

Lec8-3 实时调度 - 调度异常

调度异常

- 互斥
 - 优先级反转
 - 优先级继承
 - 优先级天花板
- 多处理器调度
 - Richard异常

考慮互斥

- 当线程访问共享资源时，它们需要使用互斥来确保数据的完整性
- 互斥也会使调度复杂化

回想一下pthreads中的互斥机制

```
#include <pthread.h>
...
pthread_mutex_t lock;
void* addListener(notify listener) {
    pthread_mutex_lock(&lock);
    ...
    pthread_mutex_unlock(&lock);
}
void* update(int newValue) {
    pthread_mutex_lock(&lock);
    value = newValue;
    elementType* element = head;
    while (element != 0) {
        (*(element->listener))(newValue);
        element = element->next;
    }
    pthread_mutex_unlock(&lock);
}
int main(void) {
    pthread_mutex_init(&lock, NULL);
    ...
}
```

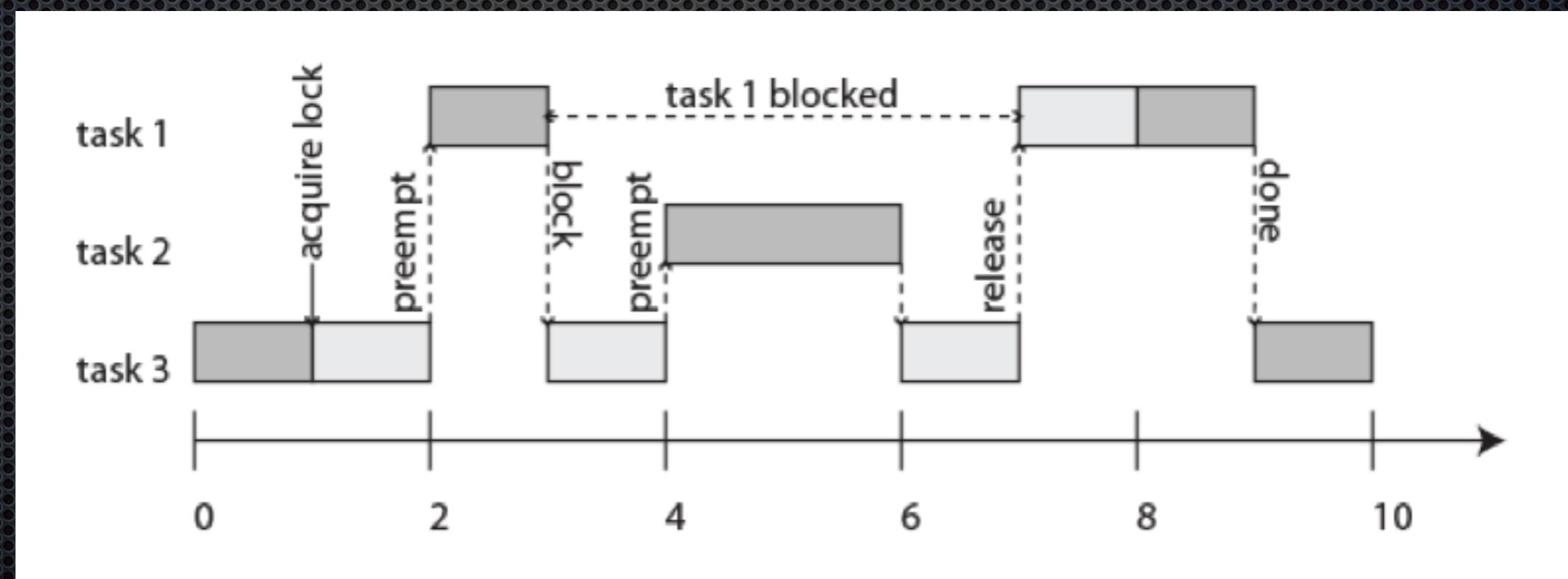
每当跨线程共享数据结构时，对数据结构的访问通常必须是原子的。

这是通过互斥或互斥锁来实现的。

持有锁时执行的代码称为临界区。

优先级反转：互斥的危险

- 任务1优先级最高，任务3优先级最低
- 任务3获取共享对象的锁，进入临界区后被任务1抢占，然后任务1试图获取锁而阻塞。任务2在时刻4抢占任务3，将高优先级的任务1阻塞无限长的时间
- 实际上，任务1和任务2的优先级颠倒了，因为任务2可以让任务1等待任意长的时间



火星探路者的问题

“火星探路者（ Mars Pathfinder, MPF ）”于1997年07月04日在火星表面着陆。在开始的几天内工作稳定，并传回大量数据，但几天后，也就是在探路者号开始收集气象数据后不久，探测器就开始经历全面的系统重置，每次都会导致数据丢失。



火星探路者的问题(2)

- VxWorks提供了线程的抢占式优先级调度。探路者号航天器上的任务是作为线程执行的，优先级以通常的方式分配，反映了这些任务的相对紧迫性
- 探路者号包含一个‘信息总线’，你可以把它想象成一个共享内存区域，用于在航天器的不同组件之间传递信息



总线管理任务以高优先级频繁运行，以便将某些类型的数据移进和移出信息总线，对总线的访问用互斥锁(互斥)同步。

火星探路者的问题(3)

- 气象数据收集任务作为低优先级线程低频率运行，…在发布数据时，它会获取一个互斥锁，对总线进行写操作，然后释放互斥锁
- 该航天器还包含一个中等优先级的通信任务

高优先级:从共享内存中检索数据

中优先级:通信任务

低优先级:收集气象数据的线程

The MARS Pathfinder problem (4)

“大多数时间，这种组合工作正常。然而，当(高优先级)信息总线线程被阻塞等待(低优先级)气象数据线程时，导致(中等优先级)通信任务在短时间间隔内被调度，这种情况可能偶尔引起中断。在这种情况下，长时间运行的通信任务具有比气象任务更高的优先级，会阻止气象任务运行，从而阻止阻塞的信息总线任务运行。过了一段时间后，看门狗计时器会运行，注意到数据总线任务已经有一段时间没有被执行了，推断出现了严重错误，因而启动整个系统重置。这种场景是优先级反转的典型案例。”

优先级反转的解决

- 优先级继承解决了火星探路者问题：探路者中使用的VxWorks操作系统使用标志位用于调用互斥原语。此标志设置为“on”将使用优先级继承。当软件发布时，被设置为“off”

火星上的探路者号问题通过使用VxWorks的调试工具将标志更改为“on”来纠正[Jones, 1997]



优先级继承协议

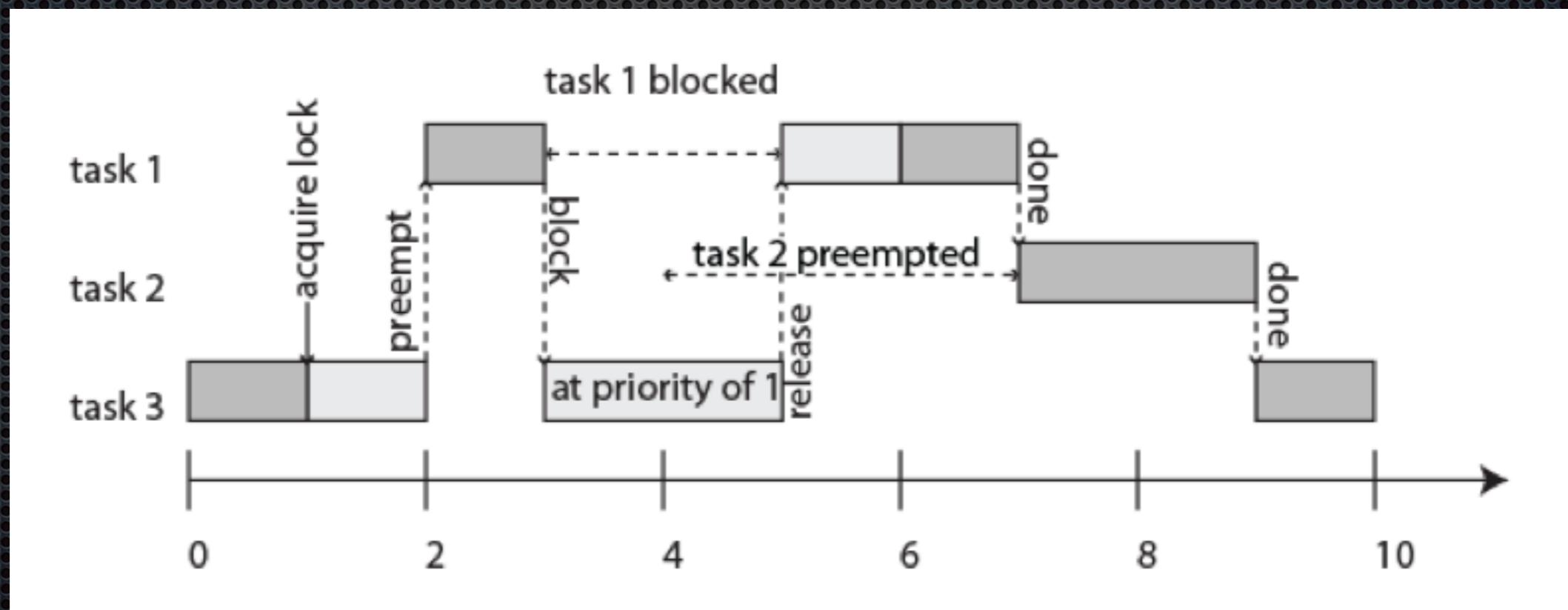
- *Priority Inheritance Protocol (PIP)* (Sha, Rajkumar, Lehoczky, 1990)
- 假设:
 - n 个任务通过 m 个共享资源进行协作
 - 固定优先级，一个资源上的所有临界区都以 $\text{wait}(S_i)$ 开始，以 $\text{signal}(S_i)$ 操作结束
- 基本思想:
 - 当任务 J_i 阻塞一个或多个高优先级任务时，它暂时假定(继承)被阻塞任务的最高优先级
- 术语:
 - 我们区分固定的名义优先级 (nominal priority) P_i 和大于或等于 P_i 的动态优先级 (active priority)，作业 $J_1 \dots J_n$ 按照名义优先级排序，其中 J_1 具有最高优先级，作业不会自动中止

- 算法:

- 作业是根据其动态优先级调度的，具有相同优先级的作业以FCFS规则执行
- 当作业 J_i 试图进入临界区，而资源被较低优先级的作业独占使用时，作业 J_i 被阻塞，否则就进入临界区
- 当作业 J_i 被阻塞时，它将其动态优先级传递给持有该信号量的作业 J_k 。 J_k 恢复并以 $P_k = P_i$ 的优先级执行其临界区的其余部分(它继承被它阻塞的所有作业的最高优先级的优先级)
- 当 J_k 退出临界区时，它释放该信号量，并唤醒阻塞在该信号量上的最高优先级作业。如果没有其他作业被 J_k 阻塞，则将 P_k 设置为其名义优先级 P_k ，否则将其设置为 J_k 阻塞的作业的最高优先级
- 优先级继承是可传递的，即如果 1 被 2 阻塞，2 被 3 阻塞，则 3 通过 2 继承 1 的优先级

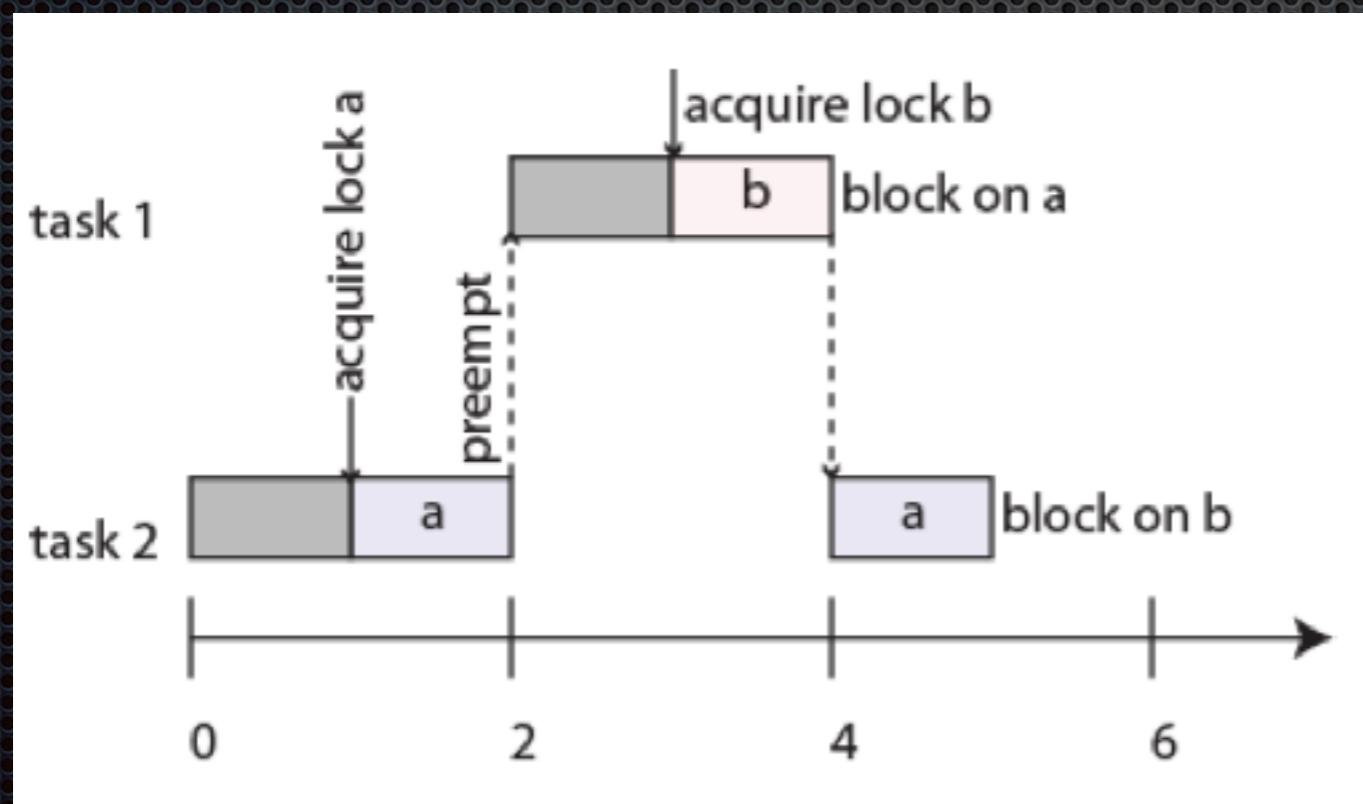
PIP例子

- 任务1优先级最高，任务3优先级最低
- 任务3获取共享对象上的锁，进入临界区。它被任务1抢占，然后任务1尝试获取锁，结果阻塞。任务3继承任务1的优先级，防止被任务2抢占



死锁

- 低优先级的任务首先启动并获取锁 a , 然后被高优先级的任务抢占, 高优先级的任务获取锁 b , 运行, 在尝试获取锁 a 时被阻塞。然后低优先级的任务尝试获取锁 b 时被阻塞, 不可能再执行下去



```
#include <pthread.h>
...
pthread_mutex_t lock_a, lock_b;
void* thread_1_function(void* arg) {
    pthread_mutex_lock(&lock_b);
    ...
    pthread_mutex_lock(&lock_a);
    ...
    pthread_mutex_unlock(&lock_a);
    ...
    pthread_mutex_unlock(&lock_b);
    ...
}
void* thread_2_function(void* arg) {
    pthread_mutex_lock(&lock_a);
    ...
    pthread_mutex_lock(&lock_b);
    ...
    pthread_mutex_unlock(&lock_b);
    ...
    pthread_mutex_unlock(&lock_a);
    ...
}
```

优先级上限协议

- *Priority Ceiling Protocol (PCP)* (Sha, Rajkumar, Lehoczky, 1990)
- 每个锁或信号量都被分配了一个优先级上限，该上限等于可以锁定它的最高优先级任务的优先级
 - 是否可以自动计算优先级上限？
- 只有当任务 T 的优先级严格高于其他任务当前持有的所有锁的优先级上限时，任务 T 才能获得锁
 - 未被任何任务持有的锁不会影响该任务
- 这可以防止一些死锁的发生
- 有一些扩展支持动态优先级和锁的动态创建

OCPP and ICPP

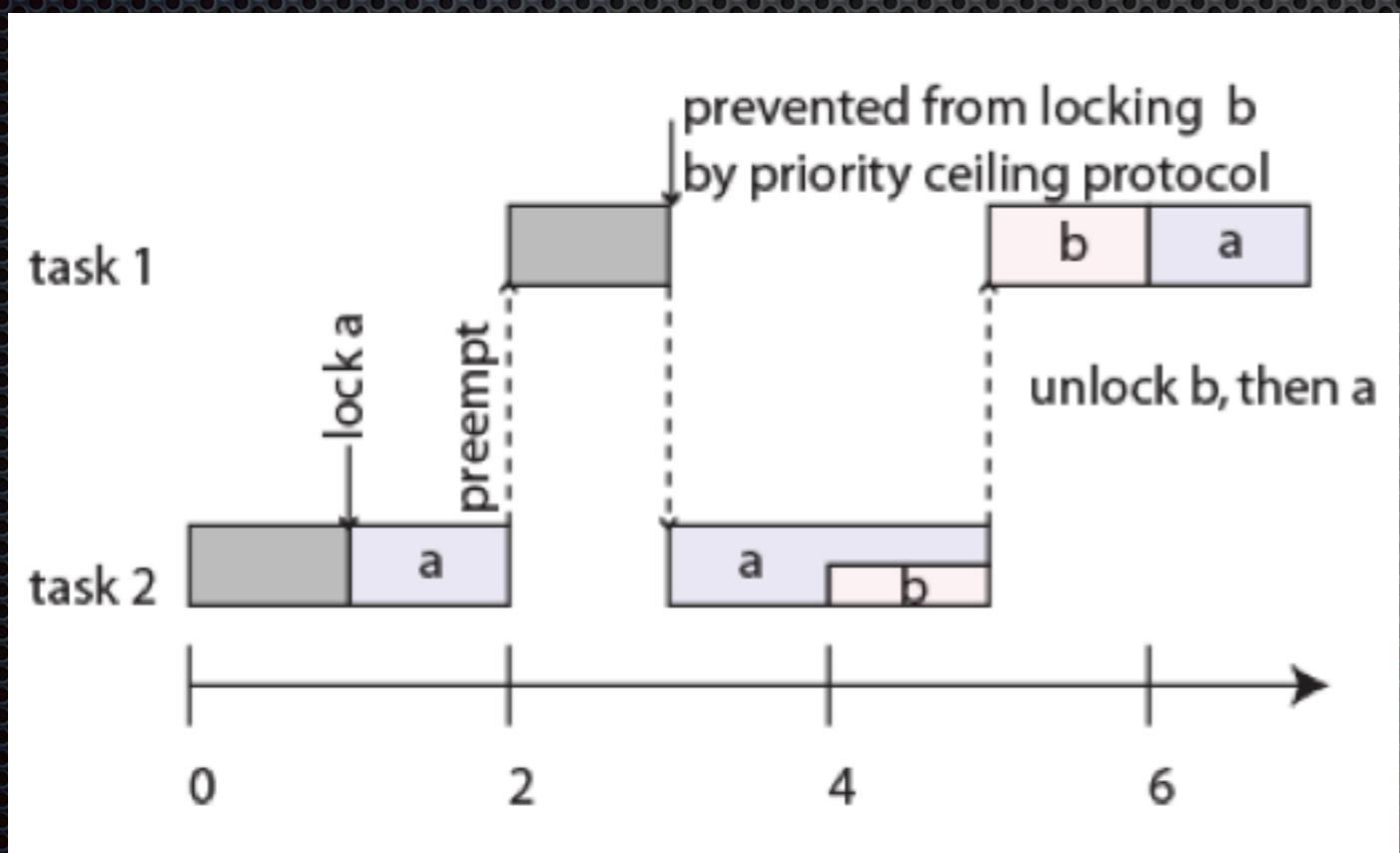
- 原始优先级上限协议(OCPP): 在OCPP中, 当更高优先级的任务 Y 试图获取任务 X 锁定的资源时, 任务 X 的优先级会被提升到资源的优先级上限, 确保任务 X 尽快完成其临界区, 以解锁资源; 只有当一个任务的动态优先级高于其他任务锁定的所有资源的优先级上限时, 才允许该任务锁定资源, 否则, 任务将阻塞, 等待资源
- 立即优先级上限协议(ICPP): 在ICPP中, 当一个任务锁定一个资源时, 它的优先级立即被提升为资源的优先级上限, 因此任何可能锁定资源的任务都无法被调度

OCPP与ICPP的对比

- 从调度的角度来看，这两种上限方案的最坏情况是相同的。然而，也有一些不同之处：
 - ICPP比OCPP更容易实现，因为不需要监视阻塞关系
 - ICPP导致更少的上下文切换，因为阻塞是在首次执行之前进行的
 - ICPP需要更多的优先级传递，因为这发生在所有资源使用中
 - OCPP仅在实际发生阻塞时才更改优先级

PCP应用示例

- 锁 a 和锁 b 的优先级上限等于任务1的优先级。在时刻3，任务1试图锁定 b ，未果，因为任务2目前持有锁 a ，其优先级上限等于任务1的优先级



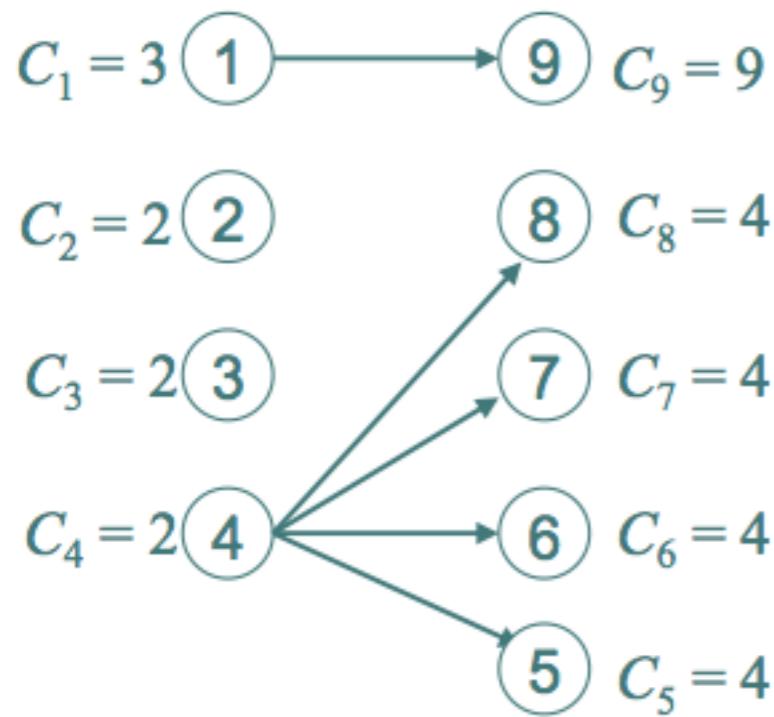
```
#include <pthread.h>
...
pthread_mutex_t lock_a, lock_b;
void* thread_1_function(void* arg) {
    pthread_mutex_lock(&lock_b);
    ...
    pthread_mutex_lock(&lock_a);
    ...
    pthread_mutex_unlock(&lock_a);
    ...
    pthread_mutex_unlock(&lock_b);
    ...
}

void* thread_2_function(void* arg) {
    pthread_mutex_lock(&lock_a);
    ...
    pthread_mutex_lock(&lock_b);
    ...
    pthread_mutex_unlock(&lock_b);
    ...
    pthread_mutex_unlock(&lock_a);
    ...
}
```

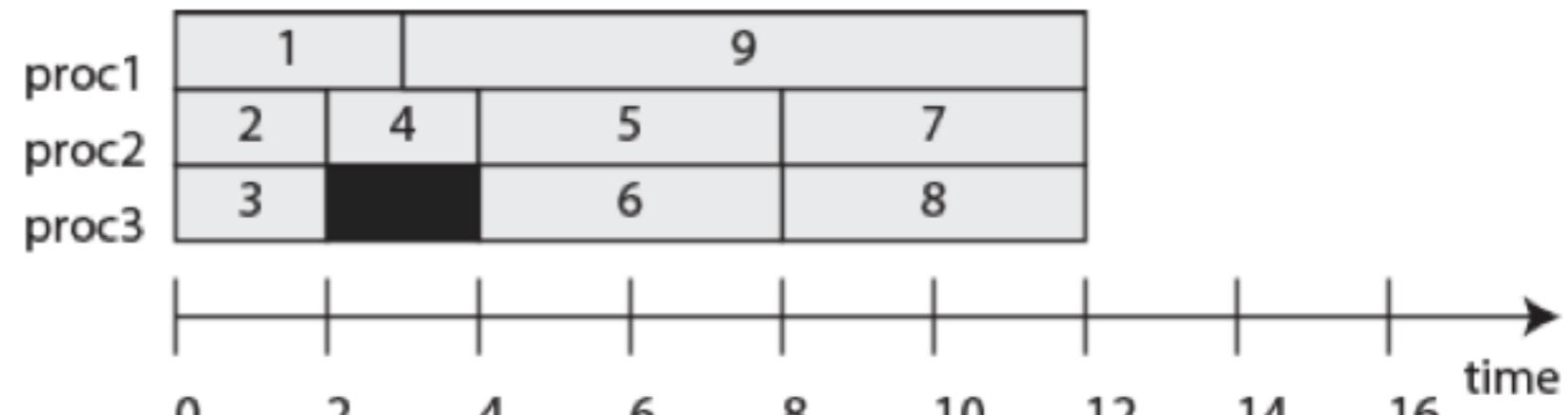
脆弱性

- 一般来说，所有线程调度算法都是脆弱的：微小的更改可能会产生意想不到的严重后果，下面将用多处理器(或多核)调度来说明这一点
- 定理(Richard Graham, 1976): 如果一个具有固定优先级、执行时间和优先级约束的任务集在固定数量的处理器上按照优先级进行调度，那么增加处理器数量、减少执行时间或削弱优先序约束可能增加调度时长

Richard异常

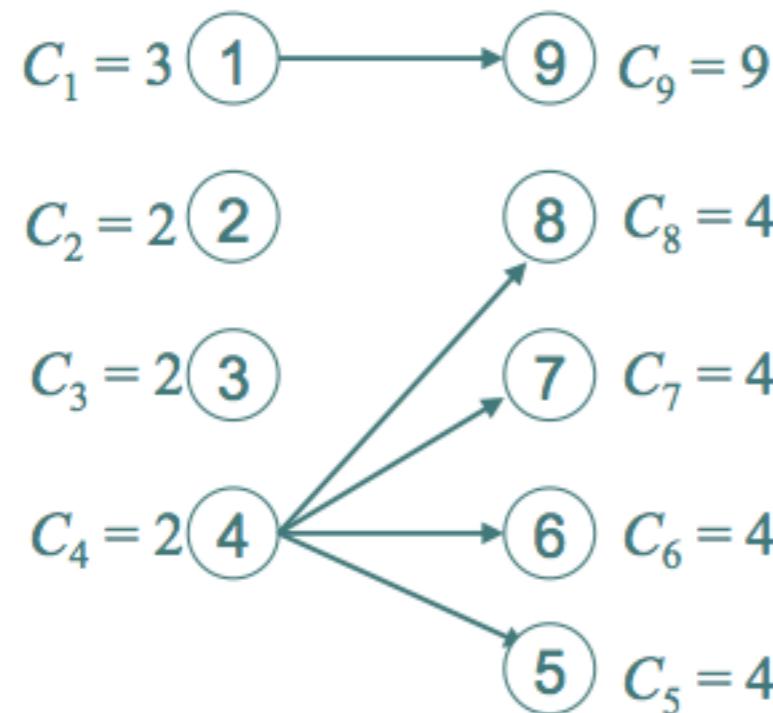


9个任务，如图所示的优先序和执行时间 C_i ，其中编号较低的任务比编号较高的任务具有更高的优先级，基于优先级的3个处理器时的调度结果如下：



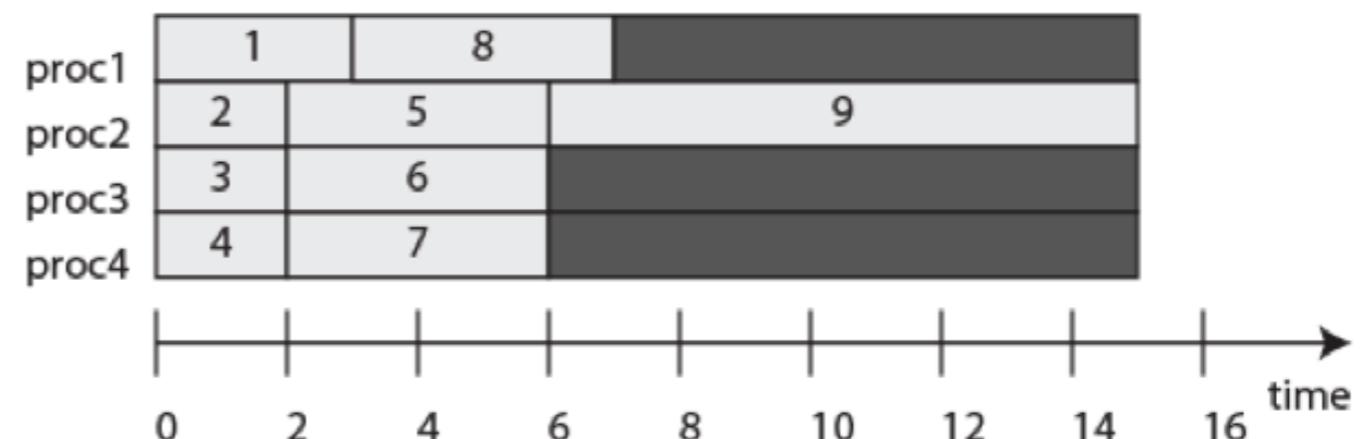
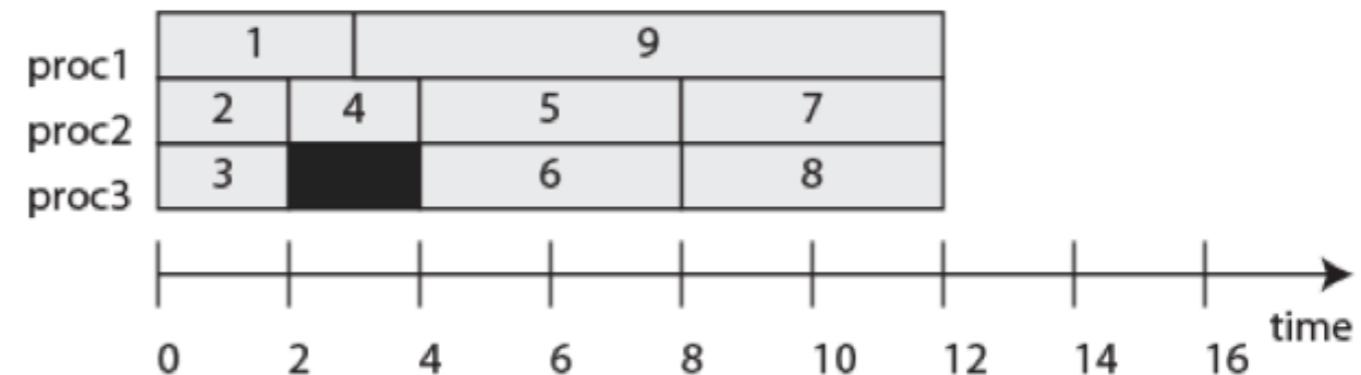
如果将处理器数量增加到4个会发生什么？

增加处理器数量

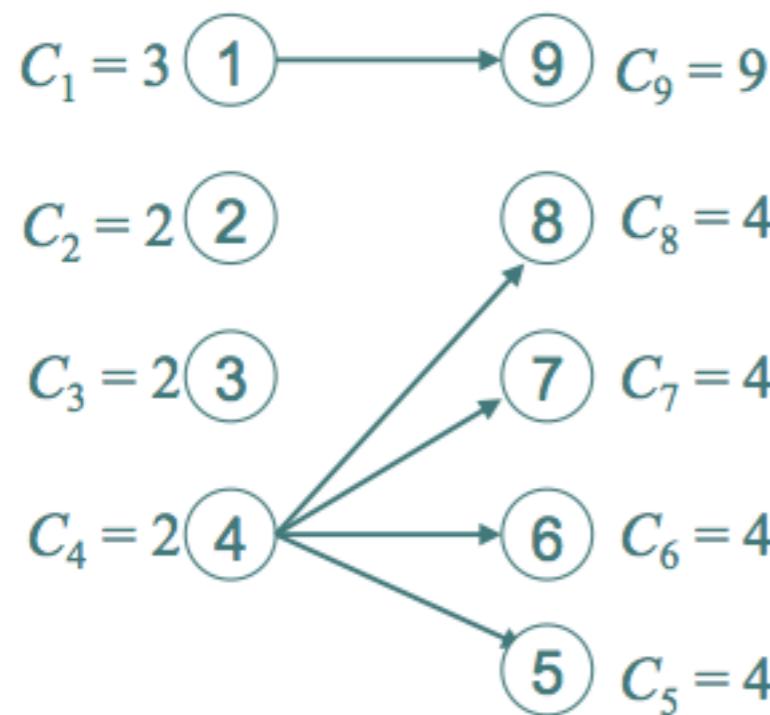


增加到四个处理器的基于优先级的调度反而具有更长的执行时间

9个任务，如图所示的优先序和执行时间 C_i ，其中编号较低的任务比编号较高的任务具有更高的优先级，基于优先级的3个处理器时的调度结果如下：

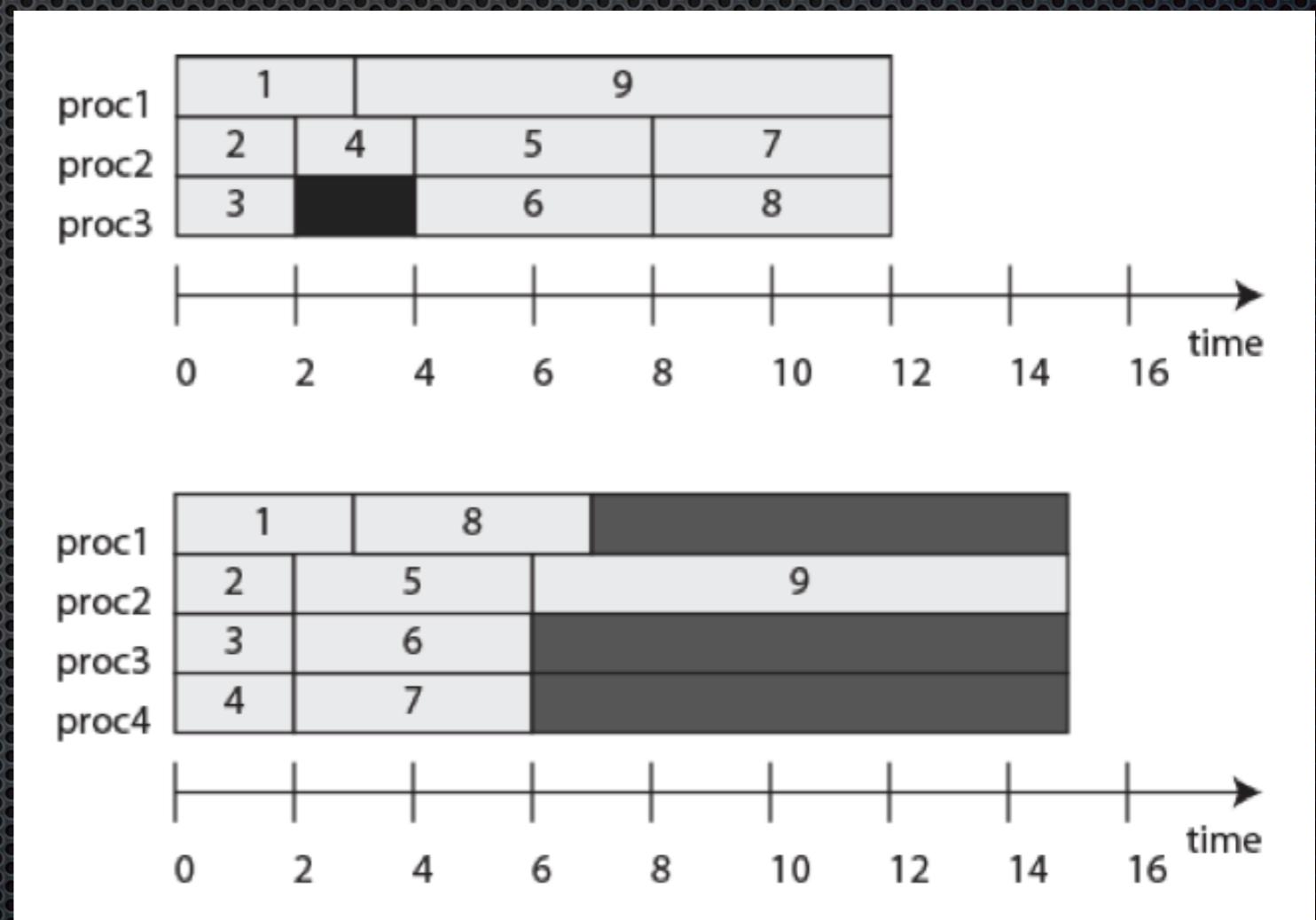


贪心调度

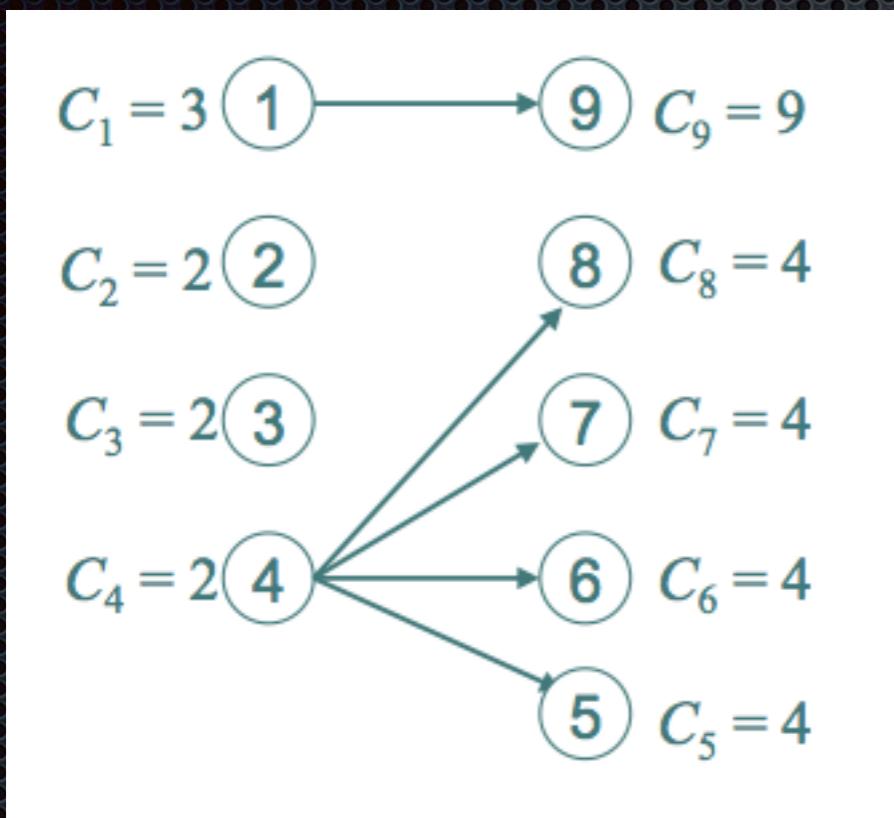


基于优先级的调度是“贪心的”。对于本例，更智能的调度程序可以推迟调度5、6或7，使处理器空闲一个时间单元

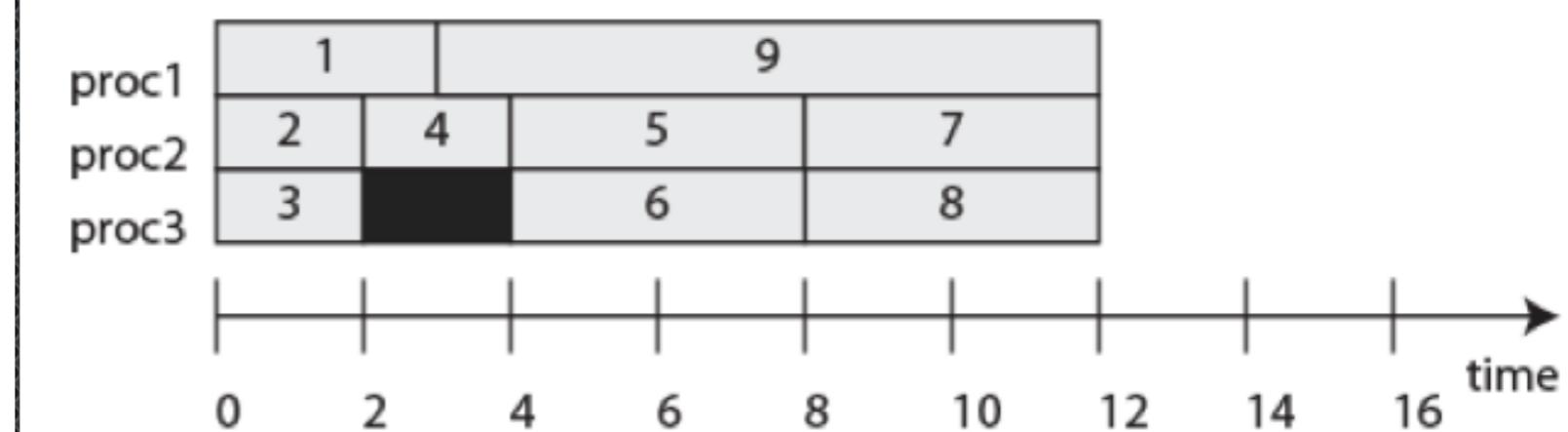
9个任务，如图所示的优先序和执行时间 C_i ，其中编号较低的任务比编号较高的任务具有更高的优先级，基于优先级的3个处理器时的调度结果如下：



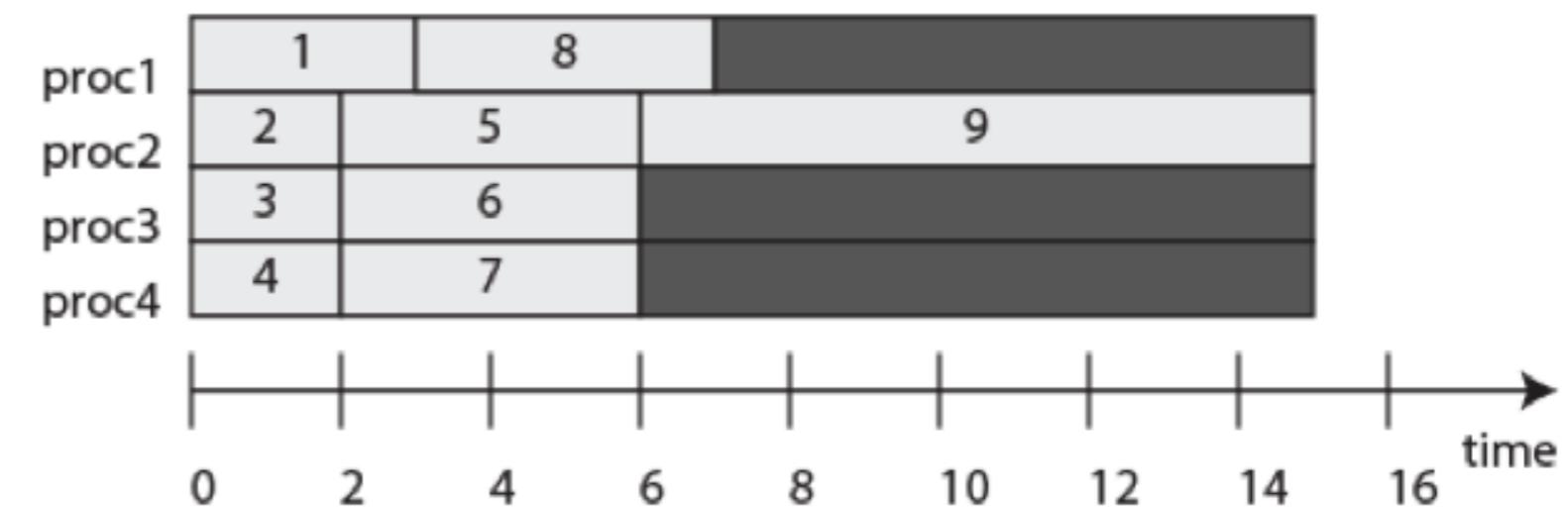
贪心调度可能是唯一可行的选项



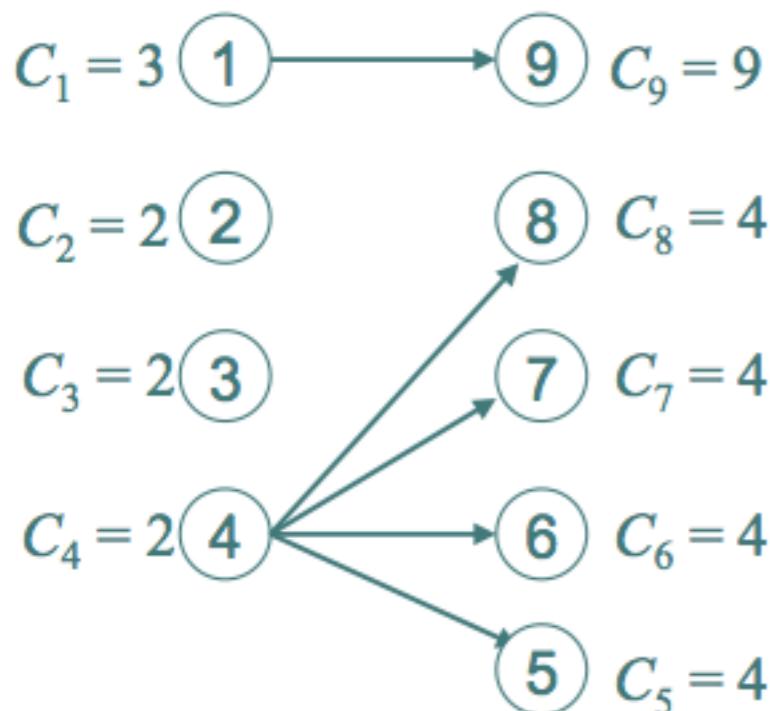
9个任务，如图所示的优先序和执行时间 C_i ，其中编号较低的任务比编号较高的任务具有更高的优先级，基于优先级的3个处理器时的调度结果如下：



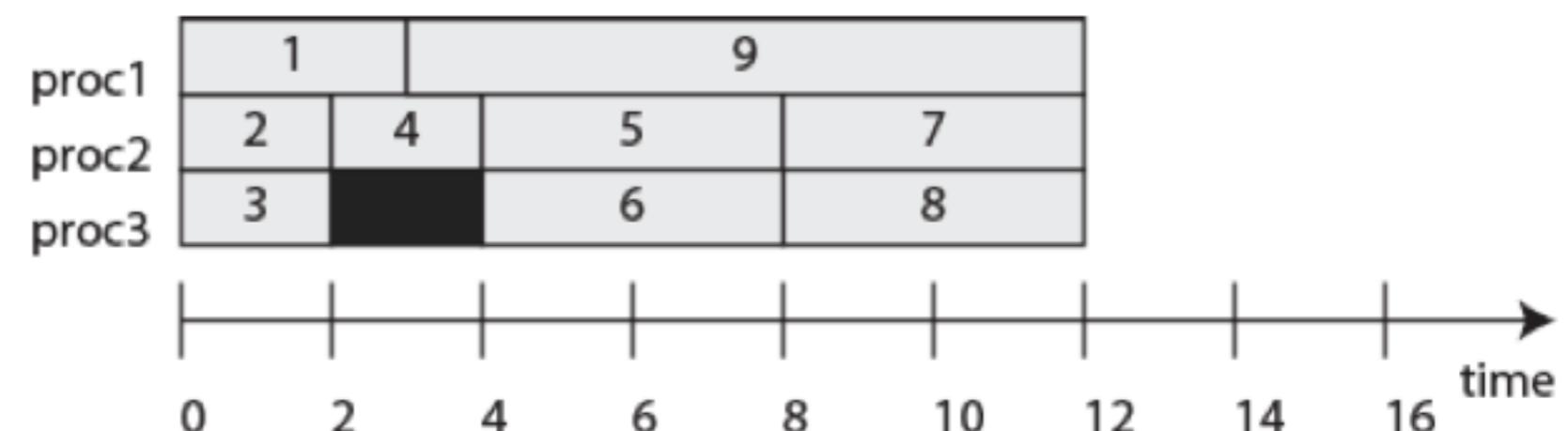
如果任务只有在其前序任务完成后才“到达”(调度器才获知)，那么贪心调度可能是唯一可行的选项



Richard异常

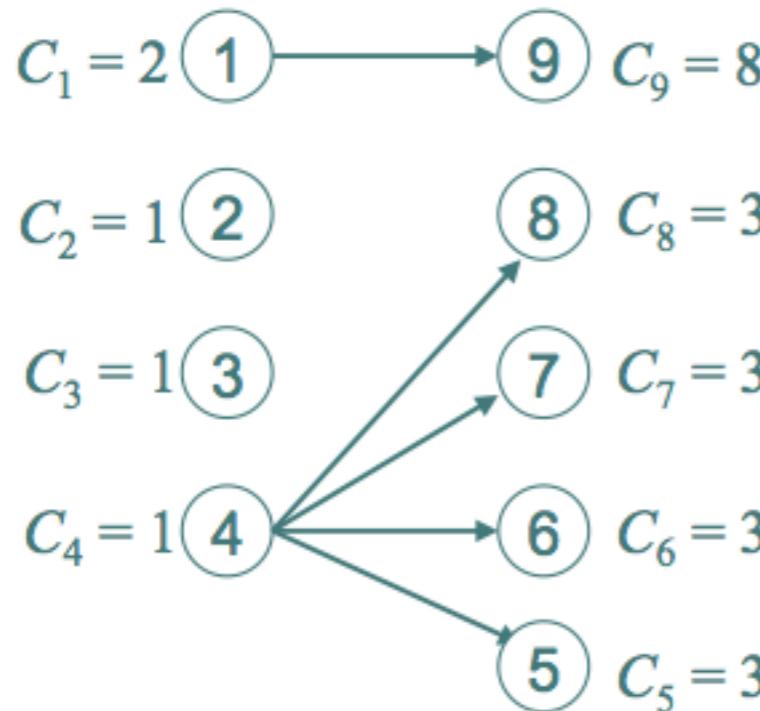


9个任务，如图所示的优先序和执行时间 C_i ，其中编号较低的任务比编号较高的任务具有更高的优先级，基于优先级的3个处理器时的调度结果如下：

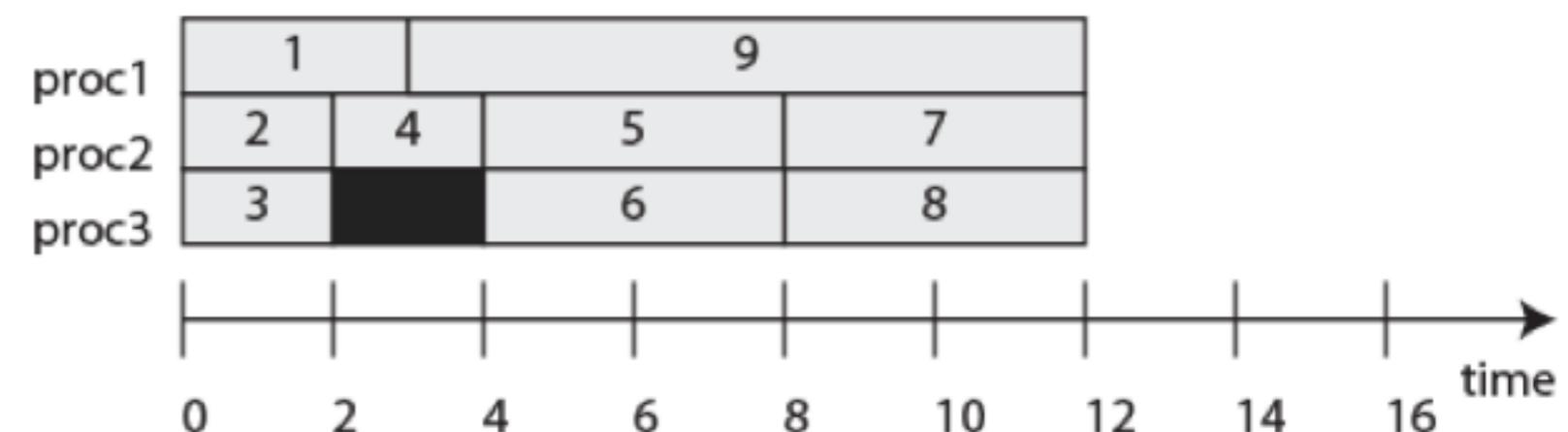


如果把所有的计算时间减少1会发生什么？

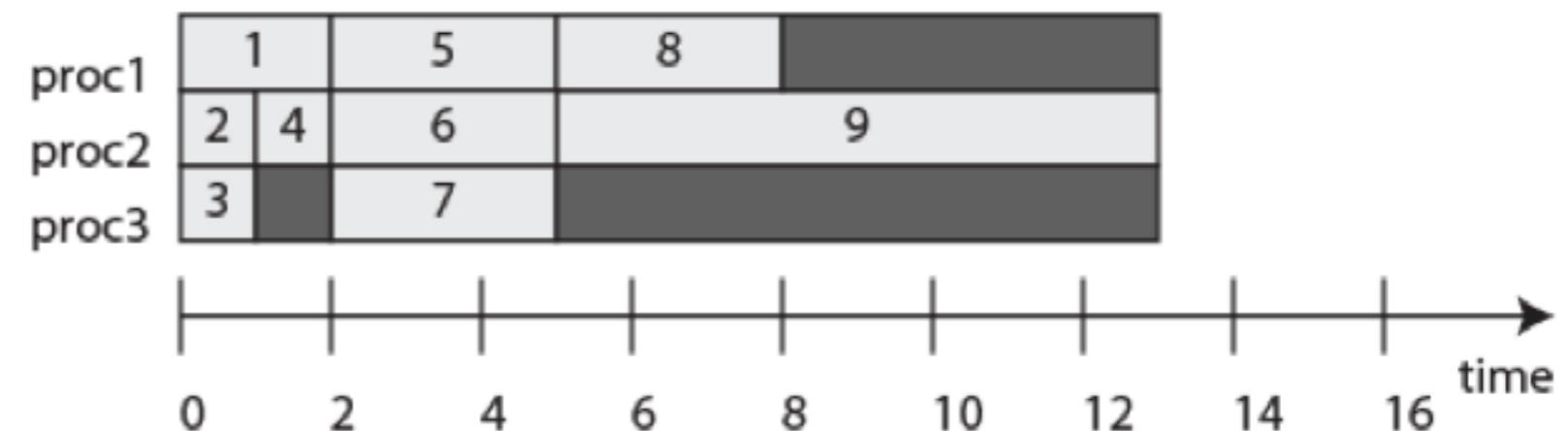
减少计算时间



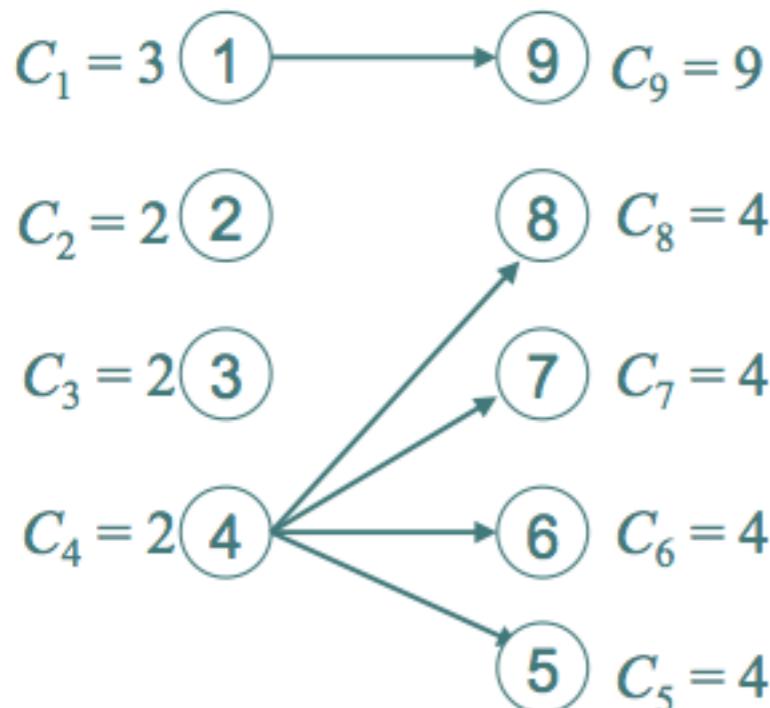
9个任务，如图所示的优先序和执行时间 C_i ，其中编号较低的任务比编号较高的任务具有更高的优先级，基于优先级的3个处理器时的调度结果如下：



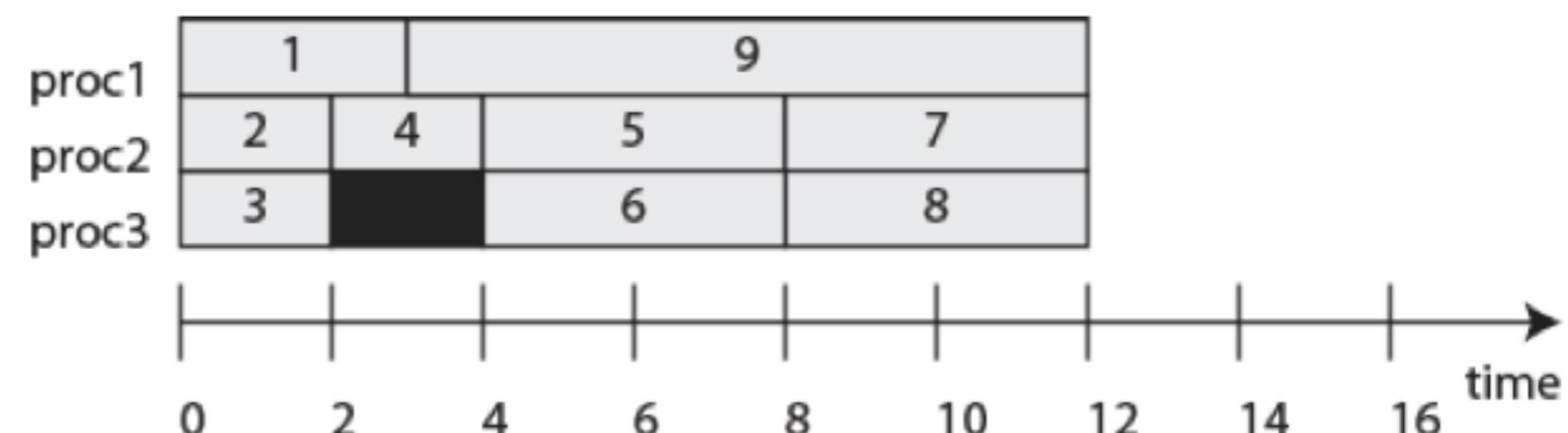
将计算时间减少1也会导致更长的执行时间



Richard异常

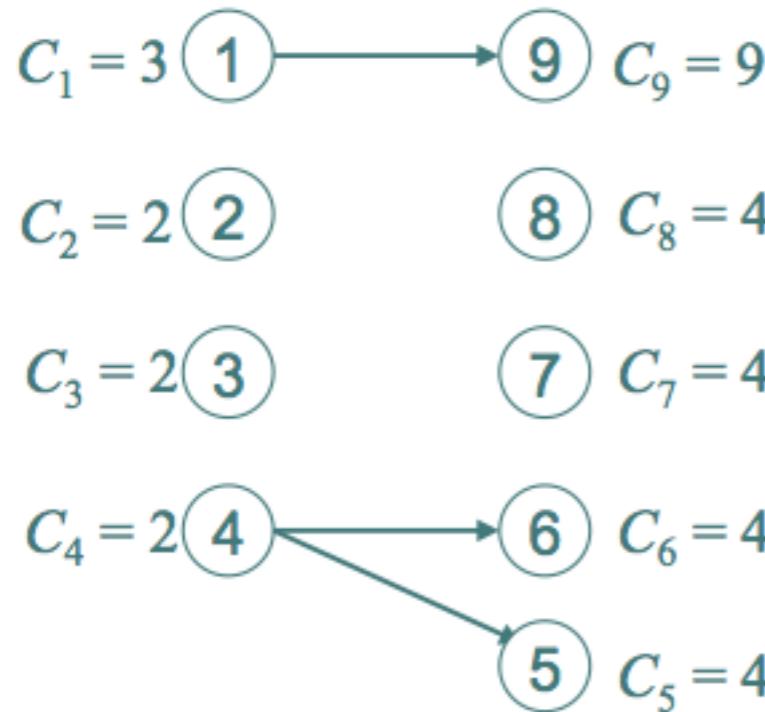


9个任务，如图所示的优先序和执行时间 C_i ，其中编号较低的任务比编号较高的任务具有更高的优先级，基于优先级的3个处理器时的调度结果如下：

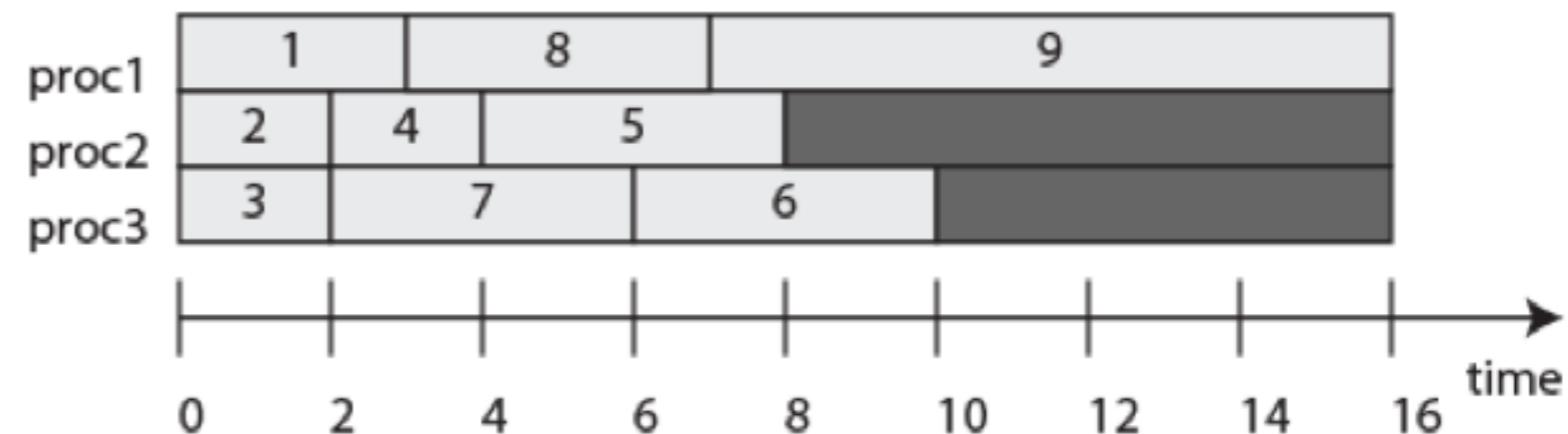
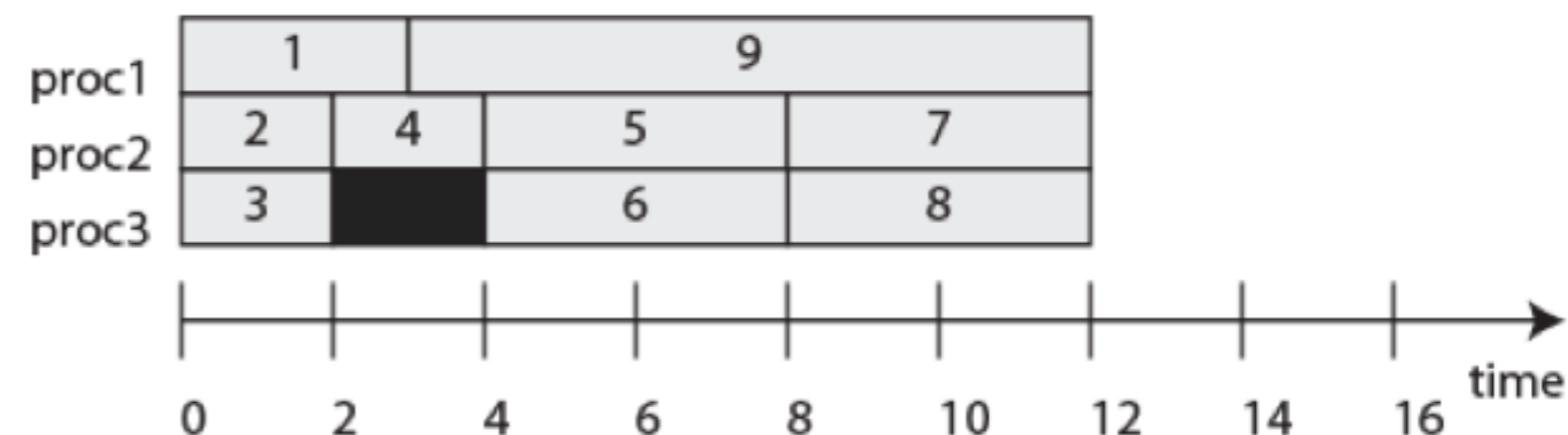


如果移除优先序约束(4,8)和(4,7)会发生什么？

弱化优先序约束

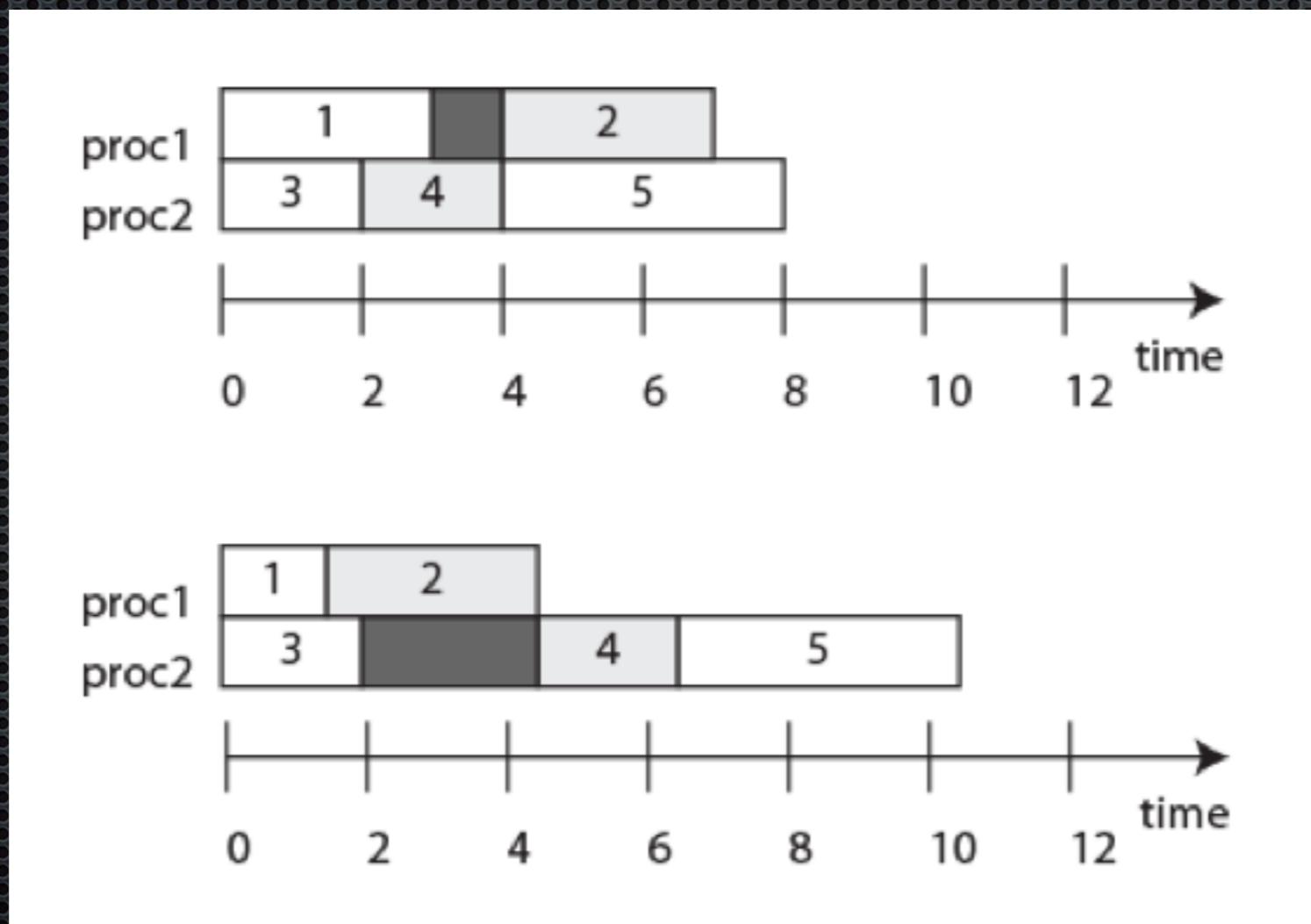


弱化优先级约束还会导致更长的执行时间



Richard互斥异常: 减少执行时间

- 假设任务2和任务4以独占模式共享相同的资源，任务被静态地分配给处理器。那么，如果任务 1的执行时间减少，则执行时间增加：



结论

- 在所有已知的任务调度策略下，时限行为都是脆弱的。小的改变会产生大的(意想不到的)后果
- 不幸的是，由于执行时间很难预测，这种脆弱性可能导致意外的系统故障

参考文献

- Edward Ashford Lee, Sanjit Arunkumar Seshia. Introduction to Embedded Systems: A Cyber-Physical Systems Approach, chapter 11. Lulu.com. (嵌入式系统导论：CPS方法)

