I/O通道

> 这部分看自己的书，手头的资料实在没找到什么详细的对通道的介绍。。。

通道指令类型单一，没有自己的内存，通道所执行的通道程序是放在主机内存中的，也就是说通道与CPU共享内存

最短路径

Dijkstra算法

贪心

> **参考链接：** https://zhidao.baidu.com/question/654923862590347765.html
>
> dijkstra由于是贪心的，每次都找一个距源点最近的点（dmin），然后将该距离定为这个点到源点的最短路径（d[i]<--dmin）；**但如果存在负权边，那就有可能先通过并不是距源点最近的一个次优点（dmin'），再通过这个负权边L(L<0)，使得路径之和更小（dmin'+L<dmin）,则dmin'+L成为最短路径，并不是dmin**，这样dijkstra就被囧掉了

```
const int INF = 0x3fffffff;

/**
vector<node> G[]: 图(邻接表表示)
int s: 起点
int n: 图的顶点数
*/
void Dijkstra(vector<node> G[], int s, int n) {
    int d[n];    // 记录最短路径
    bool vis[n]; // 访问控制
    fill(d, d+n, INF);
    fill(vis, vis+n, false);

    d[s] = 0; // 初始化起点为0

    for (int i = 0; i < n; i++) {
        /*
            从未访问的顶点集中找到与起点s的最短距离最小的一个顶点，记为u
            此处可用优先队列优化
        */
        int u = -1, MIN = INF;
```

```
            for (int j = 0; j < n; j++) {
                if (vis[j] == false && d[j] < MIN) {
                    u = j;
                    MIN = d[j];
                }
            }

            // 找不到满足要求的顶点，说明剩下的顶点和起点s不连通
            // 或者说所有顶点都处理完了
            if (u == -1) return;

            // 标记顶点u已访问
            vis[u] = true;

            /*
                处理与顶点u相邻的未访问的顶点
            */
            for (int j = 0; j < G[u].size(); j++) {
                /*
                    u->v
                    如果v未访问，且以u为中介点可以是d[v]更短
                    则更新d[v]
                */
                int v = G[u][j].v;
                if (vis[v] == false && d[u]+G[u][j].dis < d[v]) {
                    d[v] = d[u]+G[u][j].dis;
                }
            }
        }
    }
```

Floyd算法

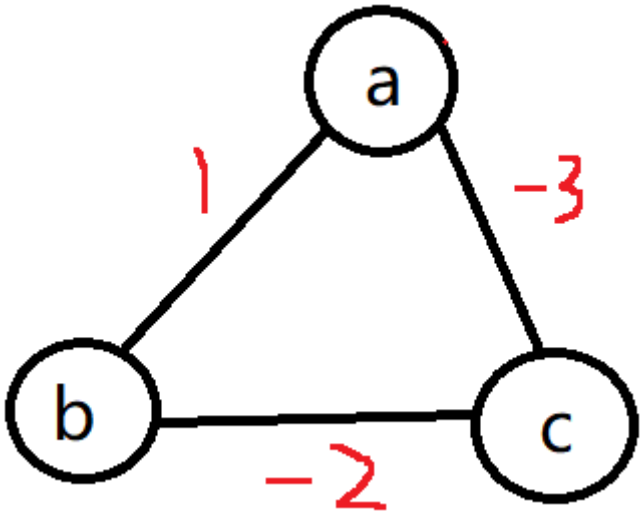
动态规划的状态转移方程： `d[i][j] = min(d[i][j], d[i][k]+d[k][j])`

适用于稠密图

Floyd算法允许图中有带负权值的边，但不允许包含负权边组成的回路

理论点的说法就是有了负边环后算法的最优子结构就不满足了

举例子的话以下面这个图为例，可能会a->b->c->a->b->c……，算法绕着负环反复走，就一直小下去了




```
d[0][0] = INF;
d[0][1] = 1;
d[0][2] = -3;
d[1][0] = 1;
d[1][1] = INF;
d[1][2] = -2;
d[2][0] = -3;
d[2][1] = -2;
d[2][2] = INF;
Floyd();
```

```
C:\Users\Adm
-6 -5 -9
-5 -4 -8
-9 -8 -12

Process retur
Press any key
```

Floyd算法具体流程

1. 枚举顶点 `k` ∈[1, n]
2. 以k为中介点，枚举所有顶点对 `i` 和 `j` (`i` ∈[1, n], `j` ∈[1, n])
3. 如果 `d[i][k]+d[k][j] < d[i][j]` 则更新路径，在路径中加入顶点k为中间结点： `d[i][j] = d[i][k] + d[k][j];`

   【逐步尝试在原路径中加入顶点k作为中间结点，若增加中间结点后，得到的路径比原来的路径长度减少了，则以此新路径代替原路径】

```
const int n = 10; // 顶点数
int d[n][n];      // 带权图(邻接矩阵)

// 注意k要写最外面一层循环, 里面的i和j是对称的, 无所谓怎么写
void Floyd() {
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (d[i][k] != INF && d[k][j] != INF
                    && d[i][k]+d[k][j] < d[i][j]) {
                        d[i][j] = d[i][k] + d[k][j];
                    }
            }
        }
    }
}
```

IPC

进程间的通信机制，各自的缺点

- **管道**

  参考2021-03-06的QA

- **消息队列MessageQueue**
- **共享存储SharedMemory**

  多个进程共享同一块内存空间，是最快的IPC方式，可以与信号量什么的结合使用，实现进程间的同步互斥

  管道需要在内核和用户空间进行 四次 的数据拷贝：由用户空间的buf中将数据拷贝到内核中 -> 内核将数据拷贝到内存中 -> 内存到内核 -> 内核到用户空间的buf。而共享内存则只拷贝 两次 数据：用户空间到内存 -> 内存到用户空间

  此处个人觉得只要知道和管道的区别就行了，实现方式可以不看

  参考文章： https://www.cnblogs.com/melons/p/5791787.html

- **信号量Semaphore**

  参考文章： https://www.cnblogs.com/melons/p/5791794.html

  信号量和教材上讲的理论知识几乎一模一样，只看书上理论也没问题

- **信号signal**

  参考文章： https://www.cnblogs.com/melons/p/5791795.html

序号	名称	默认行为	相应事件
1	SIGHUP	终止	终端线挂断
2	SIGINT	终止	来自键盘的中断
3	SIGQUIT	终止	来自键盘的退出
4	SIGILL	终止	非法指令
5	SIGTRAP	终止并转储内存①	跟踪陷阱
6	SIGABRT	终止并转储内存①	来自 abort 函数的终止信号
7	SIGBUS	终止	总线错误
8	SIGFPE	终止并转储内存①	浮点异常
9	SIGKILL	终止②	杀死程序
10	SIGUSR1	终止	用户定义的信号 1
11	SIGSEGV	终止并转储内存①	无效的内存引用（段故障）
12	SIGUSR2	终止	用户定义的信号 2
13	SIGPIPE	终止	向一个没有读用户的管道做写操作
14	SIGALRM	终止	来自 alarm 函数的定时器信号
15	SIGTERM	终止	软件终止信号
16	SIGSTKFLT	终止	协处理器上的栈故障
17	SIGCHLD	忽略	一个子进程停止或者终止
18	SIGCONT	忽略	继续进程如果该进程停止
19	SIGSTOP	停止直到下一个 SIGCONT②	不是来自终端的停止信号
20	SIGTSTP	停止直到下一个 SIGCONT	来自终端的停止信号
21	SIGTTIN	停止直到下一个 SIGCONT	后台进程从终端读
22	SIGTTOU	停止直到下一个 SIGCONT	后台进程向终端写
23	SIGURG	忽略	套接字上的紧急情况
24	SIGXCPU	终止	CPU 时间限制超出
25	SIGXFSZ	终止	文件大小限制超出
26	SIGVTALRM	终止	虚拟定时器期满
27	SIGPROF	终止	剖析定时器期满
28	SIGWINCH	忽略	窗口大小变化
29	SIGIO	终止	在某个描述符上可执行 I/O 操作
30	SIGPWR	终止	电源故障

图 8-26　Linux 信号

> 例：`Ctrl+C`
>
> 1. 用户按下`Ctrl+C`，键盘输入产生一个硬件中断
> 2. CPU从用户态切换到内核态处理该中断
> 3. 终端驱动程序将`Ctrl+C`解释为一个`SIGINT`信号，并记录在进程的PCB中（所谓的"发送一个信号给进程"）
> 4. 当要从内核态返回用户态时，检查PCB中记录的信号，发现有个SIGINT信号待处理，便进行处理。处理的结果就是终止进程，直接杀了

- **套接字Socket**

倒排页表

- 一般用于虚拟空间比实际物理空间大的情况
- **难以实现共享内存**

  共享内存是多个虚拟地址映射到一个物理地址，但反向页表的每个物理页面只有一个虚拟页面条目

2. 倒排页表

针对页式调度层级不断增长的另一种解决方案是**倒排页表**（inverted page table），首先采用这种解决方案的处理器有PowerPC、UltraSPARC和Itanium（有时也被称作Itanic，这款处理器并没有达到Intel所期望的目标）。在这种设计中，实际内存中的每个页框对应一个表项，而不是每个虚拟页面对应一个表项。例如，对于64位虚拟地址，4KB的页，4GB的RAM，一个倒排页表仅需要1 048 576个表项。表项记录了哪一个（进程，虚拟页面）对定位于该页框。

虽然倒排页表节省了大量的空间（至少当虚拟地址空间比物理内存大得多的时候是这样的），但它也有严重的不足：从虚拟地址到物理地址的转换会变得很困难。当进程n访问虚拟页面p时，硬件不再能通过把p当作指向页表的一个索引来查找物理页框。取而代之的是，它必须搜索整个倒排页表来查找某一个表项（n，p）。此外，该搜索必须对每一个内存访问操作都要执行一次，而不仅仅是在发生缺页中断时执行。每次内存访问操作都要查找一个256K的表不是一种使机器快速运行的方法。

走出这种两难局面的办法是使用TLB。如果TLB能够记录所有频繁使用的页面，地址转换就可能变得像通常的页表一样快。但是，当发生TLB失效时，需要用软件搜索整个倒排页表。实现该搜索的一个可行的方法是建立一张散列表，用虚拟地址来散列。当前所有在内存中的具有相同散列值的虚拟页面被链接在一起，如图3-14所示。如果散列表中的槽数与机器中物理页面数一样多，那么散列表的冲突链的平均长度将会是1个表项的长度，这将会大大提高映射速度。一旦页框号被找到，新的（虚拟页号，物理页框号）对就会被装载到TLB中。

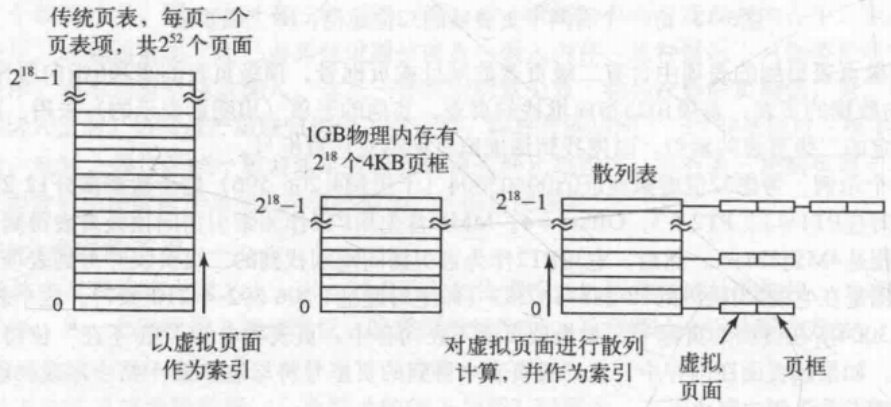


图3-14 传统页表与倒排页表的对比

数据库事务、封锁

事务

事务的概念

所谓事务是用户定义的一个数据库操作序列，这些操作 要么全做，要么全不做 ，是一个 不可分割 的工作单位

在SQL中，定义事务的语句一般有3条： `BEGIN TRANSACTION` 、 `COMMIT` 、 `ROLLBACK`

- `BEGIN TRANSACTION`：事务开始
- `COMMIT`：提交事务的所有操作（将事务中所有对数据库的更新写回到磁盘上的物理数据库中）
- `ROLLBACK`：回滚，系统将事务中对数据库的所有已完成的操作全部撤销，回滚到事务开始时的状态

事务的ACID特性

- `原子性Atomicity`：事务是数据库的逻辑工作单位，事务中包括的诸操作 要么都做要么都不做

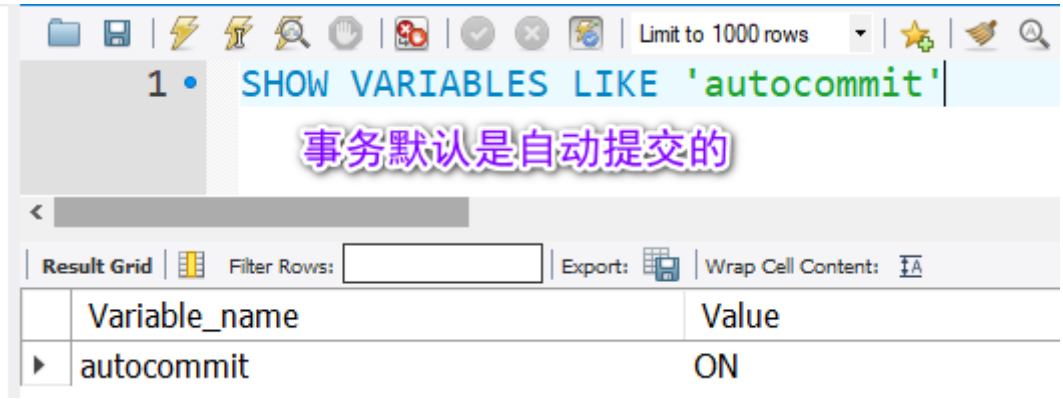
- `一致性Consistency`：事务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态

  如果事务尚未完成就被迫中断，未完成的事务操作对数据库所做的修改有一部分已经写入物理数据库，则此时数据库就处于不一致的状态
- `隔离性Isolation`：一个事务的内部操作及使用的数据对其他并发事务是隔离的，`并发执行的各个事务之间不能相互干扰`

  ⚠事务操作不要交叉进行!
- `持续性Durability`：一个事务一旦提交，它对数据库中数据的改变就应该是永久性的，其他操作或故障不应对其执行结果有任何影响

AUTOCOMMIT

`MySQL 默认采用自动提交模式`。也就是说，如果不显式使用`START TRANSACTION`语句来开始一个事务，那么每个查询都会被当做一个事务自动提交



并发一致性问题

- 丢失修改：经典例子就是操作系统中的`i+1`那个
- 不可重复读：不可重复读指在一个事务内多次读取同一数据集合。`在这一事务还未结束前，另一事务也访问了该同一数据集合并做了修改`，由于第二个事务的修改，第一次事务的两次读取的数据可能不一致

  如果`"修改"`是指`增加/删除数据`，则这种情况下的不可重复读又叫`幻影读`
- 读脏数据：读脏数据指在不同的事务下，当前事务可以读到另外事务未提交的数据（比如ROLLBACK回去的情况）

封锁

封锁粒度

封锁粒度越大，系统开销越小，并发度越小

封锁粒度越小，系统开销越大，并发度越大

- **表锁**
  - 优点：简单，上锁/释放锁的速度快；由于每次锁整个表，因此能有效避免死锁

    - 缺点：锁粒度大，并发度低
- **行锁**
    - 优点：锁粒度小，争用率低，并发度高
    - 缺点：上锁速度慢，且容易死锁

封锁类型

- **排他锁（Exclusive）**，简称**X锁**，又称**互斥锁、写锁**

> 若事务T给数据对象A加上了X锁，则其他所有事务都不能再给A加上任何类型锁

- **共享锁（Shared）**，简称**S锁**，又称**读锁**

> 若事务T给数据对象A加上了S锁，则其他所有事务都只能再给A加上S锁，而不能加X锁

- MySQL的InnoDB引擎中，共享锁和排他锁是标准的行锁，但也支持表锁

  在锁定机制的实现过程中为了让行级锁定和表级锁定**共存**，InnoDB也同样使用了**意向锁（表级锁定）**的概念

  InnoDB实现了二段锁协议

  select不会加任何锁，而update、insert、delete时默认加行级别的写锁

> # 在select语句最后加上 `lock in share mode` 可以显式的给select加上读锁

- MySQL的MyISAM引擎中，默认用的是表锁，不支持行锁

  select时默认加表级别的读锁，而update、insert、delete时默认加表级别的写锁

  在同一个事务里，如果已经获取了一个表的锁，则对没有锁的表不能进行任何操作，否则报错

> # 使用下面语句显示加锁/撤销锁
> `lock tables 表名 read/write;`
> `unlock tables;`
> # 在select语句最后加上 `for update` 也可以显式的给select加上写锁

封锁协议

- **一级封锁协议**

  事务T在修改数据R之前必须先加X锁，直到事务结束才释放

- **二级封锁协议**

  事务在读取数据之前必须加上读锁，读完后即可释放

- **三级封锁协议**

  事务在读取数据之前必须加上读锁，事务结束后才可释放

意向锁

意向锁的含义：如果对一个结点加意向锁，则说明该结点的下层结点正在被加锁；对任一结点加锁时，必须先对它的上层结点加意向锁

虽说一般教材上会有多级粒度树什么的，但 实际情况而言意向锁都是表锁 ，针对的是数据行。可以这样想，如果意向锁是行锁，那么行更新需要遍历表中所有数据，但如果意向锁是表锁的话，只要判断一次就知道有没有数据行被锁定

意向锁的作用就是当一个事务在需要获取资源锁定的时候，如果遇到自己需要的资源已经被 排他锁 占用的时候，该事务可以需要锁定行的表上面添加一个合适的意向锁。如果自己需要一个 共享锁 ，那么就在表上面添加一个 意向共享锁 。而如果自己需要的是某行（或者某些行）上面添加一个 排他锁 的话，则先在表上面添加一个 意向排他锁 。

- IS锁 和 IX锁 都是 表锁
  - IS锁 用来表示一个事务想在表中的某个数据行上加 S锁

    一个事务在获得某个数据行对象的S锁之前，必须获得表的IS锁或更强的锁
  - `IX锁 用来表示一个事务想在表中的某个数据行上加 X锁

    一个事务在获得某个数据行对象的 X 锁之前，必须先获得表的 IX 锁

锁兼容

- 任意意向锁都是相互兼容的，因为意向锁只是表示 想要 对表加锁，而不是真正的加锁

	共享锁（S）	排他锁（X）	意向共享锁（IS）	意向排他锁（IX）
共享锁（S）	✔	✘	✔	✘
排他锁（X）	✘	✘	✘	✘
意向共享锁（IS）	✔	✘	✔	✔
意向排他锁（IX）	✘	✘	✔	✔

意向锁存在的意义： 使得表锁和行锁能够共存

如果没有意向锁的话，事务A锁住表中一个数据行（写锁），事务B锁住整个表（写锁），但事务B锁住整个表的话理应能修改表中的任意一行，而其中某一行却被事务A上了写锁，因此会产生冲突