Introduction to Embedded Systems

Sanjit A. Seshia
UC Berkeley

EECS 149/249A
Fall 2015

Chapter 12: Scheduling Anomalies

Source

This lecture draws heavily from:

Giorgio C. Buttazzo, *Hard Real-Time Computing Systems,* Springer, 2004.

Recall from Last Lecture

- Rate-Monotonic Scheduling
- Earliest Deadline First
- EDF with Precedences

Today: Scheduling can interact with other constraints/requirements in Non-Intuitive ways

- Mutual exclusion
  - Priority inversion
  - Priority inheritance
  - Priority ceiling
- Multiprocessor scheduling
  - Richard’s anomalies

Accounting for Mutual Exclusion

Recall from a previous lecture:

When threads access shared resources, they need to use mutexes to ensure data integrity.

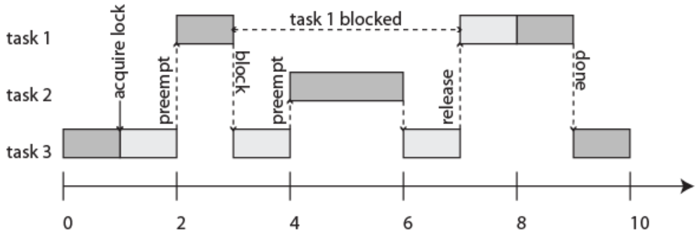
Mutexes can also complicate scheduling.

Recall mutual exclusion mechanism in pthreads

```
#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);
  ...
}
```

Whenever a data structure is shared across threads, access to the data structure must usually be atomic. This is enforced using mutexes, or mutual exclusion locks. The code executed while holding a lock is called a *critical section*.

Priority Inversion: A Hazard with Mutexes



Task 1 has highest priority, task 3 lowest. Task 3 acquires a lock on a shared object, entering a critical section. It gets preempted by task 1, which then tries to acquire the lock and blocks. Task 2 preempts task 3 at time 4, keeping the higher priority task 1 blocked for an unbounded amount of time. In effect, the priorities of tasks 1 and 2 get inverted, since task 2 can keep task 1 waiting arbitrarily long.

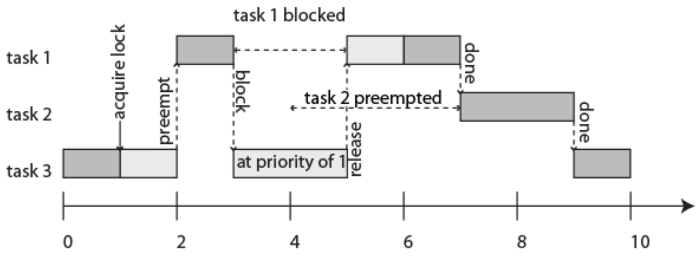
Mars Rover Pathfinder

The Mars Rover Pathfinder landed on Mars on July 4th, 1997. A few days into the mission, the Pathfinder began sporadically missing deadlines, causing total system resets, each with loss of data. The problem was diagnosed on the ground as priority inversion, where a low priority meteorological task was holding a lock blocking a high-priority task while medium priority tasks executed.

Source: RISKS-19.49 on the comp.programming.threads newsgroup, December 07, 1997, by Mike Jones (mbj@MICROSOFT.com).

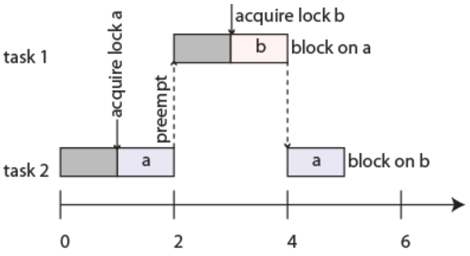
Priority Inheritance Protocol (PIP)

(Sha, Rajkumar, Lehoczky, 1990)



Task 1 has highest priority, task 3 lowest. Task 3 acquires a lock on a shared object, entering a critical section. It gets preempted by task 1, which then tries to acquire the lock and blocks. Task 3 inherits the priority of task 1, preventing preemption by task 2.

Deadlock



The lower priority task starts first and acquires lock *a*, then gets preempted by the higher priority task, which acquires lock *b* and then blocks trying to acquire lock *a*. The lower priority task then blocks trying to acquire lock *b*, and no further progress is possible.

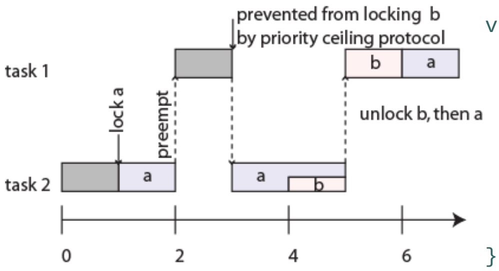
```
#include <pthread.h>
...
pthread_mutex_t lock_a, lock_b;

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);
  ...
}
```

Priority Ceiling Protocol (PCP)

(Sha, Rajkumar, Lehoczky, 1990)

- Every lock or semaphore is assigned a *priority ceiling* equal to the priority of the highest-priority task that *can* lock it.
  - Can one automatically compute the priority ceiling?
- A task T can acquire a lock only if the task's priority is strictly higher than the priority ceilings of all locks *currently held* by other tasks
  - Intuition: the task T will not later try to acquire these locks held by other tasks
  - Locks that are not held by any task don't affect the task
- This prevents deadlocks
- There are extensions supporting dynamic priorities and dynamic creations of locks (stack resource policy)

Priority Ceiling Protocol



In this version, locks a and b have priority ceilings equal to the priority of task 1. At time 3, task 1 attempts to lock *b*, but it can't because task 2 currently holds lock *a*, which has priority ceiling equal to the priority of task 1.

```
#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);
  ...
}
```

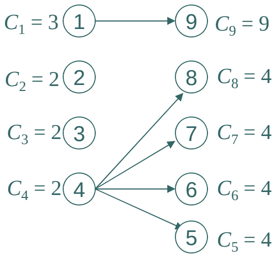
Brittleness

In general, all thread scheduling algorithms are brittle: Small changes can have big, unexpected consequences.

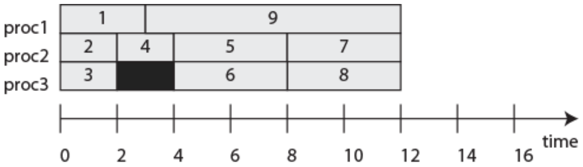
I will illustrate this with multiprocessor (or multicore) schedules.

> Theorem (Richard Graham, 1976): If a task set with fixed priorities, execution times, and precedence constraints is scheduled according to priorities on a fixed number of processors, then increasing the number of processors, reducing execution times, or weakening precedence constraints *can increase the schedule length*.

Richard' s Anomalies

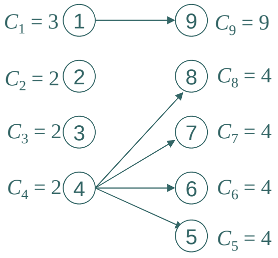


9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Priority-based 3 processor schedule:

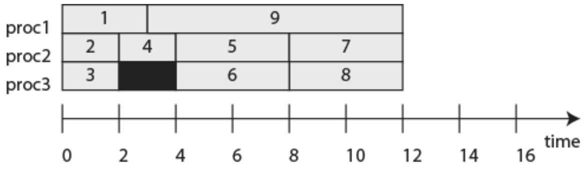


What happens if you increase the number of processors to four?

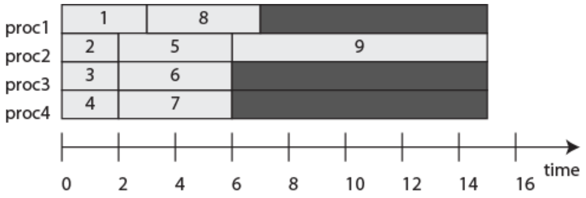
Richard’s Anomalies: Increasing the number of processors



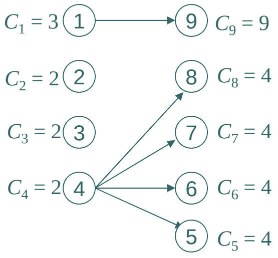
9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Priority-based 3 processor schedule:



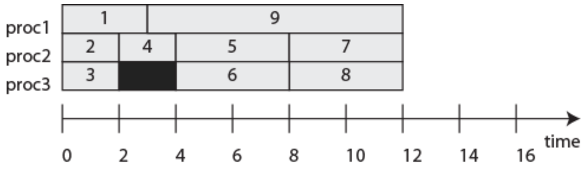
The priority-based schedule with four processors has a longer execution time.



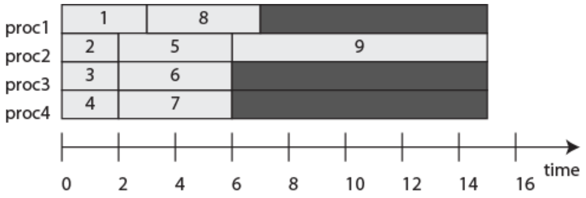
Greedy Scheduling

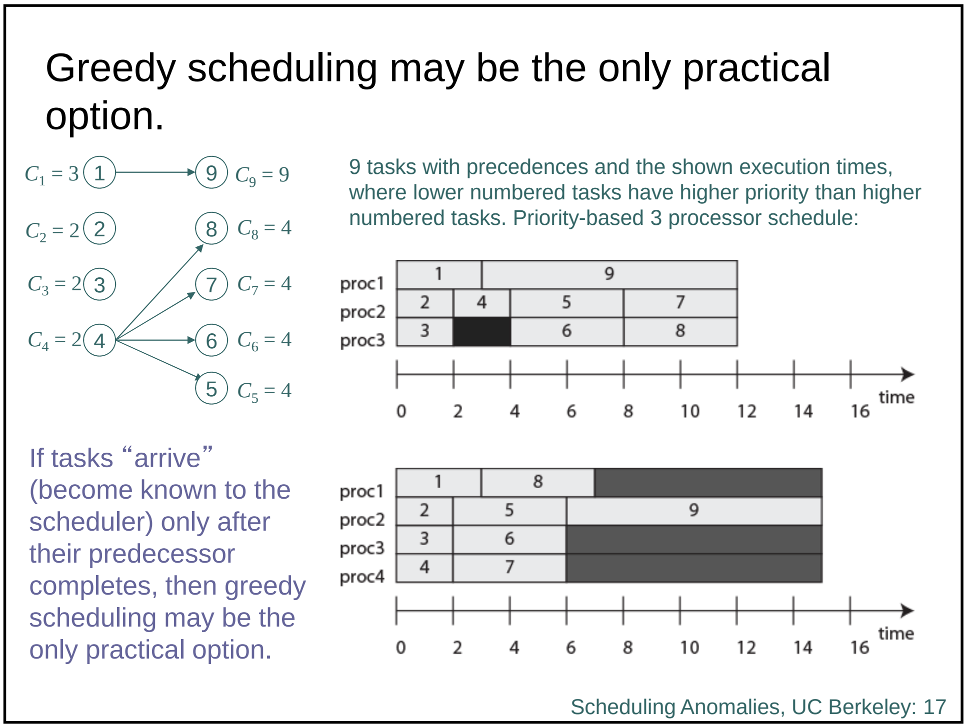


9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Priority-based 3 processor schedule:



Priority-based scheduling is “greedy.” A smarter scheduler for this example could hold off scheduling 5, 6, or 7, leaving a processor idle for one time unit.



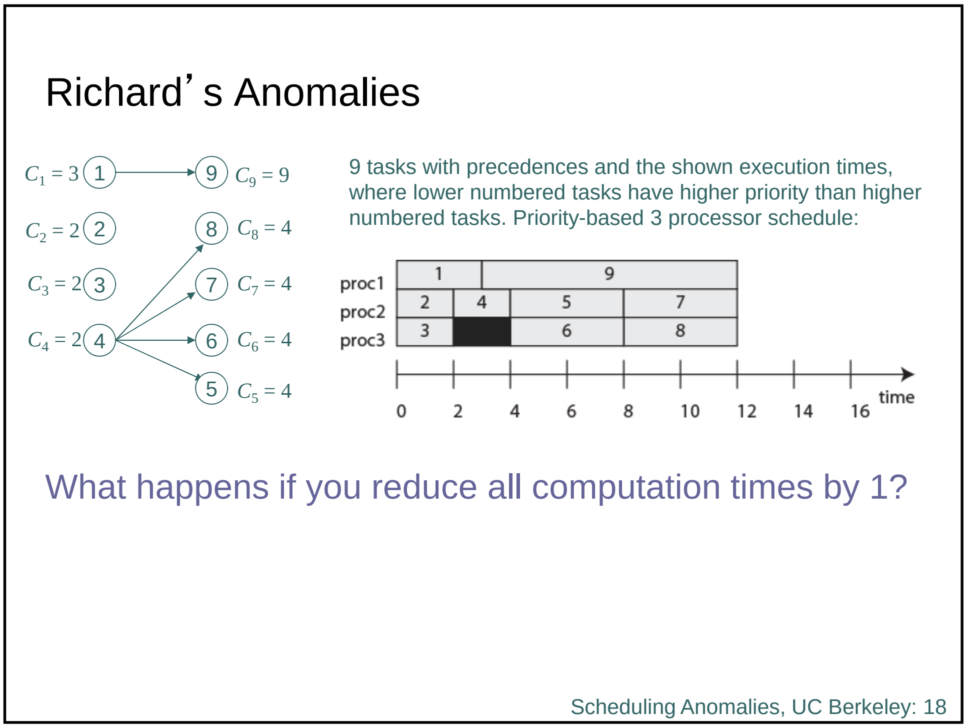


Greedy scheduling may be the only practical option.

9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Priority-based 3 processor schedule:

If tasks “arrive” (become known to the scheduler) only after their predecessor completes, then greedy scheduling may be the only practical option.

Scheduling Anomalies, UC Berkeley: 17



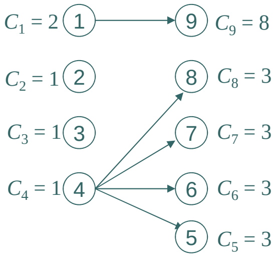
Richard’s Anomalies

9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Priority-based 3 processor schedule:

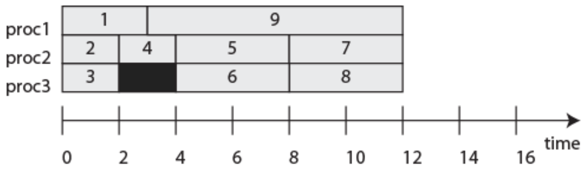
What happens if you reduce all computation times by 1?

Scheduling Anomalies, UC Berkeley: 18

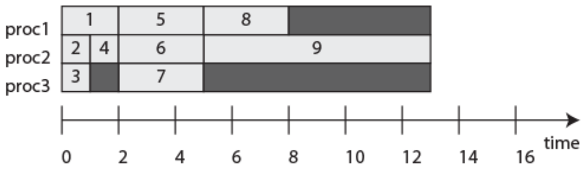
Richard’s Anomalies: Reducing computation times



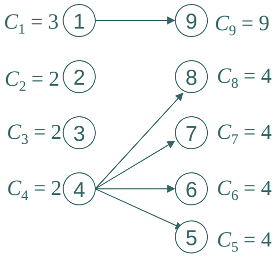
9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Priority-based 3 processor schedule:



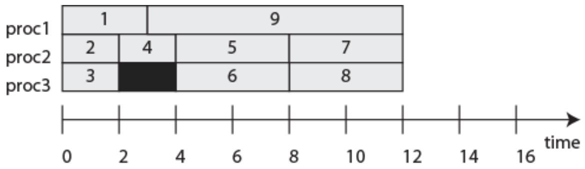
Reducing the computation times by 1 also results in a longer execution time.



Richard’s Anomalies

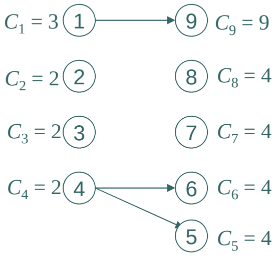


9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Priority-based 3 processor schedule:

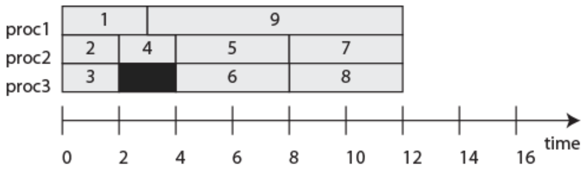


What happens if you remove the precedence constraints (4,8) and (4,7)?

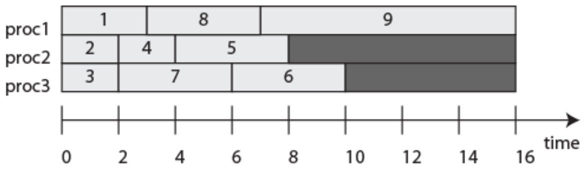
Richard's Anomalies:
Weakening the precedence constraints



9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Priority-based 3 processor schedule:

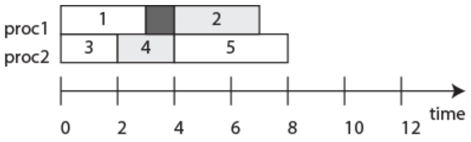


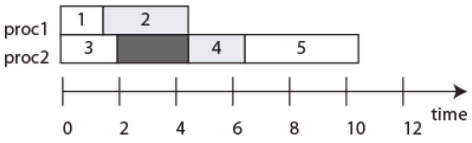
Weakening precedence constraints can also result in a longer schedule.



Richard's Anomalies with Mutexes:
Reducing Execution Time

Assume tasks 2 and 4 share the same resource in exclusive mode, and tasks are statically allocated to processors. Then if the execution time of task 1 is reduced, the schedule length increases:





Conclusion

Timing behavior under all known task scheduling strategies is brittle. Small changes can have big (and unexpected) consequences.

Unfortunately, since execution times are so hard to predict, such brittleness can result in unexpected system failures.