同步：基础
Synchronization: Basics

钮鑫涛
南京大学
2026春

本门课由蒋炎岩和钮鑫涛共同构建

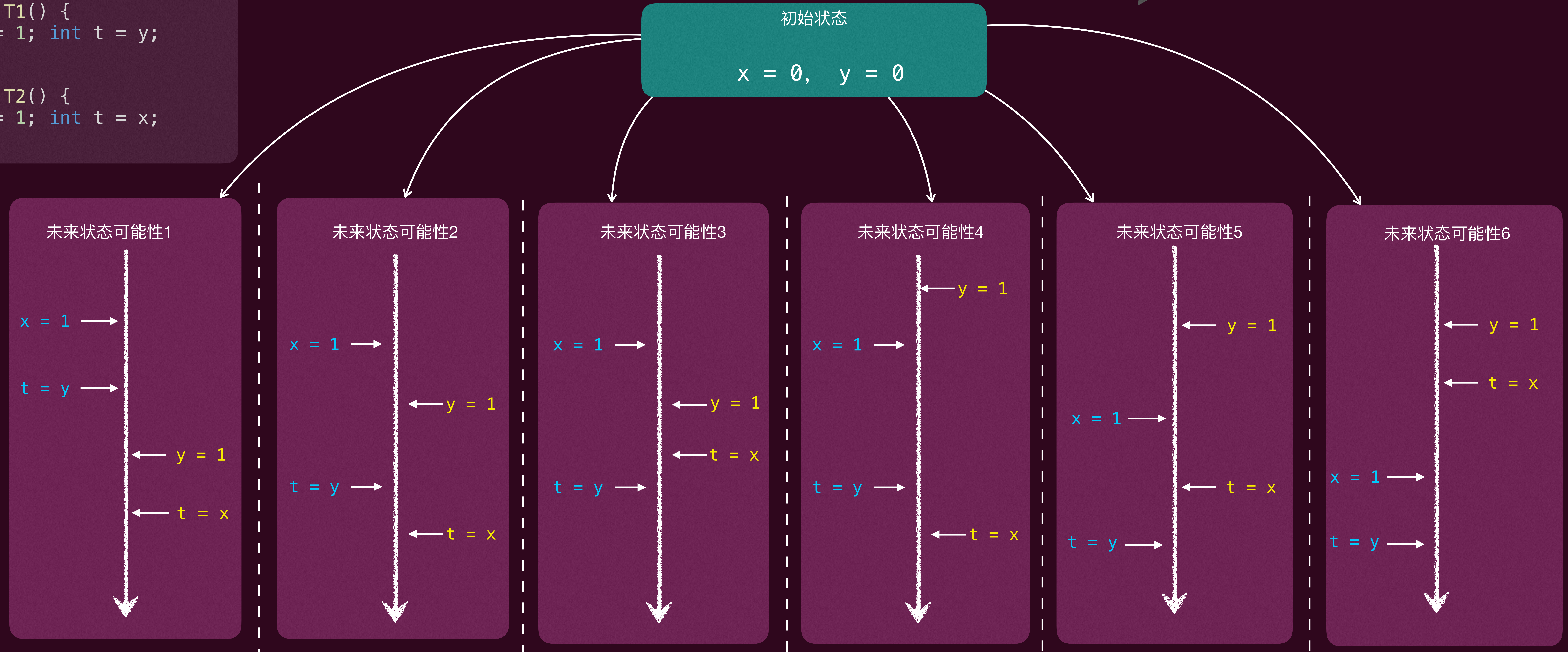
我穿越了时空 预知未来
I went forward in time to view alternate futures.
看到了这场战斗
To see all the possible outcomes
所有可能的结果
of the coming conflict.
14,000,605种
14,000,605.
我们胜了的有几种?
How many did we win?
一种
One.
~~惊队别打了，灭霸要被你打死了~~
史塔克，就是现在了，~~再不出手惊队又回来了~~

并发的困难：“未来”可能性太多

- 并发程序的语句执行顺序是不确定的

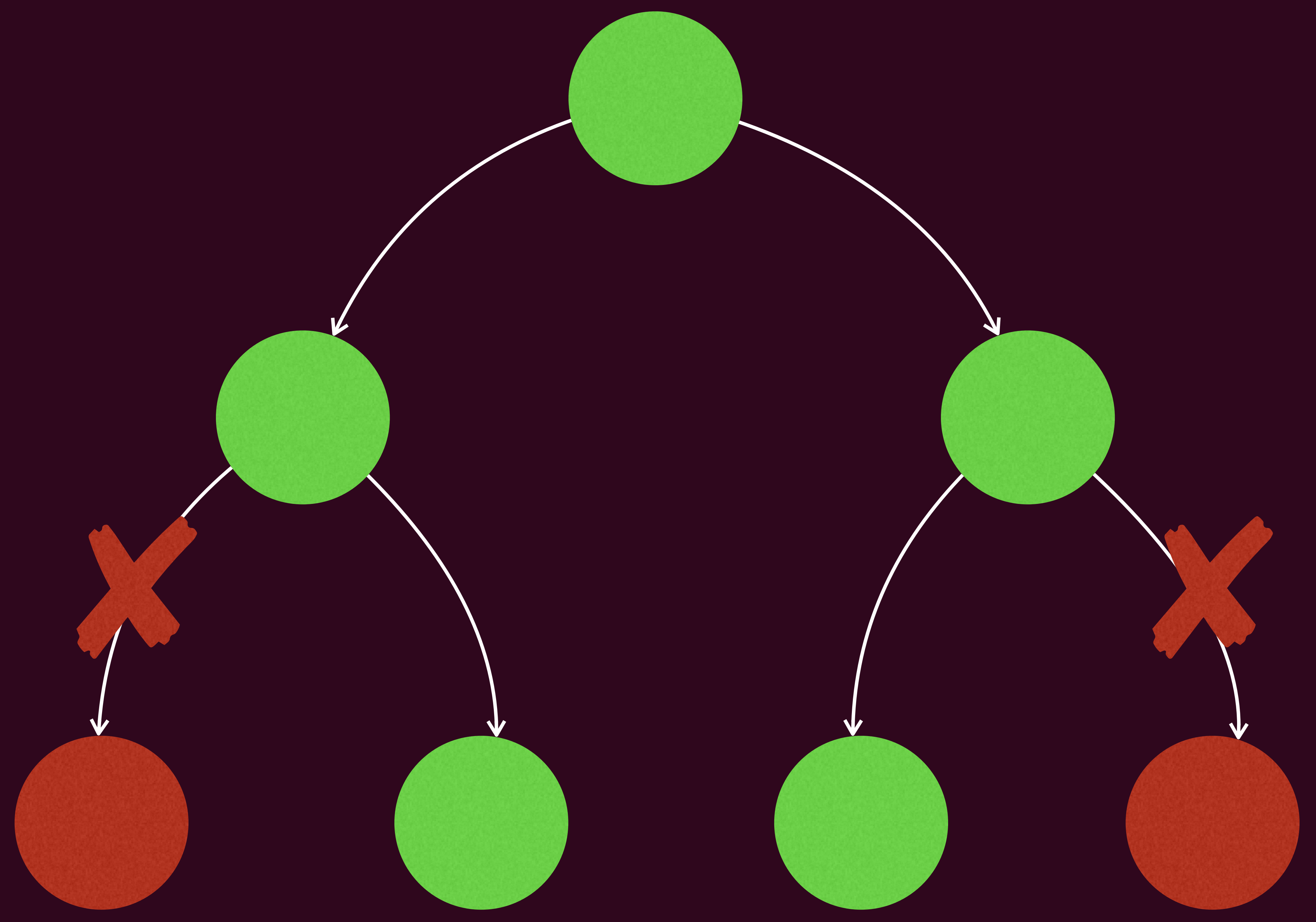
```
int x = 0, y = 0;

void T1() {
  x = 1; int t = y;
}

void T2() {
  y = 1; int t = x;
}
```

非顺序一致性内存模型有更多可能



我们关心的状态！

- 对于我们线性动物而言，很难枚举思考所有可能的状态（特别是线程数非常多的情况下）
- 然而并发程序可能会出现一些我们不想要的状态
  - 比如两个线程同时进入临界区
- 对应地，可能有一些状态我们希望并发程序达成
  - 比如当我们调用pthread_join时，我们希望达成的状态时其他线程都结束了，主线程（即创建其他线程的线程）才能继续行进



受控制的线程：同步

- 为了达成/避免一些状态，我们需要线程受控制，其往往需要：
  - 不能太快（比某些线程快）到达某个状态 （比如在某个线程出临界区之前不能提前进入临界区这个状态）
  - 不能太慢（比如某个状态达成了，就得行进下去）
- 同步(Synchronization)
  - 控制并发，使得 “两个或两个以上随时间变化的量在变化过程中保持一定的相对关系”
  - 互斥也是一种同步

只使用互斥往往很难达成想要的状态

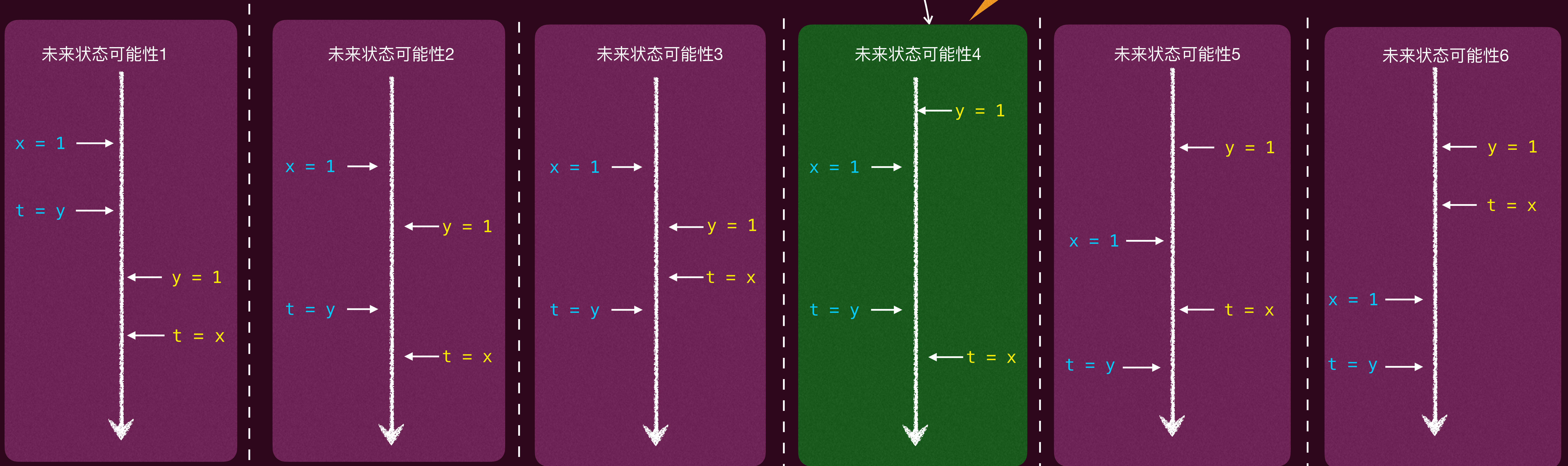
- 如果我只想要进入某个状态，简单加锁就可以吗？—当然不行

```
int x = 0, y = 0;

void T1() {
 lock(); x = 1; unlock(); lock(); int t = y; unlock();
}

void T2() {
 lock(); y = 1; unlock(); lock(); int t = x; unlock();

}
```

初始状态

x = 0, y = 0

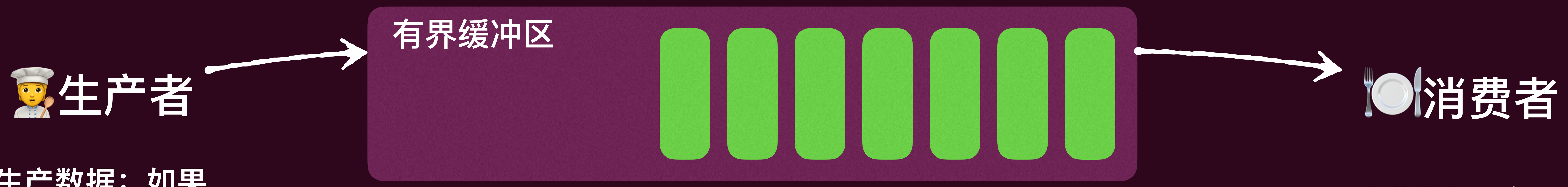
简单的互斥（加锁、解锁）并不能控制线程的执行顺序！



生产者-消费者问题

生产者-消费者问题

- 一个或多个线程共享一个数据缓冲区问题，该缓冲区容量有上限（有界缓冲区）
- 线程分为两类：一类生产数据（生产者），一类消费数据（消费者）
- 该问题由Dijkstra首先给出，可以代表绝大部分并发问题



生产数据：如果缓冲区有空位，放入；否则等待

有界缓冲区是生产者和消费者共享的一个空间

消费数据：如果缓冲区有数据，取走；否则等待

来自大神的鸡汤

EWD1055A_0

- Raise your standards as high as you can live with, avoid wasting your time on routine problems, and always try to work as closely as possible at the boundary of your abilities. Do this because it is the only way of discovering how that boundary should be moved forward.

- We all like our work to be socially relevant and scientifically sound. If we can find a topic satisfying both desires, we are lucky; if the two targets are in conflict with each other, let the requirement of scientific soundness prevail.

- Never tackle a problem of which you can be pretty sure that (now or in the near future) it will be tackled by others who are, in relation to that problem, at least as competent and well-equipped as you are.

生产者-消费者问题的简化

- 一个简化版问题（打印括号）
  - 生产 = 打印左括号 (push into buffer)
  - 消费 = 打印右括号 (pop from buffer)
  - 缓冲区：括号的嵌套深度
    - 比如：(((意味着在缓冲区生产了三个未被消费的括号
    - 比如：((())((也意味缓冲区生产了三个未被消费的括号：一开始生产了3个，然后被消费了两个，接着又生产了2个

```
void produce() { printf("("); }
void consume() { printf(")"); }
```

然而：不是所有的打印括号序列都是合法的，比如在缓冲区容量为3的前提下：
()))是非法的，此时缓冲区没有数据了，无法消费
(((()))) 也是非法的，生产了超过缓冲区容量的数据

生产者-消费者问题的简化

- 我们要实现同步，其实就是需要生产者和消费者不能不受控制的打印括号，而是在某些“条件”满足的情况下才能允许打印：受控的线程！

```
void produce() {
    wait_until(括号深度 < n) {
        printf("(");
    }
}

void consume() {
    wait_until(括号深度 > 0) {
        printf(")");
    }
}
```

括号深度的判定是trivial的，问题在于如何实现`wait_until`原语

没学过并发编程的尝试

- 生产者直接判断depth < n，为真意味着缓冲区有空间，打印(，并标记使用缓冲区的数量增加1，否则什么都不做然后自旋等待

- 消费者直接判断depth > 0，为真意味着缓冲区有数据，打印)，并标记使用缓冲区的数量减少1，否则什么都不做然后自旋等待

该尝试显然是错的，因为共享变量depth的访问部分是临界区代码，应该用互斥锁保护，否则出现竞态条件，违背安全性

```
int n; //缓冲区大小
int depth = 0; //当前使用了多少空间

void produce() {
    while(1) {
    retry:
        int ready = (depth < n);//是否还有空间
        if (!ready) goto retry; //没有空间，自旋等待

        printf("("); //只有还有空间才能打印
        depth++; //打印之后，缓冲区使用的空间加一
    }
}
```

```
void consume(){
    while(1){
    retry:
        int ready = (depth > 0);//是否有数据
        if(!ready) goto retry; //没有数据，自旋等待

        printf(")");//有数据才能打印
        depth--; //打印之后，缓冲区使用的空间减一
    }
}
```

生产者-消费者问题解决尝试1

- 利用互斥锁保护共享变量depth，使其访问都是原子的，这个尝试是对的吗？

此时的ready为true的话, depth < n 一定hold的吗？不一定！语句 “mutex_unlock(&lk);”和“if (!ready){” 之间可能depth已经被别的线程篡改了

```
void produce() {
    while(1) {
    retry:
        mutex_lock(&lk);
        int ready = (depth < n);
        mutex_unlock(&lk);
        if (!ready) goto retry;

        mutex_lock(&lk);
        printf("(");
        depth++;
        mutex_unlock(&lk);
    }
}
```

此时的ready为true的话, depth > n 一定hold的吗？不一定！同样，语句 “mutex_unlock(&lk);”和“if (ready){” 之间可能depth已经被别的线程篡改了

```
void consume(){
    while(1){
    retry:
        mutex_lock(&lk);
        int ready = (depth > 0);
        mutex_unlock(&lk);
        if (!ready) goto retry;

        mutex_lock(&lk);
        printf(")");
        depth--;
        mutex_unlock(&lk);
    }
}
```

生产者-消费者问题解决尝试2

- 之前的问题在于unlock和再次判断可能被打扰，那么我们直接把unlock推迟?
  - 虽然正确，但是会有性能问题

```
void produce() {
    while(1) {
    retry:
        mutex_lock(&lk);
        int ready = (depth < n);
        if (!ready){
                mutex_unlock(&lk);
                goto retry;
        }

        printf(“(”);
        depth++;
        mutex_unlock(&lk);
    }
}
```

```
void consume(){
    while(1){
    retry:
        mutex_lock(&lk);
        int ready = (depth > 0);
        if (!ready) {
             mutex_unlock(&lk);
             goto retry;
        }

        printf(")");
        depth--;
        mutex_unlock(&lk);
    }
}
```

生产者-消费者问题解决尝试3

- 解决这个性能问题的方法我们也已经知晓：利用操作系统的调度进行阻塞(wait)和唤醒(wakeup)

```
void produce() {
    while(1) {
    retry:
        mutex_lock(&lk);
        int ready = (depth < n);
        if (!ready){
            mutex_unlock(&lk);
            wait(&producer_waiting_list);
            goto retry;
        }

        printf("(");
        depth++;
        wakeup(&consumer_waiting_list);
        mutex_unlock(&lk);
    }
}
```

老问题，唤醒会丢失

```
void consume(){
    while(1){
    retry:
        mutex_lock(&lk);
        int ready = (depth > 0);
        if (!ready) {
            mutex_unlock(&lk);
            wait(&consumer_waiting_list);
            goto retry;
        }

        printf(")");
        depth--;
        wakeup(&producer_waiting_list);
        mutex_unlock(&lk);
    }
}
```

生产者-消费者问题解决尝试4

- 采用futex解决唤醒丢失问题

```
int cnd1 = 0;  //生产者能够生产的条件
int cnd2 = 0;  //消费者能够生产的条件


void produce() {
    while(1) {
    retry:
        mutex_lock(&lk);
        int ready = (depth < n);
        if (!ready){
            int val = cnd1;
            mutex_unlock(&lk);
            futex_wait(&cnd1, val);
            goto retry;
        }

        printf("(");
        depth++;
        atomic_add(&cnd2, 1);
        futex_wake(&cnd2);
        mutex_unlock(&lk);
    }
}
```

```
void consume(){
    while(1){
    retry:
        mutex_lock(&lk);
        int ready = (depth > 0);
        if (!ready) {
            int val = cnd2;
            mutex_unlock(&lk);
            futex_wait(&cnd2, val);
            goto retry;
        }

        printf(")");
        depth--;
        atomic_add(&cnd1, 1);
        futex_wake(&cnd1);
        mutex_unlock(&lk);
    }
}
```

我们实现了条件变量(Condition Variable)

- 条件变量：用于标记某个用于同步条件的变量， 其有两个相关的操作：
  - `cond_wait(cond_t *cv, mutex_t *lk)`
    - 调用之前默认假设lk已经持有锁lk
    - 调用之后原子的阻塞线程和释放锁
    - 被唤醒时重新acquire the lk
  - `cond_signal(cond_t *cv)`
    - 唤醒一个等在条件变量cv上的一个阻塞线程
    - 如果没有线程阻塞在这个条件变量上，什么也不做

Posix对应的函数

- 和互斥锁一样，Posix同样规定了相应的API

```
#include <pthread.h>

pthread_cond_t cv;
int pthread_cond_init(pthread_cond_t *cv, NULL);
int pthread_cond_destroy(pthread_cond_t *cv);

int pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cv);
```

我们实现了条件变量(Condition Variable)

- 条件变量有两个相关操作的实现：

```
typedef struct condtional_varaible {
    unsigned value;
}cond_t;

void cond_wait(cond_t *cv, mutex_t *lk)
{
        int val = atomic_load(&cv->value);

        mutex_unlock(lk);
        futex_wait(&cv->value, val);
        mutex_lock(lk);
}
```

```
void cond_signal(cond_t *cv)
{
        atomic_fetch_add(&cv->value, 1);
        futex_wake(&cv->value);
}
```

*注：这里存在一个微妙的可能错误（包括之前的解决方案4）：整数会溢出，因此实际实现更加复杂

条件变量解决之前的生产者-消费者问题

该实现是错的！考虑如下情况：一个生产者，两个消费者C_1，C_2，首先消费者C_1等待一个空的buffer，然后生产者生产一个数据，调用唤醒函数唤醒C_1，但此时如果C_2抢先一步进入临界区，并消费掉了数据后返回，然后C_1才被真正唤醒进入临界区，但此时，临界区已经没有数据可以消费了，错误！

```
cond_t cv_p = COND_INIT();
cond_t cv_c = COND_INIT();
mutex_t lk = MUTEX_INIT();

void produce() {
    while(1) {
        mutex_lock(&lk);
        if(!(depth < n)){
            cond_wait(&cv_p, &lk);
        }

        printf("(");
        depth++;
        cond_signal(&cv_c);
        mutex_unlock(&lk);
    }
}
```

```
void consume() {
    while(1) {
        mutex_lock(&lk);
        if(!(depth > 0)){
            cond_wait(&cv_c, &lk);
        }

        printf(")");
        depth--;
        cond_signal(&cv_p);
        mutex_unlock(&lk);
    }
}
```

Signal的具体语义

- 产生上述问题的原因是由于调用Signal通知某个线程，和那个线程wait被唤醒不是原子的，这就是Hansen (or Mesa) 语义

T_1

```
cond_signal(&cv_c);
mutex_unlock(&lk);
```

非原子：hansen/mesa语义

原子：Hoare语义

T_2

```
cond_wait(&cv_p, &lk);
```

- 与之相对的，Tony Hoare提出了另一种语义，即cond_signal和cond_wait之间是原子的，cond_signal程序将互斥锁转移到被唤醒的程序，并阻塞自己（只有等被唤醒的程序返回或者再次阻塞才会返回该signal线程，当然互斥锁也需要一并转移回它），因此其他线程无法再进入临界区
  - 这保证了被唤醒线程所等待的那个条件是保持的！

Hansen/Mesa VS Hoare

- Hoare语义下的线程能够在唤醒后知道其所期待的那个条件一定成立，这个性质由原子性保证，也就是因为这个原因，在Hoare语义下其实之前的解决方案是正确的
    - 因此Hoare语义下很多程序性质非常容易证明，深受program language研究人士欢迎，但不受操作系统业界人士欢迎，因为要实现Hoare语义是复杂的
- 相应的Hansen/Mesa虽然不保证安全性，但是实现简单（原子性不保证是自然的，无需调度器做出任何承诺）
    - 因此Hansen/Mesa是目前大部分系统的默认实现（包括Posix、Java中对Signal解释）

这两个语义分别对应两个图灵奖大佬站台：Butler Lampson (for Hansen/Mesa), Tony Hoare (for Hoare)

Hansen/Mesa语义下的正确方案

- Hansen/Mesa语义下的解决办法：唤醒后重新尝试判断条件是否满足,即判定条件那里使用while，而不是if，这也是为什么之前的尝试需要反复goto retry的原因

Rule of thumb: cond_wait must always be called within a loop

```
void produce() {
    while(1) {
        mutex_lock(&lk);
        while(!(depth < n)){
            cond_wait(&cv_p, &lk);
        }

        printf("(");
        depth++;
        cond_signal(&cv_c);
        mutex_unlock(&lk);
    }
}
```

```
void consume() {
    while(1) {
        mutex_lock(&lk);
        while(!(depth > 0)){
            cond_wait(&cv_c, &lk);
        }

        printf(")");
        depth--;
        cond_signal(&cv_p);
        mutex_unlock(&lk);
    }
}
```

一定要用两个条件变量吗?

- 上面的尝试中生产者cond_wait条件变量cv_p，cond_signal条件变量cv_c，而消费者cond_wait条件变量cv_c，cond_signal条件变量cv_c

- 两个条件变量都是需要的吗？能否只用一个条件变量，减少实现的复杂度：

```
void produce() {
    while(1) {
        mutex_lock(&lk);
        while(!(depth < n)){
            cond_wait(&cv, &lk);
        }

        printf("(");
        depth++;
        cond_signal(&cv);
        mutex_unlock(&lk);
    }
}
```

```
void consume() {
    while(1) {
        mutex_lock(&lk);
        while(!(depth > 0)){
            cond_wait(&cv, &lk);
        }

        printf(")");
        depth--;
        cond_signal(&cv);
        mutex_unlock(&lk);
    }
}
```

一定要用两个条件变量吗?

- 考虑如下情形:
  - 有三个线程：消费者C_1、C_2和生产者P_1，此时C_1、C_2首先依次进入临界区，发现数据为空，都阻塞等待
  - 然后P_1进入临界区，生产一个item之后唤醒一个线程，如C_1，在C_1唤醒之前，P_1再次抢先一步进入临界区，发现没有空间（假设缓冲区为1），阻塞自己
  - C_1此时醒了并进入临界区，消费了这个数据之后，试图唤醒其他线程，但唤醒谁是没有保障的（因为都是同一个条件变量）
  - 如果唤醒的是C_2，那么了C_2进入临界区，发现数据为空，阻塞等待，然后C_1也尝试进入临界区，阻塞等待, 而P_1没人唤醒它，也处于阻塞等待状态

一定要用两个条件变量吗?

C_1 C_2 P_2

```
!(depth > 0)
cond_wait(&cv, &lk);
```

```
!(depth > 0)
cond_wait(&cv, &lk);
```

```
depth++;
cond_signal(&cv);
```

```
!(depth < n)
cond_wait(&cv, &lk);
```

Hansen/Mesa 语义

```
depth--;
cond_signal(&cv);
```

错误的唤醒！消费者需要生产者生产数据

```
!(depth > 0)
cond_wait(&cv, &lk);
```

```
!(depth > 0)
cond_wait(&cv, &lk);
```

时间

结合多个好处?

- 多个条件变量使得我们可以合理的signal相应的线程，但是设计有点复杂，单个条件变量逻辑上又会出错，有没有比较好的办法?
  - 力大砖飞（条件覆盖）： `cond_broadcast(cond_t *cv)`
  - 一次唤醒所有线程即可，那些需要被正确唤醒的自然会醒来做相应的事情，而那些不该唤醒的，反正需要while循环一次在此判定是否符合条件，不符合条件就再“睡”即可，因此不会影响正确性
- Posix版本

```
int pthread_cond_broadcast(pthread_cond_t *cv);
```

条件覆盖下的生产者-消费者解决方案

- 使用cond_broadcast可以避免之前出现的问题 （但性能上会有影响）

```
void produce() {
    while(1) {
        mutex_lock(&lk);
        while(!(depth < n)){
            cond_wait(&cv, &lk);
        }

        printf("(");
        depth++;
        cond_broadcast(&cv);
        mutex_unlock(&lk);
    }
}
```

```
void consume() {
    while(1) {
        mutex_lock(&lk);
        while(!(depth > 0)){
            cond_wait(&cv, &lk);
        }

        printf(")");
        depth--;
        cond_broadcast(&cv);
        mutex_unlock(&lk);
    }
}
```

条件变量的使用经验法则

- 除了条件变量外，还得有共享的状态用来进行判定是否可以“生产”或者“消费”
- 使用互斥锁来保护共享的状态以及条件变量的操作
- 在做wait/signal/broadcast 时需要持有互斥锁
- 每次从wait中唤醒，要重新进行条件检查（wait within a while loop）
- 针对不同的条件使用不同的条件变量（除非使用broadcast）

条件变量的应用

找到计算中的依赖关系

- `wait`和`signal`的并发原语是非常强大的，只要你找到程序算法中的计算依赖图，你就可以利用`wait`和`signal`来将算法进行并行化！
- 定义计算图$G = (V, E)$, 其中V时计算事件集，$(u, v) \in E$意味着计算事件u一定要在v之前(happens-before)，即v需要用到u的计算结果，u和v只能串行执行，没有依赖关系（没有从u到v的路径）的可以并行计算

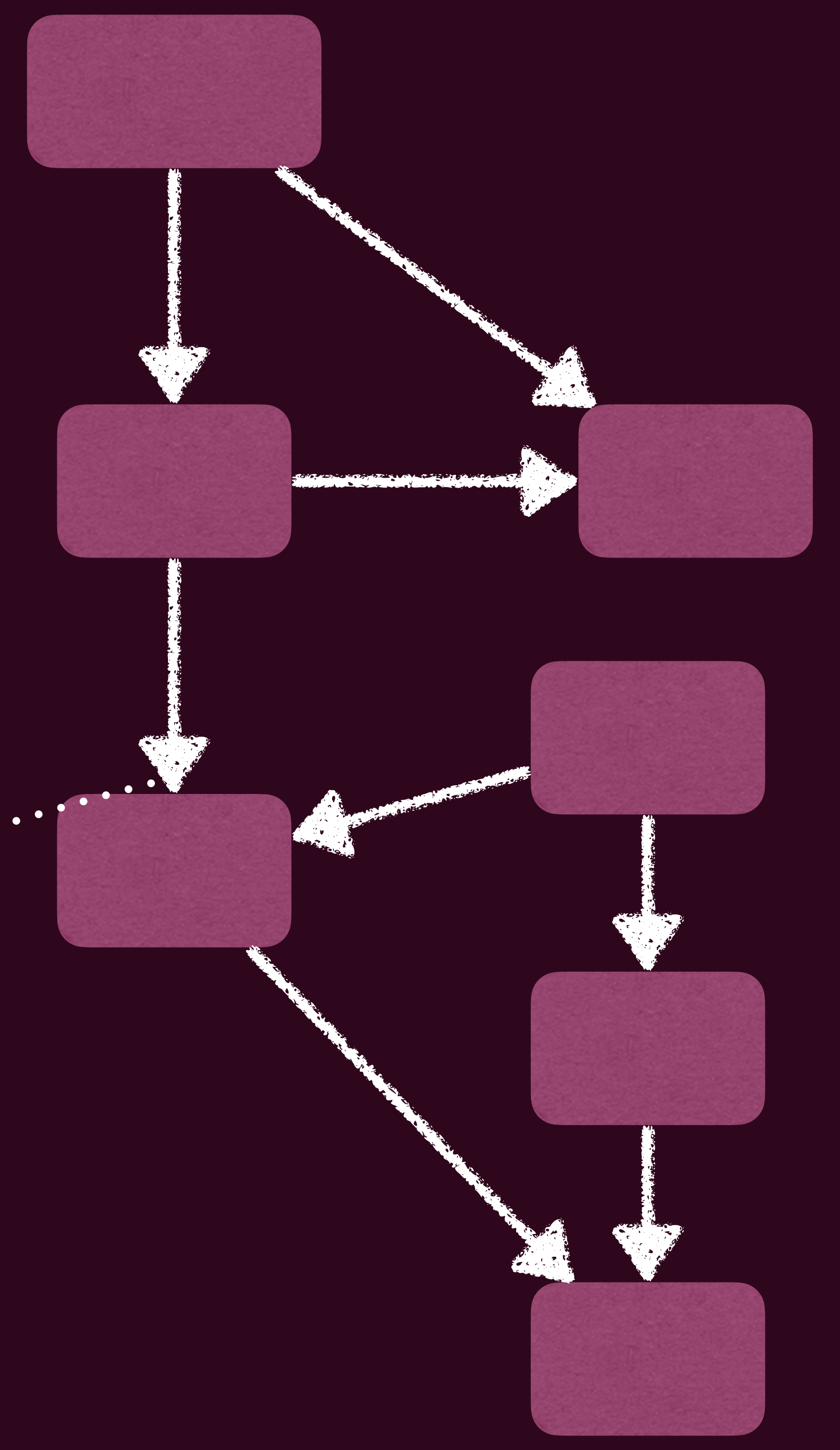
利用计算图进行并行化的方案

- 给定一个G（给定一个问题，首先构建出这样一个图），为每个节点（对应一个线程）完成如下设置
- v节点能进行计算所需要`wait`的同步条件：每一个$\forall u \in V$, s.t. $(u, v) \in E$中的u都已经完成
- u完成后，对$\forall v \in V$, s.t. $(u, v) \in E$中的v进行`signal`(或者更加简单的直接用`broadcast`)

回顾动态规划算法

- 分析问题的结构
  - E.g. [rod-cutting]: (one cut of length i) + (solution for length $n-i$)
- 递归的定义问题的最优解
  - E.g. [rod-cutting]: $r_n = \max\limits_{1 \le i \le n} (p_i + r_{n-i})$
- 计算最优解的值
  - Top-down or Bottom-up. (Usually use bottom-up)

需要构造子问题图，也就是一个有向无环图（DAG）



在单线程下，我们需要计算子问题图的拓扑序，然后依次执行。但在多线程情况下，我们可以并行执行所有这些线程，只需要额外利用条件变量按照依赖图控制线程顺序即可！

回顾例子：Edit Distance问题

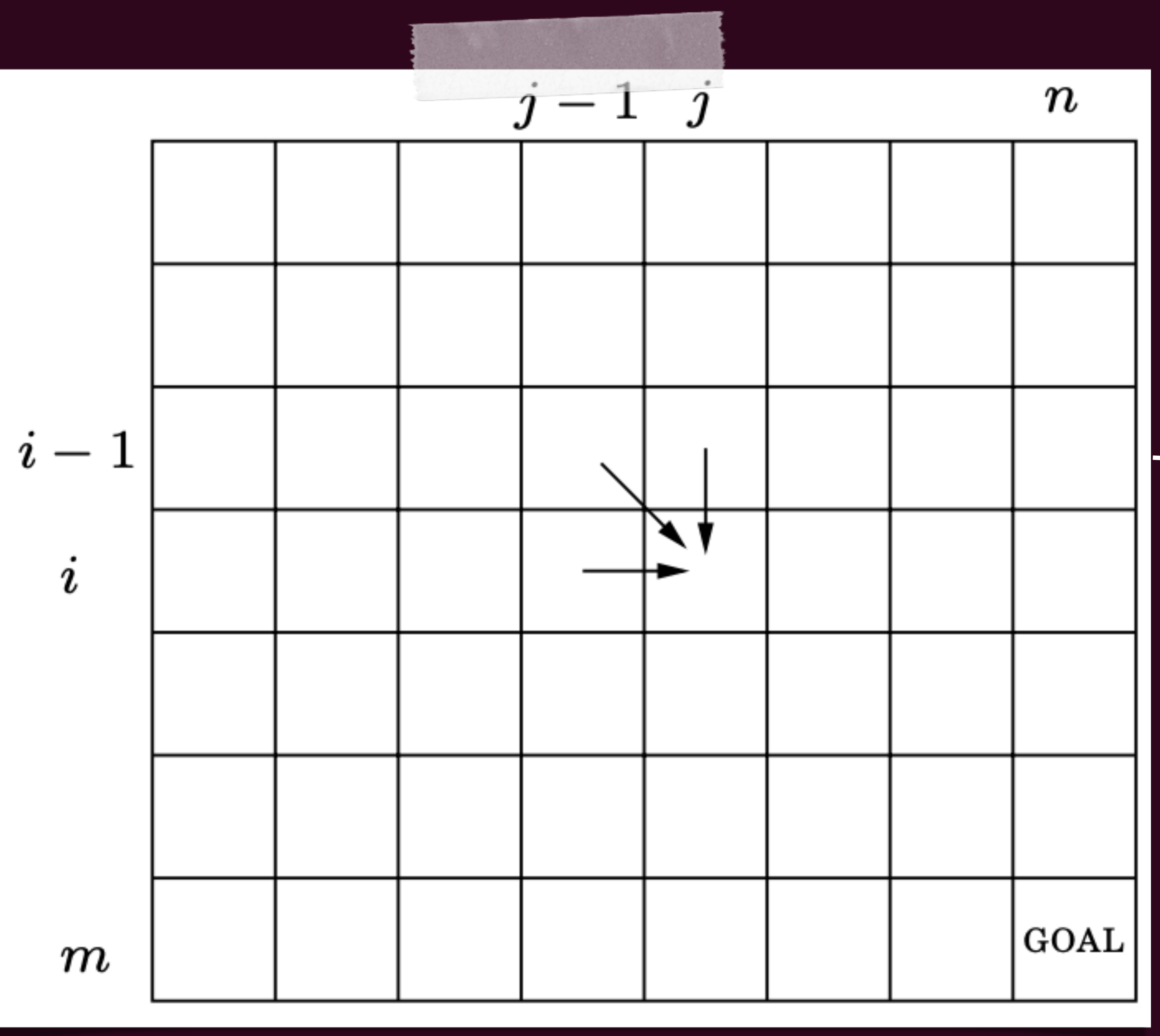
- 给定两个字符串，他们有多相似，即从一个字符串到另一个字符串最少的转化（有删除、插入和替换三种）步骤?

- 考虑匹配两个字符串$A[1 \dots m]$到$B[1 \dots n]$最后一列的情况 $\frac{-}{B[n]}$ or $\frac{A[m]}{B[n]}$ or $\frac{A[m]}{-}$，可得

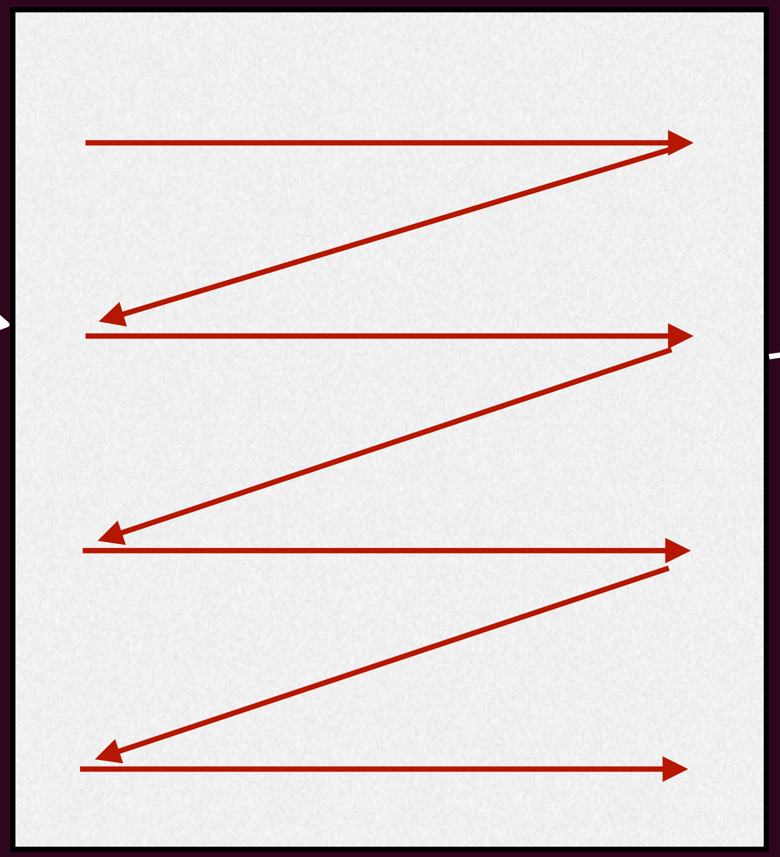
- $$dist(i,j) = \begin{cases} i & if\ j = 0 \\ j & if\ i = 0 \\ \min \left\{ \begin{array}{l} dist(i,j-1)+1 \\ dist(i-1,j)+1 \\ dist(i-1,j-1)+I[A[i] \neq B[j]] \end{array} \right\} & otherwise \end{cases}$$

回顾例子：Edit Distance问题

- 之前的做法：考虑$dist(i,j)$ 依赖于那些计算的解？$dist(i-1,j)$、$dist(i,j-1)$和$dist(i-1,j-1)$



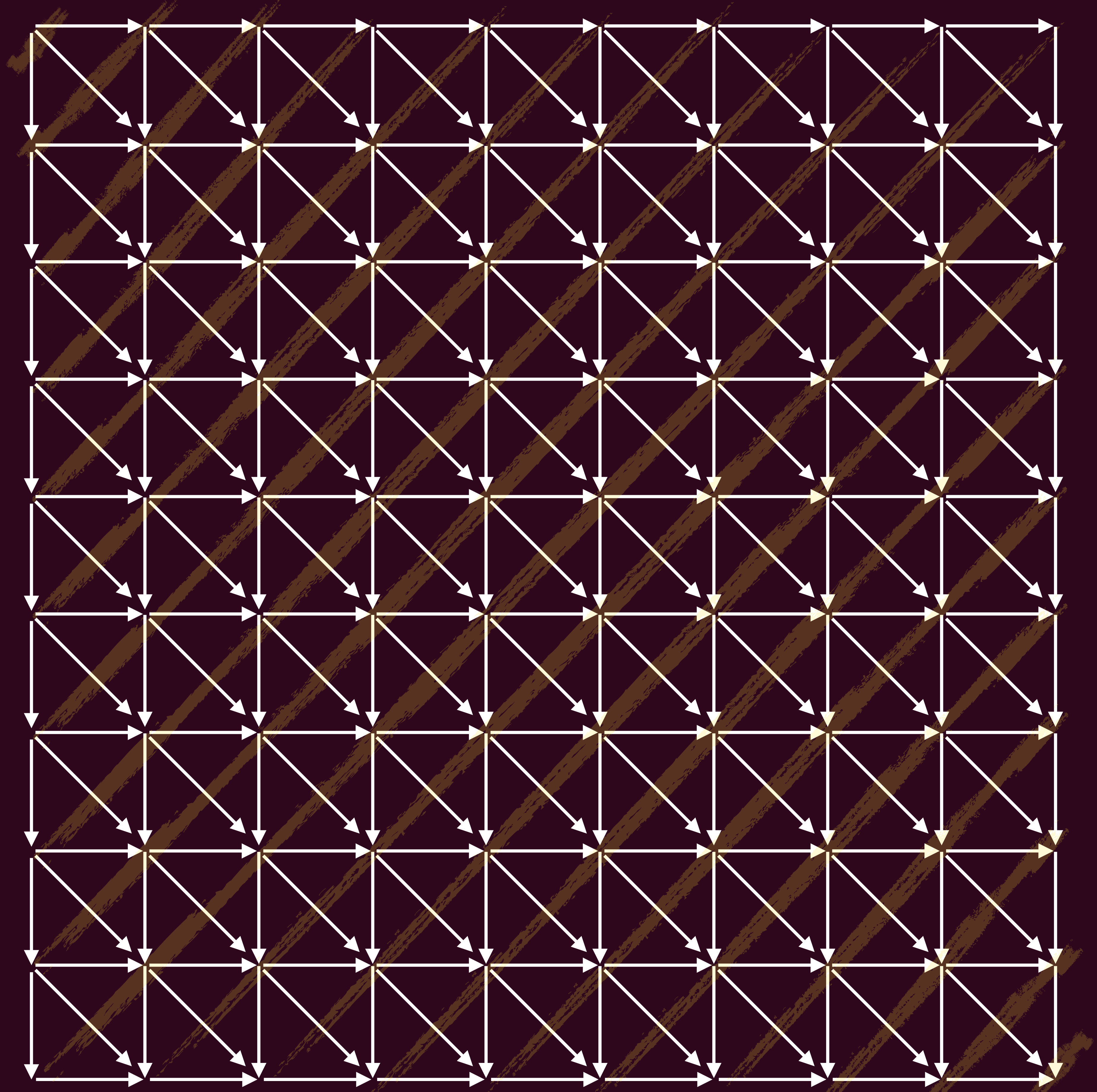
拓扑序



```
EditDistDP(A[1…m],B[1…n]):
for i := 0 to m
    dist[i, 0] := i
for j := 0 to n
    dist[0, j] := j
for i := 1 to m
    for j := 1 to n
        delDist := dist[i - 1, j] + 1
        insDist := dist[i, j - 1] + 1
        subDist := dist[i - 1, j - 1] + Diff(A[i], B[j])
        dist[i, j] := Min(delDist, insDist, subDist)
return dist
```

多线程下的解决方案

- 每一条斜边的节点都是可以其上一轮斜边对应的节点做完之后就可以继续做了（当然，是否为每个节点都设置一个线程具有性能上的考量，线程太多会增加调度成本，线程太少，并发度小）



一个比较通用的并发算法设计框架

- （生产者）可以把调度的事情全部分给一个线程，而不是一次性把所有线程都直接放进内存，这个调度线程维护一个计算图，每次都会根据当前已经做完的计算机点，和依赖关系，调度可以运行的线程，把他们放进ready列表准备运行（这就是线程池）
- （消费者）被调度的运行的线程就执行计算，计算结束后通知调度线程，告知条件可能发生变化，可以重新计算下一批就可以计算的线程

```
void T_worker() {
    while (1) {
        consume().run();
    }
}
void T_scheduler() {
    while (!jobs.empty()) {
        for (auto j : jobs.find_ready()) {
            produce(j);
        }
    }
}
```

一个更加复杂的例子：打印fish

- 有三种线程
  - T_a 若干: 无限循环打印 <
  - T_b 若干: 无限循环打印 >
  - T_c 若干: 无限循环打印 -
- 任务：对线程同步，使得屏幕打印出 <><_ 和 ><>_ 的组合
- 解决方案：使用条件变量，只要回答三个问题
  - 打印 “<” 的条件？打印 “>” 的条件？打印 “_” 的条件？

有限状态机识别“合法”字符串前缀

信号量(Semaphores)

有记忆的同步原语

- 条件变量是无记忆的，因此需要程序员手动进行额外的条件判定
  - 虽然很灵活，但是人类是不断追求简单的生物
- 一个很自然的需求就是能否有一个同步原语能够自带“状态”，然后根据预设好的约定，“状态”不满足就阻塞，否则就唤醒?
  - 一个自带“整数”状态的同步原语就是信号量！

信号量（Semaphore）

- 信号量首次在1965年Dijkstra在“THE multiprogramming system”里引入的与同步有关的原语，其代表着一个整数值变量，只能通过两个**原子**操作能够改变这个变量
  - `P(sem_t *sem)` // P for prolaag (荷兰语, 检验)，也写作decrease/down/wait/acquire
    - 如果sem的值不是正数就阻塞自己，否则将sem的值减一后返回运行
  - `V(sem_t *sem)` // V for verhoog(荷兰语,自增)，也写作increase/up/post/signal/release
    - 将信号量sem的值加1，如果有一个或多个线程阻塞在这个信号量上，选择一个唤醒

Posix对应的函数

- Posix也定义了信号量的使用

```
#include <semaphore.h>
sem_t sem;
int sem_init(sem_t *sem,  int pshared,  unsigned int value);
int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

信号量的一种实现

- 利用互斥锁和条件变量的一种信号量实现

```
typedef struct  _sem_t {
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
} sem_t;

// only one thread can call this
void sem_init(sem_t *s, int value) {
    s->value = value;
    cond_init(&s->cond);
    mutex_init(&s->lock);
}
```

```
void sem_wait(sem_t *s) {
    mutex_lock(&s->lock);
    while (s->value <= 0)
        cond_wait(&s->cond, &s->lock);

    s->value--;
    mutex_unlock(&s->lock);
}
```

```
void sem_post(sem_t *s) {
    mutex_lock(&s->lock);
    s->value++;
    cond_signal(&s->cond);
    mutex_unlock(&s->lock);
}
```

信号量的使用

- 信号量是非常易用的同步原语，可以非常容易实现互斥和控制顺序(happens-before)

- 比如实现互斥（二值信号量）：

X应该为多少?

```
sem_t sem;
sem_init(sem_t &sem, 0, X); //set sem intial value to be X
void func() {
    P(&sem);
     // critical section
    V(&sem);
    // reminder section
}
```

信号量的使用

- 实现顺序控制（happens-before）：下面程序使得T_1得等T_2完成之后才能继续

T_1

```
sem_t sem;
int  main(int argc, char *argv[]) {
    //set sem intial value to be X
    sem_init(sem_t &sem, 0, X);
    pthread_t c;
    create_thread(&c, child);
    // wait for thread
    P(&sem);
    do_something();
    return 0;
}
```

T_2

```
void* child(void *arg) {
    do_something();
    V(&sem); // signal here: child is done
    return NULL;
}
```

X应该为多少?

关于信号量的“整数”理解

- 对信号量的值赋予不同的初值可以有不同的用途
- 可以将这个值看成初始“资源数”
  - 初始“资源数”为1，意味着互斥，只有一个能够获得这个资源，其释放这个资源，其他线程才能获取这个资源
  - 初始“资源数”为0意味着阻塞，当前线程必须等待，只有未来某个线程增加一个资源，该阻塞线程才能继续行进，因此实现了“happens-before”的顺序控制
  - 更多的“资源”意味着当前有很多资源可以用来“获取”，只有资源被获取完为0，才会阻止下一个想要获取“资源”的线程。而只要已经获得资源的线程释放“资源”，下一个才能获取这个释放的“资源”
- 使用P()获取资源，使用V()增加（释放）资源

利用信号量解决生产者-消费者问题

- 使用信号量解决生产者-消费者问题是非常自然和简洁的

```
sem_t empty; //缓冲区有多少空
sem_init(sem_t &empty, 0, MAX);
sem_t full; //缓冲区有多少资源
sem_init(sem_t &full, 0, 0);

void produce() {
    while(1) {
        P(&empty);
        printf("(");
        V(&full);
    }
}
```

```
void consume() {
    while(1) {
        P(&full);
        printf(")");
        V(&empty);
    }
}
```

一个注意点：P、V操作之后都释放相应信号量里的互斥锁了，因此P、V操作之后的代码其实是没有锁保护的，如果之后的操作涉及共享资源（临界区），那么还要加互斥锁保护，这段代码里printf天生是线程安全的，所以没问题

利用信号量解决生产者-消费者问题

- 使用信号量+互斥锁解决更加一般的生成者-消费者问题

```
sem_t mutex; //用于互斥的信号量（也可以用互斥锁）
sem_init(sem_t &mutex, 0, 1);
sem_t empty; //缓冲区有多少空
sem_init(sem_t &empty, 0, MAX);
sem_t full; //缓冲区有多少资源
sem_init(sem_t &full, 0, 0);

void produce() {
    while(1) {
        P(&mutex);
        P(&empty);
        produce_shared_buffer();
        V(&full);
        V(&mutex);
    }
}
```

产生死锁：如果消费者先进行互斥，然后消费，发现没有数据，阻塞，但生产者由于互斥锁此时无法获得，无法进入临界区产生数据，因此生产者和消费者都阻塞，从而无法行进

```
void consume() {
    while(1) {
        P(&mutex);
        P(&full);
        consume_shared_buffer();
        V(&empty);
        V(&mutex);
    }
}
```

利用信号量解决生产者-消费者问题

- 正确方案：使用信号量+互斥锁解决更加一般的生成者-消费者问题

```
sem_t mutex; //用于互斥的信号量（也可以用互斥锁）
sem_init(sem_t &mutex, 0, 1);
sem_t empty; //缓冲区有多少空
sem_init(sem_t &empty, 0, MAX);
sem_t full; //缓冲区有多少资源
sem_init(sem_t &full, 0, 0);

void produce() {
    while(1) {
        P(&empty);
        P(&mutex);
        produce_shared_buffer();
        V(&mutex);
        V(&full);
    }
}
```

条件允许，才进入临界区

```
void consume() {
    while(1) {
        P(&full);
        P(&mutex);
        consume_shared_buffer();
        V(&mutex);
        V(&empty);
    }
}
```

信号量也不是擅长所有问题

- 考虑之前的打印fish问题：
  - 有三种线程 无限打印<、 >和-
  - 任务：对线程同步，使得屏幕打印出 <><_ 和 ><>_ 的组合
  - 假设刚打印完一条完整的鱼<><_ 或 ><>_，那么下一个应该是?
    - 每种线程关联一个信号量，比如打印<就是P(<)
    - 信号量的V操作具有指向性，不太好表达“二选一”

此外，信号量所记录的状态是整数形式，对于非整数的条件判定也不太适用

阅读材料

- [OSTEP] 第30, 31章