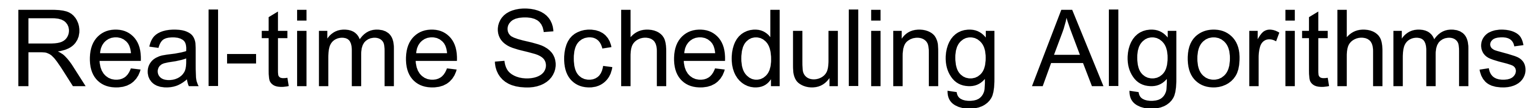
by Greg Levin

based on work with Caitlin Sadowski, Ian Pye, and Scott Brandt

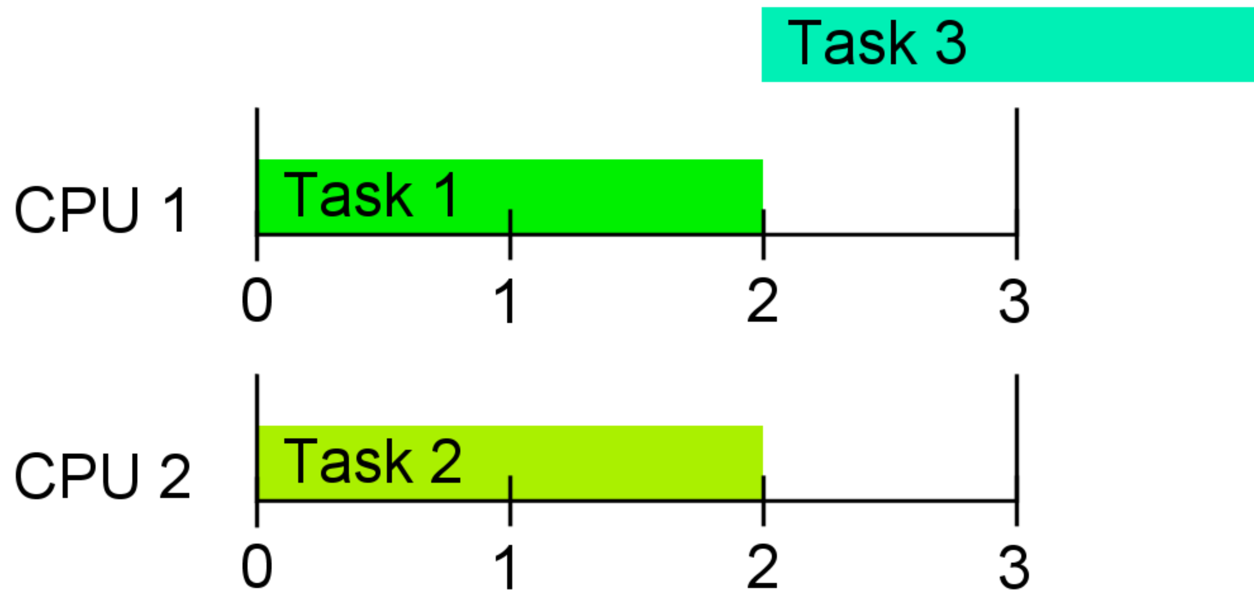


Real-time Scheduling Algorithms

- In a real-time environment, multiple processes with computational deadlines compete for processor time.
  - "Hard real-time" means it is never okay to miss deadlines
- We consider the problem of scheduling a collection of periodic processes running on a multiprocessor system.

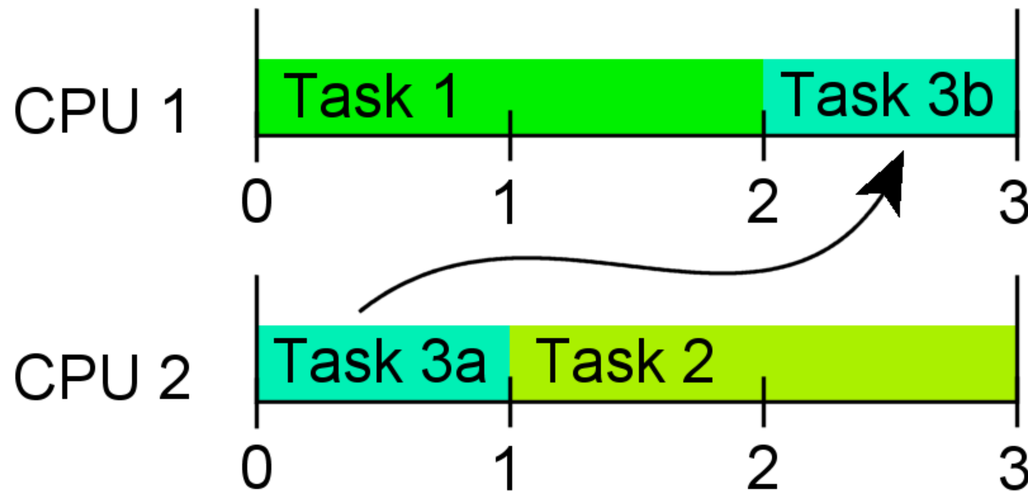
Example

- 2 processors; 3 tasks, each with 2 units of work required every 3 time units



Example

- 2 processors; 3 tasks, each with 2 units of work required every 3 time units



Periodic Tasks

- A *periodic task* is one that requires a certain amount of work be completed within each period.
  - If a task has *period* p and *workload* c, then its *utilization* $u = c / p$ is the fraction of each period that the task must be running.

Our Goal (version 1)

- Given a set of *N* tasks on *M* processors, find a feasible scheduling of tasks so that all deadlines are met (if such a scheduling exists)
  - We say that a scheduling algorithm is "optimal" if it find some successful scheduling for any task set for which some correct scheduling exists

Assumptions

- All processors are equivalent
- Tasks may migrate between processors
- Tasks are independent, and may not run simultaneously on two processors
- No overhead for context switches or migrations
  - This model is theoretical, not realistic
  - In practice, these overheads lead to the use of suboptimal scheduling algorithms

Theorem 1

- Any collection of tasks whose total (summed) utilization does not exceed *M* and whose individual utilizations do not exceed 1 has a feasible scheduling.
- **Proof:** Smaller work intervals can arbitrarily approximate a task's fluid rate curve

Our Goal (version 2)

- Given *M* processors and a set of *N* tasks with total utilization summing to *M*, find a feasible scheduling of tasks which minimizes the number of context switches and migrations
  - This goal is ironic, since we started by assuming that these operations are free

Greedy Scheduling Algorithms

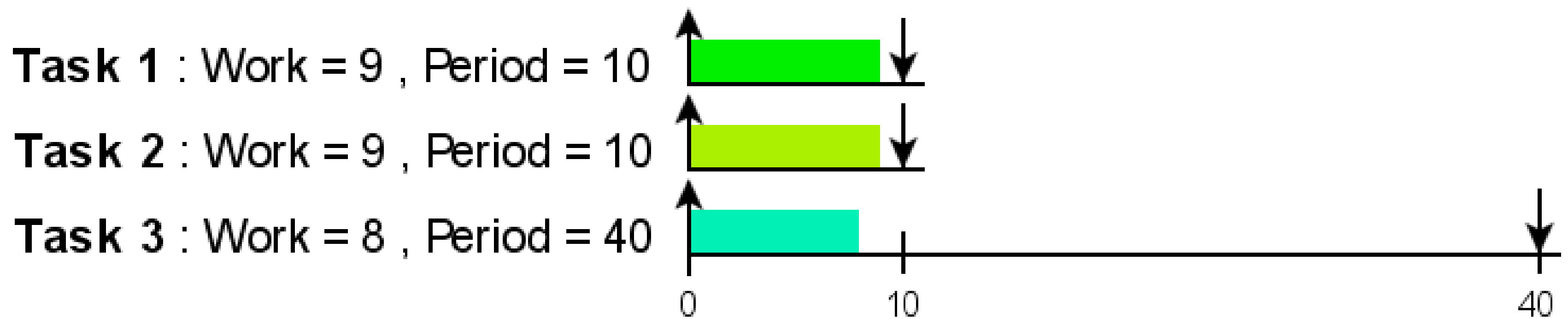
- A *Greedy Scheduling Algorithm* will, at various times, choose the *M* "best" jobs to run. We need to specify:
  - What does "best" mean? Earliest deadline? Most work remaining?
  - How often do all *N* jobs get compared to find the *M* best?

Examples of Greedy Algorithms

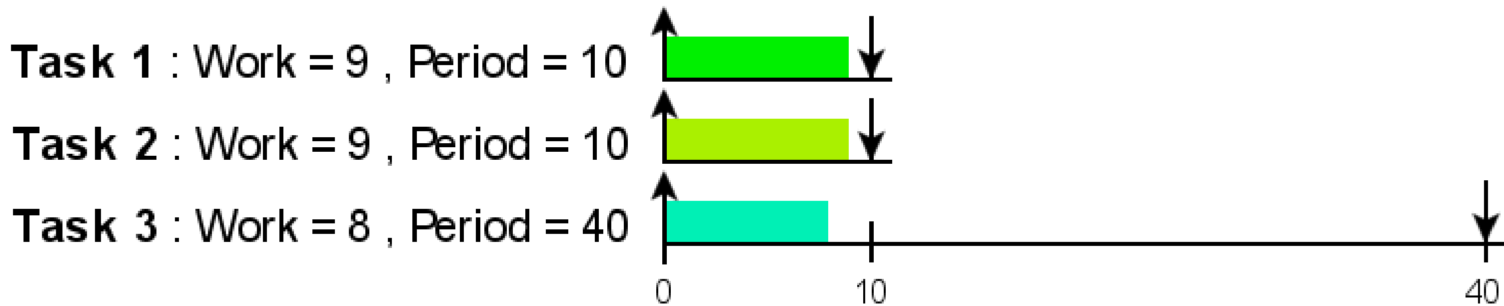
- Earliest Deadline First (EDF)
  - Schedule the *M* tasks with the earliest deadlines
- Least Laxity First (LLF)
  - *Laxity* is a task's remaining possible idle time before it *must* be scheduled in order to finish its workload by its next deadline

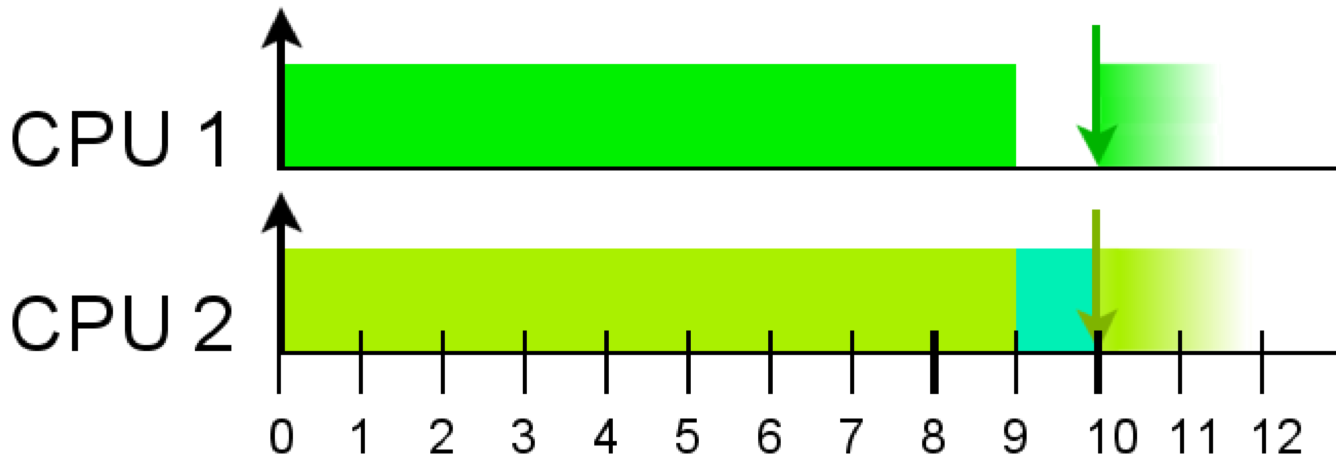
Why Greedy Algorithms Fail On Multiprocessors

- Example:



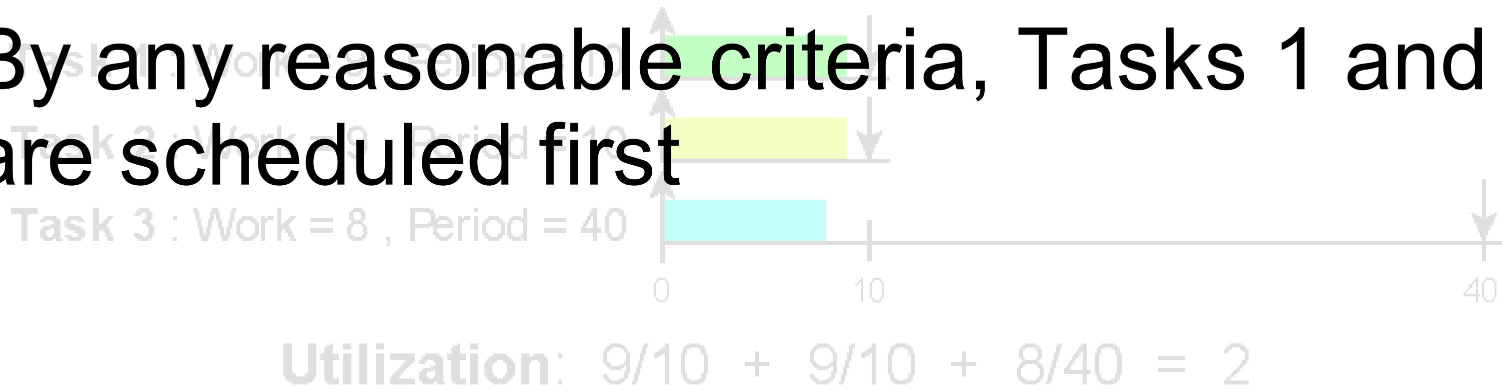
Why Greedy Algorithms Fail On Multiprocessors

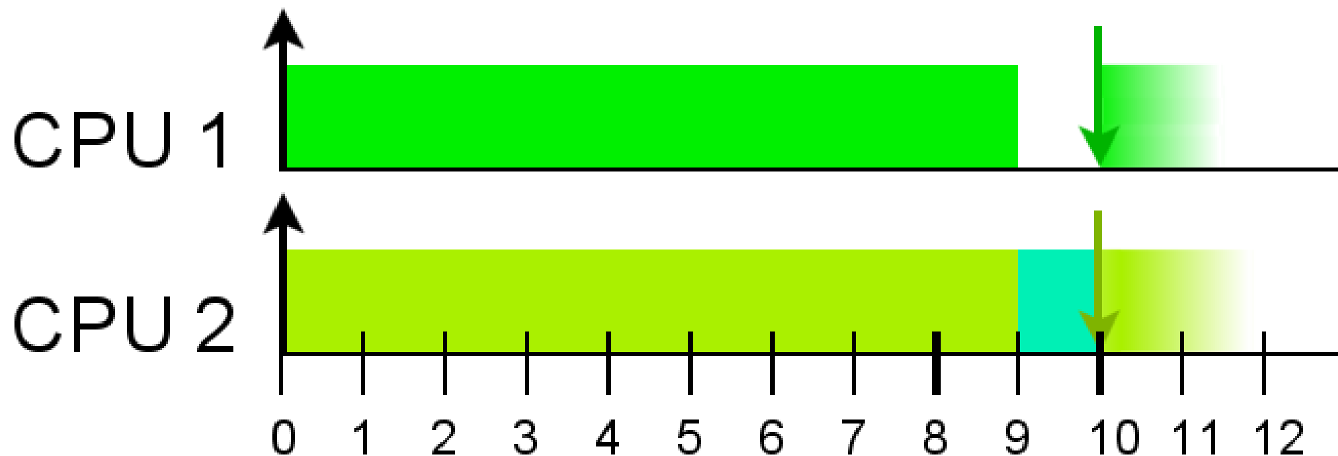




Why Greedy Algorithms Fail On Multiprocessors

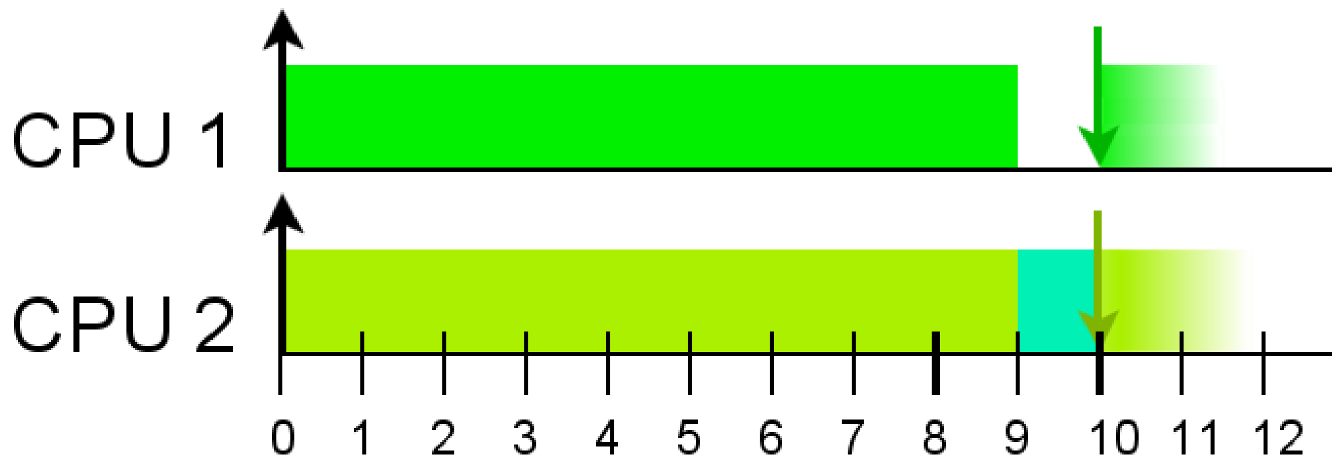
- By any reasonable criteria, Tasks 1 and 2 are scheduled first





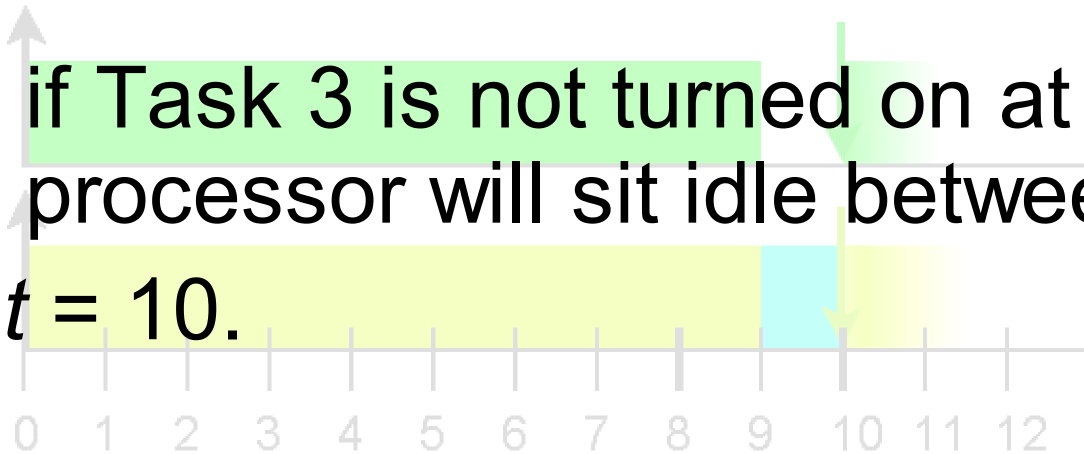
Why Greedy Algorithms Fail On Multiprocessors

- By any reasonable criteria, Tasks 1 and 2 are scheduled first
- Even at time t = 8, Tasks 1 and 2 are the obvious greedy choices



Why Greedy Algorithms Fail On Multiprocessors

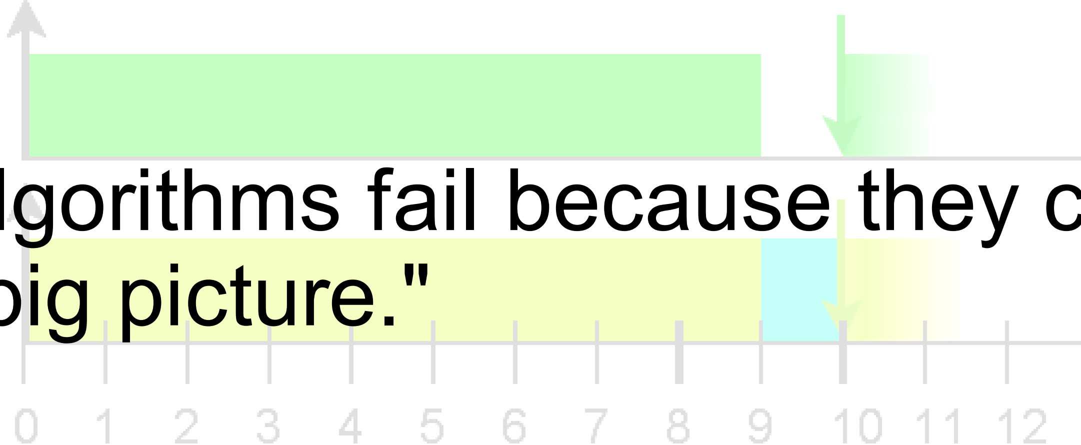
- By any reasonable criteria, Tasks 1 and 2 are scheduled first
- Even at time $t = 8$, Tasks 1 and 2 are the obvious greedy choices
- However, if Task 3 is not turned on at time $t = 8$, one processor will sit idle between $t = 9$ and $t = 10$.



Why Greedy Algorithms Fail On Multiprocessors

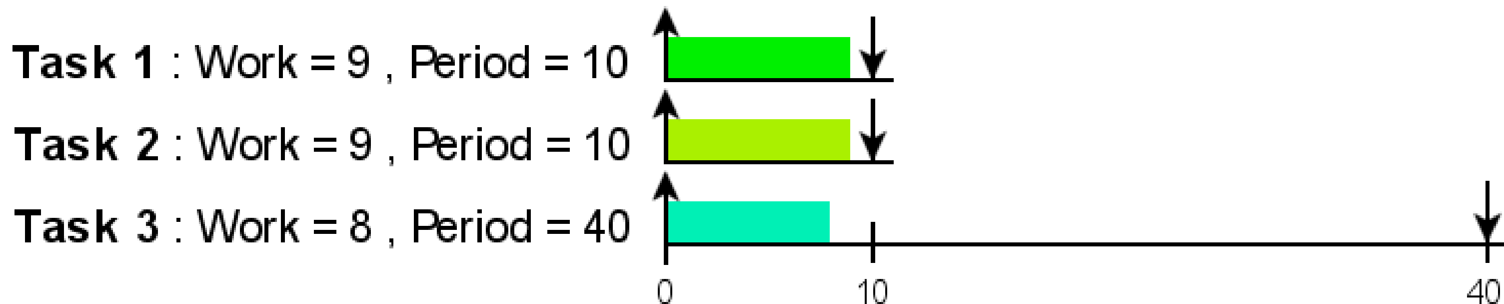
- Before $t = 40$, the two processors can do 80 units of work, and there are

  2 x (9 x 4) + 8 = 80 units of work to do. If there is any idle time, not all deadlines can be met.

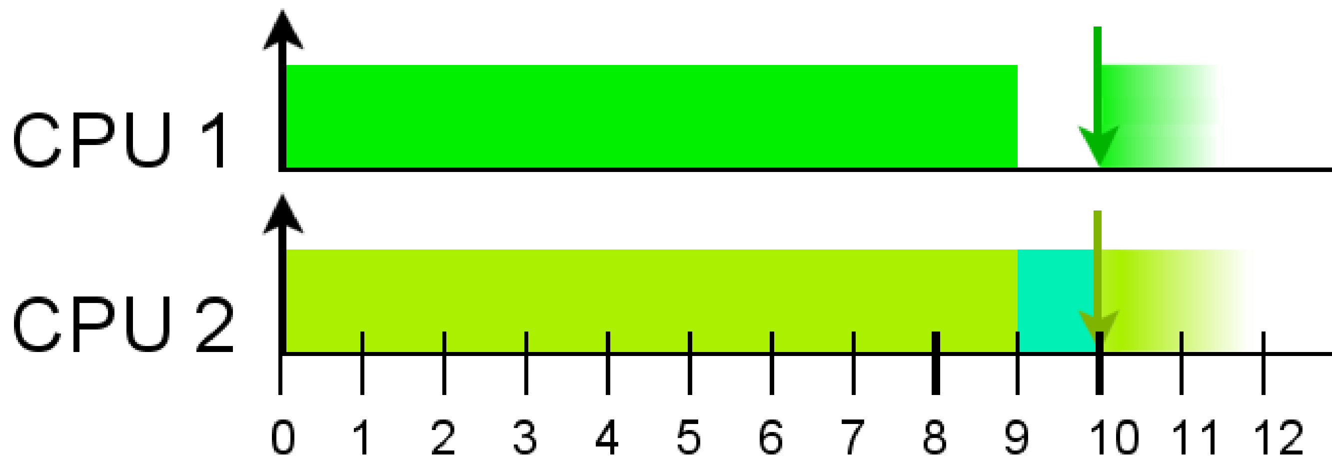


- Greedy algorithms fail because they can't see the "big picture."

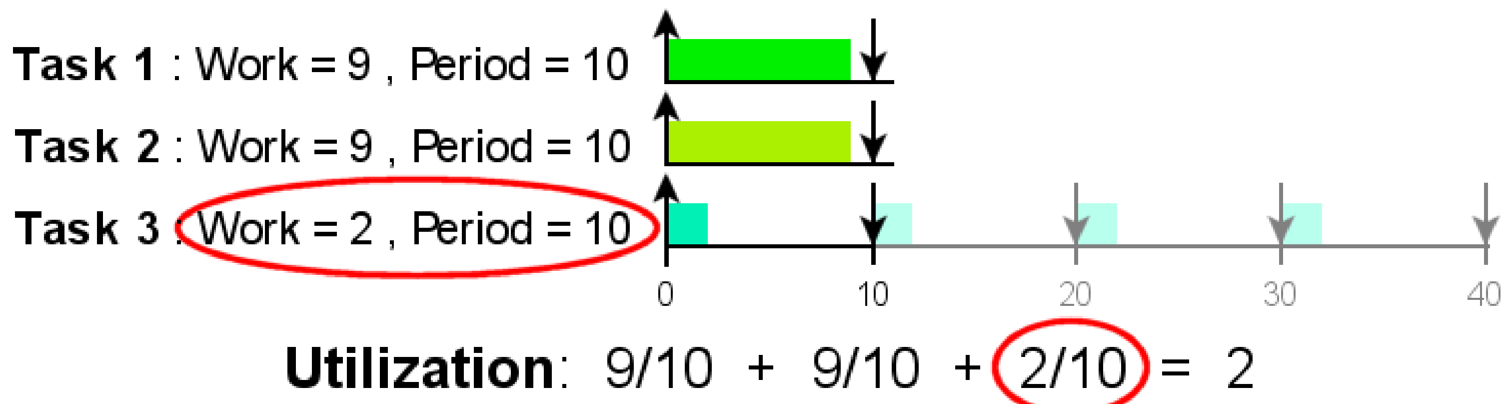
Why Greedy Algorithms Fail On Multiprocessors

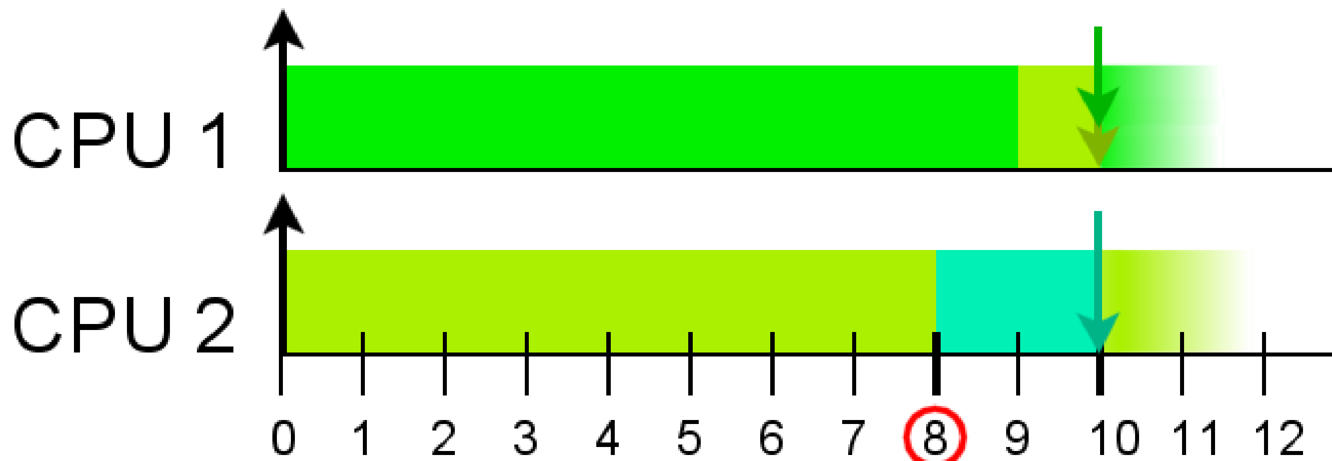


Utilization: 9/10 + 9/10 + 8/40 = 2



Proportioned Algorithms Succeed On Multiprocessors





- **Insight:** Scheduling tasks is much easier when they all have the same deadline
- **Application:** Give all task deadlines to all jobs, and within each such time window, assign each job work proportional to its utilization

- Previous proportional fairness algorithms:
  - **pfair** (1994) - *Baruah, Cohen, Plaxton, Varvel*
  - **LLREF** (2006) - *Cho, Ravindran, Jensen*
  - **EKG** (2006) - *Andersson, Tovar*

- ... but they were all using proportional fairness without understanding its simplicity

Theorem 2: "3 Rules"

- Given a collection of tasks with total utilization *M*, if all tasks are subdivided by assigning all deadlines to all tasks, then a scheduling algorithm within a single time window will succeed if and only if:
  - It always runs any job with zero laxity
  - It never runs any job which is completed
  - *M* distinct jobs are always running

Theorem 2: Proof

That these conditions are necessary is clear: if laxity for a job becomes negative, it cannot be completed on time; running a completed job is locally equivalent to an idle processor, and an idle processor means that less than a total of $M \times t_f$ work will be done in the window, implying failure. We now show that these conditions are also sufficient.

The shortening of local deadlines will create new scheduling events. The WORK COMPLETE and ZERO LAXITY events that occur based on the local deadline at the end of a time window will be referred to as *secondary scheduling events.*

Lemma 1 *A scheduling of jobs which follows Rules (1)-(3) will fail to complete all tasks on time if and only if, at some time before t_f, there are (at least) $M+1$ jobs with zero laxity.*

Proof. $M+1$ jobs with zero laxity cannot all be finished on time on M processors. On the other hand, the set of zero laxity jobs can only grow as time increases, and any job which has not hit zero laxity by time t_f will be completed before then. If there are at most M jobs with zero laxity as time reaches t_f, the other $N-M$ jobs will be completed. Rule (1) ensures that up to M zero laxity jobs may be run simultaneously and to completion. Therefore, an $(M+1)^{st}$ zero laxity job is necessary for failure. □

The point at which that $(M+1)^{st}$ job reaches zero laxity is referred to as a *critical moment*.[1] In the absence of critical moments, the M processors do the required amount of total work, and all tasks will be completed on time. We now introduce some convenient notation, deviating a bit from Cho et. al.. Anything subscripted with i, j will indicate job i at the jth secondary event (which we'll assign time t_j). $c_{i,j}$ is the *local work remaining* to job i at time t_j. $u_{i,j} = \frac{c_{i,j}}{t_f - t_j}$ is the *local remaining utilization* of job i, namely, the average rate at which it must consumed to successfully complete. Finally, we define $S_j = \sum_{i=1}^{N} u_{i,j}$ to be the *total local utilization* at time j, (the rate at which the whole system must run).

[1] The notion of a critical moment and the subsequent Lemma 2 are due to Cho et. al..

Lemma 2 **(Cho et. al.)** *If $S_j \leq M$ at time t_j, then event j is not a critical moment.*

Proof. If event j is a critical moment, then each of the $M+1$ zero laxity jobs (for simplicity, let's call them $T_1, \ldots, T_{M+1}$) has $u_{i,j} = 1$. Then $S_j = \sum_{i=1}^{N} u_{i,j} \geq \sum_{i=1}^{M+1} u_{i,j} = M+1 > M$. □

In order to show that a scheduling following Rules (1)-(3) is feasible, it suffices to show that $S_j \leq M$ at each secondary event j.

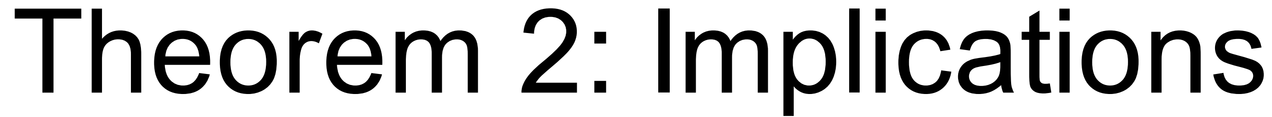
Proof of Theorem. If we let W_j be the *total work remaining* at time t_j, then

$$W_j = \sum_{i=1}^{N} c_{i,j} = \sum_{i=1}^{N} u_{i,j}(t_f - t_j) = (t_f - t_j)S_t \quad .$$

According to Lemma 2, we can only run into trouble if S_j exceeds M at some event. However, at any time t_j within the scheduling window, if all M processors have been fully utilized between times 0 and t_j, then W_j has been getting reduced at a constant rate of M-per-time unit. That is, $W_j = W_0 - t_j \cdot M$. Since $W_0 \leq t_f \cdot M$ by our general feasibility condition, we have that

$$S_t = \frac{W_0 - t_j \cdot M}{t_f - t_j} \leq \frac{t_f \cdot M - t_j \cdot M}{t_f - t_j} = M \quad .$$

So long as we fully utilize all processors when available, S_t can never exceed M, and, by Lemma 2, we can never end up with more than M tasks with zero laxity. □



- Once we've subdivided all jobs into "time windows", with all system deadlines as boundaries, correct scheduling goes from being incredibly complicated to nearly trivial.

- What is the simplest possible algorithm?

Stack-and-Slice (SNS)

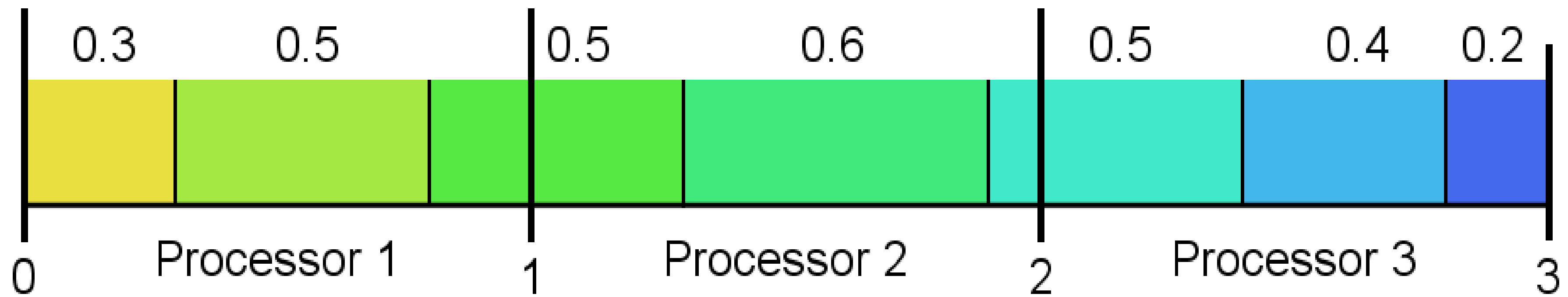
- All time windows are, up to linear scaling, equivalent, so normalize time window length to 1
- All jobs now have work equal to their utilization, and workloads add up to (no more than) M

Stack-and-Slice (SNS)

- Example: 3 processors, 7 tasks

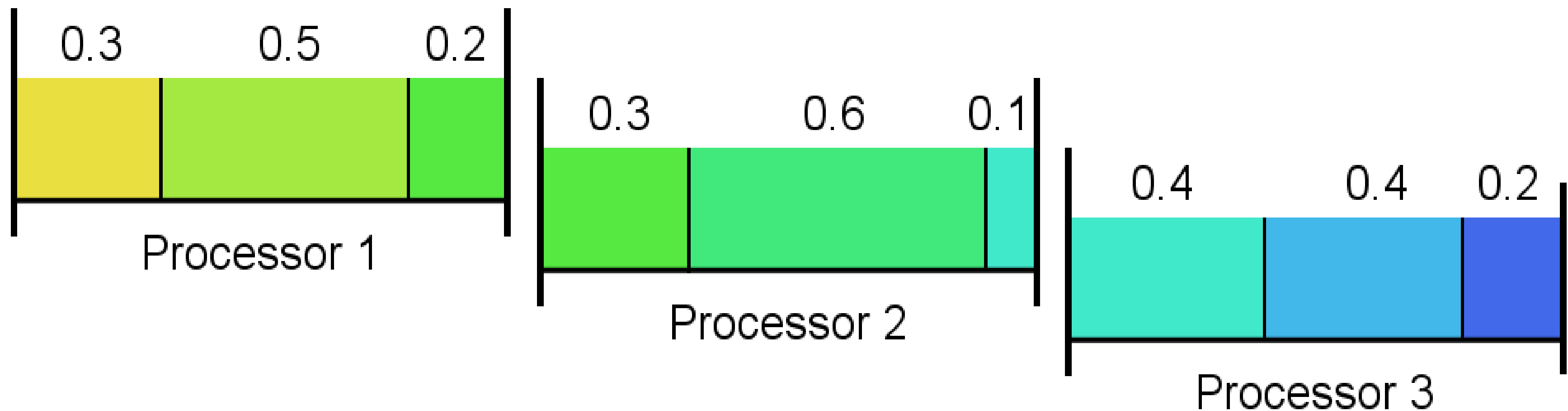
Stack-and-Slice (SNS)

- Example: 3 processors, 7 tasks

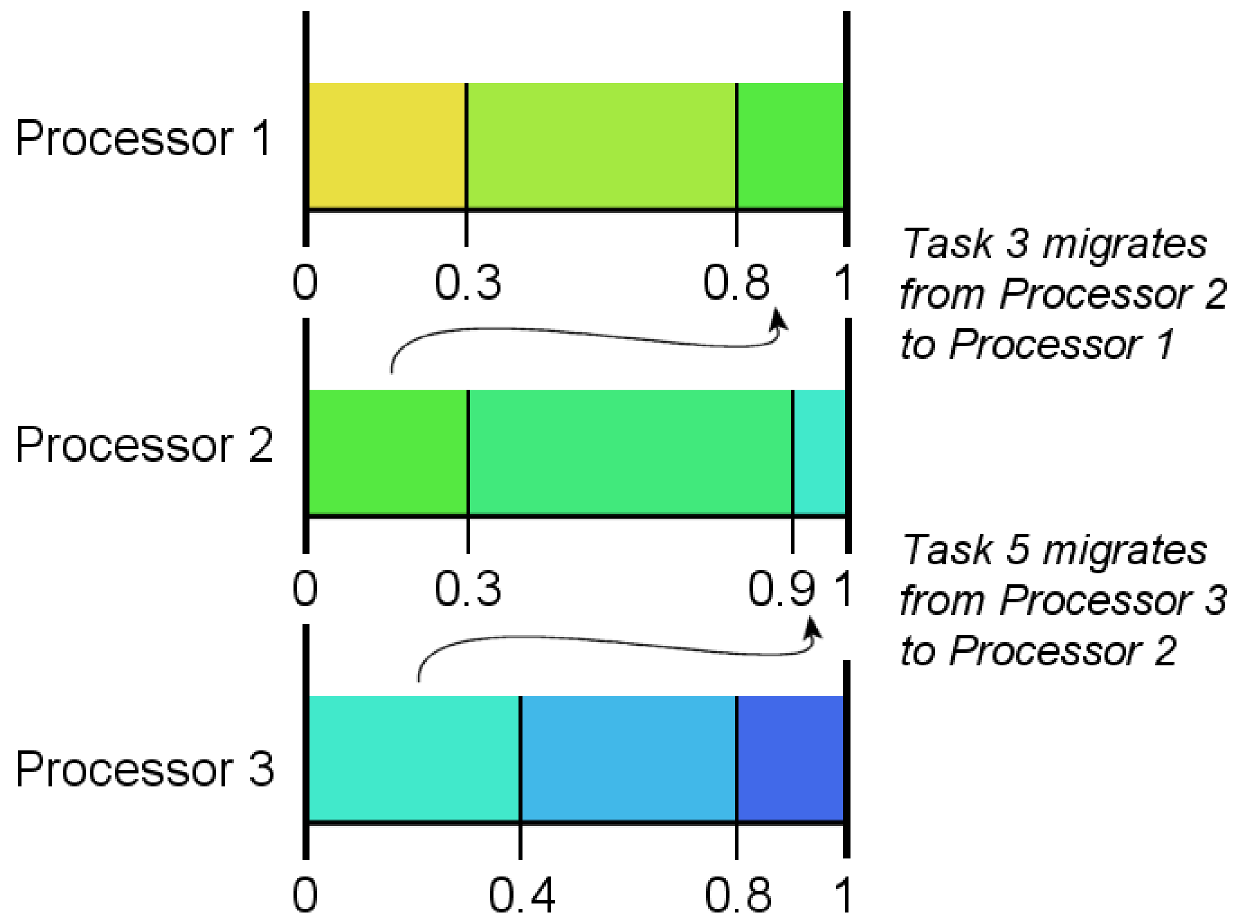


Stack-and-Slice (SNS)

- Example: 3 processors, 7 tasks



Stack-and-Slice (SNS)



SnS Performance

- $N-1$ context switches and $M-1$ migrations per time window

- This is about 1/3 as many as the LLREF algorithm, but somewhat more than EKG. However, the computational overhead is minimal compared to both.

- Multiprocessor scheduling suddenly becomes very easy when all deadlines are shared with all jobs.

- This ease is demonstrated by Stack-and-Slice, the simplest known optimal scheduling algorithm for this problem.

What's Next?

- The minimal restrictions imposed by the "3 Rules" theorem leave lots of room to develop more complicated algorithms to further reduce context switches and migrations
- How can we extend these ideas to variants of this problem?
- Can we reduce the number of operations enough to make a real implementation of such a scheduler competitive?

- Questions?