

目录

| | |
|-----------------------------------|----|
| 2021/3/2..... | 2 |
| 上下文切换..... | 3 |
| 进入 Kernel 的方式: Syscall..... | 5 |
| Fork: 创建子进程..... | 5 |
| 2021/3/9..... | 6 |
| 回收子进程..... | 6 |
| 怎么实现一个 shell? | 8 |
| 2021/3/16..... | 9 |
| Signal..... | 9 |
| 硬件触发和软件触发的 Signal..... | 10 |
| Signal Handler..... | 10 |
| 阻塞 Signal..... | 10 |
| 进程组 (Process Groups) | 11 |
| 注册 Singal Handler..... | 12 |
| 2021/3/23..... | 15 |
| 写 Signal Handler 的 Guideline..... | 15 |
| 一些 Concurrency Bug..... | 17 |
| Spin Loop..... | 18 |
| 2021/3/30..... | 19 |
| Non-local Jump..... | 19 |
| System-level I/O..... | 20 |
| OS 支持文件和 IO 的形式..... | 21 |
| 2021/4/6..... | 24 |
| Robust I/O..... | 24 |
| Standard I/O..... | 25 |
| 网络..... | 27 |
| 2021/4/13..... | 27 |
| Hub、Bridge 和 Router..... | 27 |
| TCP/IP..... | 28 |
| Berkeley socket interface..... | 30 |
| 2021/4/20..... | 32 |
| 实现一个 Echo Server..... | 32 |
| 2021/4/25..... | 35 |
| http 协议..... | 35 |
| Tiny Web Server..... | 38 |
| Proxy..... | 41 |
| 第十二章：并发编程..... | 42 |
| 2021/4/27..... | 42 |
| 应用层实现并发的方法..... | 42 |
| 1. 进程..... | 43 |
| 2. 线程..... | 44 |
| 3. I/O 多路复用..... | 47 |

| | |
|--|----|
| 2021/5/11..... | 49 |
| 2021/5/18..... | 49 |
| 生产者和消费者..... | 50 |
| 多线程求和..... | 52 |
| 2021/5/25..... | 54 |
| 可重入函数（Reentrant Functions）..... | 55 |
| TLS（线程局部存储）..... | 56 |
| 线程竞争问题..... | 57 |
| 死锁问题..... | 58 |
| 锁..... | 59 |
| 基于 TestAndSet 来实现 Lock..... | 60 |
| 2021/6/1..... | 61 |
| 主动放弃 CPU 占用（Yield）..... | 61 |
| Wakeup/Waiting Race..... | 63 |
| 条件变量..... | 64 |
| 2021/6/8..... | 67 |
| 进程调度（process scheduling）..... | 68 |
| FIFO（先来先服务）..... | 69 |
| Shortest Job First..... | 70 |
| Shortest Time-to-Completion First..... | 71 |
| 轮询..... | 72 |
| IO 和 CPU 协作..... | 73 |
| MLFQ 策略..... | 73 |

2021/3/2

如何进行异步异常处理？

用一个特殊的硬件 interrupt controller，告诉 CPU 换一个指令流执行，类似于一个很远的 jump。

exception 发生了以后，去执行 kernel 中的代码。切换到 interrupt 对应的 exception handler 的代码。只是和当前程序无关，是被动的打断。因为无关，所以处理完了要回到当前程序的下一条指令的去执行。

异步异常包括了以下例子

1.I/O 打断

比如说在命令行中按下了 **ctrl+C**；网络包来了；硬盘扇区数据到了等。

2.硬重置打断（硬件关电源）

3.软重置打断（**ctrl+alt+delete**）

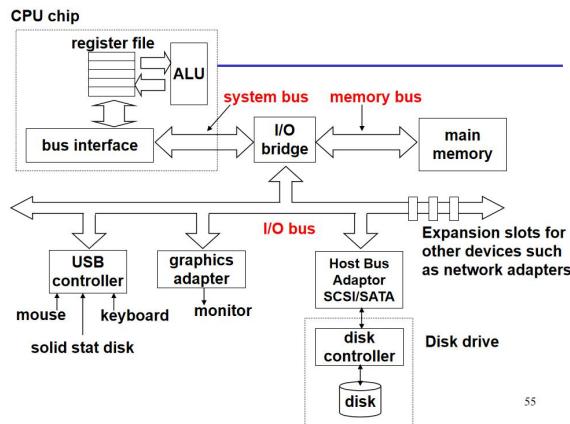
要支持程序的切换，比如说执行到一个地方做不下去了，需要等待一个信息，所以要有切换和唤醒的机制。

interrupt controller 有一根引脚连到 CPU，来通知 CPU 执行打断操作。

总线把设备连在一起。软硬件协同：硬件和软件一块去完成一件事情、上层语义（软件）来实现具体的功能（比如 **page fault** 如何实现复杂的逻辑，不能靠事先烧在硬件中），硬件只负责打断。

这是典型 IntelCpu 中的架构

BUS 分成几条，因为需求有不一样。因为连在 bus 上的数据的传输速度是不一样的。



55

显然 **system bus** 会非常快，功能基本上都是把硬件连在一块。

外设和 CPU 如何通信？

CPU 直接能访问的只有内存。如何访问设备？

外设有 **I/O port**，把 **I/O** 抽象为 **port** 可以认为是发命令的途径，可以把命令通过 **port** 来发给外部设备，称为 **in/out** 指令，都是 **driver** 来使用，而不是自己写程序来执行。

I/O port 可以映射到内存中，我们所面对的内层是经过操作系统抽象的 **virtual memory** 的内存。所有可以映射到 **I/O devices** 上。然后对外设的操作就变成了内存访问。

每个设备会占用不同数量的 **port**，来申请对应的资源。

其中有一些和 **interrupt** 有关，磁盘比较慢读数据到内存里面。

CPU 可以发起指令让磁盘数据到内存中，CPU 不会傻等，需要有异步机制用之前的 **interrupt** 机制。

1. CPU 首先向磁盘发命令，告诉磁盘要读哪块的数据以及读到内存的哪里。
2. 磁盘可以直接往内存里写数据（比较慢），此时 CPU 不需要参与这个步骤。此时可以专门设计一个 **DMA(direct memory access)** 直接来回读写，CPU 就可以不用管这件事情。把任务交给 **DMA**，CPU 干其他事情。**DMA** 完成 **copy** 任务后，可以通过 **interrupt** 的方式告诉 CPU 这件事情完成了。CPU 有针对这个 **interrupt** 的处理预案，可以从 **copy** 完成的内存地址去读数据了。

凡是异步的情况，都会返回到被打断的进程的下一条指令。

Interrupt 是硬件提供的一种支持。之后所有程序的 **process** 都是基于这个功能的。

上下文切换

Process 是 CPU 上执行程序的一个单位，每个 **process** 自己干自己的事情，但是共享 CPU。

所以我们要介绍 context switch，完成对 CPU 的分时共享（并发）。

实际的行为是 shell 会为你创建一个 process 来为你运行一个程序，加载到对应的内存然后运行。

Process 之间不同点在于 context 不同。Context 是来作为 process 的一个当前的状态，包括占用的内存，等资源。

首先要创建一个进程，可以运行自己的程序也可以运行其他的程序。

Process 比较低级，只能通过 fork 的形式复制一个自己，但是每个进程的代码都是要写在里面的。Fork 情况比较快，copy 自己就可以了。创建别的程序的进程就还要加载内层等。我们要先搞清楚每个进程如何使用 CPU，CPU 归你这个进程的时候，就是接受指令流来执行。

最早开发的 CPU 就是实现很简单，程序塞进去一个人用就可以了。当有多个进程的时候，有多个执行指令流，只能把不同进程的指令流 merge 在一块变成一根指令流，然后交给 CPU 去做。就可以让一个 CPU 去跑一百个进程。操作系统提供的 merge 抽象很重要，但是这是一个动态的过程，牵涉到一个 context switch。

在不同进程之间切换就代表，进程都处于 active 状态，称为并发进程（concurrency）。一个相似的名字是并行（parallel），指几个程序同时执行。并发是指生命周期重叠（overlap）。并发不一定是并行的。

Context switch 不需要切换的足够快，还有内存的条件，不同进程使用不同块的内存。上下文切换必须要有操作系统来完成，要从 user 切换到 kernel 来使用，kernel 的代码是一样的，再回到另一个程序的用户态，然后回到 kernel 里上下文切换的指令，然后回到 user 态。也就是恢复也是一层层往上恢复的。

上下文包含了 code 和 data（内存中），记录执行到哪，需要有 PC，状态寄存器，user 和 kernel 的 stack，记下这个进程中的 kernel 对应的状态。

Kernel 的代码是被所有进程共享的，kernel 中的数据是每个进程 specific 所特有的，存储了一些 context 中的状态。

上下文切换中，真正要 copy 的东西是很少的，内存中的东西都不会被修改，所以只要记住在哪里就可以了。一个进程切换到底切换给谁呢，都要记录在 process table 里面，。

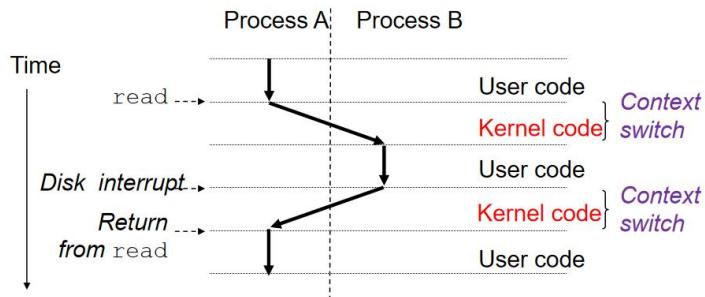
把会被擦掉的主要信息来保存起来。支持多任务，多任务不是为了提升效率的。而是要让一台机器完成多个任务，让用户体验能上去，对于每个程序来说，效率是变慢的，从整个系统的角度来说，性能也是下降的，因为要做上下文切换的指令。100 条里可能 20 条是为了切换的。

一个进程获得 CPU 的连续时间，称为一个时间片。时间片到了就要切换到下一个进程，时间片就是一个切换的阶段。时间片可以手动设置，时间片越小，CPU 消耗在切换上的指令就会多。服务器时间片会比较大。时间片小意味着用户体验会更好。所以有 timer 的硬件，定时给 CPU 发中断信号。

Signal（进程之间的通信）也是基于这个的，JAVA 的 exception 也是基于这种底层实现的。

时间片切换是被动切换，block 是主动切换（eg. 程序中有 read 或 sleep）。

Context switching



读完磁盘后, DMA 会来一个 interrupt 告诉 CPU 磁盘读完了, 经过调度评估后, 会回到 userA。
A->Kernel->B->Kernel->A

上下文切换有一个 scheduler, 其决定下面执行谁。决定是不是剥夺当前的进程, 然后执行 save 和 restore 的逻辑。Scheduler 是一个代码的集合。其对 process 是透明的, 只要保证回来后能继续执行就行了。

上下文切换的代码本身是 overhead, 需要尽可能快。

一个进程的三种状态:

Running (可以被执行, 但不一定在执行)

Stopped (blocked, 没执行完, 但是这时 cpu 给它也执行不下去, 因为在等网络包回来或者其他耗时间的行为。)

Terminated (已经执行完了, 但是还未被回收。一个进程被回收是一个过程, 不是一结束就直接回收的。比如 zombie 状态, 虽然 terminated, 但是其遗留的信息比较重要, 比如 main 的返回值)

最主要的切换在 running 和 stopped 之间, 比如主动进 (read), ctrl+d (debugger 把一个程序停在那边, 进程受到了一个 stopped 消息)

Signal 信号: 比如 ctrl+d 发出了一个 SIGSTOP 让程序停下来。

从 stopped 回到 running 状态, 比如受到包的时候会向收到包的进程发一个 SIGCONT

进入 Kernel 的方式: Syscall

用户主动进入 kernel 的方式只有 syscall, syscall 返回值是一个统一的整形返回值 (进行了一些编码, 负数代表出错, 会把出错信息放在 errno 里)

任何 syscall 在调用的时候要考虑到是会出错的, 比如说 malloc 是一个 syscall, 但可能分配失败。Syscall 都会被打断, 要检查 syscall 的返回值, 要有对应的出错处理。

书中大写的都是包装过的 syscall, 做过检查的。出错时要用全局变量 errno 打印错误信息

每个进程都会有一个 ID 号 (进程的唯一标识), PID (大于 0)。Getppid (得到父进程的 pid)

Fork: 创建子进程

Fork 是创建子进程的方式。是创建一个和自己完全一样的进程出来, 执行一样的代码。

几乎完全一样，就差了 `pid` 不一样。需要在一份代码里分出两块来，一块是子进程执行，一块是父进程执行，代码在两个进程中执行不同的路径。`Code` 和数据都有两份，初始值都一样。

`Fork` 的特殊行为，调用一次，返回两次。因为调用前是一个进程，调用完是两个进程。通过返回值（父进程返回子进程 ID（大于 0），子进程返回 0）可以来判断出父进程和子进程。然后两个进程可以进行协作了。两行打印的先后是不确定的，因为成为了两个进程。要看操作系统的调度策略。

2021/3/9

命令行执行的程序是控制台 `fork` 产生的，不是创建一个空的进程再把程序填进去，因为这样效率很低。

为了服务更多的 `client`, `webserver` 会创建很多进程，所以可以用多个进程来完成一个目的。所以 `fork` 接口的设计，也是为了服务这种通用的目的来实现的。创建一个和主进程一样的进程，效率会比较高。而如果创建一个空的进程，效率反而会低。`fork` 的返回点就是子进程执行的第一条指令，区别只有返回值不同。两个进程的虚拟地址是一样的，数据，堆，栈，完全相同。通过文件传递信息可以实现父子进程之间的交互。`fork` 进程的先后顺序是不确定的。文件标识符是可以共享的，而其他的东西都是复制过在分别的内存空间里。

回收子进程

首先介绍 `reaping child process`, 然后再学习 `console shell` 控制台的行为是怎么样的。

回收子进程：换句话说自己是不能把自己回收的，进程的回收默认情况下是由父进程来回收的。进程有三种状态（`1.running`, `2.stop` 不会被调度因为它在等一些事情的发生，比如等网络包发生，或者被 `gdb` 停住，可以回到 `running` 状态。`3.terminate` 状态，程序已经结束了，比如在 `main` 函数结束了，但是并没有被马上回收，因为有些进程还对这个进程的遗言比较好奇，所以系统不会马上回收这个资源。）所以子进程如何把消息传递给父进程呢？所以进程在退出后需要主动地回收。所以 `terminate` 后就进入了 `zombie` 状态，等待父进程回收。首先需要有特定的 `syscall`, 在这个过程中可以检查子进程的状态，比如返回值。系统保护的默认机制。如果父进程不回收子进程，系统有 `init` 函数。如果有子进程的父进程退出了，其子进程会转化为 `init` 的子进程来进行回收。`init` 的 `pid` 是 1。kernel 创建了 `init` 进程，然后切换到 `init` 进行就切换到了用户态。

比如守护进程或者一些服务，需要运行很长时间，这些叫做 `long-running` 进程。对于这种进程，如果不回收子进程，又在 `create` 子进程，那么子进程就会越来越多，子进程打开的空间和内存就不会被回收。最终内存就会受到影响。每个系统进程和打开的文件都有上线，所以要及时回收子进程。由父进程调用 `wait` 或者 `waitpid` 来进行回收子进程。返回值是被回收掉的子进程的 `pid`。一次调用只会回收一个处于 `zombie` 子进程。`waitpid` 可能会停在那边，因为子进程可能还没死。确定性回收，使用 `waitpid`（明确的指定子进程 `pid` 号，如果 `pid=-1`, 就是回收随便一个死掉的子进程，如果还没有人挂，那么也会等。）等到你挂了我就回收你。

如果不存在子进程，返回-1 并设置 `errno` 为 `ECHILD`。如果主进程收到了一个 `signal` 中断，返回-1，设置 `errno` 为 `EINTR`。`option` 默认为 0，从指定的子进程去等，然后回收退出。

父进程要干自己的事情，不能让 `waitpid` 一直等待，所以提供了 `WNOHANG` 看一下，没有就干自己的事情。

`status` 是用来带回返回值的状态，同时通过了编码返回值更为复杂。需要用 `wait.h` 中定义的一些宏来检查这个 `status`

`status` 提供了一些 `WIFSIGNALED` 和 `WIFSTOPPED` 方法，来进一步得到程序的死因是什么。

`waitpid` 是随机一个 `zombie` 子进程回收，不一定是先死的那个，或者是 `pid` 小的那一个。

这种情况是因为并发造成的不确定性。

接下来简单的函数

`sleep` 和 `pause` 都会让进程进入 `stop` 状态，

可以被 `signal` 打断出来

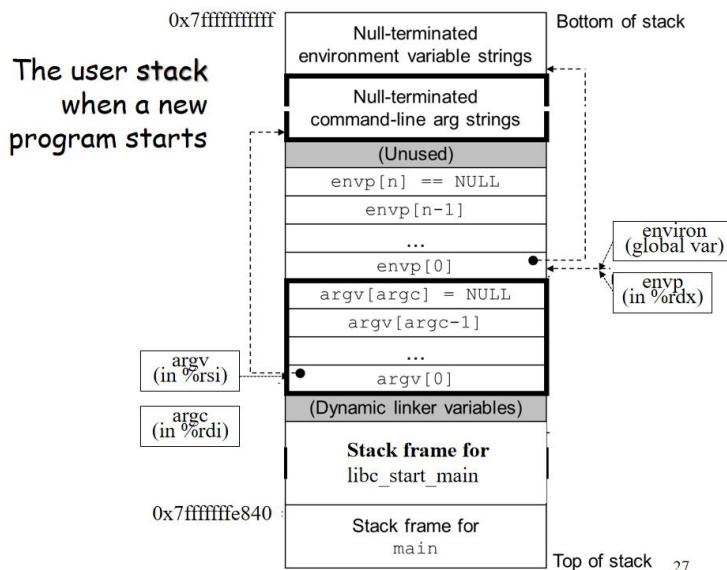
`execve` 第一个参数:可执行文件的文件名，第二个参数:程序的参数，第三个参数: 环境变量
(属于系统的全局环境变量)

这个函数会把当前程序的栈什么的都擦掉，把读入的程序状态 `copy` 进去

所以这个操作是 `load and run`

这个函数，如果不出错的情况下，是不会返回的。（一次调用，返回零次）

但是也有可能发生异常会退出，比如文件不可读，文件找不到之类的。



`Main` 函数没有提供 `environment` 的参数，很少情况下会使用环境变量。

我们可以通过键值对来指定一个环境变量。

```

#include <unistd.h>
char *getenv(const char *name);
>Returns: ptr to name if exists, NULL if no match.

int setenv(const char *name, const char *newvalue,
           int overwrite);
>Returns: 0 on success, -1 on error.

void unsetenv(const char *name);
>Returns: nothing.

```

怎么实现一个 shell?

当前大家使用的 shell 比较多，比如有 bash, cshell 等，都是在标准 shell 上增加新的功能。Shell 的主要功能就是来运行程序的。最简单的 shell 要做两件事情，1.从用户那里读命令 2.读一行命令，解析命令并执行。

所以要做字符串的解析，`parseline`，以及 `builtin_command`（内置的命令，一些关键字比命令本身更优先）

Shell 的主题就是一个循环，从用户态执行一行

```

9  int main()
10 {
11     char cmdline[MAXLINE]; /* command line */
12
13     while (1) {
14         /* read */
15         printf("> ");
16         Fgets(cmdline, MAXLINE, stdin);
17         if (feof(stdin))
18             exit(0);
19
20         /* evaluate */
21         eval(cmdline);
22     }
23 }

```

`feof` 是解决文件输入的情况。

我们关心的更多是 `eval` 部分（解析+执行）

```

/* eval - evaluate a command line */
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* argv for execve() */
    char buf[MAXLINE]; /* holds modified cmd line */
    int bg; /* should the job run in bg or fg? */
    pid_t pid; /* process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return; /* ignore empty lines */
}

```

也就是把 xxxx xxxx xxxx xxxx 按照空格分开，放到 `argument` 里去。换句话说，`argument` 是一个指针数组，把其指向对应的参数。

`Parseline` 大致就是把参数之间的空格替换为结束符。

```

/* parse the cmd line and build the argv array */
int parseline(const char *cmdline, char **argv)
{
    char *delim; /* first space delimiter */
    int argc; /* number of args */
    int bg; /* background job? */

    buf[strlen(buf)-1] = ' ';
    /* replace trailing '\n' with space */
    while ((*buf && (*buf == ' ')) /* ignore spaces */
           buf++);

```

把最后一个替换为空格

2021/3/16

运行 shell 有多种模式

一般是前台执行，等待程序执行完毕后，再回到 shell 里面，此时可以看到程序的运行结果的输出情况。不会输出多个程序同时往屏幕上打印。（同步的形式）

后台运行：在命令中最后加上&符号，和前台并发执行，生命周期会重叠。父进程会直接回到 console 执行下一行命令。

Shell 创建一个子进程，运行我们提供的二进制代码。前台运行，shell 等待子进程结束（`waitpid`），如果是后台运行就不需要等待。

Signal

如何实现 `ctrl+c` 功能终止子进程呢？

这节课介绍 `signal`，我们在讲中断的时候讲到了它是一个底层的机制。打断 `cpu` 的执行流，可以分为同步的还是异步的，有关的还是无关的，等。上层有很多基于下层硬件提供的中断机制实现的功能，都是改变程序的执行流，打断了当前的程序以后去执行其他的。比如 `java` 抛出 `exception`，程序会跳转到 `exception handler` 去执行。写一个正确的 `signal` 是很困难的。在系统中增加了 `signal` 机制会导致程序很复杂，但是能够改变执行流，这点很重要。所以现在通用计算机都采用了 `signal` 机制，单片机等小型机可能就没有这一套机制。我们主要关注的还是实现的机制，实现代码的时候需要注意什么，应用场景有很多，不在课程中介绍。

`Signal` 最早出现在 `unix` 系统，这是一个软件层去实现 `exception` 的方法。硬件上实现是通过引脚实现。`Signal` 允许一个进程去中断另一个进程，只有 `kernel` 能够操作这件事。一个进程想打断另一个进程或者给另一个进程发消息，会通过 `kernel` 以 `signal` 的形式去做。所以 `signal` 永远是 `kernel` 向进程发送消息的一种形式。我们知道一个进程去 `kernel` 需要主动地通过 `syscall`，而从 `kernel` 向进程发消息就是通过 `signal`，其原因有很多种比如是其他进程想发消息。

Shell 中命令 `Kill -9 15213` 的意思是向进程 15213 发送 9 号 signal (`SIGKILL`)
这个命令是发送 `signal` 后让进程自己去自杀掉。

硬件触发和软件触发的 Signal

Signal 分为硬件触发和软件触发。Signal 其实就是一个消息、一个数字，并不是去做什么操作，只是发了一个信号过去，至于做什么，都是由那边的进程自己完成的。这个数字不同就代表了不同的事件。硬件事件，有除零（SIGFPE）：CPU 发现不能除零，中断给操作系统，给执行这个语句的程序发送了除零操作；执行非法指令（SIGILL）；内存地址错误（SIGSEGV），也一样发现处理不好了以后发送信号给程序，然后在屏幕上打印 SIGSEGV。

软件事件，有 `ctrl+c`（SIGINT）把程序中断掉；SIGKILL 杀掉另一个进程；SIGCHLD 当一个子进程死亡或者停止了后，会向父进程发送 SIGCHILD。

发送 signal 分为硬件发送（黑盒，不需要管）和软件发送（向任意进程发 signal）

Alarm 机制，固定时间后向自己发送 signal。当然是有权限的一些问题。发送消息给一个进程，这个进程不一定会运行，所以更像是寄了一封信。或者每个 signal 维护了一个 bitmap，发送了只是置为了 1，需要接收的进程去主动检查。

所以这是一个异步的操作，什么时候被处理都是对面决定的。

`kill (pid_t pid, int sig)` 函数，出错可能是因为没有权限，进程不存在等。

Signal Handler

发生 context switch 的时候，在执行主干代码之前，会去检查当前有没有对应的 signal 去处理。也就是说正在执行自己的代码的时候，发过去会收到，但是不会处理。受到了 signal 以后会有默认行为。

默认行为分为三种处理情况

1. 直接无视，`sigchild` 必须要主动去修改槽函数。
2. 程序结束，进入 `terminate` 状态，除了终止之前打印的话不一样
3. 抓住 signal，可以写对应的 signal handler，收到了 message 后就去执行对应的 function。
注册不同的自定义 signal handler。

在 shell 中 `ctrl+c` 先向 shell 发送 `terminate` 信号，然后 shell 注册了转发给前台程序的信号处理机制。

Pending 信号：收到了，但是还没有进行处理。只能 pending 一个信号。每个 signal 对应了 0 和 1，每收到一个都会直接置 1. 可能已经收到了三四个，但因为都是置一，可能有一些 message 在 pending 过程中被丢掉了。所以不是收到了一个 message，而是有 message。

检查 Bitmap 的时候，发现可能收到了很多个 signal，会依次执行多个 signal 处理。

阻塞 Signal

同样还有一种 blocking，我当前在做一些很重要的事情，不想被打扰，此时可以关闭

消息的接收。如果设置了 block, message 不会消失, 只是不去执行它。不过 block 不是全局的, 是对于每个 signal 的 01, 所以也是个 bitmap。可以关闭一部分 signal 的接受。Message 依旧会被投递, 只是不调用处理函数。所以当 block 恢复的时候, 依旧会处理积压的 signal。

跟 signal 相关的 kernel 里会维护 pending bitvector 和 block bitvector。当有人发送了一个 signal, 就会设置 pending 的一个位。调用 signal handler 后, bit 位会被清理掉。

我们具体地看一下 sending 和 receive

进程组 (Process Groups)

我们先要引入 process groups, 在命令行执行的时候, 一般是执行一个二进制程序然后加上一个参数。但是一个复杂的命令可能会包含多个二进制程序, 比如前一个程序是后一个程序的输入, 运行了多个子进程, 其间还有信息的传递。一条命令其实就是一个前台的任务, 一个任务可能有多个进程。当运行一条命令就创建了一个进程组, 通常情况下只有一个进程, 做复杂操作的时候, 就会有多个进程。有一个进程组 ID, 通常是第一个小的进程的 ID。SIGNAL 是以进程组为对象发送的, 可以发给一个组或者组里的特定的一个进程。Setpgid 可以把某个进程设置到某个进程组里面。

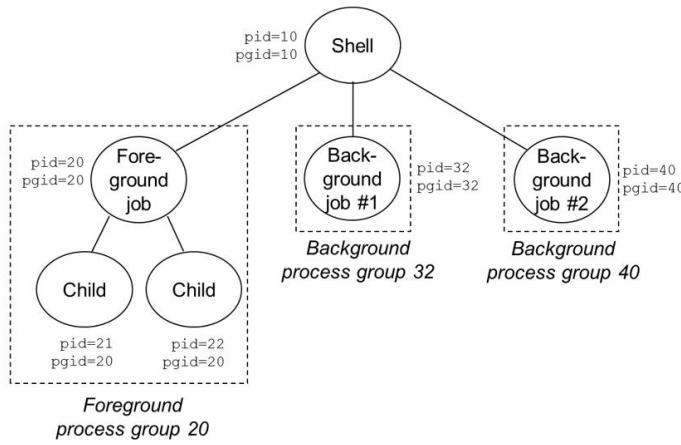
对于一个复杂命令创建多个进程得到 pid, 然后设置其在一个进程组中。

Eg: setpgid (0,0) 把自己设置进自己的进程组号里面。

创建前台 job 组, 其代表进程。

Pipe 是进程之间通讯的方法, 是一种 syscall。

前台只有一个任务 (job) 或者称为前台只有一个进程组。



25

再回顾一下, 按了 ctrl+c 以后, 硬件中断发送 sigint 发送给 shell, shell 转发信号, 找到前台的任务对每一个进程发送相同的消息。

Ctrl+z 是让进程 stop

如果 kill 参数小于 0, 是向进程组发送。进程号的获取需要通过一些额外的机制, 比如需要注册, 或者通过 killall 函数等。

Pause () 就是停在那里等待一个消息被打断。

```

1 #include "csapp.h"
2
3 int main()
4 {
5     pid_t pid;
6
7     /* child sleeps until SIGKILL signal received */
8     if ((pid = Fork()) == 0) {
9         Pause(); /* wait for a signal to arrive */
10        printf("control should never reach here!\n");
11        exit(0);
12    }
13
14    /* parent sends a SIGKILL signal to a child */
15    Kill(pid, SIGKILL);
16    exit(0);
17 }

```

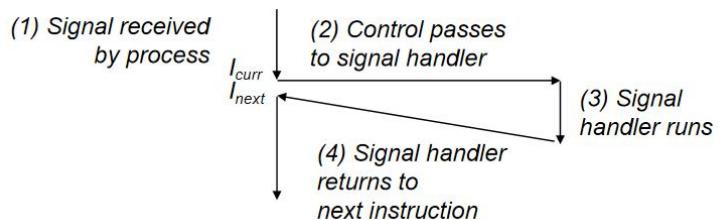
子进程先被调用：Pause 后收到一个 sigkill 直接退出，就不会

父进程先被调用：子进程已经被创建了，message 依旧会发送成功。子进程执行 pause 之前会检查 bitmap，然后发现接收到了 sigkill 直接退出。

系统里面两个信号是不能被修改的，其中包括 SIGKILL。

Alarm 调用 SIGALARM，自己给自己设置一个闹钟发送，过一段时间就会收到一个 SIGALARM，需要自己设置信号槽函数。整个系统里面只有一个 alarm，新设置的闹钟会把之前设置的还没响的闹钟覆盖掉。

Context switch 到你这个程序之前，查看 signal，如果有就回到用户态执行 signal 的代码。



并发执行流。

Signal handler 如果代码很长，在执行过程中也会发生 context switch，回来的时候可能会发生新的 signal handler，会有一层层嵌套。

调用 signal 的情况下需要满足两个条件：

1. 收到了 signal
2. 没有被 blocked 掉

如果发现了 signal 是个空集合了，就回到了原来的 main 函数控制流

有些信号会终止进程和 dump core (内存映射存到硬盘便于 debug)

SIGCHLD,

不要太傻地去回收子进程，之前我们的方法是定期 waitpid 立即返回，事实上我们可以在 sigchild handler 中去回收子进程。

注册 Singal Handler

我们可以通过 syscall signal 去注册 handler。不能覆盖 sigstop 和 sigkill。

```

#include <signal.h>
typedef void handler_t(int)
handler_t *signal(int signum, handler_t *handler)
    returns: ptr to previous handler if OK,
    SIG_ERR on error (does not set errno)

```

返回值是原来的 handler, 参数是新的 handler。

- Three ways to change default actions:
 - If **handler** is **SIG_IGN**, then signals of type **signum** are ignored.
 - If **handler** is **SIG_DFL**, then the action for signals of type **signum** reverts to the default action.
 - Otherwise, change action to **handler** (called signal handler)

有三个宏, 设置为 ignore, 恢复成默认值, 或者提供一个 handler。Handler 只能接受一个 int, 所以可以用一个 handler 为多个 signal 服务, 通过判断输入参数的 signal 号。

这个函数是功能是注册安装一个 signal handler, 操作系统会帮你调用这个 handler。

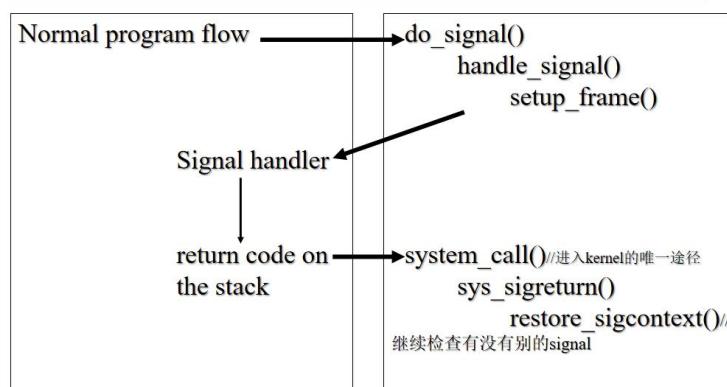
```

1 #include "csapp.h"
2
3 void handler(int sig) /* SIGINT handler */
4 {
5     printf("Caught SIGINT\n");
6     exit(0);
7 }
8
9 int main()
10 {
11     /* Install the SIGINT handler */
12     if (signal(SIGINT, handler) == SIG_ERR)
13         unix_error("signal error");
14
15     pause(); /* wait for the receipt of a signal */
16
17     exit(0);
18 }

```

注意这里是 `exit(0)` 会终止整个程序。在准备截获之前安装上自己的 handler。正常情况下处理完调用 `return` 回到原来的执行流。

Signal Function



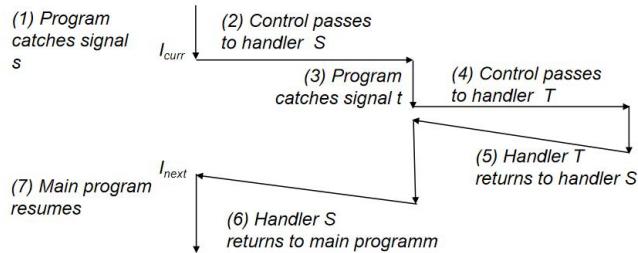
Signal handler 完成回到 kernel 态后, 需要继续检查 signal 情况。

在执行信号 10 的处理中, 信号 10 会被阻塞, 以免递归地调用同一个 signal。因为执行的是同一份 signal handler。

Q: 如果一个 handler 处理多种情况，如果有多个信号同时打断，是设置多个信号的 block 吗？

A: 不是，只设置了第一次信号的 block。所以在编写 signal handler 的时候要尽可能避免这种情况，这是异步不安全的。

Signal Handlers can be interrupted



此时存在三条并发的执行流，可能会出现各种可能的问题。

存在一个操作系统帮助 block 的情况，处理 K 的情况 K 会被 block 住。显式地 block: 使用 sigprocmask 及其一些帮助函数。

Blocking and Unblocking Signals

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set,
                 sigset_t *oldset)
int sigemptyset(sigset_t *set) ;
int sigfillset(sigset_t *set) ;
int sigaddset(sigset_t *set, int signum) ;
int sigdelset(sigset_t *set, int signum) ;
                                         Return: 0 if OK, -1 on error
int sigismember(const sigset_t *set, int signum) ;
                                         Returns: 1 if member, 0 if not, -1 on error
```

最简单的是用现在的 set 和原来的 set 做一个 and 或者 or 操作。

Explicitly Blocking Signals

- `sigprocmask`
 - Changes the set of currently blocked signals
 - `SIG_BLOCK`: add the signals in set to blocked ($\text{blocked} = \text{blocked} \mid \text{set}$)
 - `SIG_UNBLOCK`: Remove the signals in set from blocked ($\text{blocked} = \text{blocked} \& \sim \text{set}$)
 - `SIG_SETMASK`: $\text{blocked} = \text{set}$
 - If `oldset` is non-NULL, the previous value of the blocked bit vector is stored in `oldset`

要 set 第九个信号，那就是搞一个 00000100000 的和原来的 set 做一个 or 操作，其余同理。

2021/3/23

signal 是进程之间的通信方式，是反向地从 kernel 向用户态发送信息的机制（异步，并不是马上得到处理，只有进入一个 cycle 在执行 main 之前才会调用）。正向只有 syscall（同步）。这节课我们关注怎么写一个 signal handler，写对不容易，很容易造成并发问题。Main 执行流和 handler 生命周期会重叠，而且很长的情况下不知道什么时候被打断，包括还有递归打断的情况。这会带来一种特殊的 concurrent bug，处理、解决、发现都依赖于特定的执行情况，这些都由 kernel 来决定、不受控制，很 tricky。（Eg：打断的点不同，出现嵌套的情况不一样。）这些错误在 debug 过程中很难被复现。

写 Signal Handler 的 Guideline

G0:尽量保证 handler 简单，让 handler 的执行时间执行时间越短（降低嵌套的可能性，把指令留到 main 函数来执行，降低 overlap 的程度。）比如在 signal handler 里面只设置一些 flag（作为一种通知的机制），在 main 函数里来完成任务，但是这会是 main 函数要周期性地检查对应的 handler。

G1:这条影响比较大，要在 handler 里面调用函数是异步安全的（asyn-signal-safe）。必须要选取这些 safe 的函数来使用，比如说 printf sprintf malloc exit 是不安全的。Bug 不一定会发生，但是存在这些 bug。为什么会出现？因为这些函数会尝试去拿一些锁。如果在 main 函数里面也要使用相同的屏幕的锁。如果 handler 里的 printf 尝试去拿锁，但是锁在 main 函数里被拿着，这种情况下程序会进入死锁状态。锁：只有一个人可以进去，来保证进程里的代码是安全的，只被一个人访问。Signal handler 也要实现成 safe 的。Safe 函数有一个专门的列表。

Safe I/O Package

```
#include <csapp.h>
ssize_t sio_puts(long v);
ssize_t sio_puts(char s[]);
>Returns: number of bytes transferred if OK, -1 on error
void sio_error(long v);
>Returns: nothing
```

- `_exit` is an `asyn_signal_safe` variant of `exit`.

打印函数，不支持格式化输出。直接调用 syscall 来打印。

G2:保存和恢复 errno。errno（error number）是一个全局变量，是调用的库里定义的全局变量。当调用函数出错的时候，error number 会被设置。为什么要特意保存和恢复 error。在调用 syscall 的时候如果需要做一串耗时很长的指令，期间过程中会发生 context switch。最新一次 syscall 的 errornumber 会把 main 函数里面 syscall 的 errornumber 覆盖掉。会出现 errno 覆盖的情况，但是 errno 是不希望被覆盖掉的。这个和我们使用寄存器一样。参考 callee-save 和 caller-save，一般如下进行 errno 的保存。

```

1 #include "csapp.h"
2
3 void handler1(int sig)
4 {
5     int olderrno = errno; //保存外部使用时记录的errno
6
7     if ((pid = waitpid(-1, NULL, 0)) < 0)
8         sio_error("waitpid error");
9     sio_puts("Handler reaped child %d\n");
10    sleep(1);
11    errno = olderrno ;
12 }
13

```

问题：回收子进程回收不干净。

原因：signal 不会 queue，当处理 signal 的时候，signal 被 pending。如果同时来了两个 signal 不能区分出来。

每次调用的结果可能不同。所以应该在 handler 里面循环地调用 waitpid 回收子进程。

关于 signal handler 的行为的，可移植性（最低要求是用户那里可以执行）问题。

对于 signal handler 我们也希望其在类似的平台下能够处理。不同系统和版本下对于 signal 的语义约定可能有些差别。

1. syscall（可能发生上下文切换，执行时间比较长）被 signal 打断的时候怎么处理。程序接到 signal 是比较频繁的。为了设计上的简洁，我们直接 abort 掉。凡是有 signal 过来了，syscall 就直接 abort 掉，不管谁打断我都停下来，由上层来决定下一步行为是什么。Linux 的默认行为是如果 syscall 被 signal 打断了，就会重新尝试（retry）。Retry 属于 semantic，在不同操作系统下的行为可能不同，需要有意识地对 syscall 进行处理。

处理分两种 1.自己对情况进行检查处理 2.自己设置好 retry 这种情况。

2. 在执行 k 这个 signal handler 的时候，可能被其他 signal 打断，不会被自己打断（把自己的 signal block 住）。避免自己调用自己的函数，出现重入的问题。因为如果这个函数访问了全局变量，那么再进去会重写，包括说这个函数中调用了锁的情况。但是这个行为也是可以配置的，系统的默认行为可能不同，换一个场景下代码就可能出错。

sigaction Function

```

#include <signal.h>
int sigaction(int signum,
              struct sigaction *act,
              struct sigaction *oldact);
              returns: 0 if OK, -1 on error

struct sigaction {
    void (*sa_handler)();           /* addr of signal
handler,                                or SIG_IGN, or SIG_DFL */
    sigset_t sa_mask; /* additional signals to block */
    int sa_flags;                /* signal options */
} ;

```

可以控制 handler 的行为，提供了一个数据结构，其包括 handler，handler 的 mask 和 flag 其中有很多 bit 位。

```

/* block sigs of type being handled */
sigemptyset(&action.sa_mask);
/* restart syscalls if possible */
action.sa_flags = SA_RESTART;

```

系统会帮你 restart，linux 会默认帮你置上。

- Once the signal handler is installed, it remains installed until Signal is called with a handler argument of either SIG_IGN or SIG_DEF
- 无视 signal

一些 Concurrency Bug

G3: 处理共用全局变量或结构时，需要 block 对应的 signal

如果在 addjob 里发生了 context switch，比如说链表刚刚要开始执行加元素操作加了一半时，要被删除元素操作打断。使用的过程中会被打断，会出现错误。

```

32     Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
33     addjob(pid); /* Add the child to the job list */
34     Sigprocmask(SIG_SETMASK, &prev_all, NULL);

```

所以 main 函数里和 handler 里要加上对应的 block 行为，以防冲突。Block 所有 signal，然后处理完后恢复。

但是代码还不是正确的情况，如果主进程在 fork 之后，addjob 之前被打断，会怎么样？Signal handler 会先执行 deletejob，根据 deletejob 的逻辑实现不同，可能出现 bug。比如 handler delete 完，再执行了 addjob 永久加入了一个死掉的进程。

如何写一个正确的代码？

```

29     while(1) {
30         Sigprocmask(SIG_BLOCK, &mask_one, &prev_one); /* Block SIGCHLD
31         if ((pid = Fork()) == 0) { /* Child process */
32             Sigprocmask(SIG_SETMASK, &prev_one, NULL);
33             Execve("/bin/ls", argv, NULL); /* Unblock SIGCHLD
34         }

```

30 行先 Block 住 sigchld，保证在 addjob 之前都不会调用 handler 去 delete。再去 fork。Fork 出来的子进程也是被 block 住的，

32 行需要恢复原始的 sigmask，因为我们要去执行 execve，不能在 mask 变掉的情况下去 execve，否则行为会变化。

```

/* parent process */
35     Sigprocmask(SIG_BLOCK, &mask_all, NULL);
36     addjob(pid); /* Add the child to the job list */
37     Sigprocmask(SIG_SETMASK, &prev_one, NULL);
38 }
38     exit(0)
40 }

```

第 35 行是为了防止在 add 过程中被 delete 打断的情况。

然后 37 行一起恢复。等 ADDjob 完成后，才能去进 handler 里去回收子进程。

G4: 声明全局变量时一定要声明为 volatile（你这个对象是不会放到寄存器里的，每次读都会从内存中读），显式防止寄存器优化。

如果放到寄存器中，handler 会分别操作寄存器，可能会出错。

如果 Main 去读寄存器，而全局变量中在 handler 去写。Compiler 会发现 main 函数一直在读没有去写，可能会一直读寄存器。本质就是一个变量可能出现在多个地方，在寄存器或者在内存里。一个程序是不会有这种问题的。但是现在两个地方都在访问这个变量，各管各的。

汇编生成的所有 volatile 都从内存中读写这个对象。两边就能访问到正确的对象。

G5:需要声明 flag 来用 sig_atomic_t 对于这个对象的读写操作是不会被打断的。在 C 语言的一句命令，在汇编中可能是多条指令，可能被打断。两次++可能只是从 2 变成了 3.

Spin Loop

```
while (!pid)
```

```
;
```

忙等，一直在 check cpu，cpu 拉满了，浪费 cpu。让 child 还没来的时候，休息休息。

我们可以用 pause 先挂起，解放 CPU 资源。但是我们 pause 可能被别的打断，所以容易被别的打断，还是要 check。但是这段代码是会有竞争的（data race）。如果在 while 和 pause 发生了 context switch，处理完就 sigchile 再也不会来了，中间被钻了空子，容易永久 pause。

我们可以使用 sleep1 来周期性检查。这是一种折中的方案。

我们希望检查和 pause 变成一个原子操作，有个特定的函数叫做 sigsuspend (mask)

Eliminate the Spin Loop

- The **sigsuspend** function is equivalent to an **atomic** version of the following:
 - 1 `sigprocmask(SIG_SETMASK, &mask, &prev);`
 - 2 `pause();`
 - 3 `sigprocmask(SIG_SETMASK, &prev, NULL);`
- The **atomic** (uninterruptible) property guarantees that
 - the calls to `sigprocmask` (line 1) and `pause` (line 2) occur together, without being interrupted

其相当于这三句话，并且这三句话不会被打断。12 代表打开中断立马进 pause。如果在 1 之前来，中断还没有打开，所以不会处理。2 的时候来了就可以 check pid 有没有问题。

```
pid = 0;
while (!pid)
    sigsuspend(&prev);
```

12 之间不会被上下文切换掉，不会被插队。

3 代表把中断再关掉。检查一下是什么打断了，如果不是 SIGCHILD，就再次进下一次循环。包括 lock 也会导致这种东西。

2021/3/30

Non-local Jump

for 循环和 if 都是用 jump 实现的。

c 语言最多就是跳出循环跳出 switch。

从功能上，汇编可以支持 non local jump，像 c 和 java 中的 exception，从汇编层面实现很容易。但执行过程中是维护了一个栈的，call 了一个函数出来的时候栈也得是空的。如果从一个函数直接跳到另一个，栈的环境可能就不一样了，可能栈就直接乱了。所以，nonlocal jump 难在要恢复这个状态。

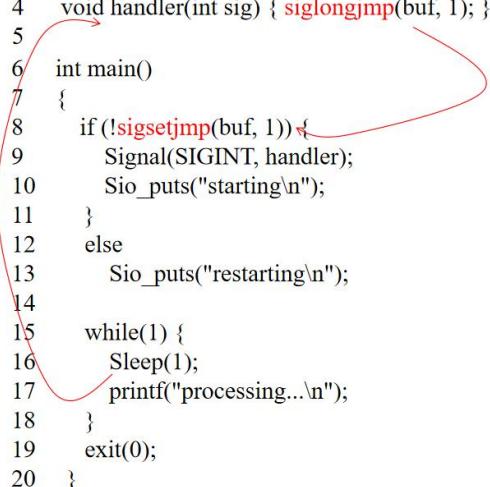
Nonlocaljump 需要分成两个步骤，功能就是在两个函数之间发生跳转。提供了 user 级提供了统一的异常控制流。

Setjmp 函数，是设置传送门，变量是当时栈情况（记录下开启传送门时栈的状态）。可以设置多个传送门变量，然后选取门去跳。

Longjmp 就跳走了，所以 retval 返回到了 setjmp 函数

还有 sigsetjmp 可以从 signal 中跳转到 main 函数的其他地方。要保存一个 sigmask 的信息。

```
1 #include "csapp.h"
2 sigjmp_buf buf;
3
4 void handler(int sig) { siglongjmp(buf, 1); }
5
6 int main()
7 {
8     if (!sigsetjmp(buf, 1))
9         Signal(SIGINT, handler);
10    Sio_puts("starting\n");
11 }
12 else
13    Sio_puts("restarting\n");
14
15 while(1) {
16     Sleep(1);
17     printf("processing...\n");
18 }
19 exit(0);
20 }
```



进程就讲完了，可以写一些多进程合作的进程。学会了 GUI 和 console。可以通过 signal 传递信息。调度是不确定的。Signal 是一个并发的执行过程。有一些特殊的 bug 就可以理解了。

System-level I/O

三大抽象:进程、内存、I/O

I/O 就是输入输出设备，除了内存、主板基本上都是输入输出设备。从 CPU 来讲输入输出设备，键盘是输入，屏幕是输出。我们要介绍 system level I/O。Printf 知道是打印。如果一个程序没有输入输出，跑完了没结果。没有输入输出机器就没有用了。把所有输入输出设备抽象为文件。默认打开的文件：屏幕和键盘。所谓的系统级的 IO 就是一些低层级的 IO。要和硬件打交道，就必须要通过操作系统。

为什么要底层的 IO，IO 分成标准 IO（高层级的用户级 IO）、UnixIO（由 kernel 直接提供，在 syscall 上直接进行处理）。IO 可以更好地理解进程。不同的进程使用 IO 的样子是怎么样的？Eg. 进程打开 IO，fork 了以后去打开 IO 呢？打开两次呢？

UnixIO 是把 IO 抽象成了文件。把所有外接设备的交互过程作为字符串。输入就是接受一个 byte，输出就是写一串 byte。在 IO 设备上，最重要的就是 read 和 write 这个 syscall。包括网络、磁盘、终端设备都是用这个。

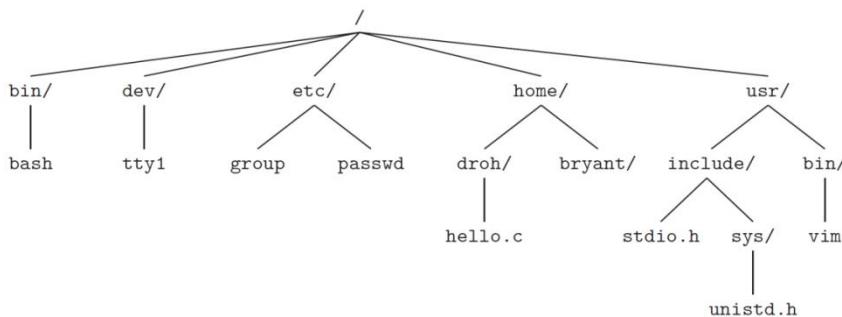
文件会分成不同的类型。有 regular 文件，普通的文本文件。Text file：可以用 asc2 码编码。Binary file：二进制文件。

除了 regular 之外还有很多其他的文件类型。里面有一些 abcd，也有一些不可见的字符。以 byte 读出来的时候是可以读到这些特殊字符的。比如回车\n。所有文件都可以抽象为一个 byte 流。

还有其他的文件：目录文件，里面是一系列的链接，可以通过操作系统解释成文件的指针。目录会自带两个创建时生成的 link。一个点代表当前目录。两个点代表上级目录的 link。

Mkdir、ls、rmdir 都是可执行文件，是前人写好的东西。这些都属于 IO 相关的编程。

Linux 对于下面的磁盘采用了树形的结构。一个磁盘让用户直接去访问太复杂了，所以需要有一个抽象。把磁盘抽象成文件进行管理，叫做文件系统。根节点在 linux 下是斜杠。下面就是一层层的目录用/分割。



所有的 IO 设备也在 dev 设备下面。所以文件系统包括了磁盘上存在的文件，以及硬件。还包括了一些 pipe。可以用一个管道把前一个文件的输出作为后一个文件的输入。管道可以把进程之间打通。Symbolic links 就是快捷连接文件。还有一些字符和 block devices，这就是外设。

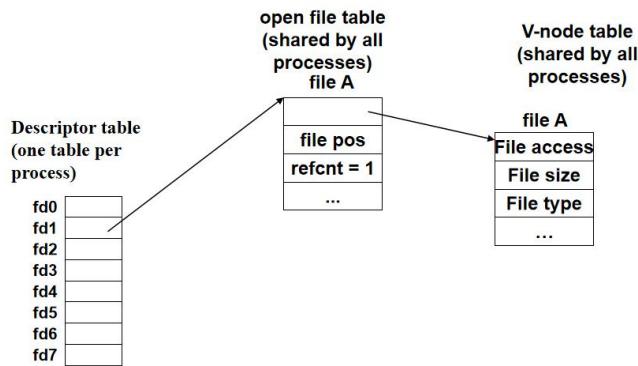
Cd 可以改变当前的工作目录。每一个进程在其之中就可以改变工作目录。

- relative pathname: ./hello.c, ../home/droh/hello.c
(if /home/bryant is the current working directory)

比如要执行当前目录下的 cd 文件，如果没有点就回去找系统中的可执行文件。可能需要加./cd

OS 支持文件和 IO 的形式

这些 IO 设备在操作之前都有一个准备阶段。Printf 下一次打印是跟在上一条打印后面的，一般是相对位置的操作。这些相对位置信息都是要记录在 kernel 里面的。内核会为每一次打开文件建立特别的数据结构。一般情况下，io 设备都有 open 和 close 操作。Open 之后会返回一个标识，分为一个 file descriptor（文件描述符）。文件退出的时候是不会马上关闭的，只有回收的时候会收回这个资源。文件描述符是 link 中数组的一个下标，存了一个 link。Fd 维护了文件的当前的位置（上次读完的位置，fileposition）、文件大小、文件使用人等。我们需要了解的就是内核为每个进程维护的这一套信息。



由三个部分组成。左边是每个进程都有一个 **descriptor table**。每个文件打开以后都会填入一个 **fd**。打开一个文件时，就是用 **fd** 指向这个文件信息。为了避免 **descriptor table** 很大，会找到一个最小的 **fd**。指向的就是 **open file table**，这由多个进程共享的。比如实现进程之间的信息传递，把两个进程联系在一块。这个 **table** 里面记录的就是这次打开相关的信息，一个是 **position**，第二个是 **reference count**（这是用来计数的，当没有人使用的时候要回收掉）但是因为会被进程共享，所以造成这个 **file** 创建的进程和造成其回收的进程可能不是同一个进程。当 **refcnt** 为 0 的时候，就可以回收这个资源了。

Vnode table 是一个静态的信息，在文件创建的时候就有了。进程哪怕关掉了文件还会存在。**File table** 可以被共享，更多情况下是每个进程自己创建。

Seek 是一个 **syscall**，就是调整当前文件的 **file pos**。（**seek current file position**）

Open 函数有三个参数，第一个是文件名可以是绝对路径可以是相对路径。返回-1 失败，返回 $>=0$ 成功。

Flag 是只读、只写、读写打开（唯一）。还有三个可选项，**O_CREAT**，创建文件。**Trunc** 覆盖文件。**Append** 就是追加到文件的最后。

Mode 参数是模式，当创建文件的时候使用模式。**Mode 777** 就是所有权限都有。**Mode** 分为读、写、执行。分为 **user**（普通人）、**group**（组）、**other**（其他人）。Eg 777, 664.

一个文件会打开标准输入输出和出错。所以第一个打开文件的 **fd** 应该是 3。如果没有自己关闭，进程回收的时候会关闭 **fd**。一个文件 **close** 两次也会报错。

文件有一个文件尾。**Read** 是外设读到内存的操作。需要有标志告诉你读完了，所以需要文件尾，这是逻辑上的，文件里是没有的。只是在 **pos** 到最后的时候，**read** 返回 0，代表只能读到 0 的 byte。

```

#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
    returns: number of bytes read if OK,
              0 on EOF, -1 on error
ssize_t write(int fd, const void *buf, size_t count);
    returns: number of bytes written if OK,
              -1 on error

```

Buf 是实现分配的地址，如果分配小了就有可能出现 segmentation fault。Read 返回 0 读到了的文件尾，返回-1 是出错了。

Write 中，写进去的数据先放到 buf 里面。然后 count 写进去的数量。

Reading and Writing Files

```

1 #include "csapp.h"
2
3 int main(void)
4 {
5     char c;
6
7     /* copy stdin to stdout, one byte at a time */
8     while(Read(STDIN_FILENO, &c, 1) != 0)
9         Write(STDOUT_FILENO, &c, 1);
10    exit(0);
11 }

STDIN_FILENO(0), STDOUT_FILENO(1), STDERR_FILENO(2)

```

键盘上敲一个，屏幕上输出一个。但是键盘上敲不出 EOF，所以可能需要 **ctrl+c**。

Ssize_t 是有符号的 **int**, **size_t** 是无符号 **int**。如果 **read** 和 **write** 读了特别大的数，可能是出现一些问题。Short count 问题，即 **count < 返回值**。比如说 **read** 读到了 EOF，或者 **syscall** 打断了 **read**，导致读了一半。读到的数量小于预期的数量，这就叫做 short count。

还有 **stat** 函数和 **fstat** 函数，访问的是文件的 **vnode**。所以 **stat** 通过路径去访问文件的 **vnode**，可以不需要打开。在打开 **fd** 的情况下，可以通过 **fstat** 去读信息。

要访问 **stat** 的时候一定要通过 **unixio** 来做。

S_ISREG 可以检查文件的模式。

```

int main(int argc, char **argv)
{
    struct stat stat ;
    char *type, *readok ;
    Stat(argv[1], &stat) ;
    if (S_ISREG(stat.st_mode)) /* Determine file type */
        type = "regular" ;
    else if (S_ISDIR(stat.st_mode))
        type = "directory" ;
    else
        type = "other" ;
    if ((stat.mode & S_IRUSR)) /* check read access */
        readok = "yes";
    else
        readok = "no";
    printf("type: %s, read: %s\n", type, readok) ;
    exit(0) ;
}

```

S_ISREG() Is this a regular file?
S_ISDIR() Is this a directory file?

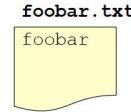
其包含了访问控制权限。

Read Directory Contents

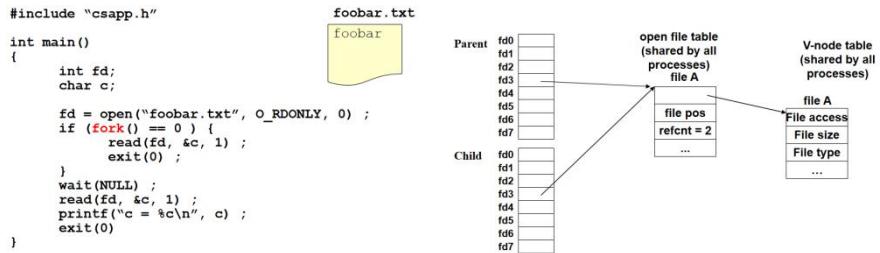
```
struct dirent {  
    ino_t d_ino; /*inode number*/  
    char d_name[256]; /*file name */  
}  
  
#include <sys/types.h>  
#include <dirent.h>  
  
DIR *opendir(const char *filename);  
    returns: pointer to handle if OK, NULL on error  
struct dirent *readdir(DIR *dirp);  
    returns: pointer to next directory entry if OK,  
            NULL if no more entry or error  
int closedir(DIR *dirp);  
    returns: 0 if OK, -1 on error
```

以上打开目录，可以一次一次地读出其包含的文件信息。可以用 `while` 来判空。
一个文件打开两次会怎样？

```
#include "csapp.h"  
  
int main()  
{  
    int fd1, fd2;  
    char c;  
  
    fd1 = open("foobar.txt", O_RDONLY, 0) ;  
    fd2 = open("foobar.txt", O_RDONLY, 0) ;  
    read(fd1, &c, 1) ;  
    read(fd2, &c, 1) ;  
    printf("c = %c\n", c) ;  
    exit(0)  
}
```



输出 f 的话，说明有两个 open file。输入 o 的话，说明有一个 open file 被两个 fd 共享。



Fork 出来的 descriptor table 和原来的一样。Kernel 是会帮你 copy 一份 open file 还是帮你共享一份呢？事实上创建子进程的时候，open file table 是共享同一个 open file 的。这提供了一种途径让进程之间进行共享。Kernel 维护一个 Refcnt++罢了。

Redirect 重定向文件的输出。Shell 在创建 ls 的时候，把标准输出的文件 fd 改成 foo.txt。往 fd1 写的时候，就写到 foo.txt 里去了。所以 shell 只需要重定向，而 ls 的代码就写到 fd1 里去，不需要进行任何修改。在做 shell lab 的时候，要实现重定向就要写 dup2 函数。

2021/4/6

今天会把 I/O 部分结束，上节课介绍了操作系统如何支持 I/O 的，维护一些数据结构来提供给上层的程序。一个是 **fb**，一种索引在数据结构中找到当前打开的位置。最主要的是，一个是一个 **array**，指向 **file** 对应的信息，第三层的一些物理上的文件。我们还介绍了一些 **syscall**，底层的操作系统支持的 I/O 系统。是读和写、打开和关闭，加上 **status** 函数，就可以知道文件所有的信息了。但是这比较麻烦，处于 **syscall** 之上比较底层。这不是我们直接使用的 **io**。我们经常使用的是 **std io**。

Robust I/O

这节课我们介绍 robust I/O 和 standard I/O，它在 unix I/O 之上去构建。

首先 **rio**，所谓 **robust** 就是功能更多，在一些特点上比如性能上会有一些特点。最多的情况下会被使用在网络里。因为普通的程序基本上使用 **std io** 即可。但是终端设备和网络上不能使用 **std io** 直接进行使用的。**syscall** 过长可能被 **signal** 打断的问题（**short count** 问题，写的比预想的要少。），需要自己去解决。**rio** 分为 **unbuffered** 和 **buffered**。也就是有没有缓冲区。屏幕打印和 **gpu** 也有对应的缓冲区，可以理解为是一段内存。我们需要回答用 **buffer** 能解决什么问题。

rio_readn 和 **rio_writen** 是无缓冲区的 **buffer**。接口和原来的 **syscall** 基本上一样，尽可能地读 **count** 个字节。完成以下逻辑：

- 如果出错了，那就返回。
 - 如果 **read** 被中断打断了，会帮你完成一个重试。
 - 如果读到的数据少了，也会帮你去重试。
 - 如果读到了文件的 **eof**，这种情况下会 **break** 出循环，正常的进行返回。
- 对于 **rio_writen** 的情况，就比较容易，如果被打断了也会重试。

`rio_readlineb` 和 `rio_readnb` 是读一行，如果一个个读 `byte`，每个 `byte` 从硬盘去读，会非常慢。这时候我们就使用 `batch` 的思想，一次读多一点。如果我们读的超过了一行该怎么办？有的地方可以退回去。但是网络包该怎么退回去？

接下来，我们介绍缓冲区 `Io`，在直接做一个操作的时候，缓冲区进行一个介入，因为原来的操作太慢。分为读缓冲和写缓冲。真实的读和写操作的触发不是完全匹配的，我们比如说先读 1024 个 `byte` 读到缓冲区里。换句话说，增加了一些内存的开销，来避免多次读写硬盘。一个 `buffer` 里面的数据可能分为几块，每次读不一定把 `buffer` 填满（比如磁盘中已经没有数据了）读上来的数据可以分为应用已经读的，和应用程序还没有读的。这里我们维护了 `buffer` 层的数据结构。第一个参数是 `fd`，对应被打开的文件。有 `cnt`（还没有被读的）、`pointer`（`buffer` 中的 `position`）、数组（`buffer` 内存）。

如何来使用呢？我们要了解 `buffer` 和 `disk` 中的数据的对应关系。`buffer` 就等于是文件中的一个子区间，是文件最新填充到 `buffer` 中的数据，存在内存里面，当然 `disk` 上也有一份。应用程序不和 `disk` 打交道，直接访问缓冲区。

初始化的时候，`fd` 作为参数传进来，返回这个初始化完成的数据结构。`pointer` 指向开头，`cnt` 为 0。

这个 `rio_readline` 函数功能不太好实现，极端情况下可能这一行比 `buffer` 还长。首先我们要实现一个 `rio_read` 函数。

`readline`

这个代码先判断 `buffer` 是否已经空了，只要 `buffer` 不空就会判断以下情况：

- 想读的比 `buffer` 大，直接返回整个 `buffer`。
- 想读的比 `buffer` 小，返回对应的值。

然后把 `buffer` 中的数据 `copy` 到应用程序中的内存中。

然后调整 `pointer` 和 `rio_cnt`（`buffer` 中剩余的数量）

如果 `buffer` 为空，从 `disc` 中读取 `buffer size` 的数据。

·如果出错，返回 -1

·如果被中断了，重试尝试读 `buffer`。

·如果读上来的值是 0，代表读到文件尾，返回 0。

然后需要条件 `pointer` 到开头，以及 `cnt` 要设置为新填入的数据数量。

所以 `read` 提供了从 `buffer` 中尽可能读对应的数据到应用程序的内存中。

`rio_readnb`

`rio_readnb` 就是使用 `buffer` 的读 `n` 个数字，就是调用 `rio_read`。调用 `syscall` 的次数比起非 `buffer` 可能要少一些。

`rio_readlineb`

然后我们再来讨论 `rio_readlineb`。如果用 `syscall` 提供的 `read`，性能就很差。现在我们换成了 `rio_read`，这样就很快了。只要在循环中判断一个是否读到了 `\n`，多读出来的数据被放到了 `buffer` 里面，便于做下一步的操作。

Standard I/O

standard io 是一种 high level 的 io，对于 std io。暴露出来的接口用 `f` 开头，比如 `fopen`

`fclose`。其同时实现了读和写的缓冲。里面最重要的概念就是，`stream` 的抽象。在文件的基础上抽象出了 `stream` 的概念。流就像水一样，一去不复返。如果采用了这一套接口，就不太会返回到读过之前的东西。所以网络更适合的抽象应该是流。

缓冲区对功能是有限制的，比如读上缓冲区以后，如果是网络就不能再丢回文件中。

前面讲了 `std io` 很重要一点提供了完整缓冲区的功能。写操作会等凑够一定的数量才会写到 `disc` 中。如果 `printf` 如果没有 `\n` 是不会写到缓冲区中的，`\n` 就是缓冲区的一个刷新条件，还有定时刷新，还有一种强制刷新，也就是 `fflush` 来保证执行到当前位置之前的写操作都会在磁盘上生效。

之前介绍的 `rio` 和 `std io` 是建立在 `unix io` 之上的，当然应用程序可以直接调用 `unix io` 和 `syscall` 去做。那么应用程序在什么时候应该选取哪种 `io`？一般情况下，是不推荐应用程序混合使用不同种 `io` 的。如果混用有 `buffer` 和无 `buffer` 的情况，可能会导致直接读到 `buffer` 缓冲后的位置。

`unix I/O` 的利弊

这是最 `general` 的，所有功能都可以完成。是 `low level` 的，理论上单次 `io` 操作，`overhead` 是最小的。其余 `io package` 是实现在其之上的。`metadata` 是只有 `unix io` 提供了检查接口。
(`fstatus` 不属于 `std I/O`， 属于 `unix I/O`)

`Unix I/O` 是异步安全的，在 `signal handler` 里面只能使用 `unix I/O`。

实现很脆弱很敏感，容易操作应用程序代码复杂，容易出错。应用程序写出的代码可靠性低于 `std I/O`。

`std io` 的利弊

支持缓冲区，在一系列操作的时候性能更好。支持 `general` 的操作。比如支持变参数的函数 (`printf`)

不是 `signal-safe` 的，不能访问 `metadata`。因为加了缓冲，不太适合网络的操作。

我们介绍几个和 `stream` 有关的。`read buffer` 和 `write buffer` 是两个 `buffer`。`stream` 又读又写的情况，如果写缓冲区没有刷新的情况，跟了一个读该怎么办？可能会读到即将被覆盖的旧数据，所以建议不要这么做。如果一定要跟在后面，需要加上 `fflush`。包括还有一些 `eof` 的处理。

如果我们为了在一个网络流上同时进行读写。我们需要把这个网络流打开两次，会出现新的问题。把 `fd` 打开两次，有两个数据结构，当我们 `close` 的时候，可能会把一个文件 `close` 两次。所以在流上处理一些行为的时候，是非常 `tricky` 的。

综上所属，`std io` 上不适合处理网络的情况的。

一个 `general` 的规则：尽可能使用 `high-level` 的 `io`。但是遇到 `signal` 和 `metadata` 的情况，我们只能使用 `unix io`。

`标准 io`: 使用磁盘文件和 `terminal` 文件 (eg: 屏幕) 的时候。

`raw unix io`: `signal`; 对需求很清楚，需要最高性能的时候。

`rio`: 在网络的时候可以使用。

网络

网络是一个非常复杂的方向，有专门可以研究的内容。我们只需要能用，了解基本结构就行了。

我们先介绍 client-server 变成模型。还有 p2p 模式。

像我们平时访问的 web 就是 cs。手机的小程序、ftp 都是这种。cs 是当前最主流的网络编程模式。client 是访问资源的人，server 就是掌握资源的人。有很多 client 寻求 server 的帮助。在单机里，也可以看作一种 cs 模式。server 就是 os，像操作系统发送请求。向屏幕发送请求进行打印，其实也是一种 cs 模式。client 和 server 都是进程，需要跑在那里。server 和 client 会约定好按照某种格式进行交互，以 request 进行传递。request 可以限定一些接口，超出了限制的范围，server 就不会管了。因为约定好了 interface，client 和 server 就是松耦合，可以自己进行优化。只有满足约定的 request，server 才接收。

首先网络从硬件上来说是 io 的一部分，既可以 input 和 output。机器上插的叫做网卡（network adapter）

我们先要介绍网络。首先网络是一个层级结构，数亿机器设备都接入网络。要管理这些设备，要用层次化的方式来组织，像一棵树一样。box（终端设备）、wire（网线、或者无线信号）

网络越往下越 low level，最低层级的是 lan（loval area network）。lan 可以认为是网络里面很小的一块，一般按照 building 或者校园为单位。

网络里要用一个机器都懂的语言沟通在一块。在 lan 里面用的比较多的就是有线以太网（ethernet），使用以太网卡。lan 最底层就是一些 host（包括机器、手机、主机）。hub 就是把 host 连在一起的设备。hub 提供的口就叫做 port。hub 的一个特征就是广播的方法，当收到一个 request 的时候，就会广播给连在上面的所有 host。但是这个方式是不可扩展的，如果数百台机器连在上面，有效消息只有百分之一，机器自己都处理不过来。每个插上以太网的设备都会有一个 48 位的以太网 mac 地址。每一位都会有不同的含义，代表硬件的厂家等。发送的单位叫做 frame。

2021/4/13

将 network adapter 连在一块，就称为网络。而且是分层的架构，像金字塔一样构建在一块。粗略的划分是 lan 和 wan，在不同场景下可能称为不同的，只是一个粗略的概念。只是在网络发展初期的时候进行区分。现在有移动设备、手表等让这个结构更加的复杂了。

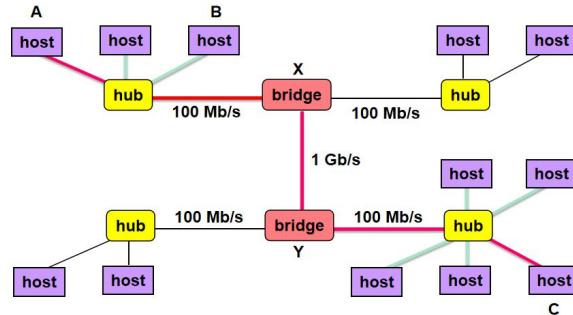
Hub、Bridge 和 Router

上节课介绍了 lan 最底层的是用 hub 进行连接的，缺点就是连接在上面的机器（host）数量不能很多。Host 过滤的方式：要有每个机器的唯一标识符，这个标识是对应了机器的 adapter 上的。每个 adapter 有 48 位的唯一标识符。虚拟机的网卡也会有 mac 地址。传输的叫做帧，也就是数据单位。

一个消息(message)包含了两部分，一部分是本身的内容（照片、文字等），叫做 payload。还有一部分是发送的对象、邮编等，这叫做 header。Header 和 payload 放在一起称作 frame，

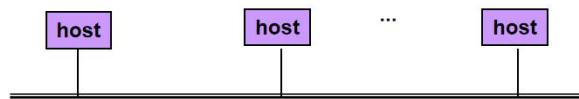
发出去。传输过程中关心的是 **header**，接受者关心的是 **payload**。

在整个建筑、校园中，都采用 **hub** 连接是不现实的。在其之上是一些更加智能的链接。越往上链接的是越高层的一些概念。在 **hub** 上有 **bridge** 的概念，需要根据消息的 **header** 的消息发送给某个 **hub**。

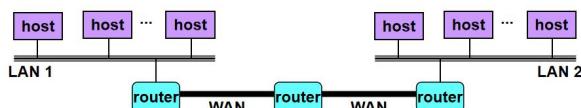


Bridge 比 **hub** 更聪明，知道应该把消息发送给哪个 **host**。**Bridge** 比 **hub** 更高层、功能更强、设备更复杂。

现在家里的 **hub** 也比较复杂，有线电视、有线网、无线网都集成在一块儿。所以现在 **bridge** 也越来越少，一部分功能去了 **hub**，一部分去了上面。所以，我们可以做以下简化，接在了一根逻辑的网络线上，都是使用相似的协议了。

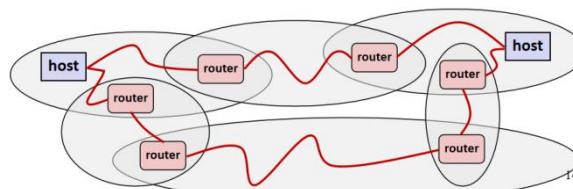


这里就牵涉到协议的一个概念，比如无线网和有线网使用的协议是完全不同的。不同的协议可以理解为是不同的国家里的人说不同的话。**Bridge** 可以认为是在一个国家内的链接。如果要把不同国家连在一起，就要对不同语言的协议做翻译。协议包括怎么写 **header**，怎么写发送的对象。在不同协议之间的翻译者（跨协议的网络链接）叫做路由器。



LAN 1 and LAN 2 might be completely different, totally incompatible
LANs (e.g. Ethernet, Wifi, DSL (Digital Subscriber Line))

通过路由器（**router**）链接不同协议的网络。路由的能力很强，可以把不同协议的消息进行互通。路由器中可以加入防火墙、检查等功能。



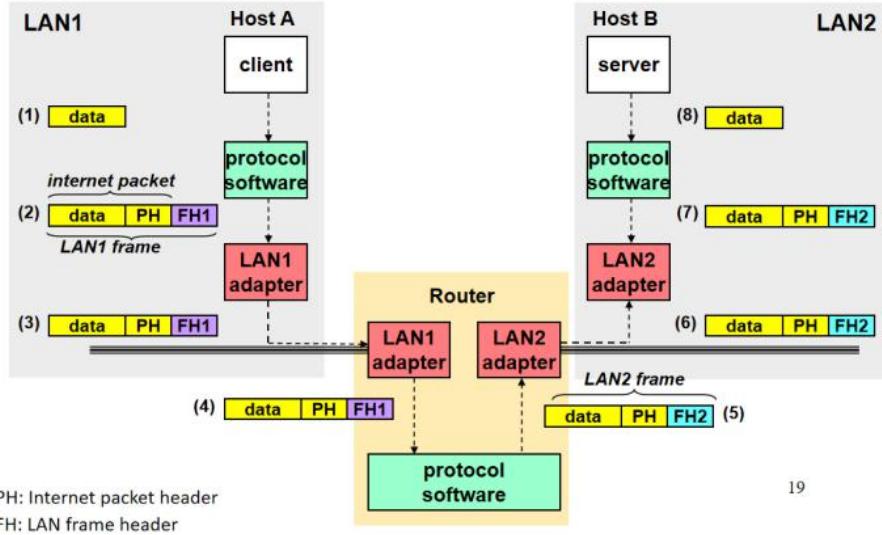
每个小的都可以有路由器进行连接，在这之间传递的就是 **package**（网络包）。前面提到的协议（**protocol**），一套协议可以认为是文字记录的表达规范。在计算机里需要由软件实现这一套协议，要发送的时候，OS 中嵌入了协议软件，来包装我们的 **payload**，包装成一个 **package**。网络各种协议的设计也称为网络栈，每一层完成一些网络。

TCP/IP

TCP/IP 是两个不同层的协议来构成的，分别是 TCP 协议和 IP 协议。

接下来我们要介绍对应的协议。协议栈分很多层，有些层比较底层，规定了硬件的行为。程序员主要需要理解的是 TCP/IP 这两个层。网络层一个重要的行为就是取名策略。名字在网络中很重要，需要能够在整个网络中被识别到。Naming 具体的实现比较复杂，要尽可能保证唯一性。所以这是全局情况下 IP 对应的协议，统一对所有 host 给出一个唯一的地址。

在其之上的一层协议是 deliver 机制。完成命名了之后，就负责发送 package。要确定投递的方式等。TCP 就属于这一层。有了这个协议以后，如何从一台机器向另一台消息发送消息呢？



19

Data 在网络上传递的时候，只是一串 01 数据。客户端如何产生，服务器如何解读都和网络没关系。Host 发送时，需要网卡的帮助。所以需要操作系统来管理。通过对应的 syscall，把这个消息交给 kernel。操作系统中有对应的协议栈，根据网卡的信息，来选择合适的 protocol，对 data 加上 header。Header 包含两部分，一个是 host 级别上的（Internet packet header），一个是 lan 级别上的 header（LAN frame header）。操作系统类似于给我们的信件加上了信封，然后帮我们向网络中投递出去。对应的封装是层次化的封装，换句话说，可能套了几层信封，在同样层中打开相应的信封。层次化包装的概念在网络中非常重要，使得我们的网络包非常适合网络这种层级结构。

Router 能够识别 FH1，router 负责翻译成对面认识的 header（换一个信封）。在 router 层级，只有蓝色和紫色是 header，里面黄色的都认为是 payload。传递到 LAN2 以后，就逐层地拆信封，最终取出 data，交给上层的应用程序。

整个网络是逻辑抽象的，可能向室友发送消息的包要走一圈才能回来。所以这是 virtual 和 physical 的一个对应关系。

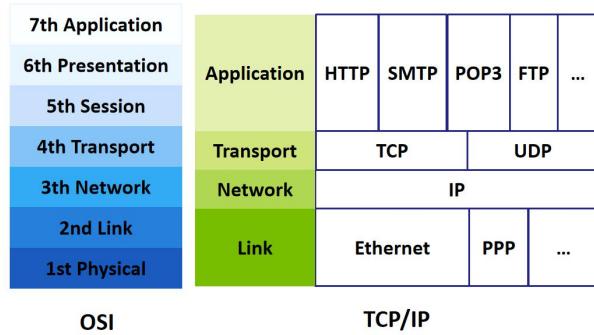
Other issues

- We are glossing over a number of important questions:
 - What if different networks have different maximum frame sizes? (segmentation)
 - How do routers know where to forward frames?
 - How are routers informed when the network topology changes?
 - What if packets get lost?

对于过大的文件，需要切割，分片后可能走不同的路径，最终如何组装起来？路由器怎么找到最近的路？如果发送的多个包有一个丢了怎么办？

互联网中，90%收到的协议都和以下介绍的协议相关的。

IP 是基于 naming 的一套协议（全球 Internet 网）。首先介绍网络栈的概念，我们要分层地去实现一个网络协议。标准组织的 OSI 模型，七层架构。

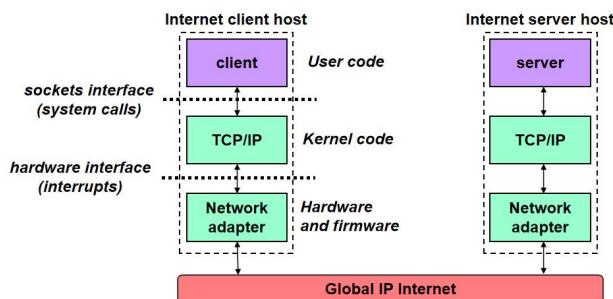


TCP/IP 是现在所用的一个事实标准（最后留存下来，大家都按照这个方式来做）。事实标准比 OSI 简化的多。TCP/IP 分别对应了网络层和传输层。

IP 层主要为上层提供定位功能。上面传输层的 TCP 构建一个稳定的通路，在两个 host 之间构建一个逻辑上的通路。在这之间如果出现了网络上的丢包，TCP 是会帮你解决的。对应的 UDP 就是不可靠的，为了更快地发送到对面，如果有丢包，也会向上把信息传递给应用程度，速度很快，比如在一个数据中心的内部，网络比较稳定，就可以用 UDP 来加速数据传输。

Berkeley socket interface

接下来我们要介绍 Berkeley socket interface，这其实是一套网络 IO 对应的接口封装，和 protocol software 对接，是 syscall 的一套 lib 包装。它的优点是把网络抽象成了文件和 linux IO 兼容。



上图紫色的就是网络应用程序，要使用网络的时候，就需要使用 socket 进入 syscall，用 kernel 中的 TCP/IP 软件栈来负责加上 header，再给到网卡。

IP 最早的时候，是一个 32 位整数来代表。为什么会变成 IPV4 呢？这就牵涉到 32 位如何进行转化。比如说 ip 映射到域名。域名也是一个层级结构，从前往后从具体到高层。Ip 地址和域名之间有一个对应关系。每一个链接就是客户端和服务器之间要建立一个 connection。

我们发现 32 位的 IP 地址和域名是没有关系的，同时服务器还可能会搬动。这是需要一张表来记忆的。Ipv4：我们把 32 位分成四段，每一段都是 0 到 255，是可以直接翻译的。

- IP address **0xca7828bc** = 202.120.40.188

```
/* Internet address structure */
struct in_addr {
    uint32_t s_addr; /* network byte order (big-endian) */
};
```

要特别强调的是，网络地址传输的时候是 big-endian 的。换句话说 IP 地址是要进行转化的。对应的翻译过程就是以下这些函数，**h** 代表 host，**n** 代表 network。即从 host 到 network 的转换。32 位是用来转 ip 地址的，16 位是用来转化 host（端口号）的。

- the following functions convert between network and host byte order
- No equivalent function for 64-bit value

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
    Returns: value in network byte order

uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
    Returns: value in host byte order
```

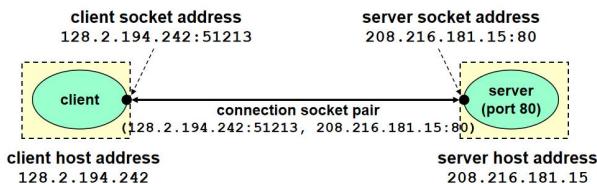
- **inet_ntop**: converts a dotted decimal string to an IP address in network byte order.
- **inet_ntop**: converts an IP address in network byte order to its corresponding dotted decimal string.
- "n" denotes network representation. "p" denotes presentation.

需要有一个把域名和 IP 的映射关系存下来的地方。即 DNS 服务器（域名服务器，domain naming system）。网络程序和域名服务器联系，在本机上把域名转化为 ip 地址，有了 ip 地址才能在网络上发包。如果连不上域名服务器，不能把域名转化为 IP。那么可能访问 ip 可以直接访问，但是通过域名就不能访问。域名服务器就是存了一张很大的表，进行域名到 ip 的翻译。比较多的是一对一映射（1-1）。也可以一个域名对应多个 ip 地址（M-1），因为服务比较大、比较重要，就可以通过服务分发的时候，把任务分发到不同的服务器。域名可能不对应任何 IP 地址（1-?）。

Localhost 可以模拟是一台机器，可以完全通过网络这条路进行链接，其实质就是自己。

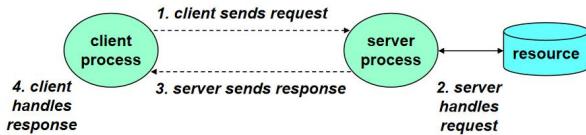
这里有一个概念叫做 Internet connection，两个 host 之间发消息，要先建立链接，再在这个链接之上发消息。这个动作是要在网络刚开始的时候要去做的。这个默认的链接就是 socket，其由两个部分组成，ip address 和 port。很多进程要使用一块网卡来发送/接收消息。消息到底发给哪个进程，需要进行一个细分，不同的服务进程可以占用不同的端口号。发送的对象具体到一个地址+一个 port。

Ip 地址和 port 都是有限的。Port 要建立一些划分的，有一些广泛认知的 port 对应了一些基础的协议。不同机器必须使用同一个 port，叫做 ephemeral port。比较有名的 port 号就是 80，如果是 <http://> 就对应了 80 端口。Smtp 和 ftp 也有固定的 port 号。

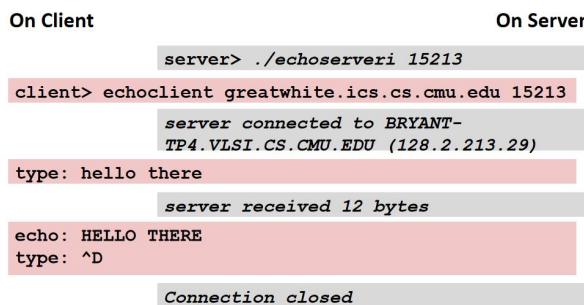


可以把这个 socket pair 抽象成一个文件。

我们回顾一下 CS 架构：



服务器通过客户端的请求返回响应的消息。浏览器就是客户端的一个进程。Nginx 就是服务器端的一个 web server。IP(32 位)和 port(16 位)构成了一个唯一标识。客户端和服务器就是按照这一套协议进行交互的。浏览器就是 http client，只是把收到的东西美化了出来。在 server 端，有 daemons (守护进程)。Server 是 7*24 等待 client 来连接的。当机器开机的时候，就会把这个机器所要提供的服务运行起来。提供 init 写一些脚本把服务开起来。



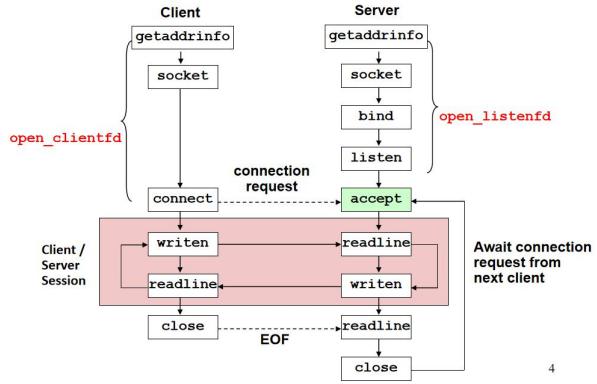
Echo server 就像是一个应答器。Client 说什么，server 就会回复什么（经过网络）。打开回声服务器，指定端口打开。echoclient 先去连这个 echoserver，建立完成链接以后。对于 echo protocol 协议，就是收到什么返回什么。

2021/4/20

实现一个 Echo Server

今天我们介绍网络相关的一些编程接口，并且以这些接口为范例，实现一个简单的 Client-Server 的程序。这个程序我们之前见过，就是 echo server，在我们这个程序里需要经过网络。client 读取输入，通过网络发送到 server 端，server 再 echo 一个返回。我们来看这样的一个程序这么构建，肯定有一个 library。在网络下面，这个 library 就是 socket，有点像 unix I/O，也是一个建立在系统调用上的库。

首先我们简单了解一下 socket，这个词翻译成套接字。它最早是伯克利在自己的操作系统中使用的一套库，经过一段时间以后，它成为了一个事实上的使用的一套库。大量网络编程的库，都是基于 socket 之上的。它的一个很重要的特点就是“协议独立性”，我们知道在网络里有很多不同的协议，在不同的层里。在 TCP/IP 之上有应用层，在核里面封装的就是不同的协议。在层里有不同协议的层的实现。socket 提供了一个相同的接口来包装下层不同协议的实现，优点等于给不同的协议提供了一个一致的接口。所以我们只要使用一套库就行了。它是在 user level 的 library。



这个图描述了整个网络里面会涉及到的最主要的功能，左右两条线是客户端、服务器上的流程。

1.getaddrinfo 函数：这个函数将主机名、主机地址、服务名和端口号的字符串转化成套接字地址结构。

```
int getaddrinfo(const char *restrict nodename, /* host 或者IP地址 */
    const char *restrict servname, /* 十进制端口号 或者常用服务名称如"ftp"、"http"等 */
    const struct addrinfo *restrict hints, /* 获取信息要求设置 */
    struct addrinfo **restrict res); /* 获取信息结果 */
```

给定 host 和 service, getaddrinfo 返回 result, result 是一个指向 addrinfo 结构的链表
IPv6 中引入了 getaddrinfo(), 它是协议无关的既可用于 IPv4 也可用于 IPv6。它能够处理名字到地址以及服务到端口两种转换, 返回的是 struct addrinfo 的结构体链表。这些结构体可由套接字函数直接使用, 这样 getaddrinfo 函数把协议相关性的安全隐藏在库函数的内部。

2.socket 函数 int socket(int domain, int type, int protocol);

客户端和服务器使用 socket 函数来创建一个套接字描述符(socket descriptor), 但是 socket 函数返回的 clientfd 描述符只是部分打开的, 还不能用于读写。

3. connect 函数 int connect (int clientfd, const struct sockaddr *addr, socklen_t addrlen);

connect 函数尝试与套接字地址为 addr 的服务器建立一个 internet 连接。其中 addrlen 是 sizeof(sockaddr_in)。Connect 函数会阻塞, 直到建立连接或者出错。如果成功, clientfd 描述符现在就准备好可以进行读写了。得到的是套接字对 (x : y, addr.sin_addr : addr.sin_port) , 其中 x 是客户端的 IP, y 是唯一对应客户端主机的客户端进程的临时端口号; addr. 就是对应服务器的 IP 地址和端口号。

4. bind 函数 int bind (int sockfd, const struct sockaddr *addr, socklen_t addrlen);

服务器使用 bind、listen 和 accept 函数和客户端建立连接。bind 函数告诉内核将 addr 中的服务器套接字地址和套接字描述符 sockfd 联系起来。参数 addrlen 就是 sizeof(sockaddr_in)。对于 socket 和 connect, 最好的方法是用 getaddrinfo 来为 bind 提供参数。

5. listen 函数 int listen (int sockfd, int backlog);

默认情况下, 系统会认为 socket 函数创建的描述符对应于主动套接字 (active socket) , 也就是客户端使用的。服务器调用 listen 函数告诉系统, 描述符是作为服务器使用的。listen 函数将 sockfd 从一个主动套接字转化为一个监听套接字(listening socket), 该套接字可以接受

来自客户端的连接请求。`backlog` 参数是系统开始拒绝连接请求之前，队列中要排队的未完成的连接请求的数量。

6. `accept` 函数 `int accept (int listenfd, struct sockaddr *addr, int *addrlen);`

`accept` 函数等待来自客户端的连接请求到达监听套接字 `listenfd`，然后在 `addr` 中填写客户端的套接字地址，并返回一个已连接描述符(`connected descriptor`)，这个描述符可被用来使用 Unix I/O 函数与客户端通信。

监听套接字描述符和已连接描述符的区别：

监听描述符是作为客户端连接请求的一个端点，它通常只被创建一次，并且存在于服务器的整个生命周期。而已连接描述符是客户端和服务器之间已经建立起来了的连接的一个端点。服务器每次接受连接请求时都会创建一次，它只存在于服务器为一个客户端服务的过程中。我们区分开这两者，允许我们可以建立并发服务器，它能够同时处理许多客户端连接。每次一个请求到达监听描述符时，我们可以 `fork` 出一个新的进程，它通过已连接描述符与客户端通信。

前面这个阶段是准备阶段，我们可以看到准备阶段上客户端和服务器要做的事情是不一样的。在 `client` 要去打开一个和 `server` 的链接，对于打开的网络链接抽象成文件，也是得到一个 `fd`，我们包装成 `open_clientfd`。服务器端，要起一个服务器，然后去等待请求。两端都对应了一个 IP 地址+端口。在 `client` 和 `server` 建立联系的时候，要分别提供对应的 ip+port，

红色部分就是完成连接以后进行的通讯。任何一端只有两个操作，读和写。需要约定好，发过去对面要及时接收。这也是 TCP/IP 等协议需要提供的一个数据传输的约定。再简单的服务之间也会有一套协议，比如我们的 `echo server` 也有这个协议。比如 `client` 发送一行等待 `server` 响应，`server` 回一句消息，期间不收新的消息。协议还会规定消息里面的格式，以及发送具体的内容。我们今天讲完以后，会重点讲 `http` 协议。我们会发现命令行的 `http` 协议和浏览器看到的 `http` 的结果是完全不一样的，之后我们要自己去实现对 `http` 协议的处理。

C/S 会话结束了以后，`client` 就关闭了。但是 `server` 端要有一个循环到上面，因为我们的 `client` 会随时发给 `server` 一个请求。所以 `server` 端一直会有这个进程(叫做 `daemon process`, 守护进程)，等待新的请求。这里有个绿色的函数 `accept`，有点像 `waitpid` 函数的行为，它是要等待 `client` 连接，如果没有 `client` 就会一直等下去。

`socket` 想尽量把文件 IO 和网络 IO 统一在一块。`socket descriptor` 和 `fd` 的含义差不多，让应用程序做对应的读写。`CS` 都能在连接上进行读写的，叫做双工。有些单向的只能往里面写或者读。

`socket` 的打开方式，不仅和文件的打开不同，`CS` 两端的 `socket` 打开也不同。`socket` 返回的是 `sockaddr`，第一个是 `family` 告诉你，返回的是哪个协议簇(协议家族)，比如 TCP/IP 等。下面的是 32 位的地址和 `port`。还有 `sin_zero[8]` 是填充的。

通用的 `sockaddr` 最多有 14 个 byte 的地址。

可以用 C 语言把通用的 `sockaddr` 转化为 `sockaddr_in` (TCP/IP 使用的 IP 地址，只使用了 6 个 byte，剩下的 8 个 byte 是空的)

```

1  #include "csapp.h"
2
3  int main(int argc, char **argv)
4  {
5      struct addrinfo *p, *listp, hints;
6      char buf[MAXLINE];
7      int rc, flags;
8
9      if (argc != 2) {
10          fprintf(stderr, "usage: %s <domain name>\n", argv[0]);
11          exit(0);
12      }
13
14      /* Get a list of addrinfo records */
15      memset(&hints, 0, sizeof(struct addrinfo));
16      hints.ai_family = AF_INET; /* IPv4 only */
17      hints.ai_socktype = SOCK_STREAM; /* Connections only */
18      if ((rc = getaddrinfo(argv[1], NULL, &hints, &listp)) != 0) {
19          fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(rc));
20          exit(1);
21      }

```

下面，我们首先介绍两个函数，一个是 `getaddrinfo` 和 `getnameinfo`。我们需要先从域名从 DNS 查到对应的 `address`。也就是输入一个域名，返回的是一个 `address` 的 list。有了 IP 地址，就可以提供给 `socket` 来建立连接。`server` 端可以调用 `getnameinfo`，从客户端的 IP 反推出域名或者 `hostname`。

2021/4/25

网络牵涉到发射方和接收方，需要有约定获取数据的步骤和格式。在 `client socket` 我们介绍 `http` 协议。`Web client` 向 `web server` 发送请求，`webserver` 负责返回网页（包括静态网页和动态网页）。动态网页是根据不同人的请求返回不同的网页。

http 协议

我们需要了解简单的 `http` 协议，然后看一下支持这个 `http` 协议的 `web server`（可以接受不同的 `web` 请求，使用不同的浏览器都可以连这个 `web server`）怎么实现。`http` 协议就是请求网页，协议本身的基础部分很简单，只是约定了发送出去的格式或者接受的格式。这边展示的是 `client` 向 `server` 发出请求的格式。`Web client` 只需要构建一个连接，然后向 `server` 请求，得到一个返回。

```

//Client: open connection to server
unix> telnet ipads.se.sjtu.edu.cn 80
//Telnet prints 3 lines to the terminal
Trying 202.120.40.88...
Connected to ipads.se.sjtu.edu.cn.
Escape character is '^>'.
//Client: request line
GET /courses/ics/index.shtml HTTP/1.1
//Client: required HTTP/1.1 HOST header
host: ipads.se.sjtu.edu.cn
//Client: empty line terminates headers

```

首先用 `telnet` 构建一个连接，无非就是告诉它域名和端口。这个和 `echo client` 所做的事情差不多。蓝色部分是 `telnet` 的一些输出（其调用 `getaddrinfo` 拿到域名所对应地址），告诉你建立已经成功了，然后等待你的输入（按 `ctrl+c` 输入）。

后面两行是你输出的一些请求，发送到 web server 端，webserver 会按照规则来解析你的请求。第一个是个 GET 操作，后面是请求的网页的路径（是 server name 的路径），你所打的网址也就是前面一段是建立连接的，后面一段是找资源的。然后是 HTTP 的 1.1 版本。下面是一些辅助信息，叫做 headers。对于初级的 http 协议，不支持对 header 的处理，这些直接丢弃。这个命令很简单，就是向 web server 要一个静态网页。

- HTTP request is a **request line**, followed by zero or more **request headers**
- Request line: **<method> <uri> <version>**
 - **<version>** is HTTP version of request (HTTP/1.0 or HTTP/1.1)
 - **<uri>** is typically URL for proxies, URL suffix for servers.
 - A URL is a type of URI (Uniform Resource Identifier)
 - See <http://www.ietf.org/rfc/rfc2396.txt>
 - **<method>** is either GET, POST, OPTIONS, HEAD, PUT, DELETE, or TRACE.

这是 http 里对请求的约定，一个是 request 部分，下面可以有多行的 request headers。Request 包含三个部分，由空格隔开。Method 最简单的是 get 操作，然后是网页的路径，然后是协议的版本。我们的 tiny web server 也只支持这一种请求。

Get 和 Post 都是最基本的取网页，大部分请求都是 GET 请求，POST 请求可以把数据包含在请求里面，往往是提交一些表单等。

Request header 相当于环境变量，可以用 header name: header data 的格式。

```
//Server: response line
HTTP/1.1 200 OK
//Server: followed by five response headers
Server: nginx/1.0.4
Date: Thu, 29 Nov 2012 10:15:38 GMT
//Server: expect HTML in the response body
Content-Type: text/html
//Server: expect 11,560 bytes in the resp body
Content-Length: 11560
//Server: empty line ("\\r\\n") terminates hdrs
```

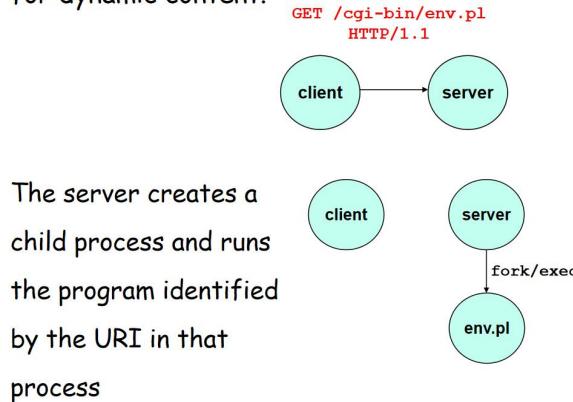
请求的返回是一个字符串流，相当于收到一封回信一样。它会有一个头，头部告诉你返回是正确的，比如响应码是 200。下面会带一些像 header 一样的状态信息，可以根据这个 header 做一些操作，或者直接忽略掉。

- HTTP response is a **response line** followed by zero or more **response headers**.
- Response line: **<version> <status code> <status msg>**
 - **<version>** is HTTP version of the response.
 - **<status code>** is numeric status.
 - **<status msg>** is corresponding English text.

| | |
|--------------------------------|--|
| 200 OK | Request was handled without error |
| 301 Moved | Provide alternate URL |
| 403 Forbidden | Server lacks permission to access file |
| 404 Not found | Server couldn't find the file |
| 501 Not Implemented | Server does not support the request method |
| 505 HTTP version not supported | Server does not support version in request |
- Response headers: **<header name>: <header data>**
 - Provide additional information about response
 - **Content-Type:** MIME type of content in response body
 - **Content-Length:** Length of content in response body

响应 header 的最有用的信息就是 Content-Type 和 Content-Length。我们 web client 就会从 web server 拿到文本格式的 html 页，因为我们没有渲染功能。我们下面更多地关注 web server 怎么去实现。

- Client sends request to server
- If request URI contains the string "/cgi-bin", then the server assumes that the request is for dynamic content.



- The server creates a child process and runs the program identified by the URI in that process

大家通常约定动态网页放在 `cgi-bin` 目录里面。

正常情况下，`server` 会通过 `fork` 出一个子进程，然后传入参数执行。把执行完的结果，通过 `server` 返回给 `client`。

对于动态网页的来说，还存在一些问题我们不知道怎么去回答。一个是参数怎么传给 `server`，第二个是 `server` 怎么把参数传递给子进程，当然可以自己构建出 `argc` 和 `argv`，然后 `execve` 的时候自动传入参数，但是不是很灵活。第三个问题，就是子进程怎么把参数返回给 `server`，`server` 还要再去还给 `client`。这些是通过 CGI (common gateway interface) - 规范来定义的。

第一件事情，是怎么从 `client` 向 `server` 传递参数。因为只接受字符串，要把参数编码进字符串里面。`<可执行文件名>?参数 1&参数 2……` 为什么浏览器里的空格都会被转化为%20呢？因为字符串里不能有空格，否则后面的 URL 就被解析为下一个参数了。

第二件事情，怎么把参数传递给 `fork` 出来的子进程？这里采用了一个环境变量的形式，所有把我所有的参数都拷贝给 `QUERY_STRING`，`QUERY_STRING` 后面的解析就交给被调用的子进程去解析。

```

if ((buf = getenv("QUERY_STRING")) != NULL) {
    p = strchr(buf, '&');
    *p = '\0';
    strcpy(arg1, buf);
    strcpy(arg2, p+1);
    n1 = atoi(arg1);
    n2 = atoi(arg2);
}

```

如果程序是被网络调用的，是个标准的 web cgi 程序的话，会解析这个环境变量。此外还有一些其他的信息，也是通过环境变量的方式，只是环境变量的名字修改了。

- Request-specific
 - QUERY_STRING (**contains** GET args)
 - SERVER_PORT
 - REQUEST_METHOD (GET, POST, etc)
 - REMOTE_HOST (**domain name of client**)
 - REMOTE_ADDR (**IP address of client**)
 - CONTENT_TYPE (**for POST, MIME type of the request body**)
 - CONTENT_LENGTH (**for POST, length in bytes**)

如果我们要这些信息，我们可以在 CGI 程序中进行解析。之前是直接把?之后的进行截断，然后赋给 QUERY_STRING，只需要按照一个规定动作来做，让 Server 做的事情变少，这样就能尽可能支持了可扩展性（server 的主题所做的事情尽可能要少）。

第三件事情，子进程如何返回值，我们知道子进程返回给 server，server 再返回给 client 是不可扩展的。因为可能同时一百个子进程返回，导致数据流都集中在 server 上，使得可扩展性变低。CGI 程序根本不需要把返回值传递给 Server，只要让 server 把 connectfd 传递给子进程。事实上子进程是复制了一份 connectfd，子进程可以直接把结果返回给 client，直接和 client 联系，这样 server 不会成为一个性能的瓶颈。对于 child 来说，采用了重定向的方式。Server 在 fork 子进程之前，子进程会默认打开三个文件（标准输出/输入/错误输出），在 fork 前，把 connectfd dup 成标准输出。子进程是不会使用 stdout 的，只要默认的使用 printf 打印，会被重定向到 connectfd 上。这样的好处是直接地在 server 端看到结果，全部调试完了以后，再重定向到 connectfd 上即可。

Tiny Web Server

下面，我们介绍 tiny web server 的简单实现，它只支持文本格式和 GET 命令，功能比较完整。Web server 无非就是解析 request，先判断是否是动态的还是静态的。如果是静态的，就创建 CGI 子进程，为其准备好参数，帮其重定向好标准输出。

```

16 int main(int argc, char **argv)
17 {
18     int listenfd, connfd;
19     char hostname[MAXLINE], port[MAXLINE];
20     socklen_t clientlen;
21     struct sockaddr_storage clientaddr;
22
23     /* check command line args */
24     if (argc != 2) {
25         fprintf(stderr, "usage: %s <port>\n", argv[0]);
26         exit(1);
27     }
28
29     listenfd = open_listenfd(argv[1]);
30     while (1) {
31         clientlen = sizeof(clientaddr);
32         connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
33         Getnameinfo((SA *)&clientaddr, clientlen, hostname, MAXLINE,
34                     port, MAXLINE, 0)
35         printf("Accepted Connection from (%s, %s)\n", hostname, port);
36         doit(connfd);
37         Close(connfd);
38     }
39 }
```

Main 函数上来，server 要启动，指定一个 port。拿到 port 以后用 openlistenfd 启动。主体函数和原来的 echo server 是差不多的。Session 结束一个 close (connfd)。差别只是在 doit 函数上。

36

用 robust-io 来初始化。可以读到 method、uri 和 version。解析完先检查 method 是不是 get。这是一个网络请求，不管对错都应该返回一个 http response，所以我们要返回一个带有出错信息的 response（用 clienterror 函数）。下面会调用 read_requesthdrs(&rio); 来处理 header，不读是不行的，因为这些东西还在网络上，我们这里读完直接扔掉就可以了。根据读到的 uri 来判断是静态的还是动态的，也就是判断有没有 cgi-bin 目录。如果没有找到文件，就返回 404。

静态这边就是要确定找到并且文件有对应的 user 可读权限。过了这个检查以后调用了 `serve_static` 传入文件名，文件大小传进去。

对于动态的，要求的就是可执行文件的执行文件，因为它所要的是程序执行的结果。采用一次连接只返回一个 `response` 的方式，可以服务更多的人。具体我们来看函数的具体实现。

```
1 void clienterror(int fd, char *cause, char *errnum,
2                   char *shortmsg, char *longmsg)
3 {
4     char buf[MAXLINE], body[MAXBUF];
5
6     /* build the HTTP response body */
7     sprintf(body, "<html><title>Tiny Error</title>");
8     sprintf(body, "%s<body bgcolor=\"ffffff\">\r\n", body);
9     sprintf(body, "%s%s: %s\r\n", body, errnum, shortmsg);
10    sprintf(body, "%s<p>%s: %s\r\n", body, longmsg, cause);
11    sprintf(body, "%s<hr><em>Tiny Web server</em>\r\n", body);
12
13    /* print the HTTP response */
14    sprintf(buf, "HTTP/1.0 %s %s\r\n", errnum, shortmsg);
15    Rio_writen(fd, buf, strlen(buf));
16    sprintf(buf, "Content-type: text/html\r\n");
17    Rio_writen(fd, buf, strlen(buf));
18    sprintf(buf, "Content-length: %d\r\n\r\n", strlen(body));
19    Rio_writen(fd, buf, strlen(buf));
20    Rio_writen(fd, body, strlen(body));
21 }
```

这是出错的实现，参数是原因，error member。返回 http version 然后是 error message。长度就是 body 的长度。先发 header 部分，再发 body 部分。

```
1 void read_requesthdrs(rio_t *rp)
2 {
3     char buf[MAXLINE];
4
5     Rio_readline(rp, buf, MAXLINE);
6     while(strcmp(buf, "\r\n")){
7         Rio_readlineb(rp, buf, MAXLINE);
8         printf("%s", buf);
9     }
10    return;
11 }
```

循环读 header，读到一个空行为止，表示请求结束了。这里是直接把 header 丢掉。

```

1 int parse_uri(char *uri, char *filename, char *cgiargs)
2 {
3     char *ptr;
4
5     if (!strstr(uri, "cgi-bin")) { /* static content */
6         strcpy(cgiargs, "");
7         strcpy(filename, ".");
8         strcat(filename, uri);
9         if (uri[strlen(uri)-1] == '/')
10            strcat(filename, "home.html");
11         return 1;
12     }
13     else { /* dynamic content */
14         ptr = index(uri, '?');
15         if (ptr) {
16             strcpy(cgiargs, ptr+1);
17             *ptr = '\0';
18         } else
19             strcpy(cgiargs, "");
20         strcpy(filename, ".");
21         strcat(filename, uri);
22         return 0;
23     }
24 }
25 }
```

是否包含 cgi-bin，如果包含就是动态的，否则就是静态的。路径是 web server 管理的子目录的路径，所以要在原来的 url 的相对路径的基础上，最前面加一个点，代表是当前目录的相对路径。静态的话，默认使用 home.html；动态的话，如果有?的话，把?后面全部截下来作为参数，传给 CGI 去解决。

```

1 void serve_static(int fd, char *filename, int filesize)
2 {
3     int srcfd;
4     char *srcp, ftype[MAXLINE], buf[MAXBUF];
5
6     /* send response headers to client */
7     get_filetype(filename, ftype);
8     sprintf(buf, "HTTP/1.0 200 OK\r\n");
9     sprintf(buf, "%sServer: Tiny Web Server\r\n", buf);
10    sprintf(buf, "%sConnection: close\r\n", buf);
11    sprintf(buf, "%sContent-length: %d\r\n", buf, filesize);
12    sprintf(buf, "%sContent-type: %s\r\n\r\n", buf, ftype);
13    Rio_writen(fd, buf, strlen(buf));
14    printf("Response headers:\n");
15    printf("%s", buf);
16
17    /* send response body to client */
18    srcfd = Open(filename, O_RDONLY, 0);
19    srcp = Mmap(0, filesize, PROT_READ, MAP_PRIVATE, srcfd, 0);
20    Close(srcfd);
21    Rio_writen(fd, srcp, filesize);
22    Munmap(srcp, filesize);
23 }
```

- 25 /*
- 26 * get_filetype - derive file type from file name
- 27 */
- 28 void get_filetype(char *filename, char *filetype)
- 29 {
- 30 if (strstr(filename, ".html"))
- 31 strcpy(filetype, "text/html");
- 32 else if (strstr(filename, ".gif"))
- 33 strcpy(filetype, "image/gif");
- 34 else if (strstr(filename, ".png"))
- 35 strcpy(filetype, "image/png");
- 36 else if (strstr(filename, ".jpg"))
- 37 strcpy(filetype, "image/jpg");
- 38 else
- 39 strcpy(filetype, "text/plain");
- 40 }

对于静态网页的处理，我们已经判断过这个存在并有权限，我们只需要把这个文件嵌入到返回的 body 即可，不需要读文件了。我们也要构造响应的访问 header。比如关闭 connection，如果要再访问的话要重新建立连接。可以通过 get_filetype 来获取文件类型。Mmap 可以把一个文件直接映射到内存空间，把映射得到的内存地址直接作为网络包发送出去。

get_filetype 里面，凡是不认识的，都作为 text/plain 发送。

```

1 void serve_dynamic(int fd, char *filename, char *cgiargs)
2 {
3     char buf[MAXLINE];
4
5     /* return first part of HTTP response */
6     sprintf(buf, "HTTP/1.0 200 OK\r\n");
7     Rio_writen(fd, buf, strlen(buf));
8     sprintf(buf, "Server: Tiny Web Server\r\n\r\n");
9     Rio_writen(fd, buf, strlen(buf));
10
11    if (Fork() == 0) { /* child */
12        /* real server would set all CGI vars here */
13        setenv("QUERY_STRING", cgiargs, 1);
14        Dup2(fd, STDOUT_FILENO); /* redirect output to client*/
15        Execve(filename, NULL, environ); /* run CGI program */
16    }
17    Wait(NULL); /* parent reaps child */
18 }

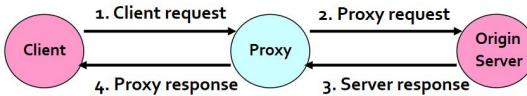
```

动态的处理，我们关注创建子进程执行文件的部分。这牵涉到 header 部分谁来返回。我们发现 header 的前面两行是由 server 来返回的，下面的子进程返回的只有 Content-length 和 Content-type 以及 body 部分。在执行之前，我们要传参数进 QUERY_STRING，并且用 connectfd 来重定向标准输出，把标准输出替换为 client 端的链接。

Proxy

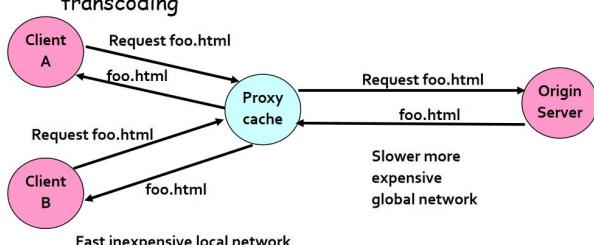
Proxies

- A **proxy** is an intermediary between a client and an origin server.
 - To the client, the proxy acts like a server.
 - To the server, the proxy acts like a client.



下面我们要介绍第九个 lab 的内容，这个 proxy 是一个网络程序。它是介于 client 和 server 之间的。对于客户端来说，它的请求发给 proxy，proxy 返回给它，proxy 就是一个服务器。对于服务器来说，它的请求发给 server，server 返回给他，proxy 就是一个客户端。所以，proxy 是同时需要有客户端和服务器的能力。

- Can perform useful functions as requests and responses pass by
 - Examples: Caching, logging, anonymization, filtering, transcoding



它有什么用呢？1.proxy 可以做 cache，第一次请求的时候发给 proxy，proxy 没有资源，proxy 向视频网站请求后，把资源返回，并在本地存一份。第二个请求来的时候，就不需要再去连接服务器了。这可以节省网络流量。2.proxy 可以记录 log，可以监视客户端请求，比如一个网络不允许打游戏。因为 proxy 是转发的东西，所以访问什么东西都可以知道。3. 区域匿名化，有了 proxy 以后，服务器都只知道是 proxy，而不知道是谁。4. 可以做一些过滤，返回的信息，如果有不良信息，proxy 可以过滤掉。网络上传递的数据可以分为 pay-load 和

header，禁掉的可能是 twitter 的 header。我们可以 header 里面去访问 proxy，proxy 接收到以后帮我们加上 twitter，就可以正常访问 twitter 了。

各种网络上的设备，比如防火墙什么都可以使用 proxy 实现。网络部分就介绍到这边，我们要关系的就是怎么用 socket 完成网络协议的实现，我们也介绍了 http 的 webserver 的实现。

第十二章：并发编程

这是一种能力，原来的程序可以写成并行的。比如 server 接受 client 请求，原来是串行的，如何变成并行的环境呢。我们的 OS 就是分时操作系统，一个程序的步骤之间可以进行并行，来获得更好的用户体验。

并发程序有多种，大家比较熟悉的是多进程的并发。我们之前说到后台运行的 console，可以通 fork 来完成多进程的并行。

我们要介绍进程、线程、和 I/O multiplexing (I/O 多路复用技术) 这三种实现并行编程的方法。

硬件的 CPU 可以执行多个程序，也是并发。硬件会使用这种并行来提高响应。在我们之前的介绍里，我们介绍进程的 handler 的时候，我们有两个执行流（main 函数和 handler 存在同时执行的情况）。这个过程操作系统支持硬件来的一些中断，这个过程也是并行。我们关心的是大量的并行程序是怎么 work 的，有什么样的好处。

并行的好处：
1. 可以处理各种异步事件（你不知道什么时候回来，来的时候想抓紧时间处理掉，最好还不要影响程序的执行，eg: handler）
2. 希望支持多 CPU 的使用，一个程序跑在多个 CPU 上来利用更多的资源。
3. 解决一些异步设备 (I/O 设备) 读写比较慢的情况。
4. 和用户打交道的情况，因为人的输入比较慢，在人输入的时候希望做一些自己的事情。
5. 网络如果按部就班一个个处理，就不能解决多人的情况。操作系统定时地轮询不同的程序，进行分配。因为操作有快有慢，我们不希望慢的操作把快的操作挡住了。

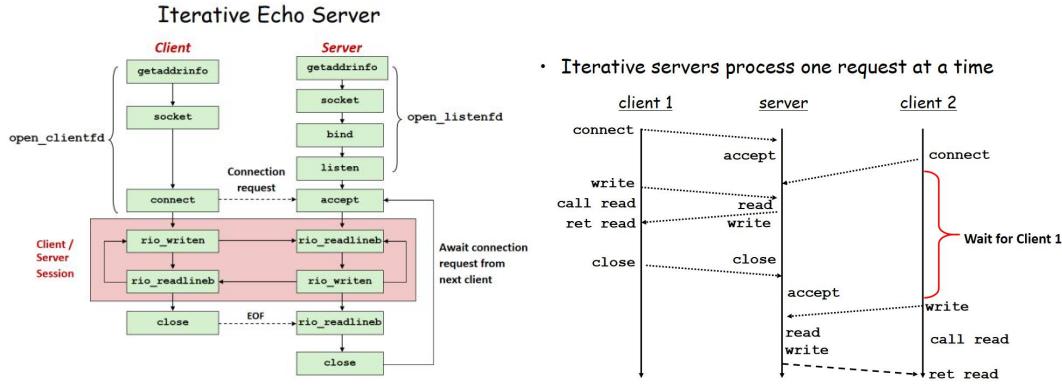
自己实现比较困难，最好是操作系统提供了一个调度接口，让操作系统帮我们去实现调度。这种就叫做并发编程。

从进程到线程到 I/O 多路复用技术，越来越轻量级。但是这些并行都是应用程序层级的，比如我们希望一个 web server 实现同时和多个 client 去打交道。*concurrency 的程序实现起来比较困难，在实现的时候，要把自己的思考方式变成一个并行的程序，程序容易出错，比如调度的顺序不确定，光这一点就会引起很多的问题。比如，竞争问题，死锁问题，饥饿问题。

2021/4/27

应用层实现并发的方法

我们如何利用更多的资源在应用层实现并发，我们可以使用进程、线程、和 I/O multiplexing (I/O 多路复用技术)。在网络这节课里，我们介绍了 CS 架构的 echo server 服务器，我们如何实现一个并发的 server 呢？我们发现之前的 server 需要一个一个地处理用户的请求。



在循环里面是一个个处理的，处理完了关闭连接以后，再获得下一个连接的 fd。因为现在服务前一个 client 的时候，后一个来的请求只能等待，需要等处理掉前一个请求，客户端才能得到下一个 fd。我们希望最终，我们的服务器可以同时地服务多个客户端。解决的思路也是分时复用，因为处理一个请求的时候，其实大部分时间也在等一个会话里的下一个网络包过来。我们强调了在 server 端和客户端建立连接，主要分成两种，一个是 `listenfd`。

1. 进程

我们可以创建多个进程来实现一个并行的 server。基本想法就是有 client 连的时候，就分配一个子进程，不管要静态的还是动态的网页，都丢给子进程去处理，不需要主进程再参与了。这种就叫做多进程的方式。只受到操作系统进程数和资源的限制。

这种相对来说是最简单的，原来的三个操作我们放在了 `fork` 的 if 里面。`Connfd` 会被关闭两次，因为在 `fork` 的时候，会把主进程的操作 `copy` 一份。子进程同时也开着 `listenfd` 和 `connectfd`。`Fork` 出来的，父进程和子进程的 `listenfd` 最终指向的是同一个东西。所以要先关掉子进程的 `listenfd`。在 `fork` 完，`connfd` 就对父进程没有用了，可以立马关掉。父进程开始处理下一个请求。也就是说，这些资源要及时的回收，因为这是守护进程（long-running），要仔细地关注资源。整个代码在原来的基础上进行简单的修改，就可以实现了进程级别的并发。这些创建的子进程会被操作系统来调度完成。

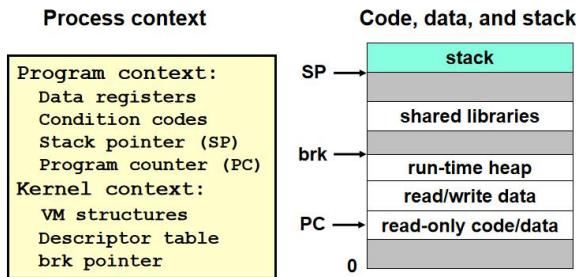
这个方案的问题就在，如果父进程要和子进程通信，采用这个模式就比较困难。需要专门建立一些传递信息的通道，叫做 `interprocess connection` 机制。需要在代码中手动实现，比如在 `fork` 前打开一个 `fd`。`signal` 会打断 `accept`，回来的时候，我们需要这个 `accept` 能够恢复。还有就是我们要写正确的 `signal handler`，去处理它们，如果我们没有正确的写，资源如果没有被及时回收，服务器很快就充满了僵尸进程挤占资源，导致出问题。

优点：可以实现并发；显式地完成共享，相对来说 bug 少一些（共享的只有 file table）；实现比较简单和直接。

缺点：在每个 client 来的时候，就要创建一个进程，要准备很多东西，额外的代价比较大。随着创建的进程越来越多，调度队列比较长，回收也比较难。对于轻量级的服务，可能大量的成本都在创建进程和销毁进程上。而且针对现在大多数请求都是短连接（一次连接只有一个来回）的情况，有非常大的额外开销。父子进程共享数据比较困难。需要 IPC（interprocess communication），比如 FIFO 队列，或者让两个进程映射到同一块共享内存。

针对这个缺点，我们想能不能实现一个更轻量级的单位，我们只希望在一个进程里实现多个

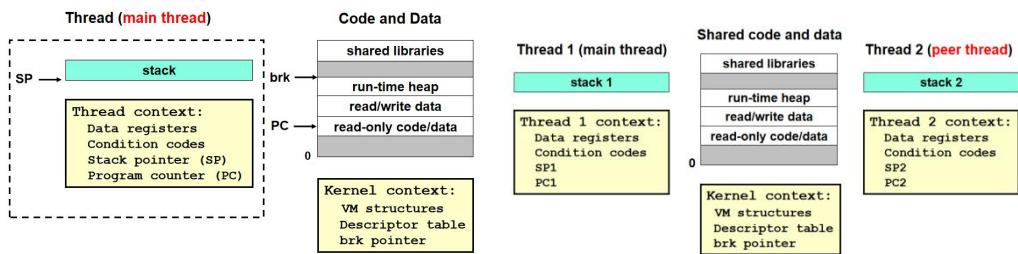
执行流，这时候我们就需要搞清楚哪一些是和执行流相关的，这就是线程。一个进程可以创建多个线程。我们要搞清楚哪些东西是线程内部独立的，哪些是依赖于进程共享的，我们希望线程内独立的部分越小，这样 overhead 更小。



一个 Context 代表了运行的一个程序。进程级的 context 分成 program 的 context(寄存器, CC, 还有 stack point, 还有 program counter(PC)), 还有一些 vm structure (一个进程, 操作系统会为你配一些记录进程相关的内存信息), descriptor table, brk pointer (堆指针)。

这一块就是进程级别的上下文 (process context) 的全部内容。还有一些在内存里面, 比如你 malloc 出来的空间。一个执行中的进程所有的东西全部包括在这里了。

- Process = thread + code, data, and kernel context

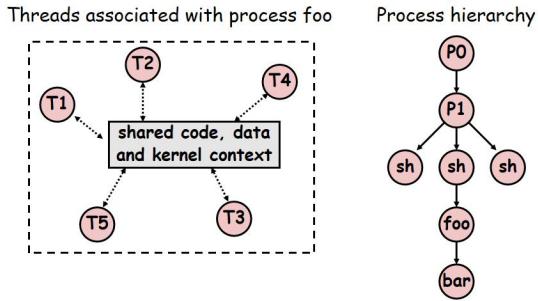


我们就要搞清楚这个进程的东西，哪些部分是要复制多份的。举个例子来说，每一个线程在执行的时候，要有自己的寄存器、PC、CC、栈顶指针（栈是代表了线程执行的整个不同的过程），所以每个线程也得有自己的 stack。这些东西，每个线程都要有一份，才能并发执行。而剩下的东西是线程可以共享的，比如全局变量、共享的库等（凡是必须的，我们都采用共享）。

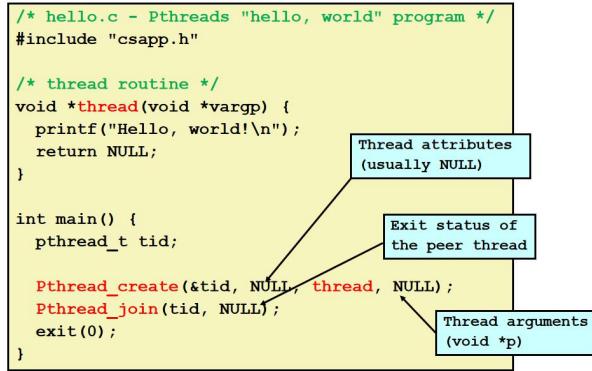
2.线程

有了这个概念以后，每个进程至少包括一个线程，叫做 **main thread**。我们现在可以在进程中创建多个线程，比较占空间的东西都会只维护一份。线程虽然小，但是可以执行不同的代码。每个线程都有自己逻辑的执行流，每个进程还会为线程维护自己的名字 (**thread id, TID**)。

- Threads associated with process form a pool of peers
 - Unlike processes which form a tree hierarchy

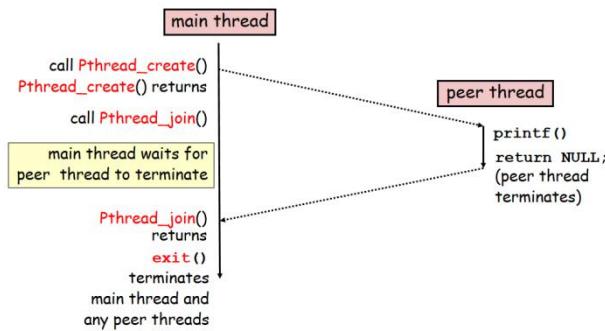


从 init 开始，都是从一个进程 fork 出来的，是一棵往下面长树状的结构。Thread 模式，就是其中的一个进程变成了方块，有很多很多的线程连着它。我们来看一些线程写出来的代码长什么样。我们希望 thread 模式有一定的通用性，所以提出了 Posix threads，使得多线程的代码有通用性。Linux 有 pthreads 一套库，封装了一些 syscall，用到的最主要的是 pthread_create、pthread_join(等待回收线程，类似于 waitpid),pthread_exit (线程退出，不要轻易地使用 exit 函数，这会导致所有的线程全部结束)



我们可以给一个 `thread` 指定一个开始的函数，我们可以把一个函数指定给多个 `thread`，只要求函数的声明是一样的，只能传入一个参数 `void*`，可以通过 cast 进程转换。`Pthread_create`，第一个参数可以带回创建的线程 ID，第二个参数用来传配置信息，第三个是执行的函数，第四个是传递给函数的参数。创建一个线程并指定它从 `thread` 这个函数开始执行。`Pthread_join` 就是等待 `tid` 这个线程退出。

线程可以认为是一个独立的控制类，一个时间线就是一个执行流，调用 `Pthread_create` 的时候，就是创建了一个执行流，操作系统就可以调度这个执行流。这和进程级别的 `fork` 不同的地方，是代码是共享的，创建 `thread` 的时候不需要考虑这些。`fork` 必须在原来进程的基础上 `copy` 出来一份，需要从 `fork` 的那个点开始执行。而在线程这里没有代码的 `copy` 这件事情，只是大家使用了代码的不同的地方。



在上例中，主线程创建完线程以后就等待线程返回。实际上情况下，操作系统是以 `thread` 为单位进行调度的，所以线程是并行执行的。如果没有 `join`，有没有可能 `main thread` 先结束，`peer thread` 后结束呢？是完全有可能的。

```

int main(int argc, char **argv){
    int listenfd, *connfdp;
    socklen_t clientlen;
    struct sockaddr_in clientaddr;
    pthread_t tid;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    listenfd = open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(clientaddr);
        connfdp = Malloc(sizeof(int));
        *connfdp = Accept(listenfd,
                          (SA *) &clientaddr, &clientlen);
        Pthread_create(&tid, NULL, thread, connfdp);
    }
}

/* thread routine */
void *thread(void *vargp)
{
    int connfd = *((int *)vargp);

    Pthread_detach(pthread_self());
    Free(vargp);
    echo(connfd);
    Close(connfd);
    return NULL;
}

```

回到服务器多线程的情况，现在有共享的 fd，谁 close 都一样。我们把 echo 的函数放到 `thread` 里面即可。在这里我们 `malloc` 了一个 `connfdp`。我们创建了一个子线程，也不会主动去回收。我们可以采用脱钩的方式，原来是 `joinable` 的，线程分完家以后，线程就不分主次了，线程结束的时候自己帮你回收。如果是 `joinable` 状态，主线程是可以杀死创建出的线程的。

`Thread` 中，先进行一个 `cast` 得到 `connfd`。然后调用了 `Pthread_detach`（分家）。然后会 `free` 这块空间，因为所有线程共享一块资源，所以主进程 `malloc` 了，可以通过子进程 `free`。`Thread` 里 `return` 是 `thread` 结束。

线程的优点：相对于进程来说，快几倍，不需要拷贝很多的内存，是轻量级的。在 `thread` 里，模型是尽可能要共享的，还会出现设计与实现的不一致的问题。

- `Pthread_create(&tid, NULL, thread, (void *) &connfd);`

如果我们直接传递 `connfd`，会出现一个对象可以被两个 `thread` 访问。服务器创建出线程以后，如果马上出来一个新的请求，会用新的请求覆盖掉局部变量 `connfd`。此时处理的线程内，会突然发现自己的 fd 变了。

进程和线程相似的地方

都有执行流，执行各自的代码；都会并行地执行；会参与到 `context` 切换中。

不同的：

| | |
|----------|-----------------|
| 线程 | 进程 |
| 全局尽量共享 | 不共享 |
| 轻量级，开销很小 | 有很多设置，有状态的 copy |

`Fork` 的时候使用了一些机制，不会一开始全部 `copy`，而是在需要的时候，再去 `copy`。

- + Easy to share data structures between threads
 - e.g., logging information, file cache.
- + Threads are more efficient than processes
- - Unintentional sharing can introduce subtle and hard-to-reproduce errors!
 - The ease with which data can be shared is both the greatest strength and the greatest weakness of threads
 - Hard to know which data shared & with private

Thread 的优缺点：1.容易共享，直接设置一些全局变量去加上去就可以了。2.线程会更高效
 1. 非显示的共享很容易导致代码出错，并且很难复现。比如直接把 connfd 传递给新的 thread，只有在局部变量还没有被赋值的时候，第二个请求来了修改了 connfd，才能复现出这个问题。

3.I/O 多路复用

最后我们介绍第三个方式，这个希望实现一个比线程更高效的并发结构。在之前，是要创建执行流，然后交给操作系统去调度，会有性能的下降。I/O 多路复用，是有限状态机的思路。在一个线程内部去调度，我们现在只有一个栈、一套寄存器。我们需要模拟时间分片，按照状态机的思路，一个 while 里面套了一堆状态的 switch。我们这里介绍的是它的特例，在 IO 这件事上，我们可以实现在一个线程下，服务多个 client。这里用到了一个叫做 select 的 syscall。它的功能就是去监听多个 fd（包括 listenfd 和 connfd），当有 fd 有消息来了的时候，它会从 select 中退出。这个相当于是大嗓门叫唯一一个服务员，服务员会赶紧过去做完这件事，然后再等别人来叫。

Select 有点复杂，可以先不管后面三个参数，我们先关心 maxfd 和 readset。Readset 是有哪些桌子被你关注的，每一个 bit 代表一个 fd，代表着要监听的桌子的范围，1 代表要监听。

Maxfd 是要监听最大的那一桌是多少。每一次从 select 出来的时候会告诉你有哪几桌叫你了。

```
/* clear all bits in fdset. */
void FD_ZERO(fd_set *fdset);

/* clear bit fd in fdset */
void FD_CLR(int fd, fd_set *fdset);

/* turn on bit fd in fdset */
void FD_SET(int fd, fd_set *fdset);

/* Is bit fd in fdset on? */
int FD_ISSET(int fd, *fdset);
```

这是一些辅助函数。我们先举一个例子。

```
#include "csapp.h"

int main(int argc, argv)
{
    int listenfd, connfd;
    socklen_t clientlen=sizeof(struct sockaddr_in);
    struct sockaddr_in clientaddr;
    fd_set read_set, ready_set;
    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    listenfd = Open_listenfd(argv[1]);
    FD_ZERO(&read_set);
    FD_SET(STDIN_FILENO, &read_set);
    FD_SET(listenfd, &read_set);
    while(1) {
        ready_set = read_set;
        Select(listenfd+1, &ready_set,
               NULL, NULL, NULL);
        if (FD_ISSET(STDIN_FILENO, &ready_set))
            /*read command line from stdin*/
            command();
        if (FD_ISSET(listenfd, &ready_set)){
            connfd = Accept(listenfd,
                            (SA *)&clientaddr, &clientlen);
            echo(connfd);
        }
    }
}
```

它关心 `listenfd` 和 `stdin` (server 上的键盘) 这两个 `fd`。先把要监听的设置为 `readset`, 然后丢到 `select` 运行, 返回值是一个 `int`, 代表有几个人叫你, `readset` 记录了谁叫你了, 没有叫你的会全部清空。如果网络上有人叫, `accept` 进去以后直接退出, 返回 `connfd`, 然后可以去做处理网络包的事情。

这可以分时服务多种情况, 切换的足够快的时候, 所有人就会被服务到。根据不同的 `event` 去做对应的事情。设计的场景就是每一个操作必须足够地短, 以免挡住别人。如果操作比较长, 我们要自己实现一个用户的打断调度, 但是会非常复杂。这里只是展示一个最简单的, 谁叫你了, 就去服务, 然后等下一个。

Concurrent Programming with I/O Multiplexing

- An event-based approach to concurrency:
 - Maintain a pool of connected descriptors.
 - Repeat the following forever:
 - use the Unix `select` function to block until:
 - (a) new connection request arrives on the listening descriptor.
 - (b) new data arrives on an existing connected descriptor.
 - If (a), add the new connection to the pool of connections.
 - If (b), read any available data from the connection
 - close connection on EOF and remove it from the pool.

为了实现我们的 server 的 io 多路复用, 一个新的请求来了, 我们要把 `connfd` 加到 `select` 里面。如果叫的人是一个 `connfd`, 那么就要过去服务, 然后再进行等待。这样的好处就是, 服务完一个 `connection` 的请求以后, 虽然保留了 `connfd`, 但是它可以去听别人的要求, 不需要等在那里。

Concurrent Programming with I/O Multiplexing

```
#include "csapp.h"
typedef struct {/* represents a pool of connected descriptors */
    int maxfd;
    fd_set read_set;
    fd_set ready_set;
    int nready;
    int maxi;
    int clientfd[FD_SETSIZE];
    rio_t clientrio[FD_SETSIZE];
} pool;

int byte_cnt=0; /*counts total bytes received by server */
```

在这里, 因为我们要服务每一桌, 所以每一桌都要自己的一些状态。每一个 `connfd` 和对应的 `rio` 的数据机构都会被记录下来, 还有一些记录了当前的 `select` 函数最多要看几桌的一些信息。

2021/5/11

Synchronizing with semaphores

- Dijkstra's P and V operations on **semaphores**
 - **semaphore**: non-negative integer synchronization variable.
 - **P(s)**: [**while (s == 0) wait(); s--;**]
 - Dutch for "Proberen" (test)
 - **V(s)**: [**s++;**]
 - Dutch for "Verhogen" (increment)

P操作的主要动作是: [\[1\]](#)

①S减1; [\[1\]](#)

②若S减1后仍大于或等于0，则进程继续执行; [\[1\]](#)

③若S减1后小于0，则该进程被阻塞后放入等待该信号量的等待队列中，然后转进程调度。 [\[1\]](#)

V操作的主要动作是: [\[1\]](#)

①S加1; [\[1\]](#)

②若相加后结果大于0，则进程继续执行; [\[1\]](#)

③若相加后结果小于或等于0，则从该信号的等待队列中释放一个等待进程，然后再返回原进程继续执行或转进程调度。 [\[1\]](#)

2021/5/18

上节课介绍了 **thread** 是什么东西，在并发的时候它是用的非常多的。不像进程需要 **fork**，**thread** 的另一个优点可以通过共享变量的方式，但是它也会造成一些潜在的问题，比如多个 **thread** 访问同一个 **count**，没有保护的情况下就会出错，出错的原因就是汇编使得数据同时出现在寄存器和内存里面。**thread** 是通过 OS 进行调用的，会按照任意的方式进行调用。我们介绍了一种方式来判断，叫做 **concurrent** 图，**x** 和 **y** 是两个不同的 **thread** 来判断不同共享变量的访问。上节课还介绍了 OS 提供的 PV 操作，以原子的方式保护想要访问的对象。**P** 是检测，当符合条件的时候，执行操作。

今天我们来看我们写一些正确的并发编程，有一些操作来帮助我们进行 **thread** 之间的共享。我们今天会介绍生产者和消费者怎么执行。最典型之前介绍 **web** 编程的时候，要实现 并发的 **web** 的程序可以使用进程、线程、IO 多路复用，每一次创建线程还是有一些开销，我们可以预先创建足够多的 **thread** 在那边。这个就可以通过生产者和消费者。主线程收到链接以后，可以变成生产者。其中一个拿到 **connectfd**，与相应的客户端进行交互。

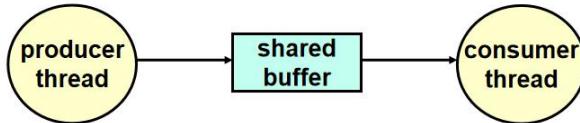
问题：生产者和消费者传递变量要正确；要保证有且只有一个消费者拿到这个 **connectfd**。还有一个 **reader** 和 **writers** 模型，这些要正确的实现都要基于对共享变量的正确的保护。我

们也需要上节课使用的 concurrency graph，我们再去会看怎么去具体地实现。

我们先来看 basic idea，PV 就是信号量来保护共享变量。凡是共享变量都应该需要保护，只有在非常清楚代码的情况下，才能设计出不保护的程序。这个信号量最早介绍过 PV 操作，在这基础上可以有不同的功能，比如 counting semaphores，binary semaphores，因为二进制信号量只允许一个 thread 进入，而我们如果把它放得更多，那么允许更多的线程进去。进去了就减下来，出来了就加上去。这样就可以衍生出不同的组合来实现不同的功能。

生产者和消费者

生产者和消费者的例子



正常场景下，是有一个 shared buffer，有两个角色，一个线程是生产者，往共享 buffer 中写入。目标是把信息从生产者传递给消费者。消费者负责从 buffer 中取出来。对于生产者来说，buffer 有容量限制，等待 buffer 中有足够的空的位置时候，就往中插入。消费者等待 buffer 中有东西了，再从中取东西。所以双方都是从 buffer 中取东西出来。我们首先看一个简单的创建，假如 buffer 只有一个 slot，生产者就检查 buffer 为空就往里插东西，消费者就是只要不空就拿出来。这时候我们就需要两个信号量来表达这件事情，full 初始化为 0，empty 初始为 1。

生产者要去 check 当前的状态是不是有空的位置，一开始初始化的时候 empty 是 1，

Initially: empty==1, full==0

Producer Thread

```
void *producer(void *arg) {
    int i, item;

    for (i=0; i<NITERS; i++) {
        /* Produce item */
        item = i;
        printf("produced %d\n", item);

        /* Write item to buf */
        P(&shared.empty);
        shared.buf = item;
        V(&shared.full);
    }
    return NULL;
}
```

Consumer Thread

```
void *consumer(void *arg) {
    int i, item;

    for (i=0; i<NITERS; i++) {
        /* Read item from buf */
        P(&shared.full);
        item = shared.buf;
        V(&shared.empty);

        /* Consume item */
        printf("consumed %d\n", item);
    }
    return NULL;
}
```

一开始生产者的 P 条件满足，把 buf 设置为 item。并且设置 V，随后消费者条件满足。轮流执行。

n 个 buffer 元素的情况，此时需要三个信号量

```

struct {
    int *buf; /* Buffer array */
    int n; /* Maximum number of slots */
    int front; /* buf[(front+1)%n] is the first item */
    int rear; /* buf[rear%n] is the last item */
    sem_t mutex; /* protects accesses to buf */
    sem_t slots; /* Counts available slots */
    sem_t items; /* Counts available items */
} sbuf_t;

```

```

void sbuf_insert(sbuf_t *sp, int item)
{
    /* Wait for available slot */
    P(&sp->slots);
    /* Lock the buffer */
    P(&sp->mutex);
    /* Insert the item */
    sp->buf[(++sp->rear)% (sp->n)] = item;
    /* Unlock the buffer */
    V(&sp->mutex);
    /* Announce available items*/
    V(&sp->items);
}

```

信号量 `mutex` 代表对 `sp->buf` 的读写操作一定是互斥的。`slots` 是 `buf` 中空闲的位置，`items` 是 `buf` 中可以被读的元素

Reader-writer 模型分为偏向 reader 的和偏好 writer 的。偏向的 reader 的就是如果有任何一个 reader 在读，一个 writer 在等待，要是此时有新的 reader 过来，因为读是不冲突的，让这个读的直接进入。

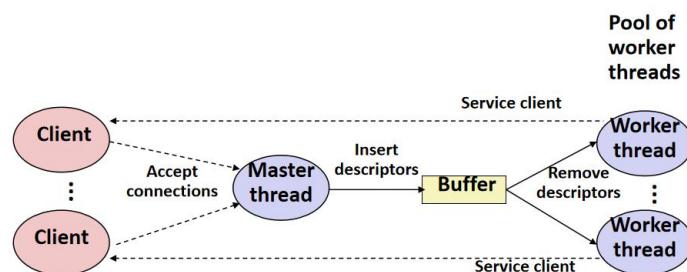
偏向 writer 的就是一旦 writer 准备写了，就尽可能快地让他执行写的这个操作。在 writer 到达之后的 reader 操作必须要等待。

Solution to First Readers-Writers Problem

| Readers: | Writers: |
|--|---|
| <pre> int readcnt; /* Initially 0 */ sem_t mutex, w; /* Both initially 1 */ void reader(void) { while (1) { P(&mutex); readcnt++; if (readcnt == 1) /* First in */ P(&w); V(&mutex); /* Reading happens here */ P(&mutex); readcnt--; if (readcnt == 0) /* Last out */ V(&w); V(&mutex); } } </pre> | <pre> void writer(void) { while (1) { P(&w); /* Writing here */ V(&w); } } </pre> |

Writer 是排他的，一定需要一个信号量进行排他地保护，只有一个 writer 可以进去。Writer 进去的时候，reader 也不能进去，`mutex` 保护的是 `readcnt`，表示当前有几个读的线程。第一个 reader 进去了以后，直到最后一个 reader 出来，期间是不能 write 的，所以 reader 那里也有 `P(&w)` 和 `V(&w)`。后面来的 reader 直接访问，不需要再去管 `w`。当所有 reader 都出来以后，就要放掉 `w` 的锁，其他的时候 `readcnt--` 直接走人就行。

Case Study: Prethreaded Concurrent Server



```

/* Remove and return the first item from buffer sp */
/* begin sbuf_remove */
int sbuf_remove(sbuf_t *sp)
{
    int item;
    P(&sp->items);                                /* Wait for available item */
    P(&sp->mutex);                                /* Lock the buffer */
    item = sp->buf[(+sp->front)% (sp->n)];      /* Remove the item */
    V(&sp->mutex);                                /* Unlock the buffer */
    V(&sp->slots);                                /* Announce available slot */
    return item;
}

void sbuf_insert(sbuf_t *sp, int item)
{
    P(&sp->slots);                                /* Wait for available slot */
    P(&sp->mutex);                                /* Lock the buffer */
    sp->buf[ (+sp->rear)% (sp->n)] = item;      /* Insert the item */
    V(&sp->mutex);                                /* Unlock the buffer */
    V(&sp->items);                                /* Announce available item */
}

sbuf_init(&sbuf, SBUFSIZE);
for (i = 0; i < NTHREADS; i++) /* Create worker threads */
    Pthread_create(&tid, NULL, thread, NULL);

while (1) {
    clientlen = sizeof(struct sockaddr_storage);
    connfd = Accept (listenfd, (SA *) &clientaddr, &clientlen);
    sbuf_insert(&sbuf, connfd); /* Insert connfd in buffer */
}
void *thread(void *vargp)
{
    Pthread_detach(pthread_self());
    while (1) {
        int connfd = sbuf_remove(&sbuf); /* Remove connfd from buffer */
        echo_cnt(connfd);             /* Service client */
        Close(connfd);
    }
}

```

```

#include "csapp.h"

static int byte_cnt;          /* byte counter */
static sem_t mutex;           /* and the mutex that protects it */

static void init_echo_cnt(void)
{
    Sem_init(&mutex, 0, 1);
    byte_cnt = 0;
}
void echo_cnt(int connfd)
{
    int n;
    char buf[MAXLINE];
    rio_t rio;
    static pthread_once_t once = PTHREAD_ONCE_INIT;
    Pthread_once(&once, init_echo_cnt);

```

主线程接到齐全后，就把 `connfd` 放入 `buffer` 中，然后线程会根据之前提到的机制从 `buffer` 中拿出 `connfd` 进行连接。

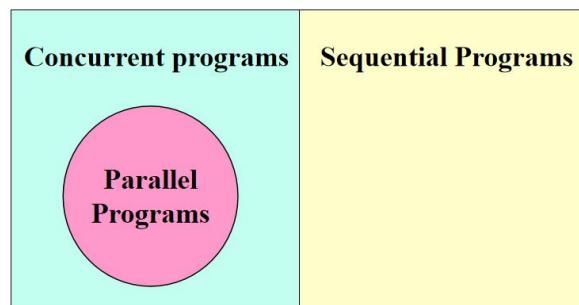
多个 `thread` 实现一个线程池，那么谁调用这个 `init_echo_cnt` 函数？不能调用多次，所以 `pthread` 库中提供了 `Pthread_once` 这个函数，这个 `echo_cnt` 会被反复调用，但是 `once` 这个函数只会被第一个线程调用一次，保证只会初始化一次。

```

Rio_readinitb(&rio, connfd);
while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
    P(&mutex);
    byte_cnt += n;
    printf("server received %d(%d) byte on fd %d\n", n, byte_cnt, connfd);
    V(&mutex);
    Rio_writen(connfd, buf, n);
}

```

All programs



串行程序和并发程序的概念： 只有一个线程看起来就是一个串行程序了，但是加了 `signal` 和 `signal handler` 就是并发程序了，所以不能通过线程数目进行判断，而应该看生命周期。
并发程序（可能交错执行）中有一类是并行程序，两个执行流同时执行（多个 CPU，多个核）。

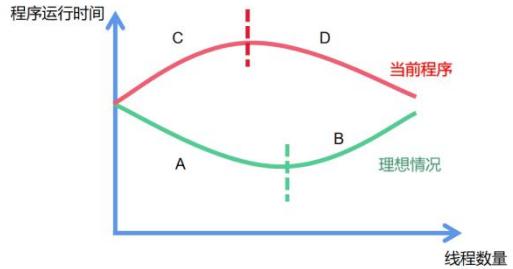
多线程求和

假如目前我们在一个 4 核 CPU 上实现了一个多线程求和的程序：

```

1 /* Thread routine for psum-mutex.c */
2 void *sum_mutex(void *vargp)
3 {
4     int myid = *((int *)vargp);           /* Extract the thread ID */
5     long start = myid * nelems_per_thread; /* Start element index */
6     long end = start + nelems_per_thread; /* End element index */
7     long i;
8
9     for (i = start; i < end; i++) {
10        p(&mutex);
11        gsum += i;
12        V(&mutex);
13    }
14    return NULL;
15 }
```

| | Number of threads | | | | |
|------------|-------------------|-----|-----|-----|-----|
| Version | 1 | 2 | 4 | 8 | 16 |
| psum-mutex | 68 | 432 | 719 | 552 | 599 |



具体效果如左图下方的表格或右图的红线所示。我们希望多线程程序能够降低总的程序执行时间，但是我们现在的程序反而增加了程序执行时间。

A 区下降的原因：线程增加了并行度，充分利用了 CPU 的 4 个核。

B 区上升的原因：CPU 一共只有 4 个核，超过 4 个线程以后会增加上下文切换(context switch)的额外开支。

C 区上升的原因：如左图代码所示，所有的线程需要抢占同一个 mutex，线程越多，抢占 mutex 的额外开销越大。

D 区下降的原因：线程多了以后，就不是所有线程抢占这个 mutex，上下文切换会导致线程分批尝试抢占这个 mutex，降低了全部线程抢占 mutex 的额外开销。

PPT 3-14 issue

- We identify four (non-disjoint) classes of thread-unsafe functions:
 - Class 1: Failing to protect shared variables.
 - Class 2: Relying on persistent state across invocations.
 - Class 3: Returning a pointer to a static variable.
 - Class 4: Calling thread-unsafe functions.

- Class 1: Failing to protect shared variables.
 - Fix: use Pthreads P/V operations.
 - Do not need to modify the calling code
 - Issue: synchronization operations will slow down code.

Thread-unsafe functions (case 2)

- Class 2: Relying on persistent state across invocations

```
unsigned int next = 1;
/* rand - return pseudo-random int on 0..32767 */
int rand(void)
{
    next = next * 110351524 + 12345 ;
    return (unsigned int)((next/65536) % 32768);
}
/* srand - set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

33

- Class 3: Returning a pointer to a static variable

```
char *ctime(const time_t *timep)
{
    static char *p;
    <get current time and converted to string>
    return &p;
}
```

- Class 3: (Fixes)

- Rewrite so caller passes pointer to struct

- Issue: Requires changes in callee

- "Lock-and-copy"

privatep = Malloc(...);
ctime_ts(&t, privatep);

```
char *ctime_ts (const time_t timep, char *privatep)
{
    char *sharedp;

    P(&mutex);
    sharedp = ctime(timep);
    strcpy(privatep, sharedp) ;
    V(&mutex);
    return privatep;
}
```

36

方法 1. 改变接口，每个 thread 自己管自己的空间

方法 2. 加 PV 操作，操作完把 sharedp 赶紧 copy 走，下一个再用 sharedp

2021/5/25

上节课介绍了 `thread` 模型下如何实现一个并发程序。并行程序最主要的问题就是共享变量的访问，比如最早的 `count`。调用的函数里或者其返回值出现了隐式地共享变量。我们 fix 的方法就是不使用函数里的隐式共享变量，这种方法好处就是可以自己去控制它。但是坏处就是要修改这个函数的接口。另一个方法就是不修改它，但是为了在别人共享之前，要在外面准备一份 `thread private` 的空间，`copy` 过去，这个步骤的时候就要保护起来，以免其他线程的干扰。

- Class 4: Calling thread-unsafe functions.
 - Calling one thread-unsafe function makes the entire function that calls it thread-unsafe
 - Fix: Modify the function so it calls only thread-safe functions ☺

最后一种是更广泛的，告诉你 `thread-unsafe` 是传递性的，当一个函数调用一个 `thread-unsafe` 函数，本身也会不安全。所以我们调用其他库的时候，我们需要确认是否是线程安全的，否则我们就要找到替代的函数。

可重入函数（Reentrant Functions）

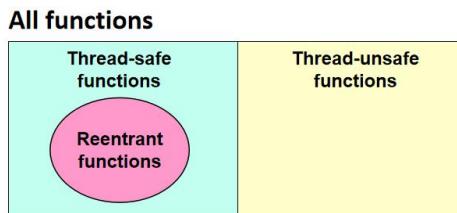
比如一个典型的函数是

Reentrant functions

- A function is **reentrant** iff it accesses NO shared variables when called from multiple threads.
 - Reentrant functions are an important subset of thread-safe functions.
 - Require no synchronization operations.

Reentrant function，它是无状态的，没有共享变量。每个 `thread` 可以重复地调用它。在一些情况下，即使是 `thread-safe` 的函数，也不是 `reentrant function`。拿着锁先做 `P` 操作，出来的时候再去做 `P` 操作，就会导致自锁。因为不存在共享变量，这类函数里也没有同步的要求。

可重入函数主要用于多任务环境中，一个可重入的函数简单来说就是可以被中断的函数，也就是说，可以在这个函数执行的任何时刻中断它，转入OS调度下去执行另外一段代码，而返回控制时不会出现什么错误；而不可重入的函数由于使用了一些系统资源，比如全局变量区，中断向量表等，所以它如果被中断的话，可能会出现问题，这类函数是不能运行在多任务环境下的。



NOTE: The fixes to Class 2 thread-unsafe functions require modifying the function to make it reentrant

- All functions in the Standard C Library (at the back of your K&R text) are thread-safe.
- Most Unix system calls are thread-safe, with a few exceptions:

| Thread-unsafe function | Class | Reentrant version |
|------------------------|-------|-------------------|
| rand | 2 | rand_r |
| Strtok | 2 | strtok_r |
| asctime | 3 | asctime_r |
| ctime | 3 | ctime_r |
| gethostbyaddr | 3 | gethostbyaddr_r |
| gethostbyname | 3 | gethostbyname_r |
| inet_ntoa | 3 | (none) |
| localtime | 3 | localtime_r |

各种函数都有 reentrant 的版本。

我们希望每个线程存在 local 的存储机制。整个在 `thread` 生命周期里都是可见的，但是每个 `thread` 之间又是不同的。我们可以用 `thread_local` 前缀来定义。代码里虽然只写了一个对象。在下例中，因为 `i` 是定义 `local_storage` 中，我们可以理解为有 `i1,i2,i3`。

Eg:`thread_local`

```
thread_local int i=0;
int main() {
    i=0;
    std::thread t1(threadfunc,1);
    std::thread t2(threadfunc,2);
    std::thread t3(threadfunc,3);
}
void f0() {
    std::cout<<i;
}
void threadfunc(int id) {
    f(id);
    ++i;
    g0();
}

"2340", "3240", "4230", "4320", "2430" or "3420"
```

41

打印的顺序不确定，但是打印的内容都是确定的。

TLS (线程局部存储)

TLS: Thread local storage

- Aka., TSS (Thread Specific Storage)
 - A variable can be declared `thread_local` as of C++11
 - Lifetime is the lifetime of the thread
- Useful for data that are logically global to the thread
 - Good for avoiding passing references to it up and down call stack
 - E.g., if data are made `extern`, or `static`, or put in a namespace, etc.

线程局部存储是编译来支持的：

对于Windows系统来说，正常情况下一个全局变量或静态变量会被放到".data"或".bss"段中，但当我们使用`__declspec(thread)`定义一个线程私有变量的时候，编译器会把这些变量放到PE文件的".tls"段中。当系统启动一个新的线程时，它会从进程的堆中分配一块足够大小的空间，然后把".tls"段中的内容复制到这块空间中，于是每个线程都有自己独立的一个".tls"副本。所以对于用`__declspec(thread)`定义的同一个变量，它们在不同线程中的地址都是不一样的。

我们知道对于一个TLS变量来说，它有可能是一个C++的全局对象，那么每个线程在启动时不仅仅是复制".tls"的内容那么简单，还需要把这些TLS对象初始化，必须逐个地调用它们的全局构造函数，而且当线程退出时，还要逐个地将它们析构，正如普通的全局对象在进程启动和退出时都要构造、析构一样。

避免了函数之间传递参数来避免全局的共享需求，TLS 可以认为是 `private` 的。TLS 允许线程之间有比较好的隔离。这些 i 都是进程的内存地址空间里的，如果我们靠传递指针的方式把一个线程的 TLS 传递给另一个线程，也是可以访问的。`Thread_local` 只是一个前缀，可以定义更复杂的数据结构。

Example-1 of TLS: Random number generator

- If your seed was a local variable within the random function
 - it would be initialized every time you called it, giving you the same number each time
 - If it was a global, threads would interfere with each other's sequences
- Using TLS
 - Each thread gets its own random number sequence, independent of other threads.

要生成一个每个 `thread` 独立的随机数的方案，就可以使用 TLS。虽然它用起来很好用，但是使用上对性能有影响，开销要比普通的变量大。

Example-2: Errno

- How is `errno` maintained?
 - A global variable
 - Modified by the library after some error happens
- What if another thread has an error at the same time?
 - Must avoid change the global `errno`
 - Solution: use TLS to save `errno`

`Errno` 应当是每个线程独有的

线程竞争问题

Issues: Races

- Correctness of a program depends on
 - one thread reaching point x in its control before another thread reaches point y

正确性依赖于两个线程竞争的结果，最典型的场景就是有一个全局指针，假设其在 `thread`

使用之前被初始化，所以使用的线程应当等待 malloc 线程初始化完再进行使用。如果没有特殊的保护，可能会出现先使用再分配的情况，这种情况就说是出现了一个 race，race 在某些情况下可能会出错。

```
Issues: Races

/*thread routine */
void *thread(void *vargp)
{
    int myid = *((int *)vargp) ;
    printf("Hello from th. %d\n", myid);
    return NULL ;
}

#define N 4
int main()
{
    pthread_t tid[N];
    int i ;
    for ( i=0 ; i<N ; i++ )
        pthread_create(&tid[i], NULL, thread, &i);
    for ( i=0 ; i<N ; i++ )
        pthread_join(tid[i], NULL) ;
    exit(0) ;
}
```

如果直接传 *i* 进去，传值是没问题的，如果传的是 pointer，这个 *thread* 拿到这个指针，取值的时候，可能就乱套了。每个 *thread* 拿到的值可能出现各种情况。期望的是 0123，出现的可能是 2223 等。*i* 是被多个线程共享的，其结果就取决于使用 *i* 和修改 *i* 的线程之间的竞争，这就是存在 race 的代码。

解决方法：

```
int main()
{
    pthread_t tid[N];
    int i, *ptr ;
    for ( i=0 ; i<N ; i++ ) {
        ptr = malloc(sizeof(int));
        *ptr = i ;
        pthread_create(&tid[i], NULL, thread, ptr);
    }
    for ( i=0 ; i<N ; i++ )
        pthread_join(tid[i], NULL) ;
    exit(0) ;
}
```

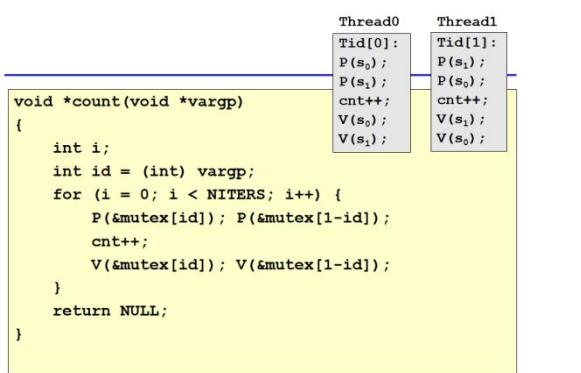
1. 消除 share
2. 对 shared variable 增加 syn 来控制顺序，做不下去只能等在那里，等待 P 解除阻塞等。

死锁问题

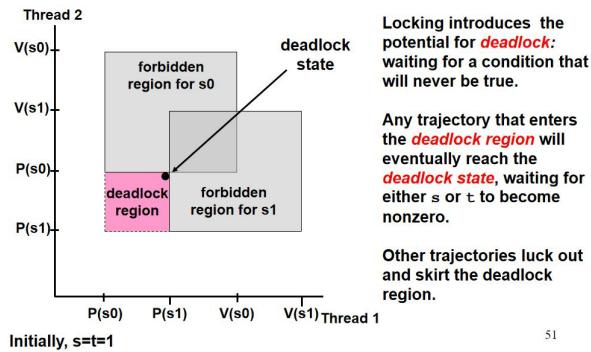
第二种典型的并发程序的问题是 deadlock

Eg:

```
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1); /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1); /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*)0);
    Pthread_create(&tid[1], NULL, count, (void*)1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}
```



有两个信号量，两个线程传进去的值分别是 0 和 1。如果一个 `thread` 一口气执行完，那么是没问题的，但是如果 `thread0` 拿到 `s0` 而 `thread1` 拿到了 `s1`，两个人抢了东西都不走，这就出现了死锁问题。



进到红色位置的时候，就只能卡死在红色区域的右上角。因为黑点进不了两个信号量保护的区域。

死锁的程序和 `race` 也是一样的，是执行的时候有可能出错，会使得程序死在那里没反应了。

解决方案：

1. 通过分析，让程序中不要存在 `deadlock region`
2. 在发现 `deadlock` 后，主动释放掉已经持有的信号量，允许往左往下回退。比如增加 `alarm`，在 `alarm` 中把持有的信号量释放掉。

要写并发程序基本上都是使用 `thread` 这个模型（效率高、容易共享）但是容易出错。它最大的核心问题就是对于共享的变量怎么使用。怎么实现同步等机制来保护它。想详细研究的话可以找书本 D. Butenhof, “Programming with Posix Threads”, Addison-Wesley, 1997

我们把 `thread` 部分介绍完了，书上的内容就到这里了。我们需要补充一点其他的知识。

第一个是 `locks`，当我们把进程、内存、网络、IO 介绍完了以后，我们对底层了解了不少。写多线程程序是未来的趋势。

下周三 `quiz`，考上次 `quiz` 之后的。

锁

其他人还在那边访问，没有办法起到保护的作用。所谓保护就是这里的 `mutual-exclusive`，关闭中断只能在单个 CPU 下才能起效果。在我们学 `signal` 的时候，`main thread` 和 `signal thread` 的生命周期可能有重叠，这时候我们就采用关闭中断的方式，可以把这个理解为 `lock` 和 `unlock`

- **First Attempt: A Simple Flag**

```

typedef struct __lock_t { int flag; } lock_t;

void init (lock_t *mutex) {
    mutex->flag = 0; // 0 -> lock is available, 1 -> held
}

void lock (lock_t *mutex) {
    while (mutex->flag == 1); // spin-wait (do nothing)
    mutex->flag = 1; // now SET it!
}

void unlock(lock_t *mutex) { mutex->flag = 0; }

```

原子的操作都是由几个操作组合在一块，OS 保证了他们是原子的

Atomic Exchange

- Test-and-set instruction
 - Read and Set Memory
 - On SPARC: ldstub
 - On x86: xchg

```

Atomically perform
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new; // store 'new' into old_ptr
    return old; // return the old value
}

```

Spin Lock: Test-and-Set

```

typedef struct __lock_t { int flag; } lock_t;

void init (lock_t *lock) {
    lock->flag = 0; // 0 -> lock is available, 1 -> held
}

void lock (lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) { lock->flag = 0; }

```

基于 TestAndSet 来实现 Lock

怎么基于 TestAndSet 来实现 Lock 呢？

如果返回的旧的值是 1，那么说明原来就已经上锁了。只有一个 thread 会从里面出来。

```

typedef struct __lock_t { int flag; } lock_t;
int flag[N]; //one flag per thread
int TestAndSet(lock_t Lock) { //set lock and return old value
    int ret;
    while (1) { //me is my index in flag
        flag[me] = 1; //warn other threads
        if (AnyoneElseInterested(me)) //is other thread warning us
            flag[me] = 0; //yes, reset my warning, try again
        else {
            ret = Lock.flag; //set ret to value of Lock
            Lock.flag = 1; //and set the Lock
            flag[me] = 0;
            return ret;
        }
    }
}

```

15

Livelock 就是两个人不停地再动，但是还在浪费 CPU，也执行不下去。

判断锁的方式：

1. correct lock

前面的 spin lock，拿不到锁的人都在那里尝试，线程越多，spin lock 的性能越差。银行排队，每个人拿一个号，到了你这个号你就去，就是 ticket lock 的机制。它也很公平，什么时候排队，什么时候拿号。在很多人抢的时候性能也比较好，公平性也兼顾了。它需要的是 fetchandadd 的支持。

```

void init (lock_t *lock) {
    lock->ticket = 0;
    lock->turn   = 0;
}
} lock_t;

typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

Atomically perform
int FetchAndAdd(int *ptr) {
    int old = *ptr; // fetch old value at ptr
    *ptr = old + 1; // add 1 and store 'new' into ptr
    return old; // return the old value
}

void lock (lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn) ← Not starvation
        ; // spin
}

void unlock(lock_t *lock) { lock->turn = lock->turn + 1; }

```

一个是 ticket，一个是 turn。Fetchandadd 先拿一个号，把 ticket 不断往上加。Eg: 进来的人拿的 ticket 旧的值 90，把 ticket 加成 91。Turn 是当前做到第几号了，每个人都观察当前的号有没有到你。`unlock` 把 turn 设置成下一个。但是这里还是一个 spin，大家还在看这个号有没有叫到你。当你发现没有轮到的时候，就 `yield` 主动释放 CPU。主动触发 context switch，把自己置成 suspend 或者 wait 状态。

```

void lock () {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

```

主动放弃 CPU 来避免 spin 的状态。

2021/6/1

quiz 取消。

主动放弃 CPU 占用（Yield）

上机课我们介绍了 ticky lock, spin lock 会导致没有拿到锁的人忙等消耗 cpu，而 ticket lock 是通过发 ticket 的形式来保证到底的顺序和拿锁的顺序相同，最普通的 ticket lock 还是会忙等，但是不会有 starvation 的状态，因为 spin lock 可能永远都抢不过别人，而 ticket lock 的公平性实现的非常好。

Lock with Test-and-Set and Yield

```

void init () { flag = 0; }

void lock () {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() { flag = 0; }

```

我们需要让没有抢到 CPU 的线程主动放弃掉 CPU，`yield` 是一种 `syscall`，主动发生 `context switch`。Kernel 在选择执行的时候，可以直接选择下一个来执行。我们假设有一种机制能把它唤醒。

假设一个 CPU，两个 thread，一个没抢到锁的线程主动 `yield` 效率非常高，原来 spin lock

的时候一定要等到需要锁的人时间片全部用完再强制发生切换。

- 100 threads on one CPU
 - One thread acquires the lock and is preempted before releasing it
 - the other 99 will each call lock(), find the lock held, and yield the CPU
 - Assuming a round-robin scheduler, each of the 99 will execute this run-and-yield pattern
 - the cost of a context switch can be substantial, and there is thus plenty of waste.

假如有 100 个 thread，极端情况下有 99 个人都 yield，那么一个有锁的线程尝试放锁的时候，该怎么唤醒其他 99 个人呢？它只能把 99 个人都叫醒，然后大家再一起抢，可能会有 98 个 yield。一个更好的方式是我们有一个队列，我们以队列的方式去主动唤醒下一个号的人。只要队列里面有人，说明前面有人排队，直接进队列即可。每次唤醒队列头的线程来拿锁。

Using Queue

- Previous problem: leave too much chance
 - Potentially lead to waste and starvation
- Explicitly select next thread to acquire lock
 - Queue: keep track of waiting threads
 - Support provided by Solaris
 - Park(): put a calling thread to sleep
 - Unpark(threadID): wake a particular thread

```
typedef struct __lock_t {  
    int flag;  
    int guard;  
    queue_t *q;  
} lock_t;  
  
void lock_init (lock_t *lock) {  
    lock->flag = 0;  
    lock->guard = 0;  
    queue_init(lock->q);  
}
```

我们需要有额外的 guard 进行保护，初始化都为 0.

```
void lock (lock_t *lock) {  
    while (TestAndSet(&lock->guard, 1) == 1)  
        ; // acquire guard lock by spinning  
    if (lock->flag == 0) {  
        lock->flag = 1; // lock is acquired  
        lock->guard = 0;  
    } else {  
        queue_add(lock->q, gettid());  
        lock->guard = 0;  
        park();  
    }  
}  
  
void unlock (lock_t *lock) {  
    while (TestAndSet(&lock->guard, 1) == 1)  
        ; // acquire guard lock by spinning  
    if (queue_empty(lock->q)) {  
        // let go of lock; no one wants it  
        lock->flag = 0;  
    } else {  
        // hold lock (for next thread!)  
        unpark(queue_remove(lock->q));  
    }  
    lock->guard = 0;  
}
```

锁本身是 flag, guard 保证拿锁的是原子的。进去的 thread 先去抢 guard(使得 guard = 1)，

如果当前没有人拿锁，队列为空，不用进队列可以直接拿锁，拿好锁（使得 `flag = 1`）然后 `guard` 释放掉（`guard = 0`）。更多情况下是锁已经被别人拿走了（`flag = 1`），我们要让它进入队列，加入自己的 `thread_id` 为了之后能通过 `signal` 的机制去通知你。然后就去 `yield` 掉。这个线程从 `park` 出来不需要再上锁，等于直接把锁传递给队列头的第一个人。这个设计的高效就在于直接叫到下一个人。

Unlock: 先进来抢到 `guard`，如果队列里没有线程，那就释放掉这把锁走人；如果有线程正在等着，只需要叫醒下一个线程就行。

Wakeup/Waiting Race

Wakeup/waiting Race

```

Thread 1 call lock() ... Thread 2 call unlock() ...

queue_add(Lock->q, mytid)
lock->guard = 0;
interrupt: switch to Thread 2

unpark();
interrupt: switch to Thread 1
park(); //mytid has been removed in unpark()

```

我们注意到这个代码可能有问题，`park` 是没被保护的，如果在 `lock->guard` 时进行了上下文切换了，有另一个线程进入了 `unlock` 中，它会发一个消息来叫醒你，但是可能线程还没睡。但是如果我们把 `lock->guard` 放在 `park` 后，又没有人放掉 `guard`。我们希望 `guard` 和 `park` 一起完成。

Wakeup/waiting Race

```

void lock (lock_t *lock) {
    while (TestAndSet(&lock->guard, 1) == 1) Solaris: setpark()
        ; // acquire guard lock by spinning
    if (lock->flag == 0) {
        lock->flag = 1; // lock is acquired
        lock->guard = 0;
    } else {
        queue_add(lock->q, gettid());
        setpark(); After calling it,
        lock->guard = 0;
        park(); Park will return
    }
}
instead of sleeping if another thread just finished unpark()

```

加了一个 `setpark`，在 `setpark` 和 `park` 之间如果有人唤醒你，会直接去做操作。

Linux-based Futex Locks

```

void mutex_lock (int *mutex) {
    int v;
    if (atomic_bit_test_set(mutex, 31) == 0) return;
    atomic_increment (mutex); // add counter
    while (1) {
        if (atomic_bit_test_set(mutex, 31) == 0) {
            atomic_decrement(mutex);
            return;
        }
        v = *mutex;
        if (v >= 0) continue; // lock has released
        futex_wait(mutex, v);
    }
}

```



Futex Locks

```

void mutex_unlock (int *mutex) {
    /* Adding 0x80000000 to the counter results in 0 if and
     * only if there are not other interested threads */
    if (atomic_add_zero(mutex, 0x80000000))
        return;

    /* There are other threads waiting for this mutex,
     * wake one of them up. */
    futex_wake(mutex);
}

```



在 linux 是怎么实现的呢？最高位 1 个 `bit` 用来代表 `lock`，刚才的设计中我们发现有 `flag` 和 `guard`，这里用了一个 32 位的 `int`，后 31 位代表有多少人在等。刚进来的时候是 `atomic_bit_test_set` 这个原子操作，代表这个就是 `mutex`，检查第 31 位并去 `set`。返回值是

`bit` 原来的值，现在我们把它改成 1 了。如果成功了，说明拿到锁了，直接 `return` 即可。`While` 循环中，置为 1 了以后再去试一下，如果这时候抢到了说明进来之前在这一段之间有人放了锁。把 `mutex` 的值拿下来 (`v`)，再对 `v` 去 `check`，如果大于 0，说明有人放锁了。注意这里没有 `guard` 的保护，它是时刻容易被人打断的。这个队列是在 `kernel` 里隐式管理的 (`futex_wait`)。出来的人是一个人还是全部人取决于 `futex` 的唤醒机制，而在这里从 `wait` 里出来只是说明获得了争抢的资格，必须在从循环一开始去尝试拿锁。

`Atomic_add_zero`，原子地加上一个数 (0x80000000) 检查加完了是否等于 0，说明情况下等于 0 呢，第一位会溢出被清除掉，说明 `count` 也为 0。如果 `count` 不为 0，那么放完锁了以后要有唤醒睡眠线程的过程。

我们发现 `futex_wait` 和 `futex_wake` 内部实现了队列和唤醒过程，在我们的代码过程中只有一个 `mutex` 对象。而我们基于 `park` 的实现，用户需要自己维护这个线程队列，而且实现过程中，是全程锁着的。而在 `linux` 的实现是在 `kernel` 内部实现这个队列，`wait` 加入队列的时候，操作系统会自动地帮我们加入到绑定的队列中。

```
Two-phase Lock (hybrid)
- 1st phase: the lock spins for a while
- 2nd phase: the caller is put to sleep
- while (1) {
    if (atomic_bit_test_set(mutex, 31) == 0) {
        atomic_decrement(mutex);
        return;
    }
    v = *mutex;
    if (v >= 0) continue; // lock has released
    futex_wait(mutex, v);
}
```

1st phase

2 阶段锁是介于乐观锁和悲观锁之间的，先试一下，没有抢到再去睡。这种看起来不是最优的，但是统计下来是比较好的情况。这种方式也是一种折中的方式。

条件变量

`Lock` 同步很难的情况下，需要条件变量。

- Locks are not the only primitives that are needed to build concurrent programs
 - A thread wishes to check whether a condition is true before continuing its execution
- ```
1 void *child(void *arg) {
2 printf("child\n");
3 // XXX how to indicate we are done?
4 return NULL;
5 }
6
```

```
7 int main(int argc, char *argv[]) {
8 printf("parent: begin\n");
9 pthread_t c;
10 Pthread_create(&c, NULL, child, NULL); // create child
11 // XXX how to wait for child?
12 printf("parent: end\n");
13 return 0;
14 }
```

Desired output  
parent: begin  
child  
parent: end

因为线程之间的调度在这个代码中是不确定的，我们希望控制两个线程执行的先后顺序需要有额外的机制，`lock` 只能保证一个人进去，但不保证进去的顺序。

```

1 volatile int done = 0; Spin-based Approach
2
3 void *child(void *arg) {
4 printf("child\n");
5 done = 1;
6 return NULL;
7 }
8
9 int main(int argc, char *argv[]) {
10 printf("parent: begin\n");
11 pthread_t c;
12 Pthread_create(&c, NULL, child, NULL); // create child
13 while (done == 0)
14 ; // spin
15 printf("parent: end\n");
16 return 0;
17 }

```

Main 线程等待成 1 了时候才继续。

- **Declare a condition variable:**
  - `pthread_cond_t c;`
  - proper initialization is also required
- A condition variable has two operations associated with it: `wait()` and `signal()`
  - `wait()`: a thread wishes to put itself to sleep
  - `signal()`: a thread has changed something in the program and thus wants to wake a sleeping thread waiting on this condition

条件变量是一个显式的队列，可以一对一通知、按需通知、也可以广播，使得可以在满足条件的情况下按需地去做。Waiting 和 signal 机制。我们需要定义一个通知变量，使得通知和被通知的人联系在一起。需要初始化，然后由 wait 和 signal 对象。Wait 是把自己变成 sleep 状态，而 signal 是叫醒一个变量。

```

pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);

• wait() call
 - Takes a mutex as a parameter
 - This mutex is locked when wait() is called
 - The responsibility of wait() is to release the lock
 and put the calling thread to sleep (atomically)
• When the thread wakes up (after some other
 thread has signaled it), it must re-acquire
 the lock before returning to the caller

```

它提供的接口比 park 好看一点，带着个锁。在调用之前是上锁状态，在睡之前会放掉这把锁。所以一个 cv 会和一个 mutex 作为一套来使用。

```

1 int done = 0;
2 pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3 pthread_cond_t c = PTHREAD_COND_INITIALIZER;
4
5 void thr_exit() {
6 Pthread_mutex_lock(&m);
7 done = 1;
8 Pthread_cond_signal(&c);
9 Pthread_mutex_unlock(&m);
10}
11

```

```

12 void *child(void *arg) {
13 printf("child\n");
14 thr_exit();
15 return NULL;
16 }
17
18 void thr_join() {
19 Pthread_mutex_lock(&m);
20 while (done == 0)
21 Pthread_cond_wait(&c, &m);
22 Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26 printf("parent: begin\n");
27 pthread_t p;
28 Pthread_create(&p, NULL, child, NULL);
29 thr_join();
30 printf("parent: end\n");
31 return 0;
32 }

```

我们发现这个代码比较冗余，为什么有 `done` 了还要加 `mutex` 和 `cv` 呢。

我们先看一下，如果没有 `done`，我们主线程这里 `wait`，子线程处理完了调用 `signal`。我们的想法是这个线程先睡了再通知，但是有可能子线程先执行可能通知完了再去睡。所以我们需要有 `done` 来避免这件事情发生。

反过来，如果没有 `lock` 直接写代码行不行，`done` 为 0 出来的时候，`wait` 之前可能已经发了 `signal` 了。

## Producer/Consumer (Broken)

---

```

1 int buffer;
2 int count = 0; // initially, empty
3
4 void put(int value) {
5 assert(count == 0);
6 count = 1;
7 buffer = value;
8 }
9
10 int get() {
11 assert(count == 1);
12 count = 0;
13 return buffer;
14 }

```

一个 `buffer`, `producer` 和 `consumer` 之间只有一个 `slot`。这个语义很简答，在 `put` 的时候要求 `count=0`，在 `get` 的时候要求 `count=0`。它们之间需要协调的，有空位了才能往里面加，有东西了才能取出来。

## Producer/Consumer (Still Broken)

### Producer/Consumer (Still Broken)

```
1 cond_t cond;
2 mutex_t mutex;
3
4 void *producer(void *arg) {
5 int i;
6 for (i = 0; i < loops; i++) {
7 Pthread_mutex_lock(&mutex); // p1
8 if (count == 1) // p2
9 Pthread_cond_wait(&cond, &mutex); // p3
10 put(i); // p4
11 Pthread_cond_signal(&cond); // p5
12 Pthread_mutex_unlock(&mutex); // p6
13 }
14 }
```

```
15
16 void *consumer(void *arg) {
17 int i;
18 for (i = 0; i < loops; i++) {
19 Pthread_mutex_lock(&mutex); // c1
20 if (count == 0) // c2
21 Pthread_cond_wait(&cond, &mutex); // c3
22 int tmp = get(); // c4
23 Pthread_cond_signal(&cond); // c5
24 Pthread_mutex_unlock(&mutex); // c6
25 printf("%d\n", tmp);
26 }
27 }
```

put 完了通知 consumer 可以拿了。Consumer 通知 producer 可以放了。Consumer 检查当前 count 是否为 0，如果有东西就 get 出来再通知 producer 可以继续放了。我们需要模拟不同的上下文切换的情况。这个代码在 1 个 producer 和 1 个 consumer 的情况下是没有问题的，互相交错地执行。

如果有 2 个 producer 和 1 个 consumer 的情况下，（例子见 PPT3-16-cv-19）consumer2 尝试去拿的时候会触发 assert，因为数据已经被 consumer1 拿走了。

## Producer/Consumer (Single buffer)

```
1 cond_t empty, fill;
2 mutex_t mutex;
3
4 void *producer(void *arg) {
5 int i;
6 for (i = 0; i < loops; i++) {
7 Pthread_mutex_lock(&mutex);
8 while (count == 1)
9 Pthread_cond_wait(&empty, &mutex);
10 put(i);
11 Pthread_cond_signal(&fill);
12 Pthread_mutex_unlock(&mutex);
13 }
14 }
```

我们 fix 的方法是：不要复用通知，从而避免掉一个 consumer 通知另一个 consumer 的情况，consumer 应该去通知另一个 producer。用两个不同的通知。

# 2021/6/8

我们要对原来的 producer-consumer 代码做一个微调。3-16-cv-page26，在课程主页的链接里 <https://pages.cs.wisc.edu/~remzi/OSTEP/threads-cv.pdf>

关于 malloc 和 free，会出现一个内存不够的情况，在 malloc 的时候，会去检查要的 size 和剩下的 size，如果不够大，就会去等。

```

1 // how many bytes of the heap are free?
2 int bytesLeft = MAX_HEAP_SIZE;
3
4 // need lock and condition too
5 cond_t c;
6 mutex_t m;
7
8 void *
9 allocate(int size) {
10 Pthread_mutex_lock(&m);
11 while (bytesLeft < size)
12 Pthread_cond_wait(&c, &m);
13 void *ptr = ...; // get mem from heap
14 bytesLeft -= size;
15 Pthread_mutex_unlock(&m);
16 return ptr;
17 }
18
19 void free(void *ptr, int size) {
20 Pthread_mutex_lock(&m);
21 bytesLeft += size;
22 Pthread_cond_signal(&c);
23 Pthread_mutex_unlock(&m);
24 }

```

28

只叫醒一个人的情况：如果我们叫醒的人发现还是不够，那么 `check` 一次被叫醒的线程又会去睡觉了。这样会导致两个线程都在睡觉。所以我们选择叫醒所有人。也就是 `Pthread_cond_broadcast()`。所以在 ticket-lock 中，叫醒所有人没有意义，因为需要等到叫自己，而在这里叫醒所有人就是有意义的。

## 进程调度 (process scheduling)

今天要讲的是进程调度 (process scheduling) 。

### From Mechanism to Policy

---

- **Mechanisms**
  - Low-level methods or protocols
    - that implement a needed piece of functionality
  - Examples
    - context switch (time-sharing)
    - Paging
- **Policies**
  - Intelligence resides on top of the mechanisms
  - algorithms for making some kind of decision
  - Examples
    - Scheduling policy (decide to run which process)
    - Replacement policy (decide which page to be replaced)

光有机制是不够的，需要一些 `policy` 来实现这些机制。比如 `lock` 是一种机制，而什么时候调用 `lock` 就是一种策略。策略和机制是独立的，当我们优化的时候我们可以用这种模块划分来解耦。策略可以独立于机制去执行，这样可以把两个研究分离开来。

我们最后一节课来介绍一些 `policy`，在应用层实现了 `mechanism` 后，操作系统的 `policy` 就来执行应用层的一些逻辑。我们不知道 OS 的实现是怎么样的，但是我们可以分析 OS 的 `policy` 在不同场景下的性能好坏。所以我们一个程序的性能，可能被这两方面影响。

我们今天来看调度进程的 `policy`，比如上面有很多 `thread`，我们怎么让这些 `thread` 来使用我们的 CPU，只要我们的 `mechanism` 提供足够多的接口，`policy` 和 `mechanism` 可以独立发展，所谓的 `policy` 就是在不同情况下做出一个决定。也有 `replacement` 的 `policy`，比如 `cache`

里访问了到底替换谁，比如 LRU 就是一个 policy，实现 LRU 就是一些机制（mechanism）。

### Workload Assumptions

---

1. Each job runs for the same amount of time
  2. All jobs arrive at the same time
  3. Once started, each job runs to completion
  4. All jobs only use the CPU
    - i.e., they perform no I/O
  5. The run-time of each job is known
- Above assumptions are unrealistic
    - Will be relaxed one by one

对于 thread 的执行，我们有很多的假设，如果我们没有假设我们也不知道一堆 thread 里找出谁给 CPU，我们需要有一个衡量标准。这边我们设置了一些约束条件，约束条件越多，我们的 policy 越简单，我们随后可以慢慢放宽这些条件，使得 policy 可以满足更多的一些情况，使得我们的 policy 更贴近实际情况，能够在实际情况中使用。

假设：

1. 每个任务运行相同的时间。
2. 每个任务在相同的时间到达。
3. 当我调度一个任务之后，就会一直占用 CPU。
4. 所有的人都使用 CPU，不考虑 IO，也就是不让出 CPU 等待 IO
5. 每个任务的运行时间已知，换句话说我们像先知一样把所有事情都放到最理想的情况下。

随后一条条去掉这些假设，我们再去考虑有什么差别。

- Metric
  - is just something we use to measure something
- Scheduling metrics
  - there are a number of different metrics that make sense in scheduling
- The turnaround time
  - the time at which the job completes minus the time at which the job arrived in the system
  - $T_{turnaround} = T_{completion} - T_{arrival}$
  - Because for now  $T_{arrival} = 0$  and hence
  - $$T_{turnaround} = T_{completion}$$

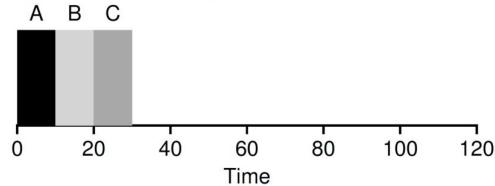
接下来，我们衡量调度需要有一些好的策略。在调度的时候到底花了多少时间完成。在这个场景底下，我们只需要考虑最理想情况下进行调度使得这些 job 完成时间尽量的短。Turn around time = 完成时间-到达时间

## FIFO (先来先服务)

第一个最简单的是先来先服务 (FIFO)，来了就把 CPU 给你，因为同时到，就随便选一个任务，开始执行。

### First In, First Out (FIFO)

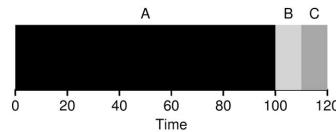
- Also known as First Come, First Served (FCFS)
- It is clearly simple and thus easy to implement
- Examples
  - Jobs A, B, C. Each runs for 10s
  - Turnaround time is  $(10+20+30)/3=20$



最后的 turn around time 分别是 10、20、30。整个这样的简单的调度，平均就是 20 秒。我们会发现这时候没有什么可以改变的 policy，因为有那么多限制条件在这里。

如果它们的执行时间不一样，假设分别是 100、10 和 10.

- Job A runs for 100s. Job B and C run for 10s
  - Turnaround time is  $(100+10+10)/3=110$
- Convey Effect
  - a number of relatively-short potential consumers of a resource get queued behind a heavyweight resource consumer

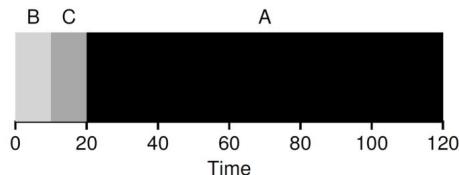


对于 turnaround 度量来说，更希望去调度任务短的，当我们执行时间有长短的时候，等待时间就和任务调度顺序有关系了。最好的方式是，对于同时到达的，希望能够把短的任务先调度。

## Shortest Job First

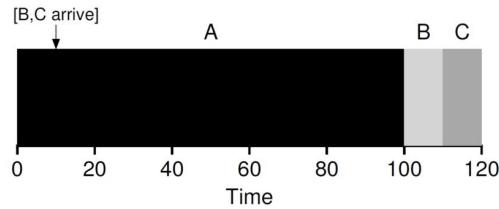
### Shortest Job First (SJF)

- Runs the shortest job first
  - then the next shortest, and so on
  - optimal
- Examples
  - Job A runs for 100s. Job B and C run for 10s
  - Turnaround time is  $(10+20+120)/3=50$



最短的任务先调度，从这个度量来考虑的情况下，SJF 就比 FIFO 就要快，可以缩短 turn around time。它不能影响任务的执行之间，但是能影响任务的等待时间。这个机制的实现可以用不同的方式。

- Assume A arrives at  $t = 0$  and needs to run for 100 seconds, whereas B and C arrive at  $t = 10$  and each need to run for 10 seconds
- Turnaround time  $(100+100+110)/3=103.33s$

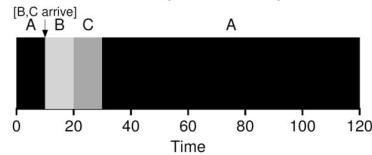


如果我们再去掉到达的时间不一样，如果快的任务到完了，因为第三条假设的存在，会导致长的任务还是一直在执行。此时 SJF 退化到了 FIFO。所以，我们还需要去掉第三条假设，也就是允许 CPU 抢占。我们需要实现新的机制和策略来允许 CPU 抢占，我们不关心新 policy 导致的新增的一些硬件和软件的实现。

## Shortest Time-to-Completion First

### Shortest Time-to-Completion First (STCF)

- 
- STCF
    - Any time a new job enters the system
    - the STCF scheduler determines
      - which of the remaining jobs (including the new job) has the least time left, and schedules that one
  - Preempt A when B and C arrived
    - Turnaround time is  $(10+20+120)/3=50$



有新的人来的时候，就可以发生一次重新调度，比较任务剩下的执行时间，任务剩下的执行时间短的优先执行。等 B 和 C 执行完了再轮到 A。有了这个机制，我们发现 turn around time 就明显缩短了。

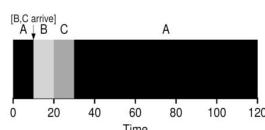
前面我们考虑的是整个的执行时间，包括了等待时间和执行时间。

第二个度量：响应时间，到达之后 CPU 第一次服务它的时间。如果在执行的过程中是以交互方式的，我们希望任务有一些反应，否则任务在完成之前都卡在那里了。在与用户交互的软件中，通常有一个进度条。

### Response time

---

- In the time sharing systems
  - Many users are interactive with computers
  - Each user does not want to wait too long
- Response time
  - the time from when the job arrives in a system to the first time it is scheduled
  - $T_{response} = T_{firstturn} - T_{arrival}$
- Example
  - $T_{response} = (0+0+10)/3=3.33$



在 STCF 策略下，响应时间比较好。SJF 的响应时间就会非常差。我们发现这种带有抢占的机制 STCF，响应时间和等待时间都比较优秀。但是我们发现，对于 C 来说，需要等到 B 结束。这时候我们引入时间片的机制，不但要抢占，也最好不要让一个人一直占用 CPU。

## 轮询

### Round Robin

- Mechanism**
    - Run a job for a time slice
    - Switch to the next job in the run queue
    - Repeat until the jobs are finished
  - Time slice**
    - scheduling quantum
    - a multiple of the timer-interrupt period (10ms)
- 

### Round Robin vs. SJF

- SJF**
    - Best for turnaround time  
 $(5+10+15)/3=10$
    - Worse for response time  
 $(0+5+10)/3=5$
  - Round Robin**
    - Worse for turnaround time  
 $(13+14+15)/3=14$
    - Better for response time  
 $(0+1+2)/3=1$
    - Must some trade-off
- 

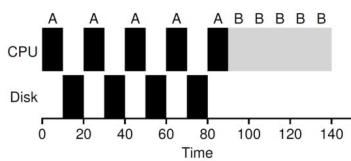
我们不但要让最快的任务优先执行，也要考虑到公平性，让大家都有机会执行。我们引入时间片，把执行时间切成一片片然后进行调度。当我们切完以后，我们就不去考虑谁的时间长和短了（这是从整体考虑），这时候我们就使用 Round Robin 也就是一个轮询的调度机制，这样的执行和原来执行，turn around time 变久了，但是 response time 大大缩短了。我们发现 turn around time 都变成了靠近最后的时间，这就是一个权衡。对于现在大量的应用来说，和用户交互的系统来说，response time 比 turn around time。而在超算任务中，肯定不会注重 response time。所以用什么样的策略和应用场景都是有关系的。

而且我们注意，在实际情况下，round robin 在每一次切换的时候会引入额外的上下文切换的开销，总的时间会比 15 秒更久一点。要 flush TLB entry 的缓存，被换进来的任务肯定是要比连续跑的 TLB 慢一点，包括流水线也得全部刷掉。所以我们也不可能把时间片无限缩小，这会加大时间片切换的额外开销。

### Incorporating I/O

#### Example

- Job A and B run on CPU for 50s each
- Job A does 4 I/O operations (10s each)
- The poor schedule does as follows



#### When a job initiates an I/O request

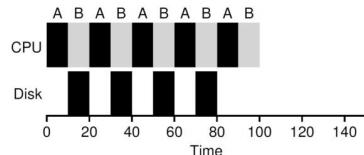
- A system call is invoked
- The job is blocked (by kernel)
- Another job is scheduled to CPU
- When a job completes an I/O request**
  - An interrupt is sent to CPU
  - The job becomes ready to run
  - The schedule determines which job to be scheduled
- Overlap I/O and CPU**
  - Maximize the utilization of systems

此时我们再去掉第四个假设，在整个过程中有访问磁盘、访问网络、等待用户，我们不可能让这个 thread 再占用 CPU。比如任务 A 有 4 个 IO 贯穿在任务过程中，如果我们没有 yield，用完 CPU，就会出现浪费 CPU 的情况。如果任务因为 IO 被阻塞了，我们应该去调度别的任务。直到一个任务完成了之后，再调度它。这时候就出现了 I.O 和 CPU 的使用时间 overlap 的情况，也就是不同的器件可以一块工作，各自独立完成一些任务，这是提高系统利用率的好方法。怎么利用起来这些 CPU 的 idle 阶段，就需要在调度上予以支持。

# IO 和 CPU 协作

## Incorporating I/O

- How should the scheduler treat each job?
  - Treat each CPU burst as a job
- Example
  - Treat each 10-ms sub-job of A as an independent job
  - Doing overlap better



在这种情况下，可能就需要两个队列来维护 `running` 和 `suspend` 状态的队列。一个任务在被调度的时候不再会以整体来考虑，而是以子任务来执行。任务会进出队列很多次，我们可以认为每一个任务都把时间片当成单独的子任务进行调度，这样就可以解决同时使用其他设备的情况。

最后要解决的是一个任务要执行多少时间到目前为止我们是知道的，比如 SJF 我们要挑选出执行时间最短的。但是在实际过程中我们不知道，而且有一些任务是 long-running 的，但是我们把每一个时间片都作为子任务的时候，我们不再关心整体执行多少时间了。所以，到目前为止 round robin 是可以用的了。

## Outline

- Multi-Level Feedback Queue Scheduler
- Suggested reading: 8

在实际使用中，linux 的核心思想就是 MLFQS 策略。

## MLFQ 策略

### Learning from History

#### Primary Goal of the MLFQ

- MLFQ would like to both
  - optimize the turnaround time without a priori knowledge of job length
  - minimize the response time
- How to schedule without perfect knowledge
  - The scheduler can learn the characteristics of the jobs it is running as the system runs,
  - and thus make better scheduling decisions

#### Systems that learn from the past to predict the future

- The multi-level feedback queue
- hardware branch predictors
- caching algorithms
- Such approaches work when jobs have phases of behavior and are thus predictable
- Must be careful with such techniques
  - they can easily be wrong and drive a system to make worse decisions than they would have with no knowledge at all

我们先看一下这个策略的主要目标，它希望在不知道任务会执行多久的情况下优化 turn around time 和 response time。它能够做出一个比较好的决策。

核心想法：从历史中进行学习，在执行过程中进行调整，来预测未来的情况。首先它有多个 running 队列，并且带有 feedback 也就是根据执行情况会有反馈来调整策略。它的有点像分支预测或者一些 cache 的策略。凡是带有预测的，基本上都是这种 learning from history 策略。它的整个想法会把任务变成很多阶段，认为一个阶段内的行为是有连续性的，行为是可以通过一个阶段内之前的行为来预测的。当然还是需要比较仔细地设计策略来防止预测错误导致的较差的结果。

## Basic Rules

---

- A number of distinct queues
    - each assigned a different priority level
  - A job that is ready to run is on a single queue
- Rule 1:** If Priority(A) > Priority(B), A runs (B doesn't)
- Rule 2:** If Priority(A) = Priority(B), A & B run in RR
- [High Priority] Q8 → (A) → (B)  
Q7  
Q6  
Q5  
Q4 → (C)  
Q3  
Q2  
[Low Priority] Q1 → (D)

这个策略挺简单的，比如有七八个队列，可以和 linux 下的 priority 对应。新来的任务就去高 priority 的，调度的时候会先考虑优先级较高的。在一个队列里可以采用随机或者轮询的策略。

重点在于我们怎么把任务放到这些不同的队列里去，而且因为是基于 learning 的，任务会在不同的队列中移动。

## How to Change Priority

---

- Our workload is a mix of
  - interactive jobs that are short-running
    - and may frequently relinquish the CPU
  - and some longer-running "CPU-bound" jobs
    - need a lot of CPU time
    - response time isn't important

**Rule 3:** When a job enters the system, it is placed at the highest priority

**Rule 4a:** If a job uses up an entire time slice while running, its priority is reduced

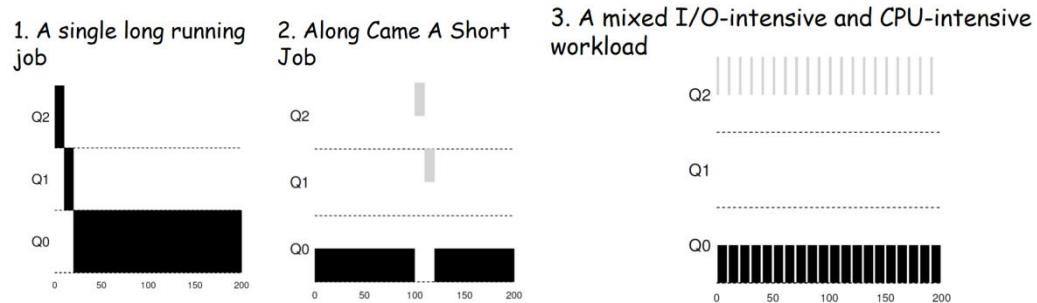
**Rule 4b:** If a job gives up the CPU before the time slice is up, it stays at the same priority level.

在实现的时候，我们可以认为这是一个逻辑队列，我们认为 work load (任务的集合) 分成

两类任务，

1. 和人打交道的任务（响应时间很重要，CPU 用的比较少（约 10%，更多的时间是等人反应），但是想用的时候要尽快给人家用），为了保证响应时间，并且它可能 CPU 的时间片不会用满，所以优先级比较高。
2. CPU-bound long-running 任务，比如说是一个计算任务，会使用很长时间的 CPU，不需要进行交互，它需要大量的 CPU 的时间，但是不是交互式的任务，所以响应时间不那么重要。所以给的 CPU 时间片就全部用满，这种就要降低它的优先级。

慢慢的，这些任务就会在队列中分开来，交互式的任务在最上面（优先级最大）。所以，如果一个任务在时间片执行过程中没用完，那么它的优先级不变，用满时间片的就不断沉到底去。这时候我们发现 SJF 等策略就是这个策略的极端情况，不需要知道执行时间也可以得到类似的性能。



上图可以看到，对于 turn around time 和 response time 的表现都非常好。右图是一个经常 IO 的任务和经常使用 CPU 的任务一起使用 CPU 的情况，表现也非常好。

#### Notes

- 
- When a job arrives, the MLFQ
    - first assumes it might be a short job
      - Thus giving the job high priority
    - If it actually is a short job
      - it will run quickly and complete
    - If it is not a short job
      - it will slowly move down the queues
      - soon prove itself to be a long-running process
    - If it is an interactive job doing a lot of I/O
      - it will relinquish the CPU before its time slice is complete
      - simply keep it at the same level without penalizing

当你到达的时候，你每次 CPU 都用完了，那么你慢慢地就沉下去（纠正预判）。为什么大家初始化先是优先级最高（short job）的呢，因为我们可以花几个时间片来预测出来，否则预测错的代价对于 long running 任务来说代价是比较小的，而对于 short 任务来说，如果预测错了，可能一直都没有执行的机会导致一开始的响应时间非常大，这个代价是更大的。大家平时使用电脑的时候，CPU 利用率都是比较低的。

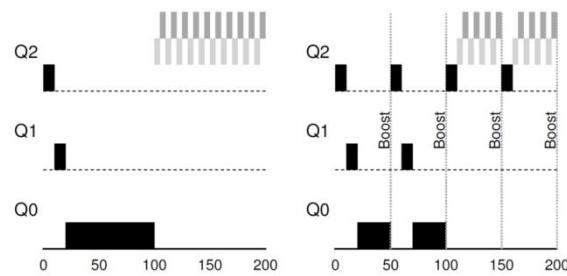
刚才讲的是比较 Naive 的，但是会有几个问题。

1. 饥饿问题，如果交互式任务太多了，下面的 long-running 任务一次都没有机会得到执行，因为没有上升的机制。
2. 一些小技巧可以骗取更多的 CPU，比如我们是一个 long-running 任务，在时间片结束之前来主动调用 yield，这个任务就可以和交互式任务来抢 CPU。
3. 程序可能会改变行为，开始是交互式的，然后变成 long-running，再变成交互式的，

这该怎么办？因为我们的策略是单向的，没有上升的策略。

为了解决这些问题，需要增加一些规则。提供一个往上的方法，采用一个非常暴力的方式，一段时间之后把下面的这些任务直接放到最上面，再让所有任务沉一次。

### Priority Boost

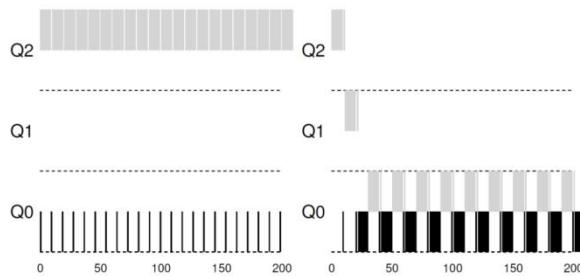


1. 解决了 starvation，放上去的时候有机会执行一点。
2. 行为变化以后，比如从 long-running 变成交互式的，放上去以后就不会再沉下去了。

当然这个周期肯定比时间片要长很多，比如 50 个时间片放一次，开销并不大，但是可以解决饥饿和程序行为变化的问题。我们不需要设计很复杂的规则，就能得到一个很好的性能。

那么怎么解决骗取 CPU 的 gaming 问题呢？其实你需要的是一个更好的计数机制，本来我们是 0 和 1 的一个维度，现在我们要统计这个时间片使用的比例是多少。如果要使用小技巧，需要占用很多 CPU，最终再自己放掉。现在我们采用的策略是看时间片里时间的使用情况，也就是我们在时间片的什么时候放弃的 CPU，只有比较早放弃的任务我们认为是交互式的任务，维持较高的优先级，否则就踢到下面去（存在一个阈值）。

### Better Accounting



我们发现越复杂的策略就会存在一些需要调参的参数，比如时间片的大小，队列的个数，时间片的阈值等。调参一般是使用评测基准（benchmark），在这个测试集合包含了典型的 workload。优先级越低的时间片应该越大，因为它得到执行说明上面都没任务，不需要很频繁的进行 context switch。

## OUSTERHOUT'S LAW

---

- VOO-DOO CONSTANTS
  - require some form of black magic to set them correctly
- Avoiding voo-doo constants is a good idea whenever possible
  - Unfortunately, it is often difficult
  - One could try to make the system learn a good value
    - but that too is not straightforward
- The frequent result:
  - a configuration file filled with default parameter values
  - a seasoned administrator can tweak when something isn't quite working correctly
    - And is left unchanged

魔法数字是要尽可能避免的，因为我们说不清它出问题到底是什么原因。这些数字被集合在一起就是配置文件。它以参数的方式暴露出来给程序员作为一些配置选项。