Task Synchronization on Multiprocessors

Raj Rajkumar
Lecture #16

Multiprocessor Task Synchronization

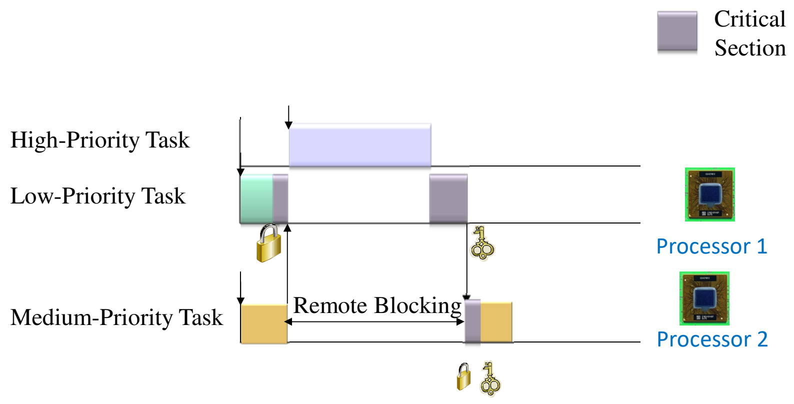
- Multiprocessor synchronization challenges
  - Remote blocking
  - Self-suspending behavior
  - Multiple priority inversions due to suspensions
- Uniprocessor solutions need to be revisited
  - Priority inheritance protocols
  - Priority ceiling protocol

Revisiting Blocking from Resource Sharing

- Consider the blocking term B_i of a task τ_i as the **additional time penalty** incurred by τ_i due to resource sharing **relative to the case when there is absolutely *no* resource sharing** in the taskset.
- On a uniprocessor, when one of the priority inheritance protocols is used, only the critical sections of lower priority tasks can contribute to B_i
  - Under the **basic priority inheritance protocol**, up to *(m, n)* critical sections of lower priority critical sections, where *n* is the number of tasks with lower priority than τ_i and *m* is the number of mutexes accessed by lower-priority tasks with a priority ceiling higher than or equal to the priority of τ_i
  - Under the **priority ceiling protocol**, at most one critical section of a lower-priority task with a priority ceiling higher than or equal to the priority of τ_i
  - The **highest locker protocol** has the same property as the priority ceiling protocol
  - Under the **non-preemption protocol**, at most any critical section of a lower-priority task

Remote Blocking

- Remote Blocking
  - Tasks wait on resources held on other cores
  - Leads to wastage of CPU utilization

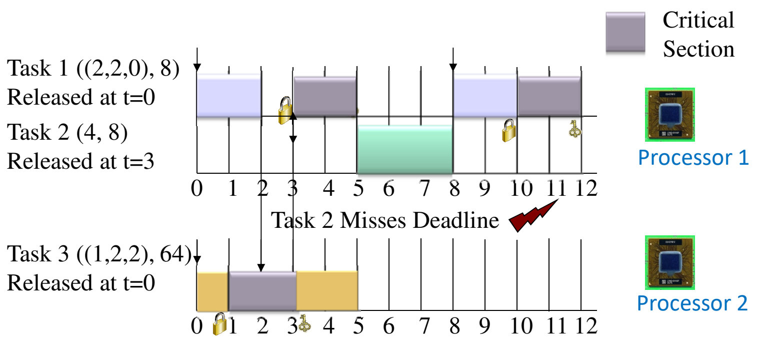


Multiprocessor Resource Sharing

- If a task shares a resource with *any* task on another (remote) processor, it pays an additional penalty in the worst case.
- So, sharing resources with tasks of **any** priority on **any** other processor can contribute to an increase in B_i.

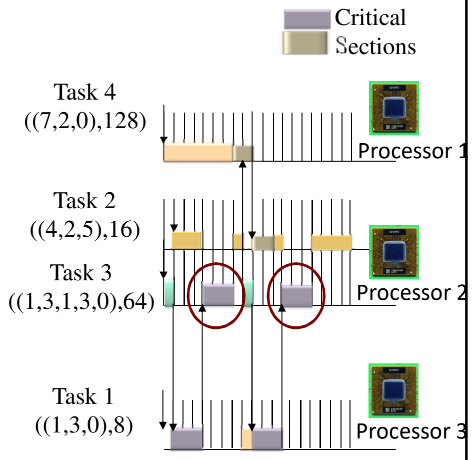
Penalties due to Self-Suspension

- Tasks suspend on remote cores
  - In the uni-processor context
    - Tasks suspend on other tasks in the same processor
  - Leads to various scheduling penalties



Multiple Priority Inversions

- When a task suspends
  - Lower-priority tasks could be released
  - These lower-priority tasks could acquire resources
    - Due to their priority ceilings, they can cause preemptions
- The worst case:
  - For each task segment,
  - One preemption from each lower-priority task



Priority Ceiling Protocol Extension

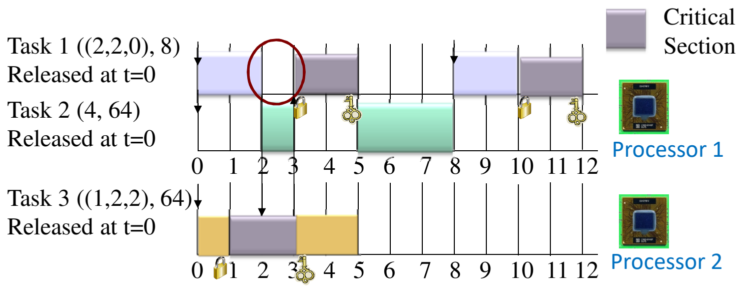
- Multiprocessor Priority Ceiling Protocol (MPCP)
  - **Global critical sections** guarded by **global mutexes**
  - Global mutexes associated with global priorities
    - **The Global Priority ceiling of global mutex G_M** is calculated as:

Max (All task normal execution priorities) +
1 + Max (Normal priorities of tasks accessing G_M)

- Critical section executions can be preempted
  - By other critical section executions with an even greater priority ceiling
- Each mutex M_G has a priority queue
  - Pending tasks acquire a mutex in priority order

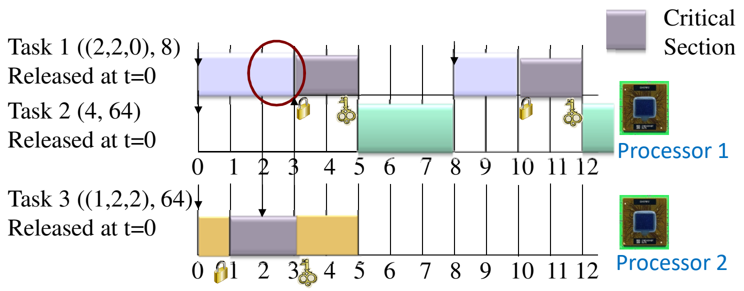
Suspension-based Protocol

- When global mutex request is pending
  - Task suspended and other active tasks execute
- Issues:
  - Self-suspension penalties, multiple priority inversions



Spinning-based Protocol

- When a global mutex request is pending
  - Task continues to "spin" till it acquires resource
- Issues:
  - Spinning-based utilization loss



To Suspend or Not to Suspend?

- Use spin-locks when the critical section duration is pretty short
  - Roughly comparable to the time it would take to context-switch to other tasks
- Use suspension otherwise and let CPU perform useful functions instead of spinning in idle loop
  - Potential exception: when there are multiple lower-priority tasks accessing several global mutexes
    - Could lead to significant remote blocking penalties

Bin-Packing with Global Mutexes

Evaluating The Effectiveness of Bin-Packing

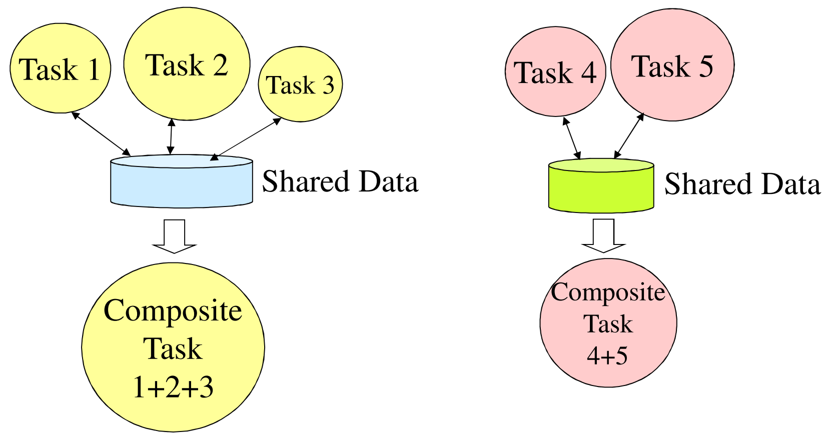
- Suppose the number of bins required by a scheme *S* to pack a set of objects is |*S*|.
- One may want to compare the performance of a bin-packing heuristic *H* with that of the *optimal* scheme *O*
- Analyses:
  - **Worst-case analysis**: theoretical study to determine the absolute worst-case number of bins required by *H* relative to that of *O*
    - i.e. find *max*(|*H*| ÷ |*O*|)
  - **Average-case analysis**: compute (|*H*| ÷ |*O*|) for randomly picked sets of objects
    - Perform over a large sample so that the results are statistically significant
    - How do we know |*O*|?
      - Finding the optimal packing is NP-hard:
      - Will take an exponential amount of time (unless P = NP)
    - **Create the optimal solution by construction**!
      - Assume fully packed bins: split into known # of objects in each bin into random sizes (for a given # of bins = |O|)
      - Unpack these objects and offer to heuristic for packing
      - Original packing we started with is optimal!

Synchronization-Aware Bin-Packing

Making Bin-Packing Synchronization-Aware

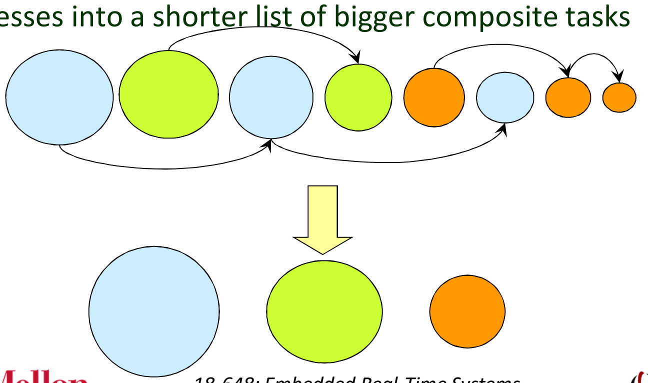
- Global critical sections lead to
  - Significantly more blocking (i.e. the B_i term becomes much bigger)
    - "Blocking" can be caused both by-higher priority and lower-priority tasks on other processors.
  - Deferred execution behaviors can result in scheduling penalties
- Conversely, local critical sections
  - Utilize the (local) priority ceiling protocols (like the highest locker priority) to bound B_i to a single (local) critical section of a lower priority task
  - Experience no deferred execution penalties
- Our bin-packing heuristics use object size as the primary criterion while packing objects into bins
- Can we try to exploit the bin-packing heuristics and try to make "global critical sections" become "local ones"?
  - Combine objects (tasks) into **composite objects**.

Synchronization-Aware Packing



Bin-Packing → Packing of Composite Objects

- If tasks A and B share data, they become a composite task
  - Applies recursively: task A and/or B can also be a composite task
  - The relationship is transitive:
    - i.e. if tasks A and B share one piece of data, and tasks B and C share another piece of data, tasks A, B and C become a single composite task
- Transform the given list of tasks with their shared resource accesses into a shorter list of bigger composite tasks



Bin-Packing of Composite Objects

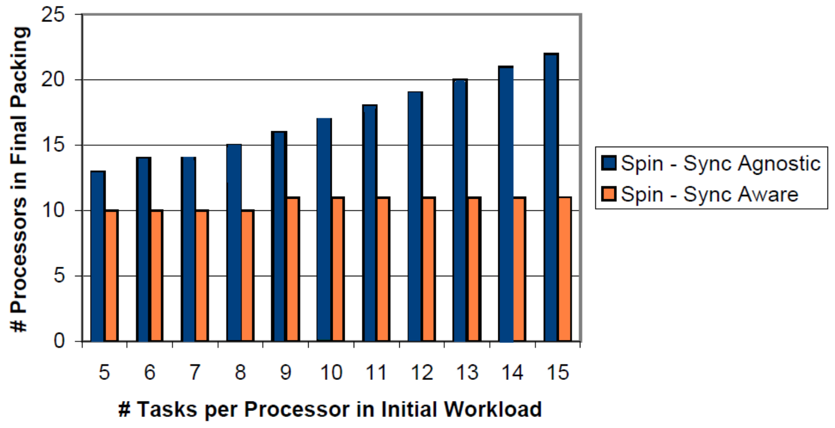
- Use bin-packing heuristics to allocate composite tasks to bins
  1. If a composite task does not fit into a bin, set it aside and allocate other unallocated tasks
  2. After the current iteration, if any unallocated tasks remain, only then try to allocate them
     - This reduces the number of global critical sections
  3. If a composite task does not fit any bin, split into two objects with *minimum synchronization cost* and allocate. If a task cannot be split (i.e. it is an *elementary* task and not a *composite* task), you must create a new bin.
  4. Go to Step 1.

Other variations are possible.

Comparison of Schemes

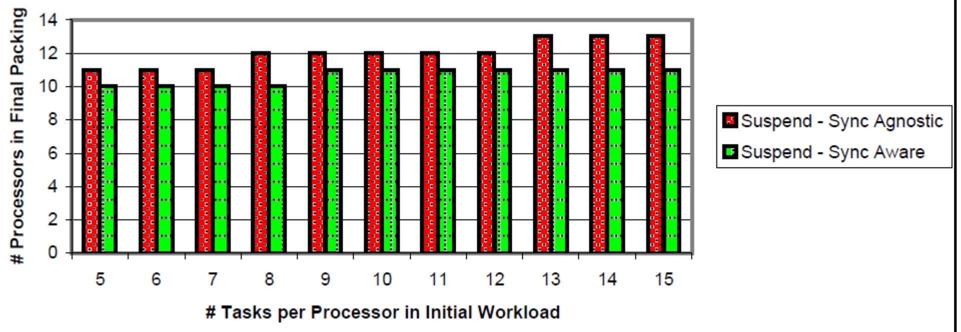
- Synchronization itself between the objects after allocation can be either spin-based or suspension-based.
- Evaluation is *very* dependent on assumptions:
  - Distribution of task utilization (object sizes)
  - Amount of sharing (# of tasks that share the same resources)
  - Number of resource accesses per task
  - Length of each critical section
  - Overhead of preemptions
  - Overhead of caching penalty during preemptions

Synchronization-Aware Bin Packing - Spinning



8 fully packed cores, 2 critical sections per task, 2 lockers per mutex
500µs critical sections, Periods: [10ms, 100ms]

Synchronization-Aware Bin Packing - Suspension



8 fully packed cores, 2 critical sections per task, 2 lockers per mutex
500µs critical sections, Periods: [10ms, 100ms]

Task Schedulability on Multiprocessors

For a task to be deemed schedulable, need to account for

- Its own execution time
- Preemption time from higher-priority tasks
- "Blocking"
  - Local blocking time *and*
  - Remote blocking time
- Jitter penalty

Conclusions

- Multiprocessor synchronization:
  - Global and local mutexes
  - Remote blocking from global mutexes is *much* worse than local blocking caused by local mutexes
  - Self-suspending penalties
    - Multiple priority inversions due to suspensions
- Global priority ceilings assigned to global mutexes
  - always higher than the priority of any task not within a global critical section
- A task may suspend or spin when it waits for a global critical section
  - Unless the global critical section is small, it is useful to suspend and not waste CPU cycles
- Allocation heuristics must target minimizing the need for global mutexes