Introduction to Embedded Systems

Edward A. Lee & Sanjit Seshia
UC Berkeley
EECS 124
Spring 2008

Lecture 20: Scheduling Anomalies

Source

This lecture draws heavily from:

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

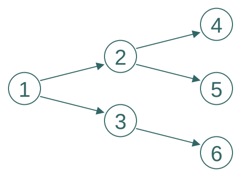
Review

- Rate-Monotonic Scheduling
- Earliest Deadline First
- Execution Time Estimation (WCET)

Outline

- Precedences
  - Acyclic precedence graphs
  - LDF scheduling
- Mutual exclusion
  - Priority inversion
  - Priority inheritance
  - Priority ceiling
- Multiprocessor scheduling
  - Richard’s anomalies

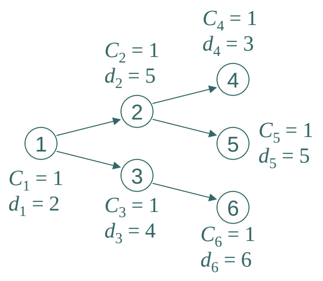
Precedence Constraints

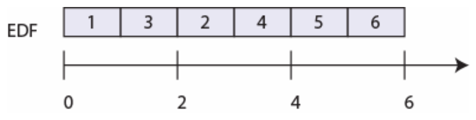


DAG, showing that task 1 must complete before tasks 2 and 3 can be started, etc.

A directed acyclic graph (DAG) shows precedences, which indicate which tasks must complete before other tasks start.

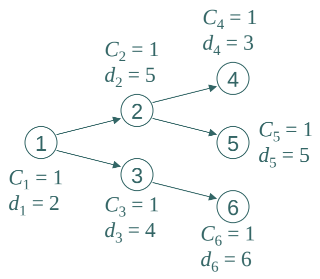
Example: EDF Schedule

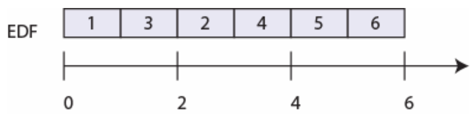




Is this feasible? Is it optimal?

EDF is not optimal under precedence constraints

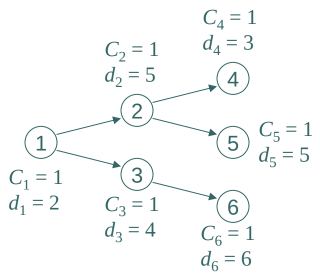


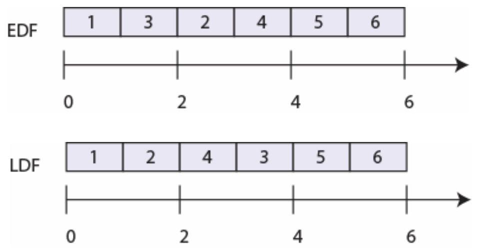


The EDF schedule chooses task 3 at time 1 because it has an earlier deadline. This choice results in task 4 missing its deadline.

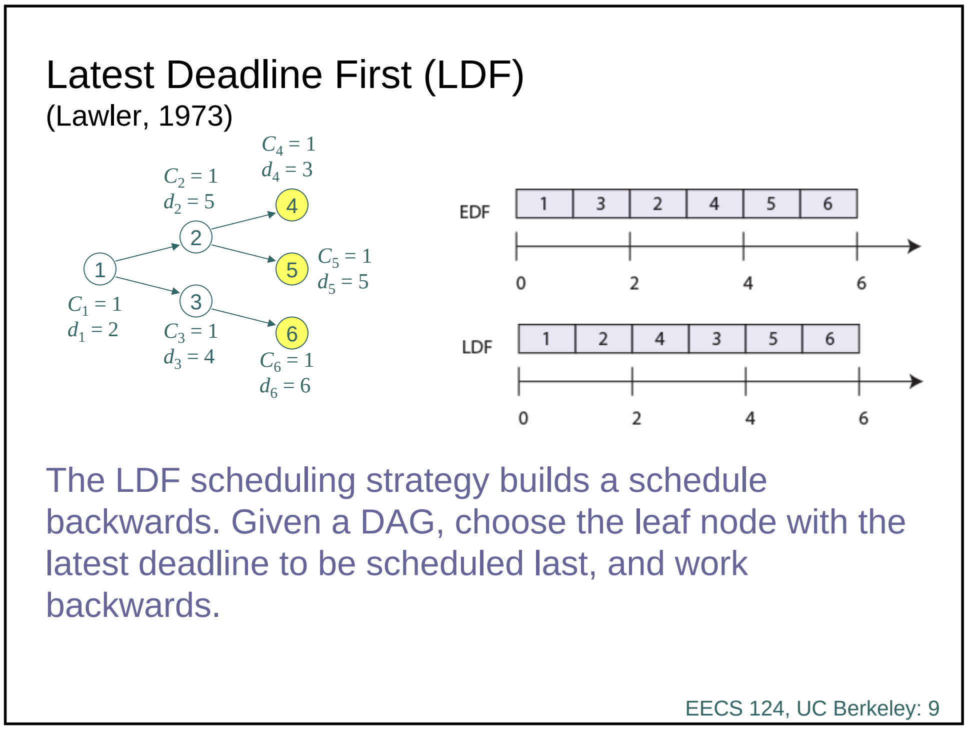
Is there a feasible schedule?

LDF is optimal under precedence constraints



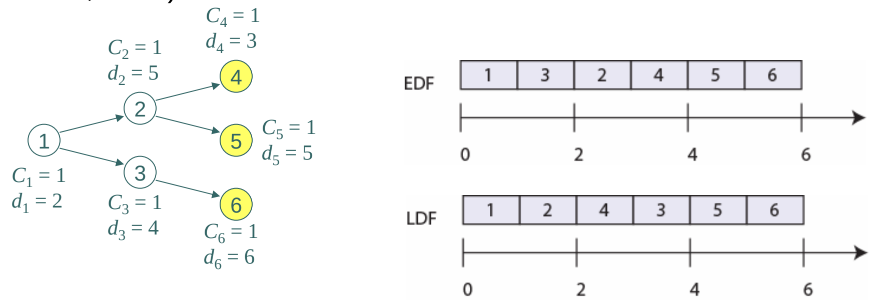


The LDF schedule shown at the bottom respects all precedences and meets all deadlines.

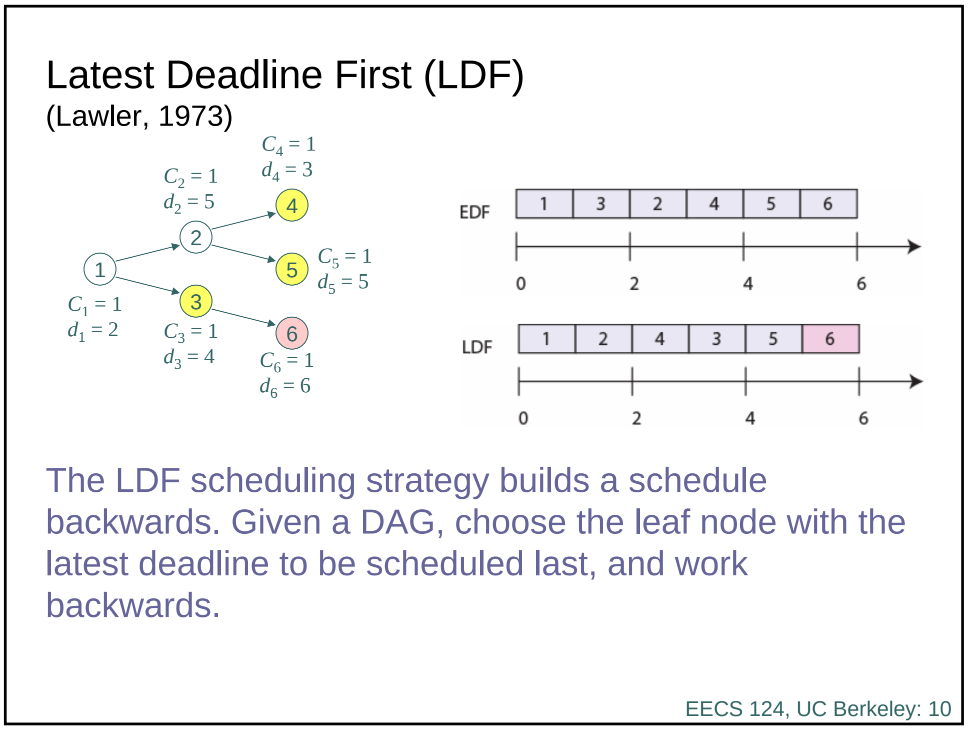


Latest Deadline First (LDF)

(Lawler, 1973)

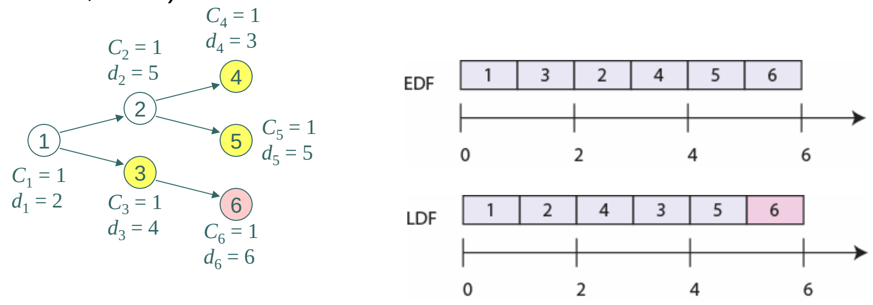


The LDF scheduling strategy builds a schedule backwards. Given a DAG, choose the leaf node with the latest deadline to be scheduled last, and work backwards.



Latest Deadline First (LDF)

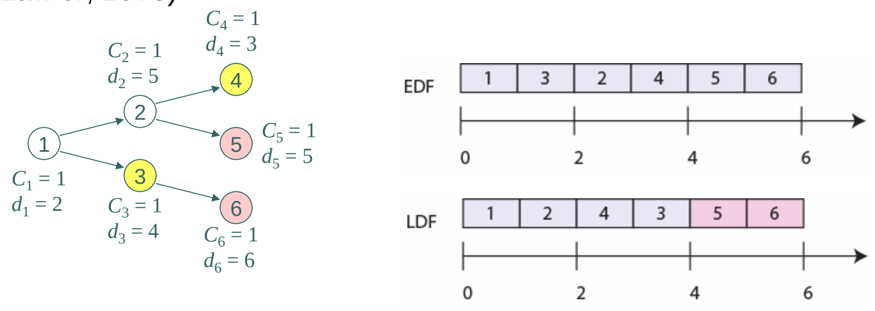
(Lawler, 1973)



The LDF scheduling strategy builds a schedule backwards. Given a DAG, choose the leaf node with the latest deadline to be scheduled last, and work backwards.

Latest Deadline First (LDF)

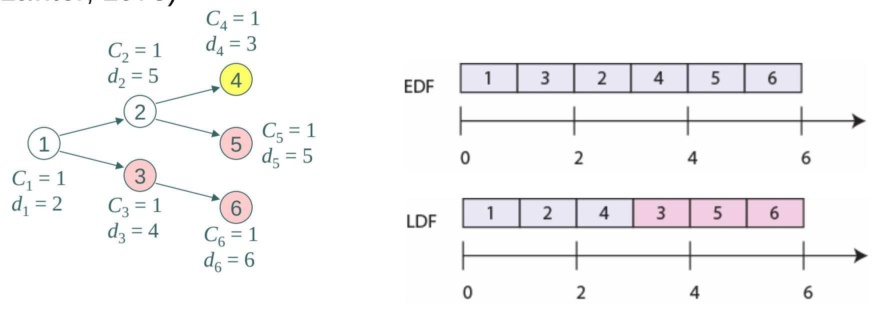
(Lawler, 1973)



The LDF scheduling strategy builds a schedule backwards. Given a DAG, choose the leaf node with the latest deadline to be scheduled last, and work backwards.

Latest Deadline First (LDF)

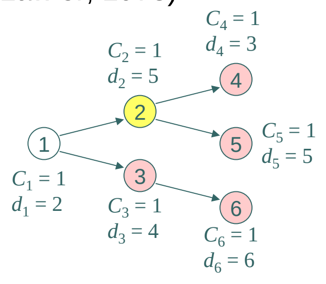
(Lawler, 1973)

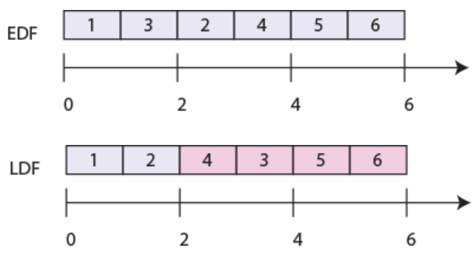


The LDF scheduling strategy builds a schedule backwards. Given a DAG, choose the leaf node with the latest deadline to be scheduled last, and work backwards.

Latest Deadline First (LDF)

(Lawler, 1973)

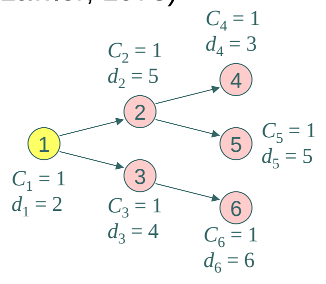


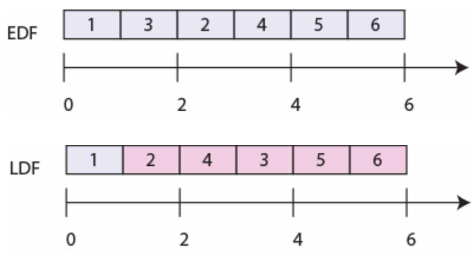


The LDF scheduling strategy builds a schedule backwards. Given a DAG, choose the leaf node with the latest deadline to be scheduled last, and work backwards.

Latest Deadline First (LDF)

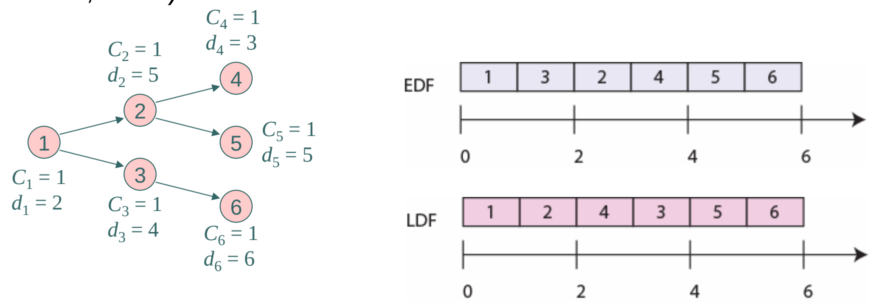
(Lawler, 1973)





The LDF scheduling strategy builds a schedule backwards. Given a DAG, choose the leaf node with the latest deadline to be scheduled last, and work backwards.

Latest Deadline First (LDF)
(Lawler, 1973)



The LDF scheduling strategy builds a schedule backwards. Given a DAG, choose the leaf node with the latest deadline to be scheduled last, and work backwards.

Latest Deadline First (LDF)
(Lawler, 1973)

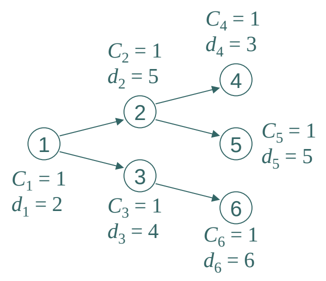
LDF is optimal in the sense that it minimizes the maximum lateness.

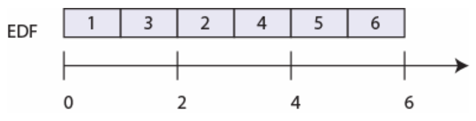
It does not require preemption.

However, it requires that all tasks be available and their precedences known before any task is executed.

EDF with Precedences

With a preemptive scheduler, EDF can be modified to account for precedences and to allow tasks to arrive at arbitrary times. Simply adjust the deadlines and arrival times according to the precedences.





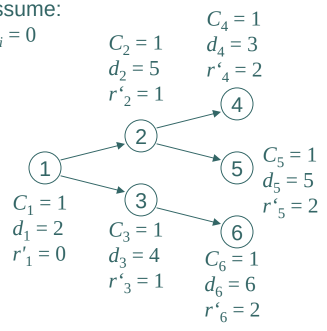
Recall that for the tasks at the left, EDF yields the schedule above, where task 4 misses its deadline.

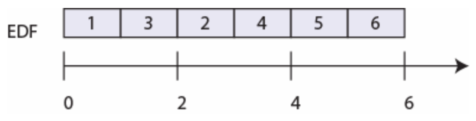
EDF with Precedences
Modifying release times

Given n tasks with precedences and release times r_i, if task i immediately precedes task j, then modify the release times as follows:

$$r'_j = \max(r_j, r_i + C_i)$$

assume:
$r_i = 0$



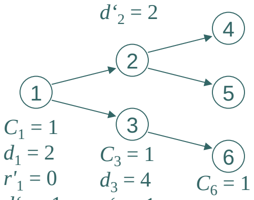


EDF with Precedences
Modifying deadlines

Given n tasks with precedences and deadlines d_i, if task i immediately precedes task j, then modify the deadlines as follows:

$$d'_i = \min(d_i, d'_j - C_j)$$

assume:
$r_i = 0$



$C_1 = 1$
$d_1 = 2$
$r'_1 = 0$
$d'_2 = 1$

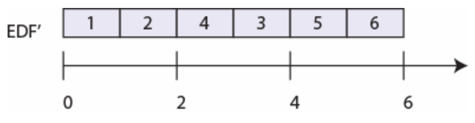
$C_2 = 1$
$d_2 = 5$
$r'_2 = 1$
$d'_2 = 2$

$C_3 = 1$
$d_3 = 4$
$r'_3 = 1$
$d'_3 = 4$

$C_4 = 1$
$d_4 = 3$
$r'_4 = 2$
$d'_4 = 3$

$C_5 = 1$
$d_5 = 5$
$r'_5 = 2$
$d'_5 = 5$

$C_6 = 1$
$d_6 = 6$
$r'_6 = 2$
$d'_6 = 6$



Using the revised release times and deadlines, the above EDF schedule is optimal and meets all deadlines.

Optimality

EDF with precedences is optimal in the sense of minimizing the maximum lateness.

Accounting for Mutual Exclusion

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

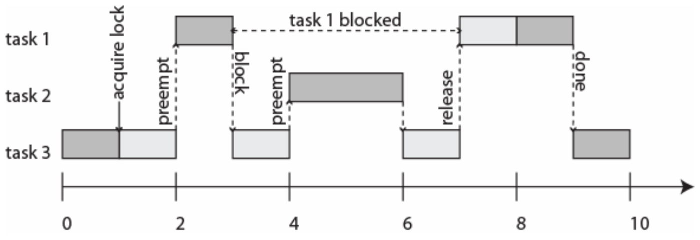
Mutexes can also complicate scheduling.

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);
  ...
}
```

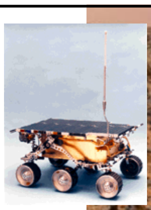
Whenever a data structure is shared across threads, access to the data structure must usually be atomic. This is enforced using mutexes, or mutual exclusion locks. The code executed while holding a lock is called a *critical section*.

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

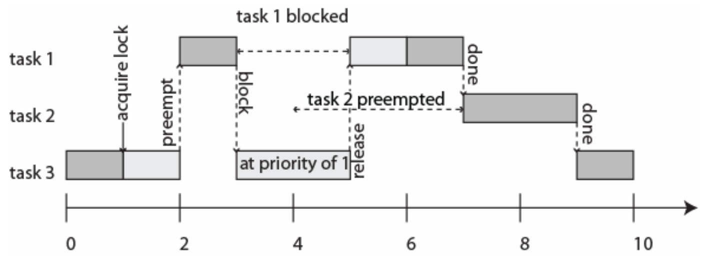


The Mars Rover Pathfinder landed on Mars on July 4th, 1997. A few days into the mission, the Pathfinder began sporadically missing deadlines, causing total system resets, each with loss of data. The problem was diagnosed on the ground as priority inversion, where a low priority meteorological task was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

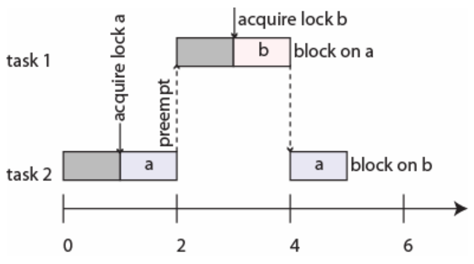
Priority Inheritance Protocol (PIP)

(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 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.

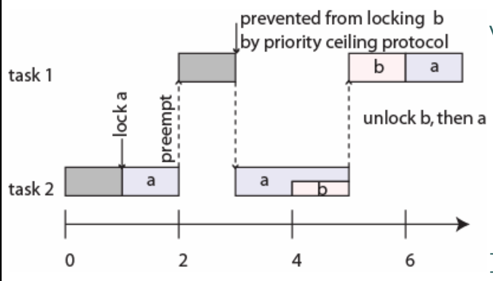
```
#include <pthread.h>
...
pthread_mutex_t lock_a, lock_b;

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.
- A task is preventing from acquiring a lock unless its priority is higher than the priority ceiling of all locks current held by other tasks.
- 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);
  ...
}
```

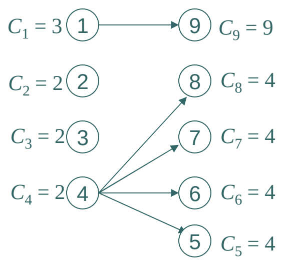
Brittleness

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

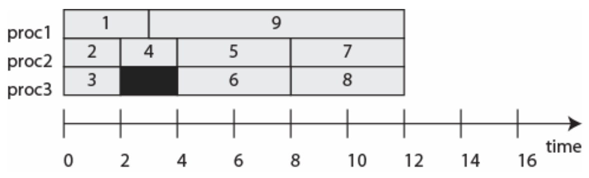
I will illustrate this with multiprocessor (or multicore) schedules.

Theorem (Richard Graham, 1976): If a task set with fixed priorities, execution times, and precedence constraints is optimally scheduled 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

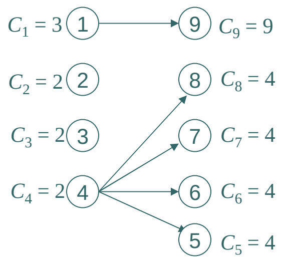


9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Optimal 3 processor schedule:

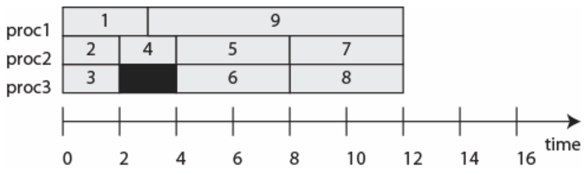


What happens if you increase the number of processors to four?

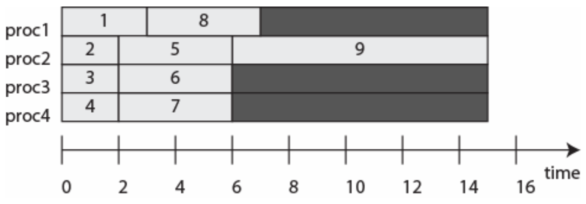
Richard’s Anomalies: Increasing the number of processors



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

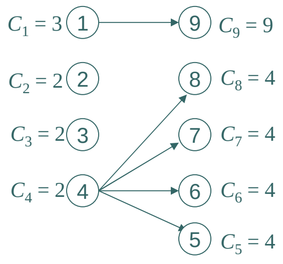


The optimal schedule with four processors has a longer execution time.

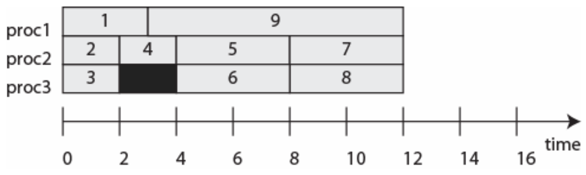


EECS 124, UC Berkeley: 31

Richard’s Anomalies



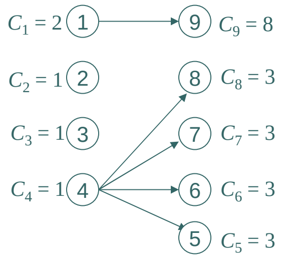
9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Optimal 3 processor schedule:



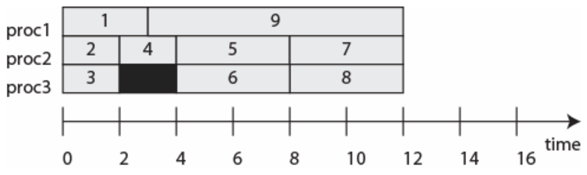
What happens if you reduce all computation times by 1?

EECS 124, UC Berkeley: 32

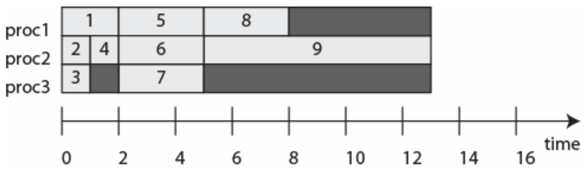
Richard’s Anomalies: Reducing computation times



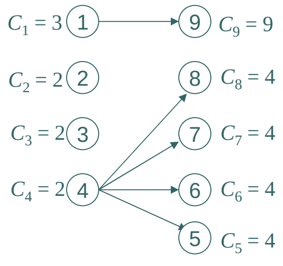
9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Optimal 3 processor schedule:



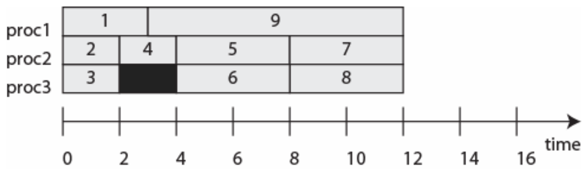
Reducing the computation times by 1 also results in a longer execution time.



Richard’s Anomalies

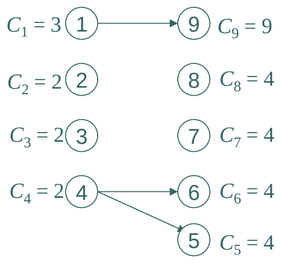


9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Optimal 3 processor schedule:

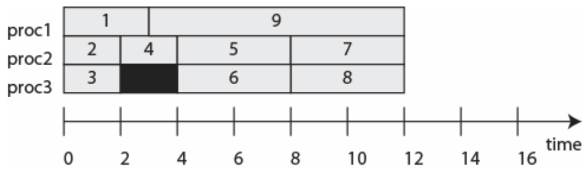


What happens if you remove the precedence constraints (4,8) and (4,7)?

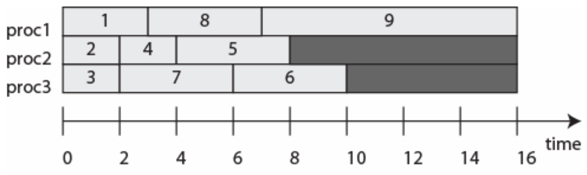
Richard's Anomalies: Weakening the precedence constraints



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

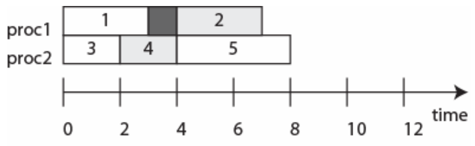


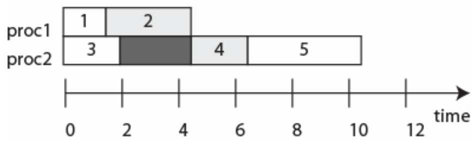
Weakening precedence constraints can also result in a longer schedule.



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.