
    vii. 更新目录修改时间
- Berkeley FFS: 恢复
    i. 扫描 inode 表
    ii. 如果发现未连接文件（不在任何目录），删除或移入 lost+found 目录
    iii. 对比 free block bitmap 和 inode 树
    iv. 扫描目录查找丢失的更新/访问时间
- 问题：扫描时间复杂度和磁盘大小成正比

Copy-on-Write

- **不覆写已有的数据块并更新 inode，而是 Copy-on-Write 更新数据所在的块，并通过指针复用旧版本未修改的块**
- 所有修改完成后，单次指针切换更新根节点：原子操作
- 看似昂贵，但
    - 可以 **batch 更新**
    - **几乎所有写请求都可以并行**
    - **追加式的写天然是顺序的**
- 应用
    - NetApp 的 WAFL（Write Anywhere File Layout）
    - ZFS, OpenZFS
- ZFS 和 OpenZFS
    - 块大小可变，从 512 B 到 128 KB
    - 对称树：拷贝操作发起时即知文件大小
    - 版本号和指针存在一起
    - buffer 写操作
    - 空闲空间表示为每个 block group 中的 extent 树
        - 维护一个日志，延迟更新，直到 block group 被激活时

更通用的 reliability 解决方法：

- 用**事务**（transaction）完成原子操作
    - 保证单个文件操作的一系列连续子操作能原子地完成
    - **如果中途崩溃，文件系统的状态只有两种可能**：
        - **完成了所有子操作**
        - **没有完成任何子操作**
    - 大部分文件系统和应用都有应用
- 为媒介 failure 提供冗余：媒介上的冗余表示（ECC）、跨媒介的备份（RAID）

事务

- 事务是将系统从一个一致状态转变为另一个一致状态的原子性的读写序列
    - **相当于同步中的临界区**
    - FFS 也可以理解为事务的一种
- 典型结构：将一系列更新封装为一个事务
    - Begin Transaction：获得事务 ID
    - 一系列更新，发生失败则回滚
    - Commit Transaction：将所有更新写入磁盘
- 日志（log）
    - 写日志是原子操作

    - 追加式写入
        - 不可变性和事件发生的顺序性
    - Seal the commitment：事务完成时写入日志
    - **日志数据强制写入磁盘**（或 NVRAM），日志自身不能丢失
- Journaled 文件系统和 Log structured 文件系统的区别
    - Journaled 文件系统：**日志是用于恢复的辅助结构，文件数据和元数据仍然存储在传统的、原地更新的文件系统中**
    - Log structured 文件系统：**日志是文件系统的主要结构，所有数据和元数据都存储在日志中。不原地更新，而是 Copy-on-Write 更新数据和元数据**

Journaling 文件系统：

- 日志分为三部分：
    - 已实际完成的事务
    - pending 的事务（在日志中记录，尚未实际更新文件系统）
        - tail 指针指向其末尾
    - 正在写入日志的事务
        - head 指针指向当前写入的位置
- **先将更新写入日志，然后再实际更新文件系统中的数据结构**（inode 指针、目录映射、...）
- 垃圾回收：**当实际的更新完成后，清除它在日志中的记录**
- Linux 采用了类 FFS 的 ext2 文件系统，并在其上添加了 journal，得到了 ext3 文件系统
    - 选项：**将所有数据写入 journal，还是只写元数据**
- 创建文件：
    i. 根据 free space map 找到空闲数据块
    ii. 找到空闲 inode 表项
    iii. 找到对应的目录
    iv. 在日志中写事务开始
    v. 在日志中写 free space map，将对应块标记为使用中
    vi. 在日志中写 inode，使其指向数据块
    vii. 在日志中写目录项，将文件名映射到 inode
    viii. 在日志中写事务提交，该事务成为 pending 事务
- 所有对文件系统的访问都要先查看日志，因为文件系统中的数据结构可能是过时的
- 最终，日志中的 pending 事务被实际应用到文件系统中，并通过垃圾回收机制从日志中被删除
- Crash recovery
    - 恢复时扫描日志，未提交的事务直接被丢弃
        - 磁盘未被实际更新
    - 已提交的事务被保留，应用到文件系统中（可以立即应用，也可让其稍后自然完成）
- 整个 Journaling 机制的意义：**原子化地更新文件系统，崩溃不会导致文件系统不一致**
- 代价较高：
    - **所有数据必须被写入两次**
    - **现代文件系统只将元数据写入日志**
        - **文件内容数据直接裸着写入文件系统，不受日志保护**

Distributed Systems

中心化系统：主要功能由中心的一台物理计算机提供（Client-Server 模型）

分布式系统：多个分布的物理计算机共同完成任务

- 早期模型：集群
- 后期模型：peer-to-peer/wide-spread collaboration

分布式系统：

- 愿景
    - 获得很多小计算机廉价、简单
    - **增量式地提高功耗**
    - 用户可以完全控制某些组件
    - **用户间合作容易很多**：有网络文件系统就不需要微信传来传去
- 现实
    - **差可用性**：依赖所有电脑都在线
        - Lamport: "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."
    - **差可靠性**：任一电脑崩溃都可能造成数据丢失

    - **差安全性**：所有人都能闯进系统
    - **必须定位共享数据的多份拷贝**
    - Trust/Security/Privacy/Denial of Service
        - Lamport: "A distributed system is one where you can't do work because some computer you didn't even know existed is successfully coordinating an attack on my system."
- 透明性：将系统的复杂性隐藏在接口背后，从而使用户不必关心
    - Location
    - Migration：资源可以在不同机器间迁移
    - Replication
    - Concurrency
    - Parallelism
    - Fault Tolerance
- 透明性和合作要求不同的处理器之间能够通信

协议（protocol）：

- 关于如何通信的协议
    - 语法（Syntax）：指明信息的结构/格式
    - 语义（Semantics）：指明信息的含义
- 可以被状态机形式地描述

分布式应用：

- 多线程同步：没有共享内存，不能用 `test&set`
- send/receive：原子操作
- 接口：
    - Mailbox（mbox）：保存消息的临时缓冲区
    - `send(message, mbox)`：将消息发送到指定的 mailbox
    - `receive(buffer, mbox)`：等待 mbox 的来信，将其拷贝到 buffer 中
        - 如果有线程在此 mbox 上休眠，唤醒它

分布式共识达成：

- 共识问题
    - 多个结点都提出值
    - 有的结点正常，有的崩溃无响应
    - 所有剩余结点需要根据被提出的值决定一个最终值
    - 值可以是 true/false，也可以是 commit/abort
- 共识决定需要是 durable 的

Two General's Paradox：

- **不可靠的网络下，两个实体无法达成“同时做”的共识**
    - Alice：早上 8 点?
    - Bob：好！（这是 Bob 不知道 Alice 是否收到自己的回应，不敢发起行动）
    - Alice：收到！（这是 Alice 也不知道 Bob 是否收到自己的收到，也不敢发起行动）
    - Bob：收到收到！（同类）
    - ...
    - 两人永远无法确定对方是否收到自己的消息，导致都不敢发起行动
- 无法保证双方**同时做**

Two-Phase Commit

- **不试图保证双方同时做，只保证双方最终会去做**
- 分布式事务：两台或多台机器可以原子性地同意或不同意做某件事

Two-Phase Commit Algorithm：

- 每台机器维护一个持久、稳定的日志，记录承诺是否发生
    - 机器崩溃重启后，首先检查日志，恢复到崩溃前状态
- 理念：先确保所有人承诺将会提交，然后再要求所有人提交
- **准备阶段**（prepare phase）
    - 全局协调者（global coordinator）向所有参与事务的结点（participants）发送 VOTE-REQ
        - **参与结点在等待 VOTE-REQ 过程中可能超时而 abort（发送 VOTE-ABORT）**

    - 每个参与结点收到请求后，检查自己是否能够成功完成事务。将承诺写入自己的日志，并向协调者响应 VOTE-COMMIT（同意提交）或 VOTE-ABORT（拒绝提交）
    - 如果任一参与结点否决（abort），或响应超时，协调者将 "Abort" 写入自己的日志，并向所有参与者发送 GLOBAL-ABORT。参与者们也将 "Abort" 写入自己的日志
    - 如果所有参与者都响应同意，协调者将 "Commit" 写入自己的日志，进入提交阶段
        - 一旦进行到此步，事务就必须被最终完成
        - 也就是说 worker 的 failure 只在准备阶段有影响
- **提交阶段**（commit phase）
    - 协调者向所有参与结点发送 GLOBAL-COMMIT
        - **等待 GLOBAL-COMMIT 的参与结点不会因超时而中止，必须一直等**
    - 参与者执行实际的提交操作，在自己的日志中记录，并响应协调者
    - 协调者收到所有参与结点的响应后，将 "Got Commit" 写入自己的日志
- 日志用于保证所有机器要么全部提交，要么全部回滚

分布式 decision making：

- 为什么需要分布式决策?
    - Fault tolerance：**部分故障不影响系统运行**
    - 决策模式之后，结果多处存储
- 为什么 2PC 不被 Two General's Paradox 影响
    - 2PC 只需要保证所有结点**最终**达到同一决策，不要求**同时性**
    - 允许重启和继续
- 2PC 的问题：Blocking
    - 站点 B 在日志中记录“准备好提交”，向协调者站点 A 发送 VOTE-COMMIT，然后崩溃
    - 站点 A 也崩溃了
    - 站点 B 重启后检查日志，发现日志中已经记录了“准备好提交”，向站点 A 发送信息询问当前状态
        - 此时 B 不能决定中止，因为更新可能已经被提交
    - B 被阻塞，直到 A 重启并响应
    - **等待 GLOBAL-COMMIT 的参与结点不会因超时而中止，必须一直等**

Storage and File Systems in Modern Computer Systems

- I/O 设备：disk with dedup
    - Dedup
- I/O：端到端管理
    - IOFlow
- 现代文件系统
    - GFS
- RAID 和 erasure coding
    - EC-Cache
- 分布式应用文件系统
    - Chord

Dedup

Deduplication：**在全局文件系统中消除重复数据（全局压缩技术）**

- `lab3-designdoc-v1.md` 和 `lab3-designdoc-v2.md` 中的重复部分只存一份
- 可以达到远超传统的小窗口压缩的压缩率
- 例子：数据备份
    - 备份的模式：全量备份、增量备份
    - 很多冗余数据块：冗余数据块只存指针

dedup 过程：

- **数据流分割为数据块，每个数据块有一个唯一的指纹，作为索引**
- **每次存储时在文件系统中查找指纹是否存在，若存在，只存指纹，否则存储数据块**

高速、高压缩率、低硬件成本：

- **Summary vector**
    - 用 bloom filter（零一向量）总结什么数据块已经被存储
    - 设数据块的指纹为 x
    - 插入时计算 $h_1(x), h_2(x), h_3(x)$，将 bloom filter 中的 $h_1(x), h_2(x), h_3(x)$ 位置置为 1

    - 查询时检查 bloom filter 中的 $h_1(x), h_2(x), h_3(x)$ 位置是否全为 1，
        - 若全为 1，则数据块可能存在（false positive）
        - 若有 0，则数据块一定不存在（不会有false negative）
- Stream informed segmemnt layout
    - 利用 duplicate locality，将来自同一个流的数据块以及元数据（索引数据）放在同一个 container 中
- Locality preserved caching (LPC)
    - 在缓存中维持 duplicate locality
    - 磁盘索引保存 $\langle \text{指纹}, \text{container ID} \rangle$ 对，它们被缓存在 index cache 中
    - 缓存替换时，在 disk index 里找到对应的 container，将 container 的所有元数据加载进 index cache
    - 也就是说，以 container 为单位进行缓存替换
- 全过程
    - 指纹先在 index cache 中查找。若找到，则说明已存在
    - 若未找到，在 summary vector 中查找。若显示不存在，则说明未存储
    - 若显示存在，则说明可能存在。继续在 disk index 中查找，并加载对应的 container metadata 到 index cache

IOFlow

企业数据中心：

- 通用目的应用，运行在虚拟机上
- VM-to-VM 通信网络和 VM-to-Storage 通信网络分离
- 虚拟化的存储
- 共享的资源

动机：可预测的应用行为和性能

- **端到端 SLA**：
    - 可保证存储带宽
    - 可保证高 IOPS 和优先级
    - 关于 I/O 路径的分应用决策控制
- **当前系统 I/O 路径有太多层，难以配置**
    - 没有 storage control plane

IOFlow：

- **解耦数据平面（enforcement）和控制平面（policy logic）**
- 贡献
    - 定义并构建了存储控制平面：**中心化的控制器**
    - **控制平面和数据平面之间的 API（IOFlow API）**
    - **控制器通过 IOFlow API 控制数据平面中的可控制队列，从而满足性能 SLA 和非性能 SLA（如 bypassing malware scanner）**
- Storage traffic 中没有通用的 I/O 头：控制器 flow name resolution
- 拥塞控制中的 rate limiting：
    - token bucket 不 work：读请求比写请求小很多，后者包含要写的数据。带宽都被读请求占了
    - 按读写的实际大小，也不 work：写比读慢很多。带宽都被写请求占了
    - 按 IOPS，还是不 work：读写的实际大小可能差别很大
    - 建立一个 cost model，分配给每个队列
- 数据平面中的可编程队列：
    - 分类：I/O 头 -> 队列
    - 队列服务
    - 路由
- 分布式、动态的 enforcement：
    - SLA 需要 per-VM enforcement
    - Max-min fair sharing：已有的是分布式的，本文提出基于中心化控制器的版本

The Google File System (GFS)

动机：

- 结点 failure 频繁发生（服务器挂掉）
- 文件数 GB
- 多数文件修改以追加为主，随机写和覆写少
- 高持续带宽比低延迟更重要

典型负载：

- 读
  - 大的流式读，一般连续
  - 小的随机读
- 大的顺序追加写
- 100 个左右客户同时追加同一个文件

接口：

- 非 POSIX，但支持 `create`、`delete`、`open`、`close`、`read`、`write`
- `snapshot`：低成本创建文件或目录的拷贝
- `record append`：并发的追加写，至少第一个写保证是原子的

架构：

- **数据流和控制流解耦**
  - 客户与 master 交互，获取文件的元数据
  - 客户的文件操作直接与 chunkservers 交互
  - 根据网络拓扑调度昂贵数据流，以优化性能
- **Master node**
  - 负责系统级操作：管理 chunk leases、回收存储空间、负载均衡
  - 维护文件系统元数据：全部在内存中，命名空间和 file-to-chunk mapping 持久地存储在 **operation log** 中
    - 文件名到 chunk 的映射
    - 每个 chunk 的位置
    - 命名空间
  - 通过 `HeartBeat` 信息周期性地与每个 chunkserver 通信，决定 chunk 位置，评估系统状态
- Operation Log
  - 元数据的唯一持久化记录
  - 也作为并发操作的序列化的逻辑时间线
  - Master 可以通过重放 operation log 恢复状态
  - Master 周期性 checkpoint

为什么 single master?

- Master 拥有全局信息，简化设计
- Master 一般不是性能瓶颈
  - Master 只处理元数据请求，客户端文件操作直接与 chunkserver 通信
- Master 通过 operation log 和 checkpoint 备份在多台机器
  - Shadow masters 可以提供文件系统的只读访问，这样在主 master 挂掉时，仍然可以访问文件系统

Chunks 和 Chunkservers：

- **文件被分割为固定大小 chunk**，每个 chunk 有一个不可变、全局唯一的 64 位 chunk handle
- Chunkservers 将 chunk 以 Linux 文件形式存储在本地磁盘
  - 元数据存储在 master 中，包括当前备份位置、引用计数（copy-on-write）、版本号
- **Chunk size：64 MB（比大多数文件系统大很多）**
  - 缺点：**内部碎片化严重、并发的小文件操作占用较多流量**
  - 优点：
    - **减少与 master 的交互开销**
    - **维持一个持久的 TCP 连接，减少网络开销**
    - **减少元数据大小，可以完全存储在内存中**

当 master 节点收到对某个 chunk 的修改操作时：

- Master 寻找持有该 chunk 的所有 chunkserver，并授权其中一个“租约”(lease)
  - 被授权的服务器称为 primary (主副本)，其他持有相同 chunk 的服务器则被称为 secondary (次副本)
  - Primary 负责确定该 chunk 所有修改操作的序列化顺序，所有 secondary 都必须遵循此顺序
  - 租约到期后 (约60秒)，master 可以将该 chunk 的 primary 身份授予另一个 chunkserver
- Master 有时也可以撤销租约 (例如，当文件正在重命名时，为了禁止修改操作)
- 只要该 chunk 持续被修改，primary 就可以无限期地请求延长租约
- 如果 master 与 primary 失去联系，也没关系：只需在旧租约到期后，授予一个新的租约即可

客户端写入流程

1. 客户端向 master 请求持有该 chunk 的所有 chunkserver (包括所有的 secondary)。

2. Master 授予一个新的租约，增加 chunk 的版本号，并通知所有副本 (replica) 也同样增加版本号。然后 master 将这些信息回复给客户端。此后，客户端在本次写入中不再需要与 master 通信。
3. 客户端将要写入的数据推送给所有的副本服务器 (不一定先推给 primary)。
4. 一旦所有副本都确认收到了数据，客户端就向 primary 发送写入请求。Primary 负责决定所有并发修改的序列化顺序，并将这些修改应用到其本地的 chunk 上。
5. Primary 完成修改后，将写入请求和序列化顺序转发给所有的 secondary，这样它们就可以按照完全相同的顺序应用这些修改。（如果 primary 在此过程中失败，这一步将不会发生。）
6. 所有的 secondary 在完成修改后，会向 primary 回复确认消息。
7. Primary 最终向客户端回复成功或错误信息。

- 如果写入在 primary 上成功，但在任何一个 secondary 上失败，此时副本间就出现了不一致的状态 -> 错误会被返回给客户端。
- 客户端可以重试第 (3) 步到第 (7) 步。
- 注意：如果一次写入操作跨越了 chunk 的边界，GFS 会将其拆分成多个独立的写入操作。

EC-Cache

数据密集型集群依靠分布式、in-memory 缓存来提高性能

集群中的不平衡导致**负载不均衡和高的读延迟**：**内存中数据单份存储不足以满足高性能要求**

- 数据流行度不均（热数据和冷数据）
- 后台网络不均衡
- 失败/不可用

流行方法：selective replication

- 将热数据备份到更多的节点上
- Erasure coding 可以在**不增加内存开销的前提下，提高读的性能，使负载更均衡**

回顾：erasure coding

- 将数据分为 k 个数据块，生成 r 个校验块
- 任意 k 个块都可以恢复整份数据

EC-Cache 鸟瞰：

- 写
  - 数据分为 k 个数据单元，encode 生成 r 个校验单元
  - **将 $k+r$ 个单元缓存到不同服务器上（均匀随机分布）**
- 读
  - **均匀随机地读 $k+\delta$ 个单元（additional reads）**
  - **使用前 k 个到达的单元 decode 并合并出数据**
- 优点：
  - 对内存的细粒度控制：Selective replication 以整份数据为单位，EC-Cache 允许更细粒度
  - 数据被分割，优化负载均衡
    - Selective replication 的负载方差是 EC-Cache 的 k 倍
  - 数据被分割，中位延迟降低，但尾延迟提高。Additional reads 可以降低尾延迟

和存储系统中的 erasure coding 在设计考量上的区别：

- 目的：
  - 存储系统：空间高效的 fault tolerance
  - EC-Cache：**降低读延迟、负载均衡**
- Erasure code 选择：
  - 存储系统：高效的存储和重建，不一定满足“any k out of $k+r$”性质
  - EC-Cache：内存 cache 中不需要重建，要求“any k out of $k+r$”性质
- 数据分割方法：
  - 存储系统：可以在对象之间，也可以在对象内部
  - EC-Cache：数据分割在对象内部

Chord

问题：**在分布式文件共享系统中，如何定位数据**

- 中心化：单点故障、需要存大量索引信息
- Naive 分布式：Flooding（我问你，你问他），开销大、延迟高

- Chord：**routed messages**

路由的挑战：

- 定义 key nearness metric
- 保持跳数较少
- 保持路由表大小合适
- 面对剧烈变化健壮

Chord：

- P2P 哈希表查找服务：键为 IP 地址
  - Chord 不存储数据
- 性质：
  - 查找发送 $O(\log N)$ 条消息
    - N 为服务器数量
  - Scalable：每个结点保存 $O(\log N)$ 个路由表项
  - 健壮（假设没有恶意结点）
- Chord ID
  - SHA-1 哈希函数
  - 键 ID：键
  - 结点 ID：IP 地址
  - 均匀分布
- Consistent Hashing：键被存储在其 ID 大于等于该键的第一个结点上
  - 例如，键 ID 为 k，结点 ID 为 n，则存储在第一个满足 $n \geq k$ 的结点上
  - 所有结点被组织为环，最大结点 ID 的下一个结点是最小结点 ID 的结点
- 查找：
  - 如果所有结点都保存全局信息，则查找只需 $O(1)$ 时间，但路由表空间开销为 $O(N)$
  - 如果所有结点只保存下一个结点信息，则查找需要 $O(N)$ 时间，但路由表空间开销为 $O(1)$
  - Finger tables：
    - **结点 n 保存结点 $n+2^i$ 的信息**，其中 $i=0,1,...,m-1$，m 是 ID 的位数
      - 它们称为结点的 fingers
    - **时间开销和空间开销均为 $O(\log N)$**
- Joining the Ring
  - 初始化新结点的 finger table
    - 若新结点 ID 为 36，则它的 finger table 包含结点 $37, 38, 40, ...$
    - 更新新结点的 finger table 表项
    - 将下一个结点中 ID 小于新结点 ID 的键传送到新结点