Lecture 20: Real-Time Scheduling III

Seyed-Hosein Attarzadeh-Niaki

Based on the Slides by Edward Lee and Rodolfo Pellizzoni

Review

- Earliest Due Date (EDD) and Earliest Deadline First (EDF) scheduling
  - Optimality
- Precedence Constraints
  - Latest Deadline First (LDF) scheduling
  - EDF* scheduling

Outline

- Mutual exclusion
  - Priority inversion
    - Priority inheritance protocol
  - Deadlock
    - Priority ceiling protocol
- Aperiodic scheduling
  - Polling server
  - Sporadic server
- Multiprocessor scheduling
  - Brittleness
  - Richard's anomalies

Accounting for Mutual Exclusion

Recall from a previous lectures:

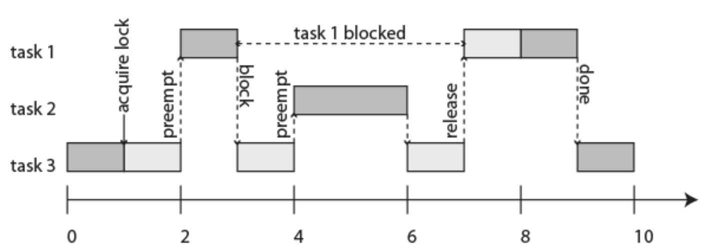
- When threads access shared resources, they need to use mutexes to ensure data integrity.
- Mutexes can also complicate scheduling.

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.

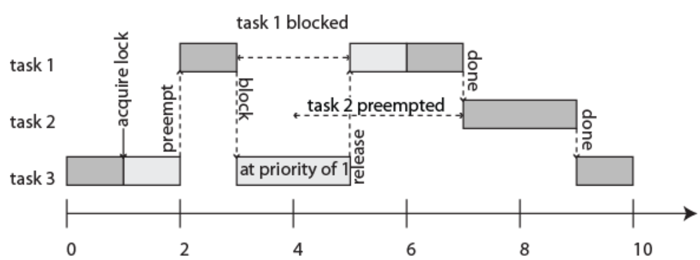
Mars Rover Pathfinder

- 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)

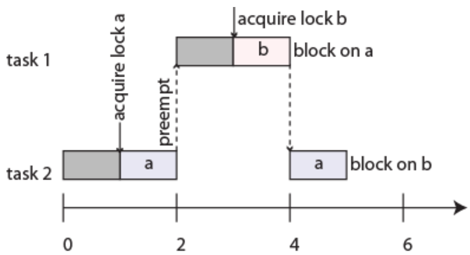
(Sha, Rajkumar, Lehoczky, 1990)



➢ The task that holds the lock inherits the priority of the blocked task.

- 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.
- <u>Task 3 inherits the priority of task 1</u>, 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*.

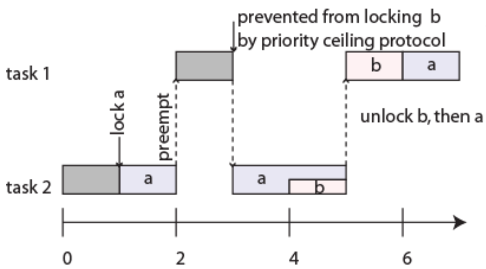
```
#include <pthread.h>
...
pthread_mutex_t lock_a, lock_b;

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



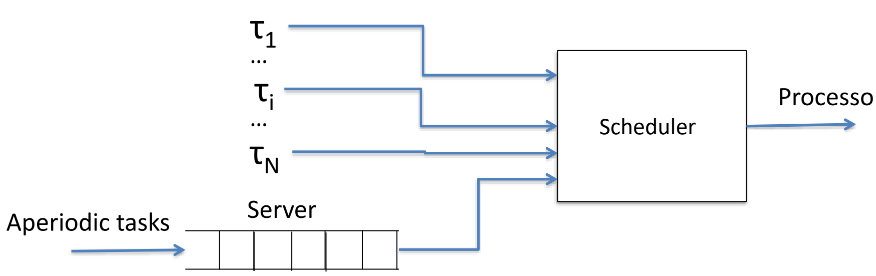
In this version, locks a and b have priority ceilings equal to the priority of task 1.

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);
  ...
}
```

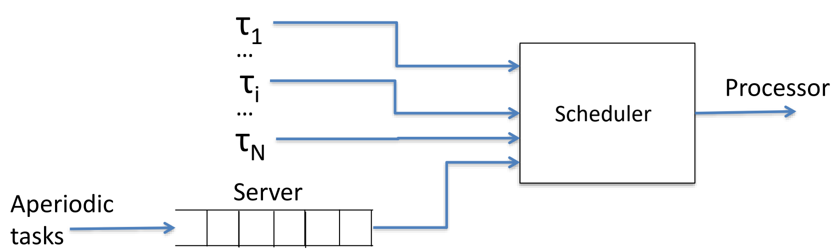

Aperiodic Tasks

- What happens if we mix aperiodic and periodic tasks?
- **Main idea**: ensure that *periodic tasks remain schedulable* no matter what.
- Aperiodic server
  - Insert aperiodic tasks into a queue (server)
  - Scheduler picks among periodic tasks or the aperiodic server
- Server Goals
  - Minimize response time of aperiodic tasks
  - Low overhead



Aperiodic Servers
Solution#1: background server

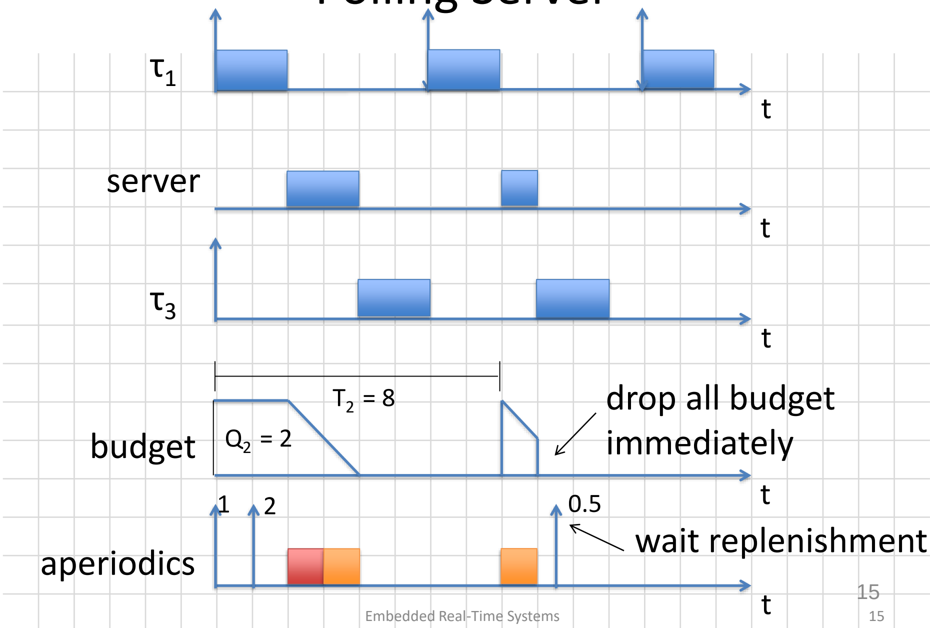
- Execute aperiodics whenever the CPU is not running a periodic task (i.e., the server has lowest priority)
- Problem: response time can be very high.



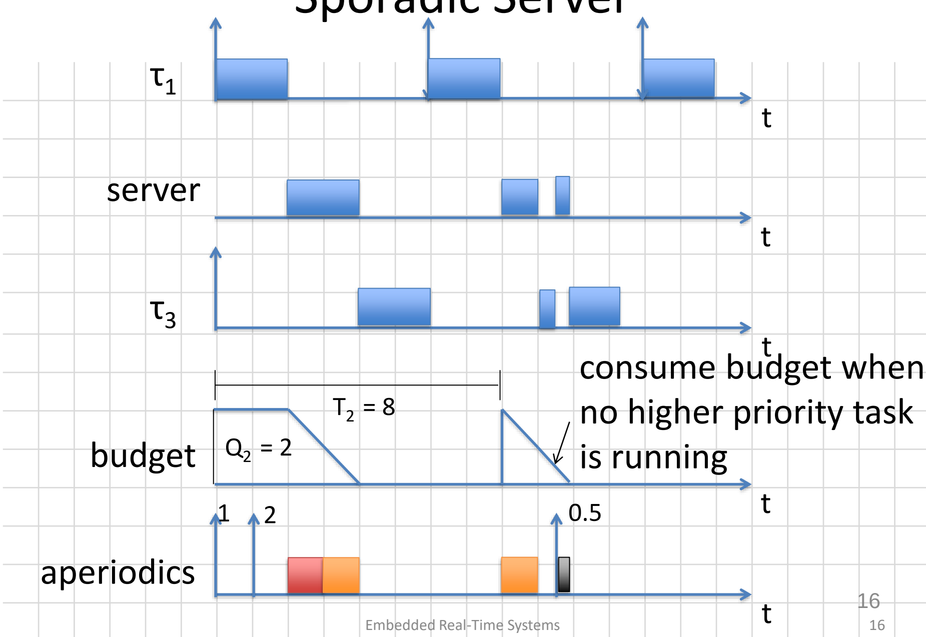
Aperiodic Servers
Solution#2: budget-based server

- Server is assigned budget Q_i, period T_i.
- The server behaves like a periodic task with $C_i = Q_i$ and period T_i.
- When the scheduler picks the server, if there is budget left, the server executes an aperiodic in the queue consuming its budget.
- When budget=0, server waits until next period, then replenish budget to Q_i.
- Problem: what happens if the scheduler picks the server and there are no queued aperiodic tasks?
  - **"Dumb" servers** (polling server) lose budget.
  - **"Smart" servers** (ex: sporadic server) keep the budget but modify their activation (recharge) time.

Polling Server



Sporadic Server



Scheduling on Multiprocessor

Solution #1: partitioning

- Statically assign tasks among M processors.
- Ex: EDF. Each core is schedulable if sum of utilizations of tasks assigned to that core <= 1.
- Problem can be rephrased as: given a set of objects with known sizes (task utilizations), place them into M equal-size containers.
  - Classic bin-packing problem
  - NP-hard

Solution #2: global scheduling

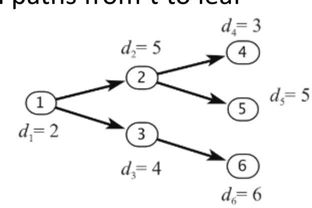
- Keep a global scheduling queue.
- Whenever there is a free core, pick one task from the queue and schedule it on the core.

Scheduling on Multiprocessor

- Typical scheduling goal: minimizing makespan.
- In practice, real-time adoption of multiprocessor is limited, especially for hard (real-time) systems.
- Partitioned scheduling preferred.
- Three *issues with global scheduling*
  1. Increases unpredictability – tasks can migrate among cores.
  2. Much more complex to implement.
  3. Does not necessarily perform better than partitioned.

Global Scheduling Strategies

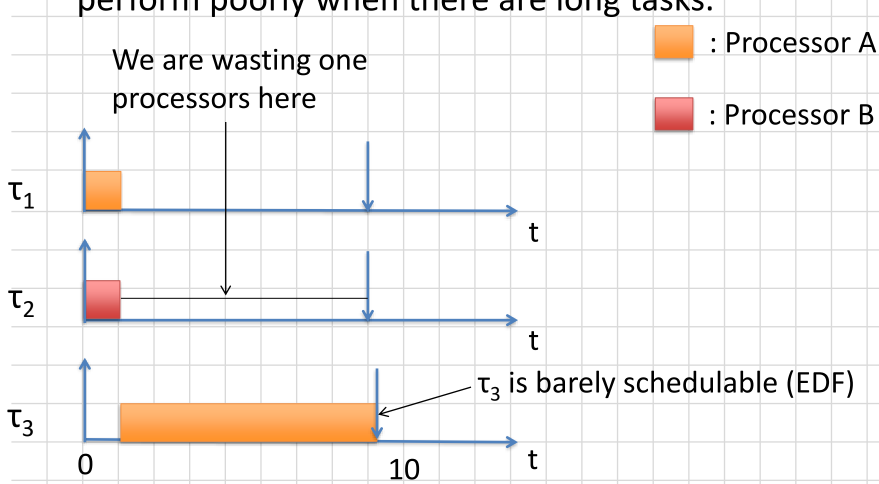
- Hu level scheduling algorithm
  - Assigns a priority to each task τ based on the *level*
    - Largest sum of exec times of tasks on paths from τ to leaf tasks.
    - Larger level -> higher priority
    - Critical path-based
    - Example priorities:
      - High τ_1; Medium τ_2, τ_3; Low τ_4, τ_5, τ_6



- List scheduler
  - Sorts the tasks by priorities,
  - assigns them to processors in the order of the sorted list as processors become available.

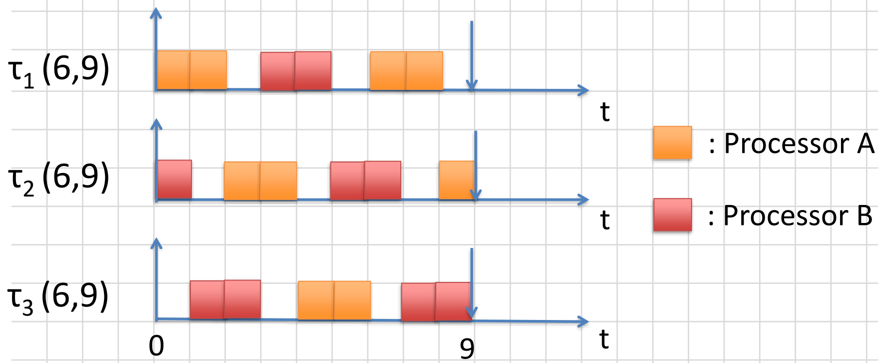
Improving Global Scheduling

- Problem: both fixed-priority and EDF scheduling perform poorly when there are long tasks.



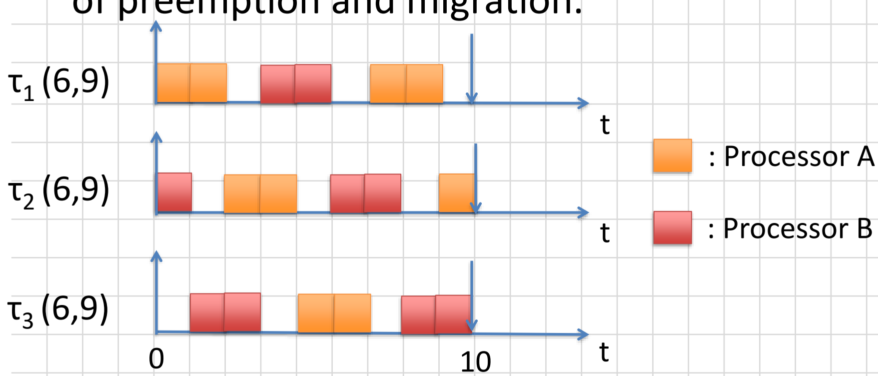
Improving Global Scheduling

- Optimal algorithm: p-fair.
- Split the tasks into small chunks.
- Allocate them on the cores in a “fair” way.



Improving Global Scheduling

- Pros: task set schedulable on M cores iff U <= M ; the algorithm is optimal.
- Cons: this does not take into account the cost of preemption and migration.



Scheduling on Multiprocessor

Some other details…

- There are several (only sufficient) schedulability analyses for EDF and FP – both based on utilization bounds and response time…
- There are extensions for resource sharing protocols and aperiodic servers to multicores…

- Very active research topic, but often ignores the main problem – how do we determine the worst-case computation time?

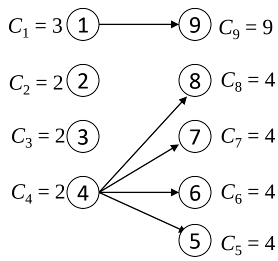
Brittleness

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

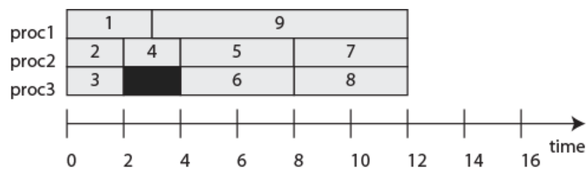
- A good illustration of this is with multiprocessor (or multicore) schedules.

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

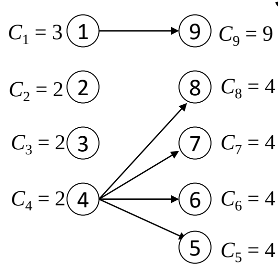


9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Priority-based 3 processor schedule:

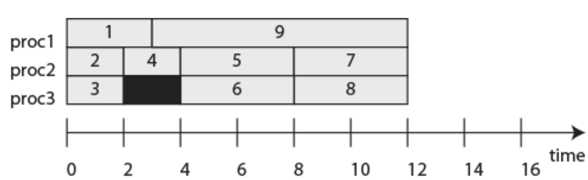


What happens if you increase the number of processors to four?

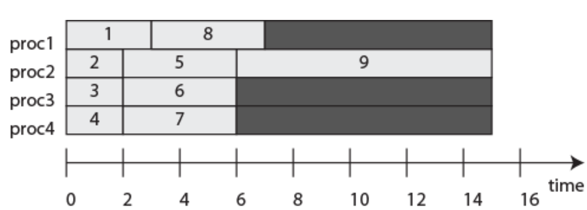
Richard's Anomalies: Increasing the number of processors



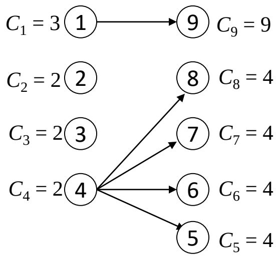
9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Priority-based 3 processor schedule:



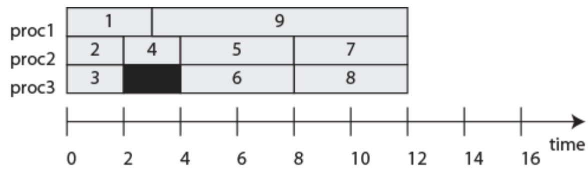
The priority-based schedule with four processors has a longer execution time.



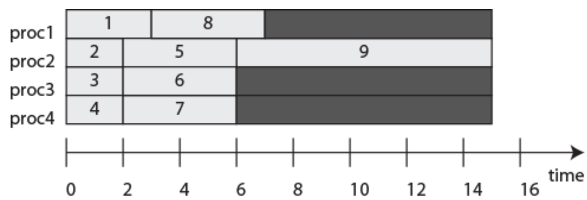
Greedy Scheduling



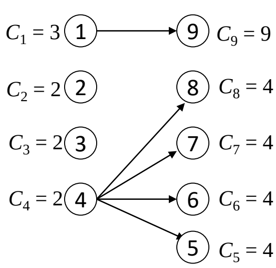
9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Priority-based 3 processor schedule:



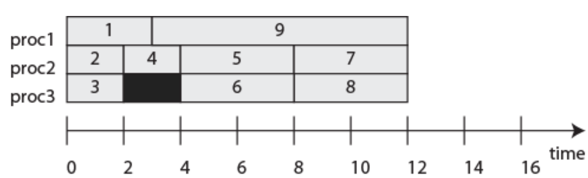
Priority-based scheduling is "greedy." A smarter scheduler for this example could hold off scheduling 5, 6, or 7, leaving a processor idle for one time unit.



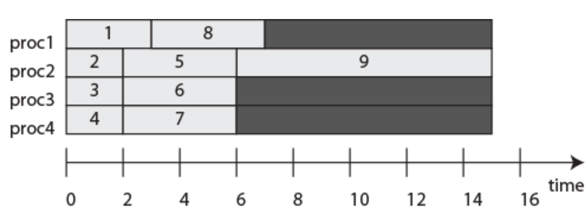
Greedy scheduling may be the only practical option.



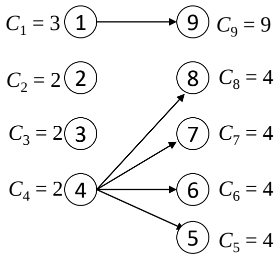
9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Priority-based 3 processor schedule:



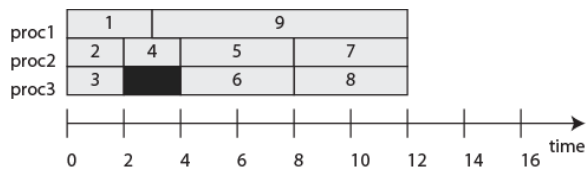
If tasks "arrive" (become known to the scheduler) only after their predecessor completes, then greedy scheduling may be the only practical option.



Richard’s Anomalies

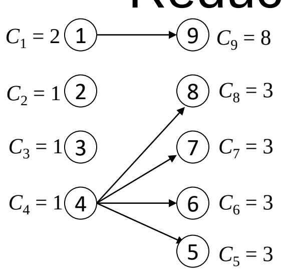


9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Priority-based 3 processor schedule:

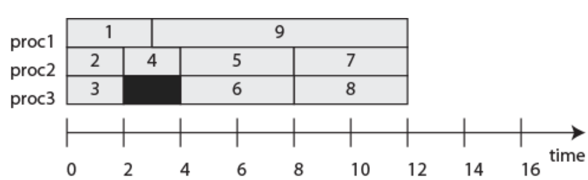


What happens if you reduce all computation times by 1?

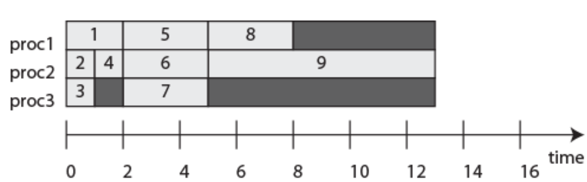
Richard’s Anomalies: Reducing computation times



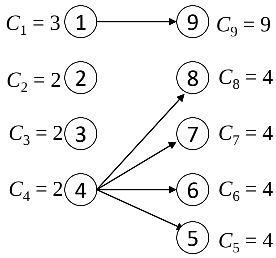
9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Priority-based 3 processor schedule:



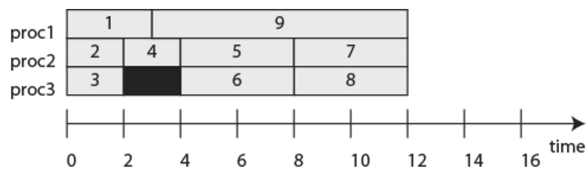
Reducing the computation times by 1 also results in a longer execution time.



Richard's Anomalies

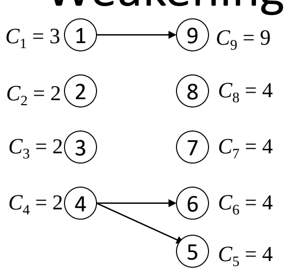


9 tasks with precedences and the shown execution times, where lower numbered tasks have higher priority than higher numbered tasks. Priority-based 3 processor schedule:

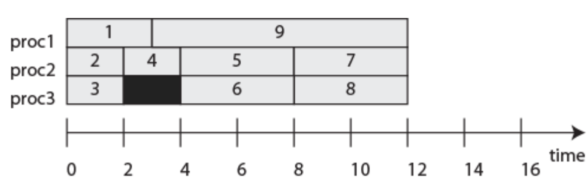


What happens if you remove the precedence constraints (4,8) and (4,7)?

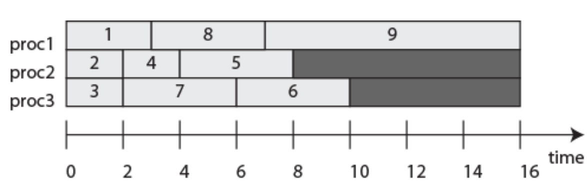
Richard's Anomalies: Weakening the precedence constraints



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:

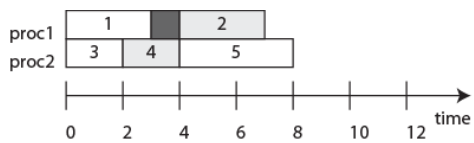


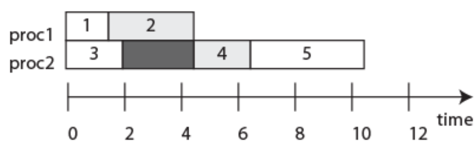
Weakening precedence constraints can also result in a longer schedule.



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.
- If the execution time of task 1 is reduced, the schedule length increases:





In short…

- 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.