

Unix系统高级编程

PART 3

DAY05

内容

上午	09:00 ~ 09:30	作业讲解和回顾
	09:30 ~ 10:20	互斥体与信号量
	10:30 ~ 11:20	
	11:30 ~ 12:20	条件变量
下午	14:00 ~ 14:50	局域网聊天室
	15:00 ~ 15:50	
	16:00 ~ 16:50	总结和答疑
	17:00 ~ 17:30	



互斥体与信号量



并发冲突



并发冲突

- 当多个线程同时访问其共享的进程资源时，如果不能相互协调配合，就难免会出现数据不一致或不完整的问题。这种现象被称为线程间的并发访问冲突，简称并发冲突

- 假设有整型全局变量g_cn被初始化为0

```
int g_cn = 0;
```

- 启动两个线程，同时执行如下线程过程函数，分别对该全局变量做一百万次累加

```
void* start_routine (void* arg) {  
    int i;  
    for (i = 0; i < 1000000; ++i) ++g_cn;  
    return NULL; }  
}
```

- 两个线程结束后，g_cn的值理想情况下应该是两百万，但实际情况却往往少于两百万，且每次运行的结果不尽相同

并发冲突 (续1)

- 理想中的原子 “++”

线程1		内存	线程2	
指令	eax	g_cn	eax	指令
<code>movl g_cn, %eax</code>	0	0		
<code>addl \$1, %eax</code>	1	0		
<code>movl %eax, g_cn</code>	1	1		
		1	1	<code>movl g_cn, %eax</code>
		1	2	<code>addl \$1, %eax</code>
		②	2	<code>movl %eax, g_cn</code>

- 原子操作通常被认为是不可分割的，即在执行完该操作之前不会被任何其它任务或事件中中断。理论上只有在单条指令中完成的操作才可被视为原子操作



并发冲突 (续2)

- 现实中的非原子 “++”

线程1		内存	线程2	
指令	eax	g_cn	eax	指令
movl g_cn, %eax	0	0		
		0	0	movl g_cn, %eax
addl \$1, %eax	1	0		
		0	1	addl \$1, %eax
movl %eax, g_cn	1	1		
		①	1	movl %eax, g_cn

- 一组非原子化的操作常常会因为线程切换而导致未定义的结果。这时就必须借助人力的方法，迫使其被原子化



并发冲突

【参见：TTS COOKBOOK】

- 并发冲突

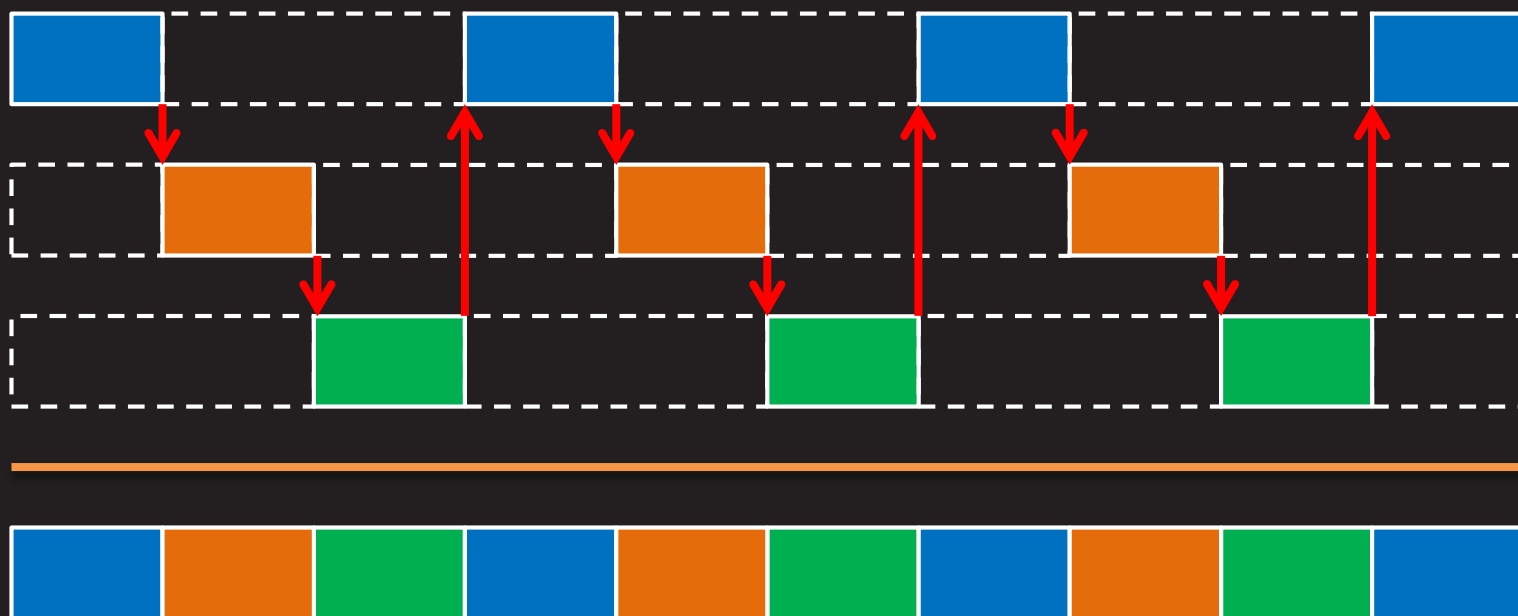


线程同步



线程同步

- 缺省情况下，一个进程中的线程是以异步方式运行的，即各自运行各自的，彼此间不需要保持步调的协调一致
- 某些情况下，需要在线程之间建立起某种停等机制，即一或多个线程有时必须停下来，等待另外一或多个线程执行完一个特定的步骤以后才能继续执行，这就叫同步



线程同步（续1）

- 并发冲突问题
 - 任何时候只允许一个线程持有共享数据，其它线程必须阻塞于调度队列之外，直到数据持有者不再持有该数据为止
- 资源竞争问题
 - 任何时候只允许部分线程拥有有限的资源，其它线程必须阻塞于调度队列之外，直到资源拥有者主动释放其所拥有的资源为止
- 条件等待问题
 - 当某些条件一时无法满足时，一些线程必须阻塞于调度队列之外，直到令该条件满足的线程用信号唤醒它们为止



互斥体



互斥体

- 初始化互斥体

```
#include <pthread_h>
```

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

成功返回0，失败返回错误码

- *mutex* : 互斥体
 - *attr* : 互斥体属性，同步一个进程内的多个线程，取NULL
- 例如
 - `pthread_mutex_init (&g_mutex, NULL);`
 - `pthread_mutex_t g_mutex =`
`PTHREAD_MUTEX_INITIALIZER;`



互斥体（续1）

- 销毁互斥体

```
#include <pthread.h>
```

```
int pthread_mutex_destroy (pthread_mutex_t* mutex);
```

成功返回0，失败返回错误码

- *mutex*：互斥体

- 释放和互斥体*mutex*有关的一切内核资源
- 例如
 - `pthread_mutex_destroy (&g_mutex);`



互斥体（续2）

- 锁定互斥体

```
#include <pthread.h>
```

```
int pthread_mutex_lock (pthread_mutex_t* mutex);
```

成功返回0，失败返回错误码

- *mutex* : 互斥体

- 例如

- `pthread_mutex_lock (&g_mutex);`

- 任何时刻只会有一个线程对特定的互斥体加锁成功，其它试图对其加锁的线程会在此函数的阻塞中等待，直到该互斥体的持有者线程将其解锁



互斥体（续3）

- 解锁互斥体

```
#include <pthread.h>
```

```
int pthread_mutex_unlock (pthread_mutex_t* mutex);
```

成功返回0，失败返回错误码

- *mutex* : 互斥体

- 例如

- `pthread_mutex_unlock (&g_mutex);`

- 对特定互斥体加锁成功的线程通过此函数将其解锁，那些阻塞于对该互斥体加锁的线程中的一个会被唤醒，得到该互斥体，并从pthread_mutex_lock函数中返回



互斥体（续4）

- 例如

- for (i = 0; i < 1000000; ++i) {
 pthread_mutex_lock (&g_mutex);
 ++g_cn;
 pthread_mutex_unlock (&g_mutex);
}

- 任何时候都只会有一个线程执行++g_cn，其它试图执行++g_cn的线程则阻塞于pthread_mutex_lock函数，直到那个执行++g_cn的线程在完成计算后，通过pthread_mutex_unlock函数解锁该互斥体，这时处于阻塞状态的线程之一将被唤醒，并从pthread_mutex_lock函数中返回，执行++g_cn，其它线程继续在阻塞中等待



基于互斥体的线程同步

【参见：TTS COOKBOOK】

- 基于互斥体的线程同步



信号量



信号量

- 初始化信号量

```
#include <semaphore.h>
```

```
int sem_init (sem_t* sem, int pshared,  
              unsigned int value);
```

成功返回0，失败返回-1

- *sem* : 信号量
- *pshared* : 0表示该信号量用于线程间的同步；非0表示该信号量用于进程间的同步。对于后者，信号量被存储在共享内存中
- *value* : 信号量初值



信号量（续1）

- 例如
 - `sem_t g_sem;`
 - `sem_init (&g_sem, 0, 5);`
- 信号量的本质就是一个由系统内核维护的全局计数器，跟踪当前可用资源的数量，借以限制分享资源的线程数
- 信号量初值即资源总数。当有线程获得资源时，信号量计数器的值会相应减少。当其不够减时，试图获取资源的线程会在阻塞中等待
- 已获得资源的线程在其不需要继续使用资源时，会释放手中的资源，令信号量计数器的值回升。这时处于等待状态的线程之一会被唤醒，获得它想要的资源，同时令信号量计数器的值再度减少，以反映可用资源数的变化



信号量 (续2)

- 销毁信号量

```
#include <semaphore.h>

int sem_destroy (sem_t* sem);
```

成功返回0，失败返回-1

- *sem* : 信号量

- 释放和信号量*sem*有关的一切内核资源
- 例如
 - `sem_destroy (&g_sem);`



信号量 (续3)

- 等待信号量

```
#include <semaphore.h>
```

```
int sem_wait (sem_t* sem);  
int sem_trywait (sem_t* sem);  
int sem_timedwait (sem_t* sem,  
    const struct timespec* abs_timeout);
```

成功返回0，失败返回-1

- *sem* : 信号量
- *abs_timeout* : 等候时限(始自UTC19700101T000000)

```
struct timespec {  
    time_t tv_sec; // 秒  
    long   tv_nsec; // 纳秒，取值范围0-999999999  
};
```



信号量（续4）

- 调用以上函数时，若信号量计数器的值大于1，则将其减1并立即返回0，否则
 - `sem_wait`函数会阻塞，直到信号量计数器的值够减时，将其减1并返回0
 - `sem_trywait`函数立即返回-1，并置`errno`为EAGAIN
 - `sem_timedwait`函数会阻塞，直到信号量计数器的值够减时，将其减1并返回0，但最多阻塞到`abs_timeout`时间，一旦超时立即返回-1，并置`errno`为ETIMEDOUT
- 例如
 - `sem_wait (&g_sem);`



信号量 (续5)

- 释放信号量

```
#include <semaphore.h>

int sem_post (sem_t* sem);
```

成功返回0，失败返回-1

- *sem* : 信号量

- 调用sem_post函数将直接导致*sem*信号量计数器的值加1。那些此刻正阻塞于针对该信号量的sem_wait函数调用的线程中的一个将被唤醒，令*sem*信号量计数器的值减1并从sem_wait函数中返回
- 例如

- `sem_post (&g_sem);`

信号量 (续6)

- 获取信号量计数器的当前值

```
#include <semaphore.h>
```

```
int sem_getvalue (sem_t* sem, int* sval);
```

成功返回0，失败返回-1

- *sem* : 信号量
- *sval* : 输出计数器的当前值
- 例如
 - int *sval*;
 - `sem_getvalue` (&*g_sem*, &*sval*);
 - `printf` ("%d\n", *sval*);



信号量（续7）

- 信号量函数的声明不在<pthread.h>中，而是在<semaphore.h>中，成功虽然返回0，但失败并不返回错误码，而是返回-1，同时设置errno
- 互斥量任何时候都只允许一个线程访问共享资源，而信号量则允许最多 *value* 个线程同时访问共享资源。 *value* 为1的信号量与互斥量等价



基于信号量的线程同步

【参见：TTS COOKBOOK】

- 基于信号量的线程同步

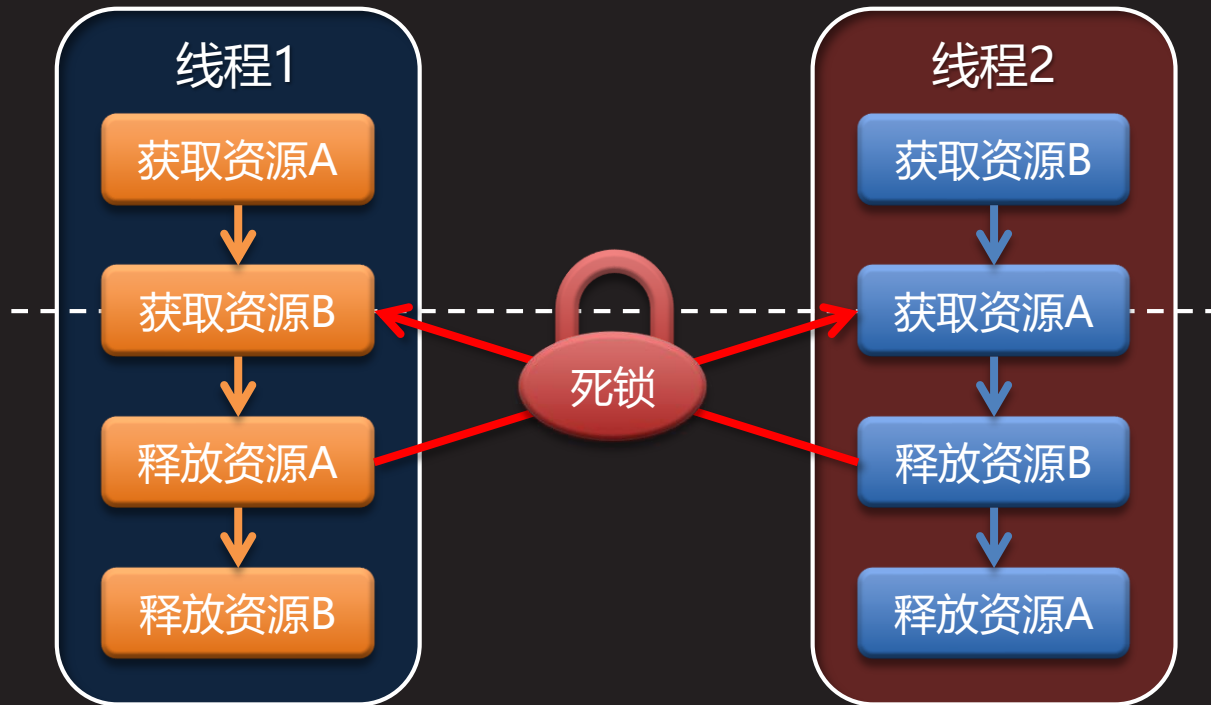


死锁问题



死锁问题

- 死锁系两个或两个以上的线程在运行过程中为了争夺资源，而形成的一种互相等待现象。若无外力作用，它们都将无法进行下去。此时称系统处于死锁状态或系统产生了死锁。这些处于互等待状态的线程则称为死锁线程



死锁问题（续1）

- 产生死锁的四个必要条件

- 独占排它

- 线程以独占的方式使用其所获得的资源，即在一段时间内不允许其它线程使用该资源。这段时间内，任何试图请求该资源的线程只能在阻塞中等待，直到资源被其拥有者主动释放

- 请求保持

- 线程已经拥有了至少一个资源，但又试图获取已被其它线程拥有的资源，因此只能在阻塞中等待，同时对自己已经获得的资源坚守不放

- 不可剥夺

- 线程已经获得的资源，在其未被使用完之前，不可被强制剥夺，而只能由其拥有者自己释放

- 循环等待

- 线程集合 $\{T_0, T_1, T_2, \dots, T_n\}$ 中， T_0 等待 T_1 占有的资源， T_1 等待 T_2 占有的资源，……， T_n 等待 T_0 占有的资源，形成环路



死锁问题（续2）

- 死锁问题的解决方案

- 事前预防

- 通过设置某些限制条件，破坏产生死锁的四个必要条件中的一个或几个，以避免死锁的发生。但如果所设置的限制条件过于严苛，则极有可能导致系统资源利用率和吞吐量的下降

- 事中规避

- 无需事先通过各种限制措施破坏产生死锁的四个必要条件，而是在资源的动态分配过程中，通过某种方法防止系统进入不安全状态，从而避免死锁的发生

- 事后补救

- 允许系统发生死锁，但可以通过预设的检测机制，及时发现死锁的产生，并精确定位与死锁有关的线程和资源。而后取消或挂起一些线程，回收其资源，再将这部分资源分配给那些于阻塞中等待资源的线程，使之进入就绪状态，继续运行



死锁问题

【参见：TTS COOKBOOK】

课堂
练习

- 死锁问题



条件变量



条件变量



条件变量

- 初始化条件变量

```
#include <pthread.h>
```

```
int pthread_cond_init (pthread_cond_t* cond,  
const pthread_condattr_t* attr);
```

成功返回0，失败返回错误码

- *cond* : 条件变量
 - *attr* : 条件变量属性，同步一个进程内的多个线程，取NULL
- 例如
 - `pthread_cond_init (&g_cond, NULL);`
 - `pthread_cond_t g_cond = PTHREAD_COND_INITIALIZER;`



条件变量（续1）

- 销毁条件变量

```
#include <pthread.h>
```

```
int pthread_cond_destroy (pthread_cond_t* cond);
```

成功返回0，失败返回错误码

- *cond*：条件变量

- 释放和条件变量cond有关的一切内核资源
- 例如
 - `pthread_cond_destroy (&g_cond);`



条件变量（续2）

- 等待条件变量

```
#include <pthread.h>
```

```
int pthread_cond_wait (pthread_cond_t* cond,  
pthread_mutex_t* mutex);
```

```
int pthread_cond_timedwait (pthread_cond_t* cond,  
pthread_mutex_t* mutex,  
const struct timespec* abs_timeout);
```

成功返回0，失败返回错误码

- *cond* : 条件变量
- *mutex* : 互斥体
- *abs_timeout* : 等候时限(始自UTC19700101T000000)



条件变量（续3）

- 以上函数会令调用线程进入阻塞状态，直到条件变量 *cond* 收到信号为止，阻塞期间互斥体 *mutex* 被解锁
- 条件变量必须与互斥体配合使用，以防止多个线程同时进入条件等待队列时发生竞争
 - 线程在调用 `pthread_cond_wait` 函数前必须先通过 `pthread_mutex_lock` 函数锁定 *mutex* 互斥体
 - 在调用线程进入条件等待队列之前，*mutex* 互斥体一直处于锁定状态，直到调用线程进入条件等待队列后才被解锁
 - 当调用线程即将从 `pthread_cond_wait` 函数返回时，*mutex* 互斥体会被重新锁定，回到调用该函数之前的状态



条件变量（续4）

- 和pthread_cond_wait相比pthread_cond_timedwait函数多了一个*abs_timeout*参数，在条件变量*cond*收到信号之前，如果等候时限已到，该函数会提前解除阻塞，并返回ETIMEDOUT错误码
- 例如
 - pthread_mutex_lock (&g_mutex);
 - ...
 - if (条件满足)
 - pthread_cond_wait (&g_cond, &g_mutex);
 - ...
 - pthread_mutex_unlock (&g_mutex);



条件变量（续5）

- 向指定的条件变量发送信号

```
#include <pthread.h>
```

```
int pthread_cond_signal (pthread_cond_t* cond);  
int pthread_cond_broadcast (pthread_cond_t* cond);
```

成功返回0，失败返回错误码

- *cond* : 条件变量
- 通过pthread_cond_signal函数向条件变量*cond*发送信号，与该条件变量相对应的条件等待队列中的第一个线程，将离开条件等待队列，并在重新锁定*mutex*互斥体后，从pthread_cond_wait函数中返回



条件变量（续6）

- 通过pthread_cond_broadcast函数向条件变量`cond`发送信号，与该条件变量相对应的条件等待队列中的所有线程，将同时离开条件等待队列，但只有第一个重新锁定`mutex`互斥体的线程，能够从pthread_cond_wait函数中返回，其余线程则继续为等待`mutex`互斥体而阻塞
- 例如
 - `pthread_cond_signal (&g_cond);`
 - `pthread_cond_broadcast (&g_cond);`



条件变量（续7）

- 被pthread_cond_broadcast函数唤醒的线程，在从pthread_cond_wait函数返回后，并不能确定自己一定是第一个获得*mutex*互斥体的线程。导致其进入等待状态的条件很可能因先于该线程获得*mutex*互斥体的线程动作而重新得到满足。因此有必要对该条件再做一次判断，以决定是向下执行还是继续等待

- pthread_mutex_lock (&g_mutex);

...

while (条件满足)

pthread_cond_wait (&g_cond, &g_mutex);

...

pthread_mutex_unlock (&g_mutex);



生产者消费者问题



生产者消费者问题

- 生产者消费者(Producer-Consumer)问题，亦称有界缓冲区(Bounded-Buffer)问题
- 两个线程共享一个公共的固定大小的缓冲区，其中一个线程作为生产者，负责将消息放入缓冲区；而另一个线程则作为消费者，负责从缓冲区中提取消息
- 假设缓冲区已满，若生产者线程还想放入消息，就必须等待消费者线程从缓冲区中提取消息以产生足够的空间
- 假设缓冲区已空，若消费者线程还想提取消息，就必须等待生产者线程向缓冲区中放入消息以产生足够的数据
- 生产者和消费者线程之间必须建立某种形式的同步，以确保为其所共享的缓冲区既不发生上溢，也不发生下溢



生产者消费者问题

【参见：TTS COOKBOOK】

课堂
练习

- 生产者消费者问题

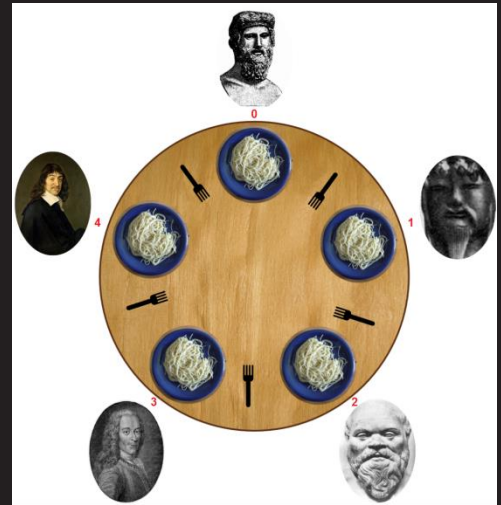


哲学家就餐问题



哲学家就餐问题

- 1965年，著名计算机科学家艾兹格·迪科斯彻，提出并解决了一个他称之为“哲学家就餐”的同步问题。从那时起，几乎每个发明同步原语的人，都希望通过解决“哲学家就餐”问题来展示其同步原语的精妙之处
- 五个哲学家围坐在一张圆桌周围，每个哲学家面前都有一盘面条。面条很滑，必须用一双，即两根筷子才能夹住。相邻两个盘子之间放有一根筷子。哲学家的一生只做两件事，吃饭和思考，交替进行。当一个哲学家感觉饿了时，他就试图分两次去取其左右两边的筷子，每次拿一根，不分次序。如果成功地得到了两根筷子，就开始吃面条，吃完后放下筷子继续思考



哲学家就餐问题（续1）

- 为每个哲学家编写一段描述其行为的程序，使整个系统既能保持最大限度的并发性，同时又不会发生死锁
- 如果五位哲学家同时拿起各自左边的筷子，那么就沒有人能够拿到他们各自右边的筷子，于是发生了死锁
- 如果每位哲学家在拿到左边的筷子后，发现其右边的筷子不可用，就先放下左边的筷子，等待一段时间，再重新尝试，那么就可以保证其它哲学家有同时获得两根筷子的机会。但在某一瞬间，所有的哲学家都同时拿起左筷，看到右筷不可用，又都同时放下左筷，等一会儿，又都同时拿起左筷，如此重复下去。虽然程序在不停运行，但都无法取得进展。这种现象通常被称为活锁



哲学家就餐问题（续2）

- 解决问题的关键在于，必须保证任意一位哲学家只有在其左右两个邻居都没有在进餐时，才允许其进入进餐状态。这样做不仅不会发生死锁，而且对于任意数量的哲学家都能获得最大限度的并行性
- 每个哲学家都持有一个条件变量。当其感觉饥饿时，环顾左右，发现至少有一个邻居正在吃饭，于是睡入条件变量。任一哲学家在其吃饱并开始思考时，向其左右邻居的条件变量发送信号。被唤醒的邻居们继续检查是否满足吃饭的条件，若满足则进入吃饭状态，否则继续睡眠于条件变量



哲学家就餐问题

【参见：TTS COOKBOOK】

课堂练习

- 哲学家就餐问题



局域网聊天室



需求分析



需求分析

- 局域网聊天室采用客户机/服务器模式，即在局域网内的一台机器上部署聊天室服务器，同时与该局域网中的多个聊天室客户机通信。服务器端采用TCP并发线程模式
- 每个聊天室客户机包括发送机和接收机两部分。发送机负责将用户从控制台输入的文本发送到服务器；而接收机则负责接收服务器广播的任何消息，打印在控制台上

系统> 热烈欢迎可爱的小猫咪进入聊天室！
 可爱的小猫咪> 大家好！
 可爱的小猫咪> 今天学校里有什么好玩的事儿吗？
 系统> 热烈欢迎轻风夜语进入聊天室！
 轻风夜语> 明天要考试了，大家都复习得怎么样了？
 可爱的小猫咪> 还好啦，这是测验哦。
 系统> 热烈欢迎班长进入聊天室！
 班长> 明天放学去游泳馆吧。
 轻风夜语> 好啊！好啊！响应号召！
 可爱的小猫咪> 写作业去喽，88~~
 系统> 可爱的小猫咪挥一挥衣袖，不带走一片云彩...
 班长> 小猫咪这么快撤了？

可爱的小猫咪> 大家好！
 可爱的小猫咪> 今天学校里有什么好玩的事儿吗？
 可爱的小猫咪> 还好啦，这是测验哦。
 可爱的小猫咪> 写作业去喽，88~~
 可爱的小猫咪> ！

接受来自127.0.0.1:41794接收器的连接请求。
 接受来自127.0.0.1:41796发送器的连接请求。

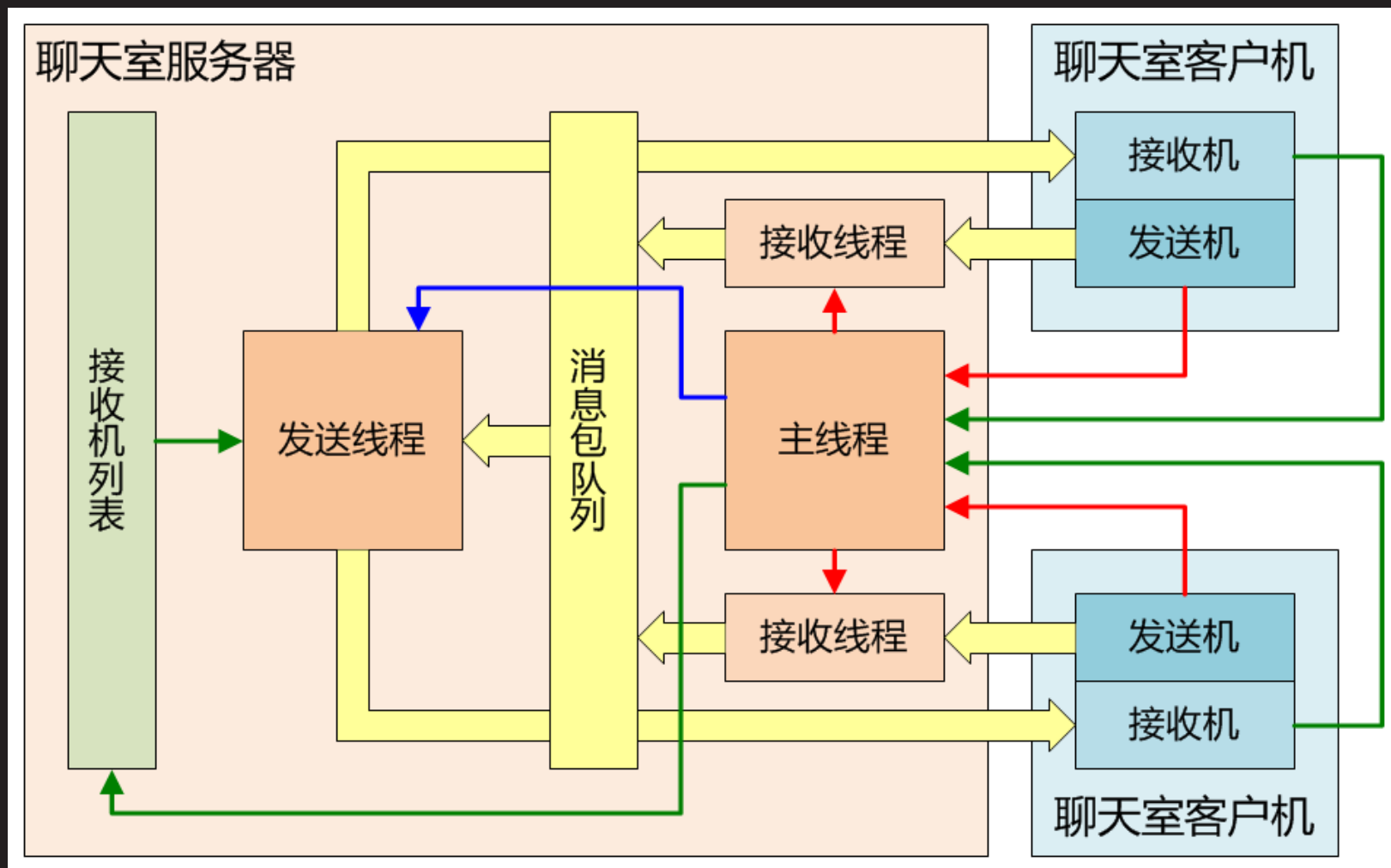


概要设计



概要设计

知识讲解



概要设计（续1）

- 聊天室客户机分为接收机和发送机两个独立进程
 - 接收机和聊天室服务器建立TCP连接后，不断接收服务器下发的各种消息，并将所接收到的消息内容打印在屏幕上
 - 发送机和聊天室服务器建立TCP连接后，阻塞于对标准输入的读取，将用户输入的文本打包发送到聊天室服务器端
- 聊天室服务器以多线程并发模式运行，包括一个主线程、一个发送线程和与在线发送机数量相当的若干接收线程
 - 主线程负责等待并接受来自接收机和发送机的连接请求，对于前者，将其排入接收机列表；对于后者，则创建专门的接收线程，将来自特定发送机的消息包压入消息包队列
 - 发送线程不断从消息包队列中弹出队首消息，并通过遍历接收机列表，将该消息向所有在线接收机广播



开发计划



开发计划

- 聊天室客户机
 - 接收机：receiver.c
 - 发送机：sender.c
- 聊天室服务器
 - 泛型化的单向线性链表：list.c
 - 服务器应用：chatroom.c
 - 主线程函数：main
 - 发送线程函数：send_proc
 - 接收线程函数：recv_proc
 - 其它功能函数：login、logout、wait_client，等等
- 构建脚本：makefile



局域网聊天室

【参见：TTS COOKBOOK】

课堂练习

- 局域网聊天室



总结和答疑

