UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

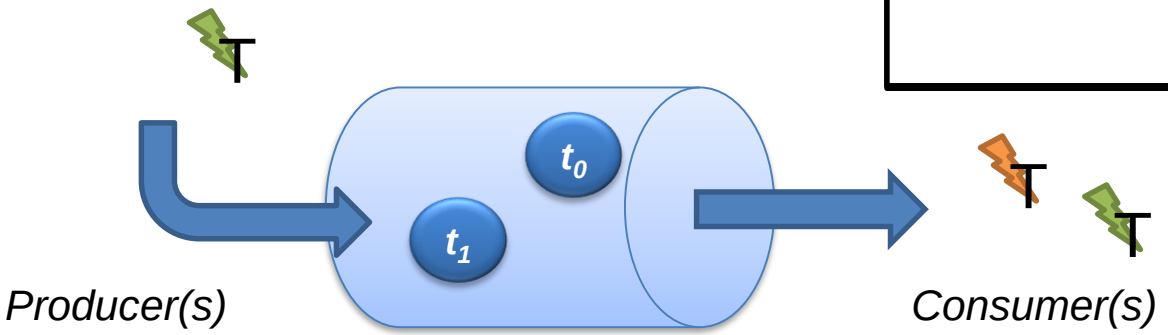
Scheduling on multi- and many-cores

Paolo Burgio
paolo.burgio@unimore.it

Recall: OpenMP tasks

- Work ~~queue~~ pool paradigm!
  - Timing de-couple
- Typically, one producer, one consumer
  - Simple
  - More manageable
- How to do this?

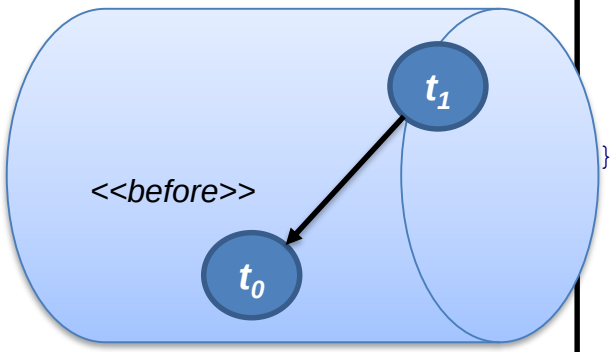
```
/* Create threads */
#pragma omp parallel num_treads(3)
{
  #pragma omp single
  {
    #pragma omp task
      t0();

    #pragma omp task
      t1();
  }

} // Implicit barrier
```



A problem

- A system with two tasks
  - t_0 created before t_1
  - t_1 must execute before t_0
  - Also on single-core system
- …sounds familiar?

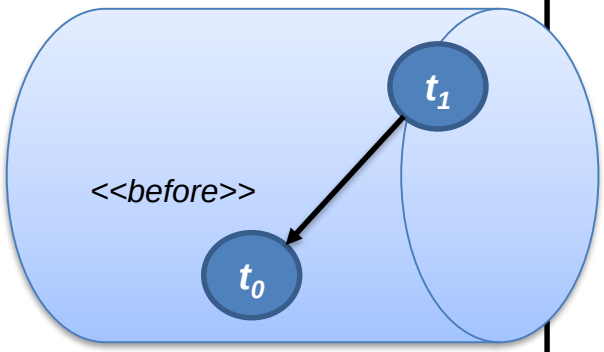
```
#pragma omp parallel
{
  #pragma omp single
  {



    #pragma omp task
    {


      t0();
    }

    #pragma omp task
    {
      t1();


    }
  } // bar&TSO
} // parreg end
```



A problem

- A system with two tasks
  - t_0 created before t_1
  - t_1 must execute before t_0
  - Also on single-core system
- …sounds familiar?

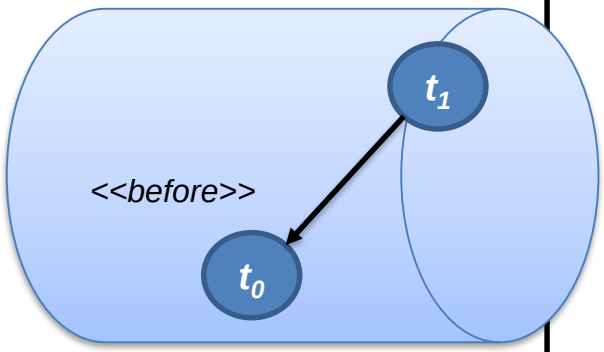
```
/* Lock var */
omp_lock_t lock;

#pragma omp parallel
{
  #pragma omp single
  {
    /* Init as "locked" */
    omp_init_lock(&lock);
    omp_set_lock(&lock);

    #pragma omp task
    {
      /* Immediately wait */
      omp_set_lock(&lock);

      t0();
    }

    #pragma omp task
    {
      t1();
      /* Done: release */
      omp_unset_lock(&lock);
    }
  } // bar&TSO
 // parreg end
```



A further problem

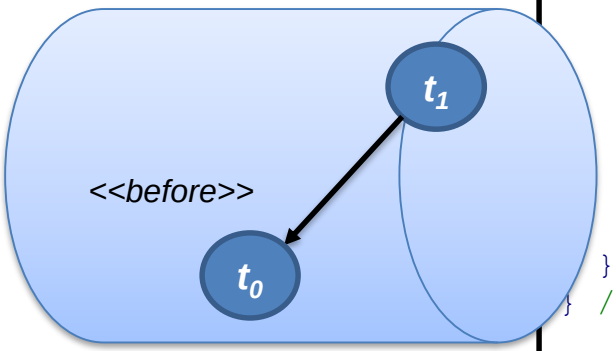
- A system with two tasks
  - t_0 created before t_1
  - t_1 must execute before t_0
  - Also on single-core system
- Thread blocked on t_0 will never work on t_1
  - Not enough parallelism in the machine

```
/* Lock var */
omp_lock_t lock;

#pragma omp parallel
{
  #pragma omp single
  {
    /* Init as "locked" */
    omp_init_lock(&lock);
    omp_set_lock(&lock);

    #pragma omp task
    {
      /* Immediately wait */
      omp_set_lock(&lock);

      t0();
    }

    #pragma omp task
    {
      t1();
      /* Done: release */
      omp_unset_lock(&lock);
    }
  } // bar&TSO
 // parreg end
```



A further problem

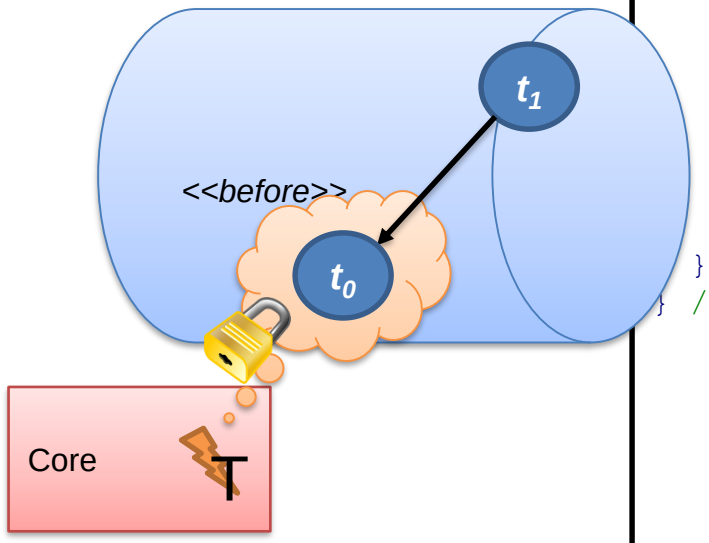
- `omp_set_lock` is blocking, but not a TSP!!
  - There can be thread switch
  - There is NO task switch!
- Thread is stuck on t0
- In single-thread systems, deadlock!




```
/* Lock var */
omp_lock_t lock;

#pragma omp parallel
{
  #pragma omp single
  {
    /* Init as "locked" */
    omp_init_lock(&lock);
    omp_set_lock(&lock);

    #pragma omp task
    {
      /* Immediately wait */
      omp_set_lock(&lock);

      t0();
    }

    #pragma omp task
    {
      t1();
      /* Done: release */
      omp_unset_lock(&lock);
    }
  } // bar&TSO
} // parreg end
```

A further problem

- `omp_set_lock` is blocking, but not a TSP!!
  - There can be thread switch
  - There is NO task switch!
- Thread is stuck on t0
- In single-thread systems, deadlock!



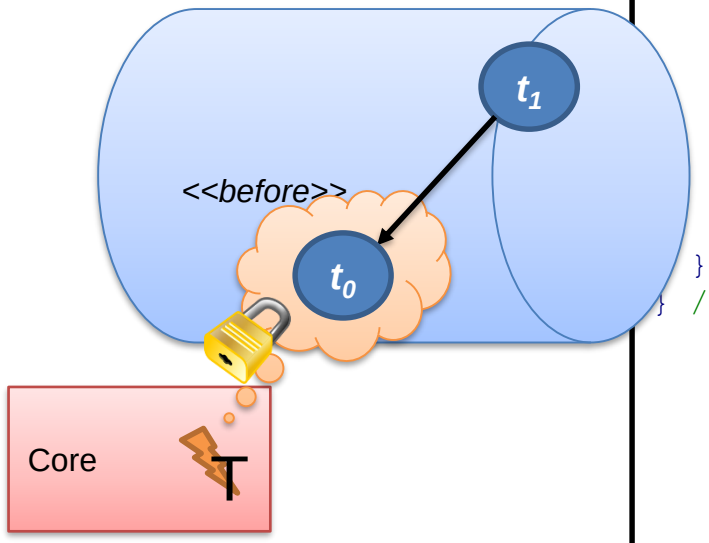

```
/* Lock var */
omp_lock_t lock;

#pragma omp parallel
{
  #pragma omp single
  {
    /* Init as "locked" */
    omp_init_lock(&lock);
    omp_set_lock(&lock);

    #pragma omp task
    {
      /* Immediately wait */
      omp_set_lock(&lock);

      t0();
    }

    #pragma omp task
    {
      t1();
      /* Done: release */
      omp_unset_lock(&lock);
    }
  } // bar&TSO
} // parreg end
```

A further problem

- ✓ `omp_set_lock` is blocking, but not a TSP!!
  - There can be thre
  - There is NO task
- ✓ Thread is stuck on t0
- ✓ In single-thread systems, deadlock!

```
/* Wait + TSP */
while(!omp_test_lock(&lock))
{
  #pragma omp taskyeld
}
```

```
/* Lock var */
omp_lock_t lock;

#pragma omp parallel
{
   single

   s "locked" */
  lock(&lock);
  ock(&lock);

   agma omp task

   /* Immediat y wait */
   omp_set_lo &lock);

   t0();
 }

 #pragma omp task
 {
   t1();
   /* Done: release */
   omp_unset_lock(&lock);
 }
} // bar&TSO
} // parreg end
```

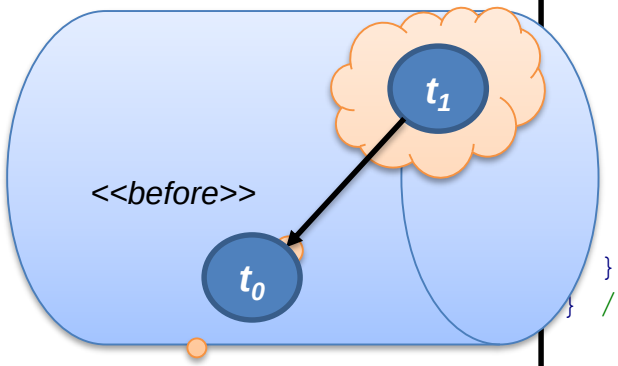


A further problem

- ✓ `omp_set_lock` is blocking, but not a TSP!!
  - There can be thre
  - There is NO task
- ✓ Thread is stuck on t0
- ✓ In single-thread systems, deadlock!

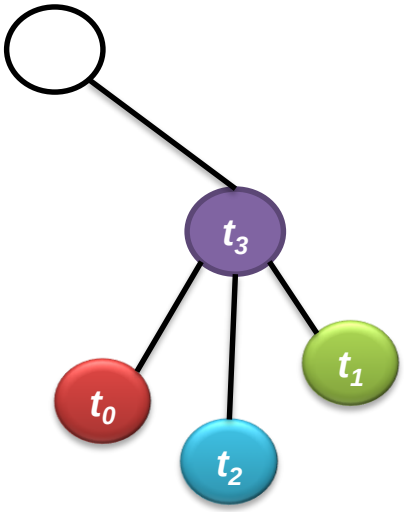
```
/* Wait + TSP */
while(!omp_test_lock(&lock))
{
  #pragma omp taskyeld
}
```

```
/* Lock var */
omp_lock_t lock;

#pragma omp parallel
{
   single

   s "locked" */
  lock(&lock);
  ock(&lock);

   agma omp task

   /* Immediat y wait */
   omp_set_lo &lock);

   t0();
 }

 #pragma omp task
 {
   t1();
   /* Done: release */
   omp_unset_lock(&lock);
 }
} // bar&TSO
} // parreg end
```

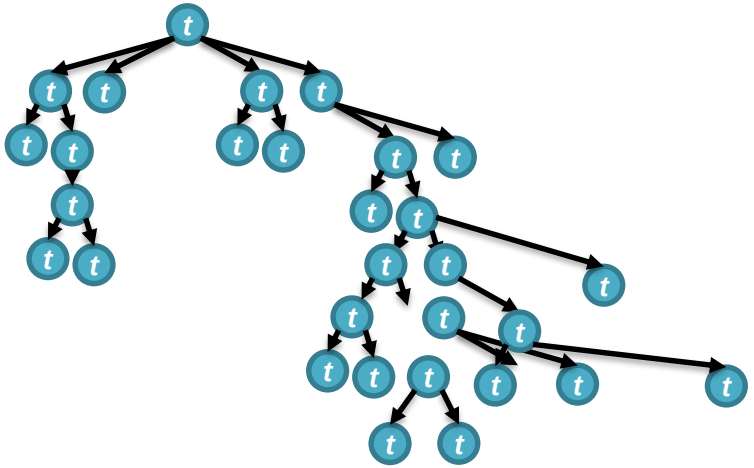


..but we want more!

- ✓ Instead of this…



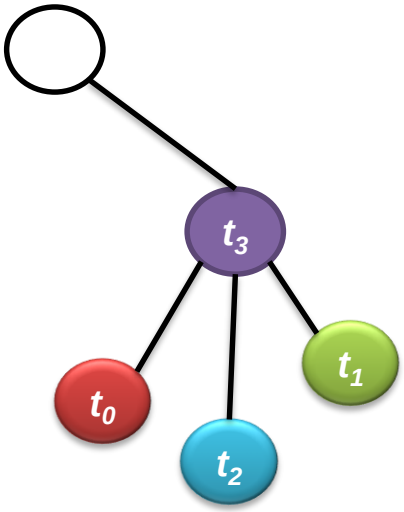
- ✓ Want to express this



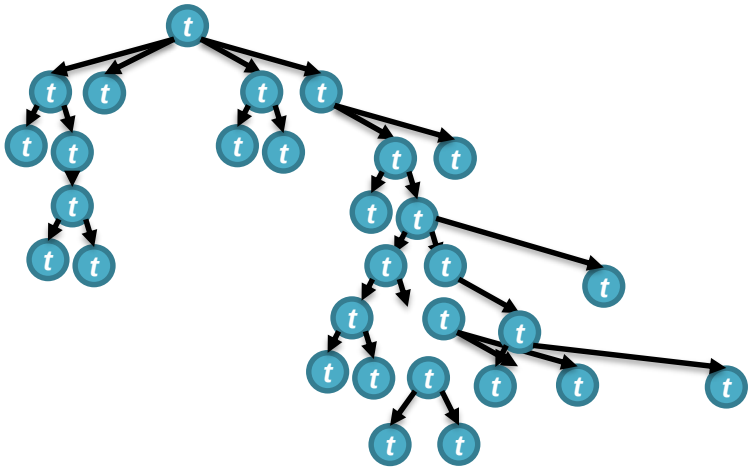
..but we want more!

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task
    {
        #pragma omp task
          t0();
        #pragma omp task
          t1();
        #pragma omp task
          t2();
        t3();
    } // end of task
} // parreg end
```

- ✓ Instead of this…



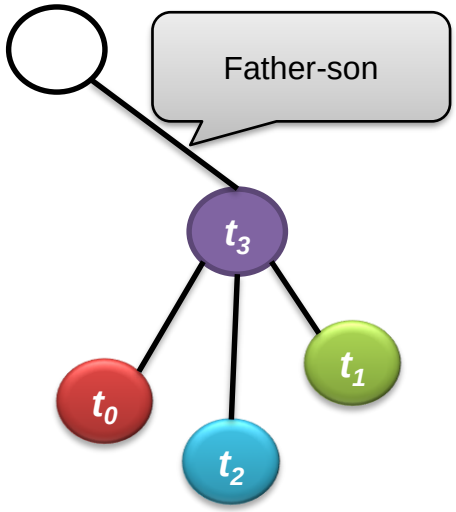
- ✓ Want to express this

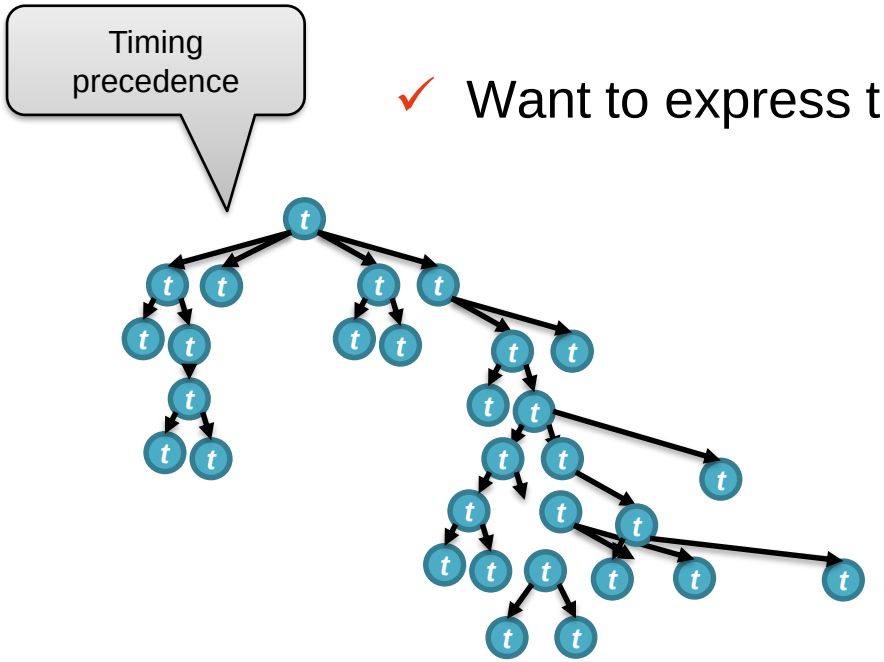


..but we want more!

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task
    {
        #pragma omp task
          t0();
        #pragma omp task
          t1();
        #pragma omp task
          t2();
        t3();
    } // end of task
} // parreg end
```

- ✓ Instead of this…





- ✓ Want to express this

The depend clause

```
#pragma omp task [clause [[,] clause]...] new-line
  structured-block
```

Where clauses can be:

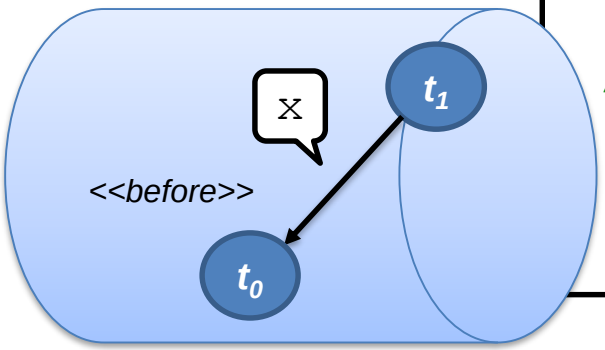
```
if([ task : ]scalar-expression)
final(scalar-expression)
untied
default(shared | none)
mergeable
private(list)
firstprivate(list)
shared(list)
depend(dependence-type : list)
priority(priority-value)
```

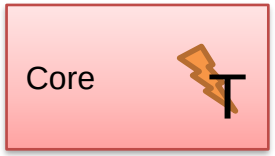
- ✓ Expresses dependencies among tasks
  - Tasks cannot proceed until the dependencies are satisfied

The depend clause

- ✓ Set a variable to act as placeholder for the dependency

```
#pragma omp parallel
{
  #pragma omp single
  {
    // Dependency is represented as a var
    int x = 0;

    #pragma omp task depend(in:x)
    {
      t0();
    }

    #pragma omp task depend(out:x)
    {
      t1();
    }
  } // bar&TSO
  // parreg end
```





OpenMP is just a language

- ✓ At the end, OpenMP can be seen as a mechanism!
  - Still, good, but let's move on!
- ✓ Use OpenMP to express complex graphs
- ✓ For the sake of completeness, i cheated a bit
  - Also an OpenMP task can be decomposed

```
#pragma omp parallel
#pragma omp single // task TO
  p00 (x=0; y=0;)
  // task T1
  #pragma omp task depend(out:x,y) { p1 }
  p01
  // task T2
  #pragma omp task depend(in:x) { P2 }
  p02
  // task T3
  #pragma omp task depend(in:y) { P3 }
  p03
```

OpenMP is just a language

- ✓ At the end, OpenMP can be seen as a mechanism!
  - Still, good, but let's move on!
- ✓ Use OpenMP to express complex graphs
- ✓ For the sake of completeness, i cheated a bit
  - Also an OpenMP task can be decomposed

```
#pragma omp parallel
#pragma omp single // task TO
 p00 (x=0; y=0;)
  // task T1
  #pragma omp task depend(out:x,y) { p1 }
 p01
  // task T2
  #pragma omp task depend(in:x) { P2 }
 p02
  // task T3
  #pragma omp task depend(in:y) { P3 }
 p03
```

task-parts

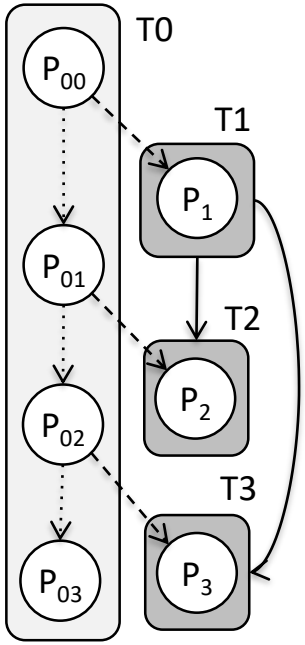
OpenMP is just a language

- ✓ At the end, OpenMP can be seen as a mechanism!
  - Still, good, but let's move on!
- ✓ Use OpenMP to express complex graphs
- ✓ For the sake of completeness, i cheated a bit
  - Also an OpenMP task can be decomposed

```
#pragma omp parallel
#pragma omp single // task TO
 p00 (x=0; y=0;)
  // task T1
  #pragma omp task depend(out:x,y) { p1 }
 p01
  // task T2
  #pragma omp task depend(in:x) { p2 }
 p02
  // task T3
  #pragma omp task depend(in:y) { p3 }
 p03
```

task-parts

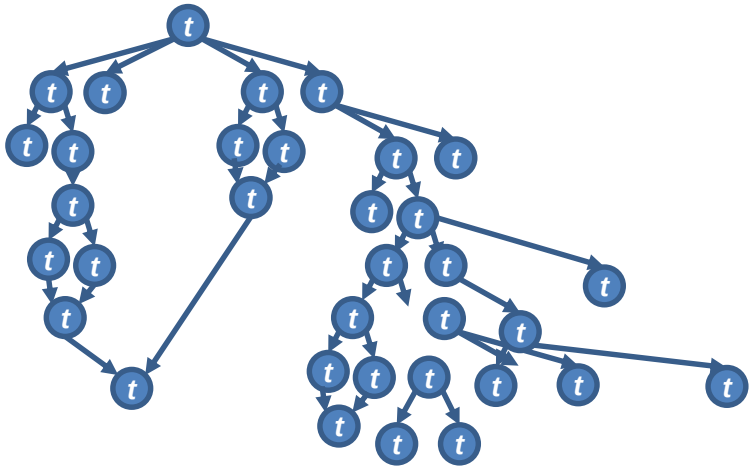
OpenMP-DAG



Example: DAGs

Directed Acyclic Graphs

- ✓ Nodes are in a parent-children relationship, no cycles (back arcs)
  - No loops!
- ✓ As opposite to, program as a "full" graph
  - Basic blocks in compiler internals
- ✓ In some situations, preferrable for representing the flow of a program
  - Today, we will see mainly these



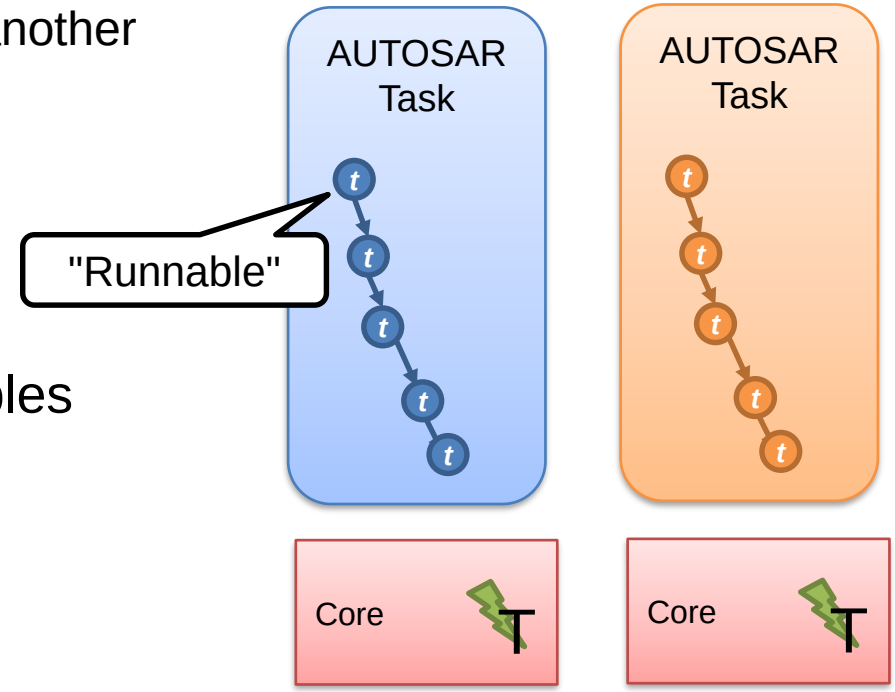
Example: AUTOSAR

- ✓ Very-well known standard for automotive
  - Clean model
  - Composability of isolated software "AUTOSAR components"
  - "Black box" approach

- ✓ AUTOSAR *tasks* are composed by *runnables*
  - Runnables are sequential one another
  - Tasks are schedulable to cores
  - A bit complex...

AUTOSAR runnable -> our "task"

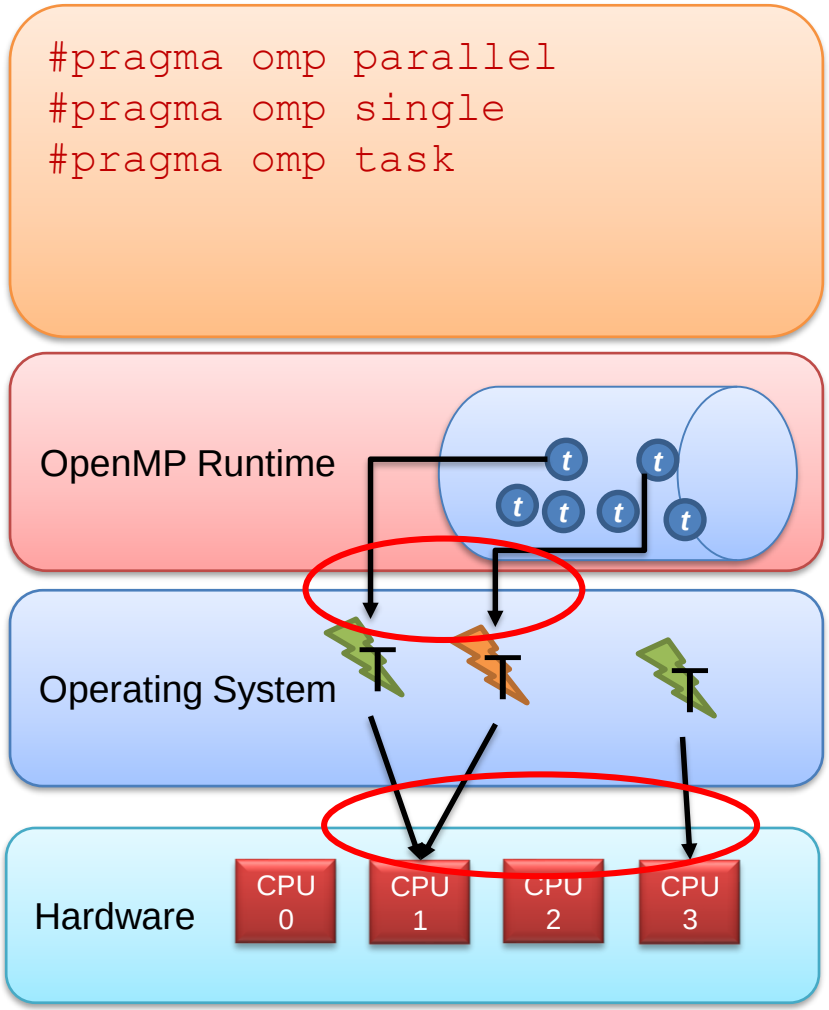
AUTOSAR task -> Group of runnables



Generic parallel stack

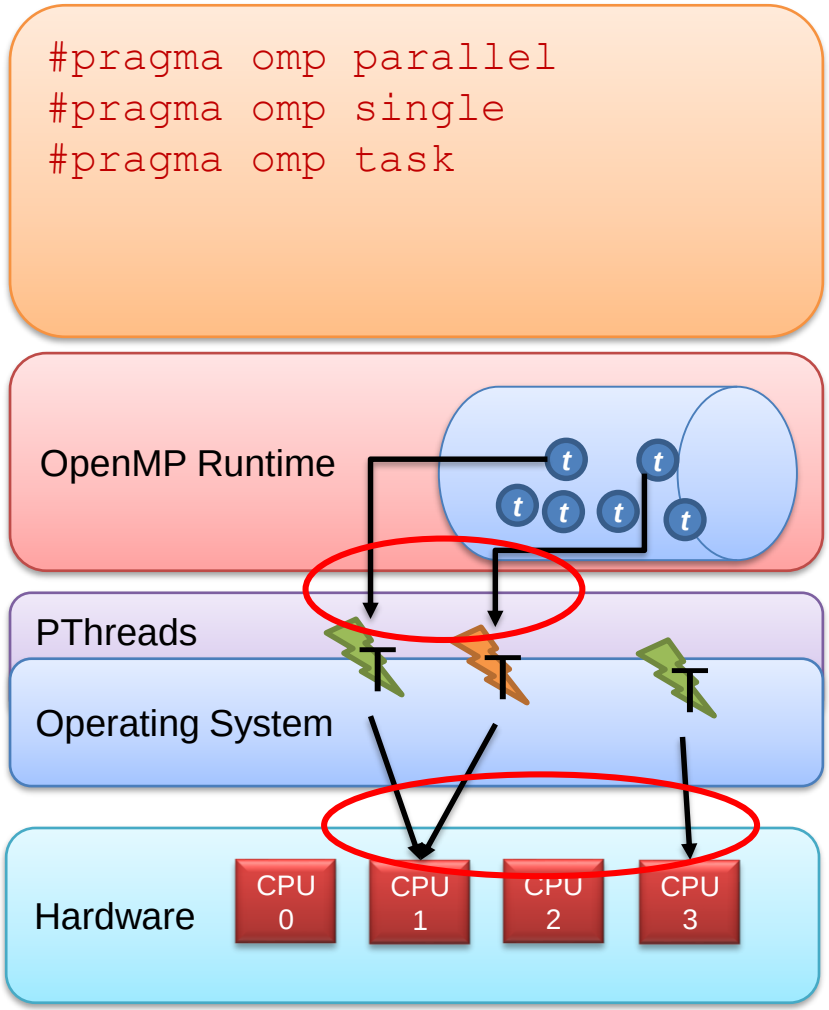
- For instance, OpenMP lets us specifying
  - Threads in parregs
  - Tasks in a parreg

- There are two scheduling levels
  - Tasks -> Threads
  - Threads -> Cores



Example #1: OMP + GNU/Linux

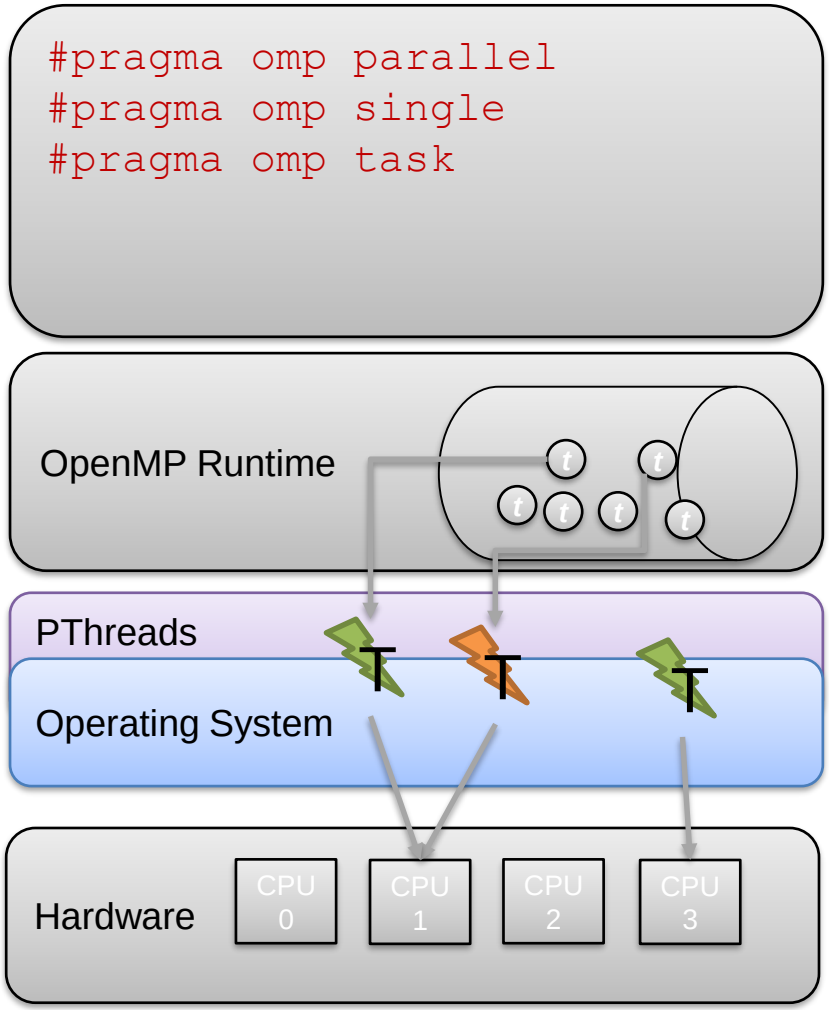
- ✓ Example of parallel stack
  - OMP runtime (e.g., GCC-OMP)
  - GNU/Linux w/Pthreads
  - 4 cores



OpenMP-to-OS threading

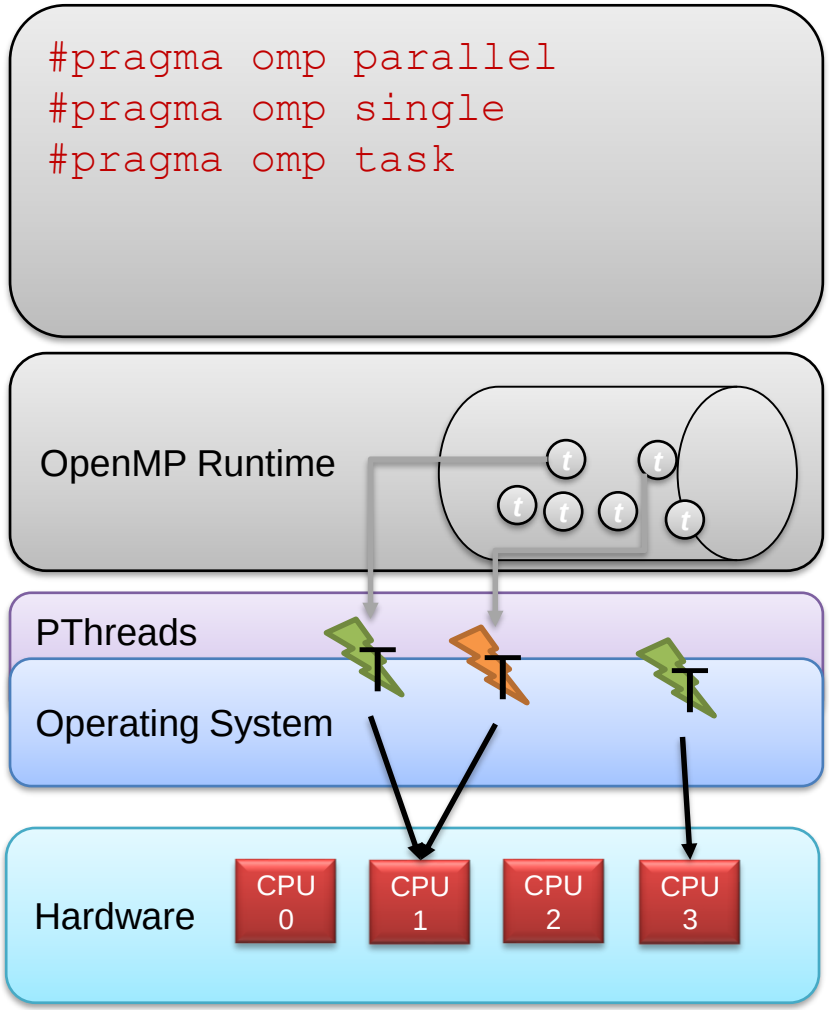
- ✓ OpenMP runtime creates "OMP threads"
  - Leveraging on OS threads
  - E.g., Pthreads
  - OMP threads <-> Pthread

Let's see this!



Thread scheduling

- ✓ OMP threads become OS threads
  - E.g., Pthreads
- ✓ Linux has his own thread scheduling policies
- ✓ Pthreads layer adds its own!
- ✓ Runtime+App have poor control!
  - Only `proc_bind` clause
  - NO notion of priority

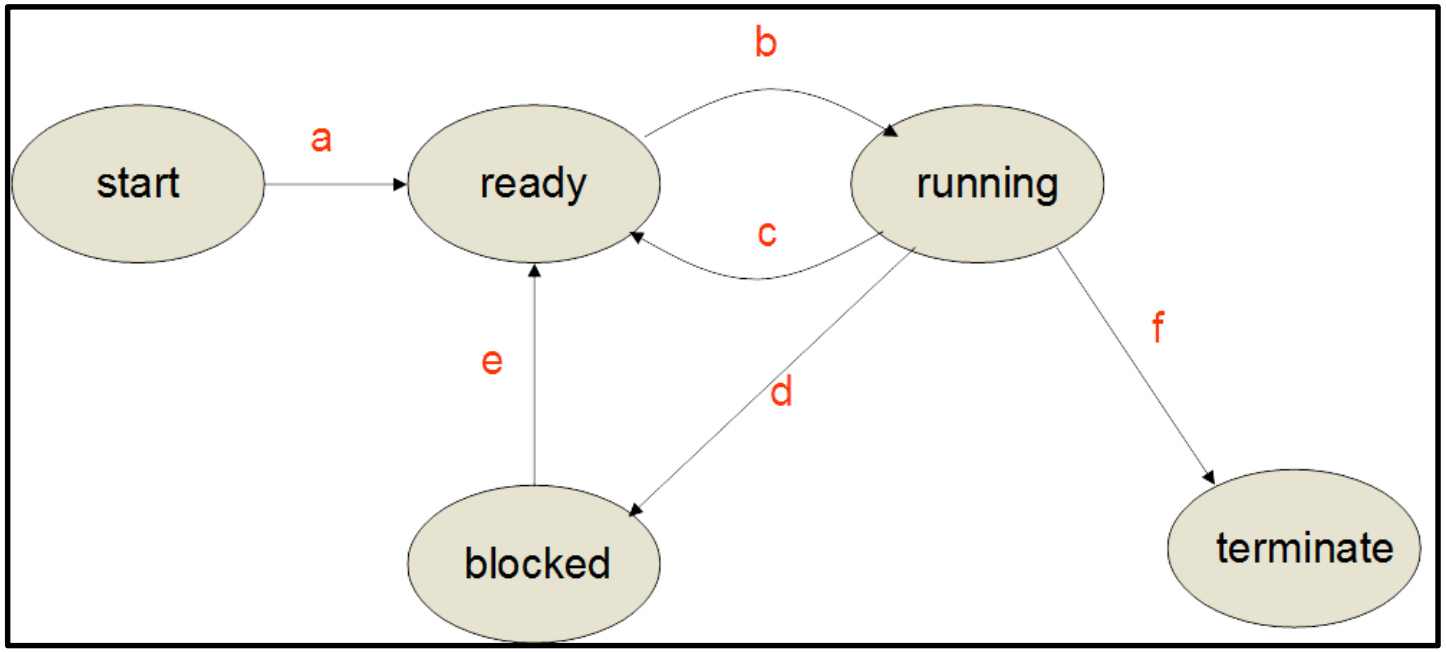


UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Thread scheduling

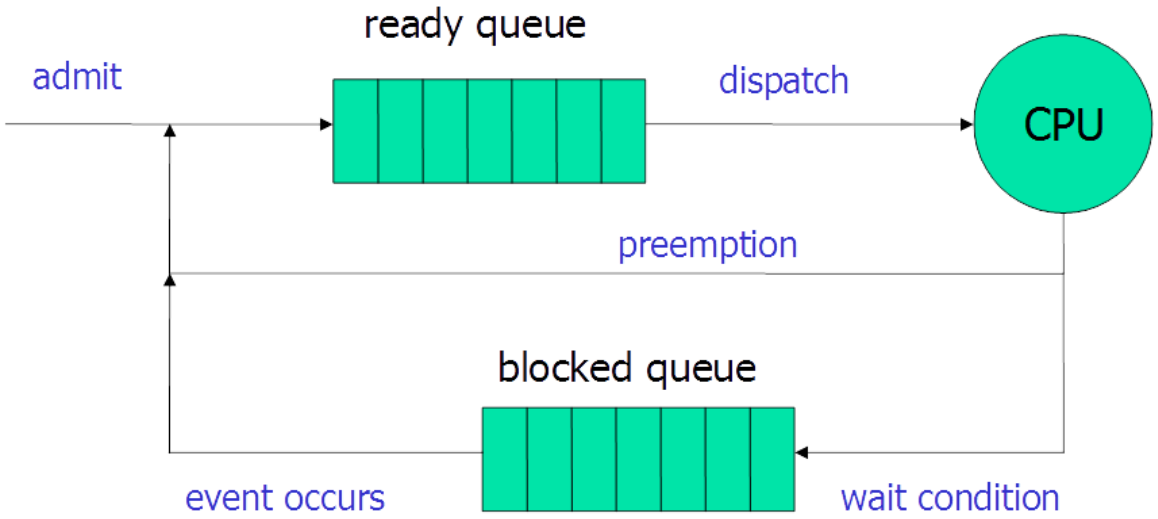
Generic thread scheduling

- Each process (and its threads) can be in one of the following states:
  - starting — being created (e.g., before `main`)
  - ready — ready to be executed)
  - executing — executing on a core
  - blocked — waiting on a condition (e.g., a lock)
  - terminating — about to terminate (e.g., after `return`)



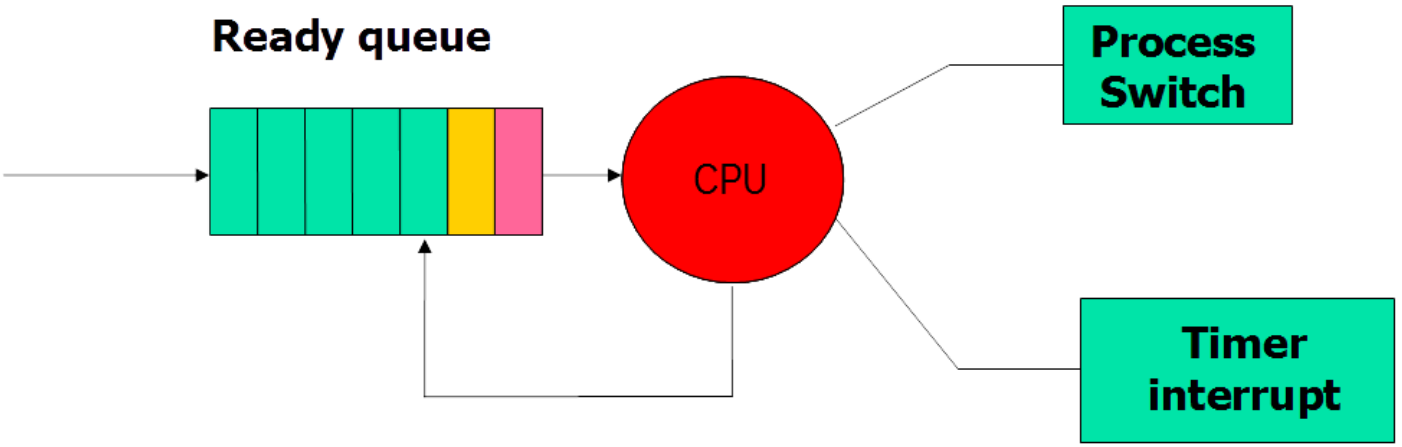
OS thread queues

- ✓ Single processor



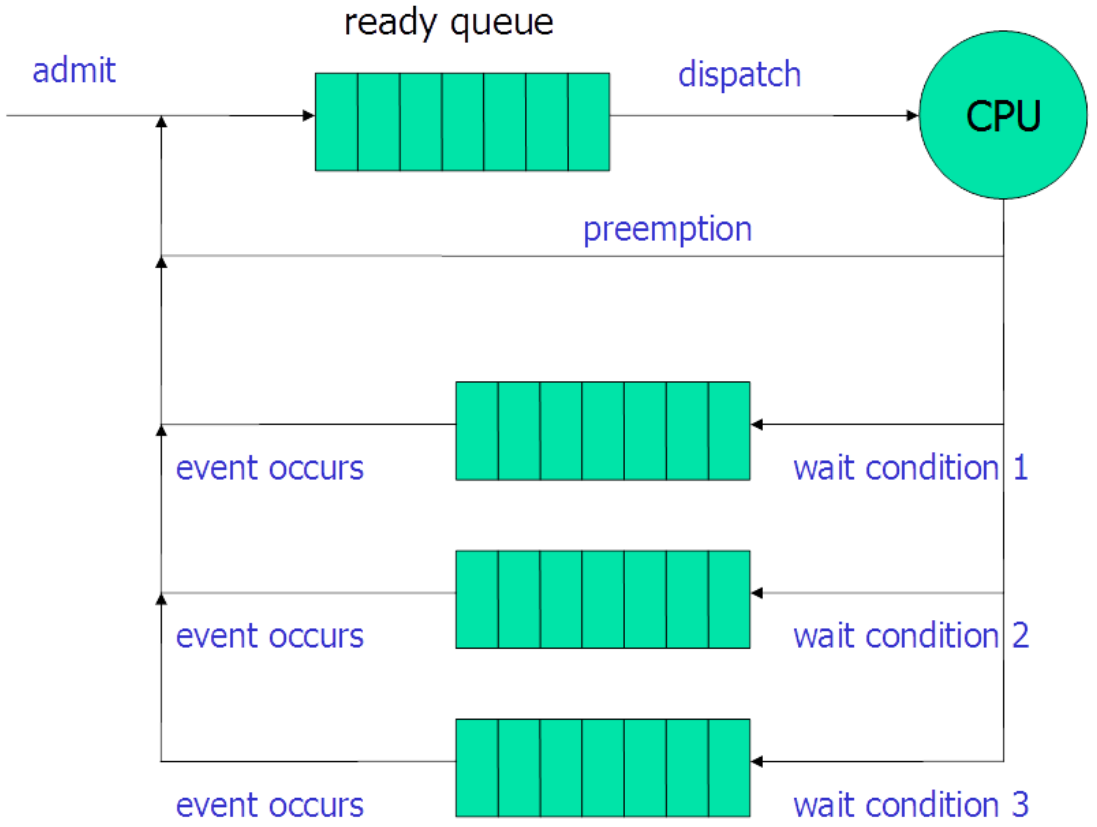
Time sharing: fairness

- ✓ Given a time T (e.g., 1 sec)
- ✓ be sure we allocate the CPU at least T/N, where N = #threads



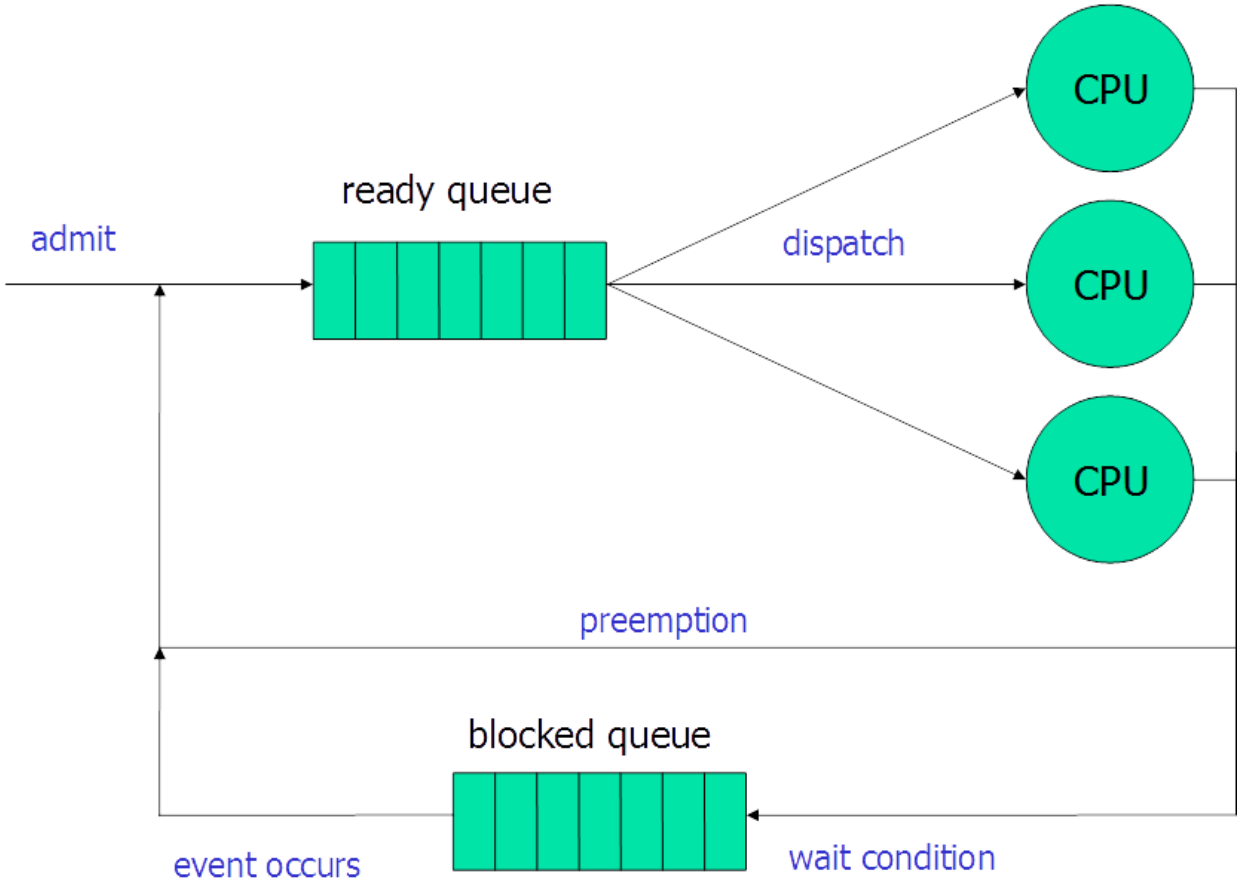
OS thread queues

- ✓ Multi-processor with multiple wait queues



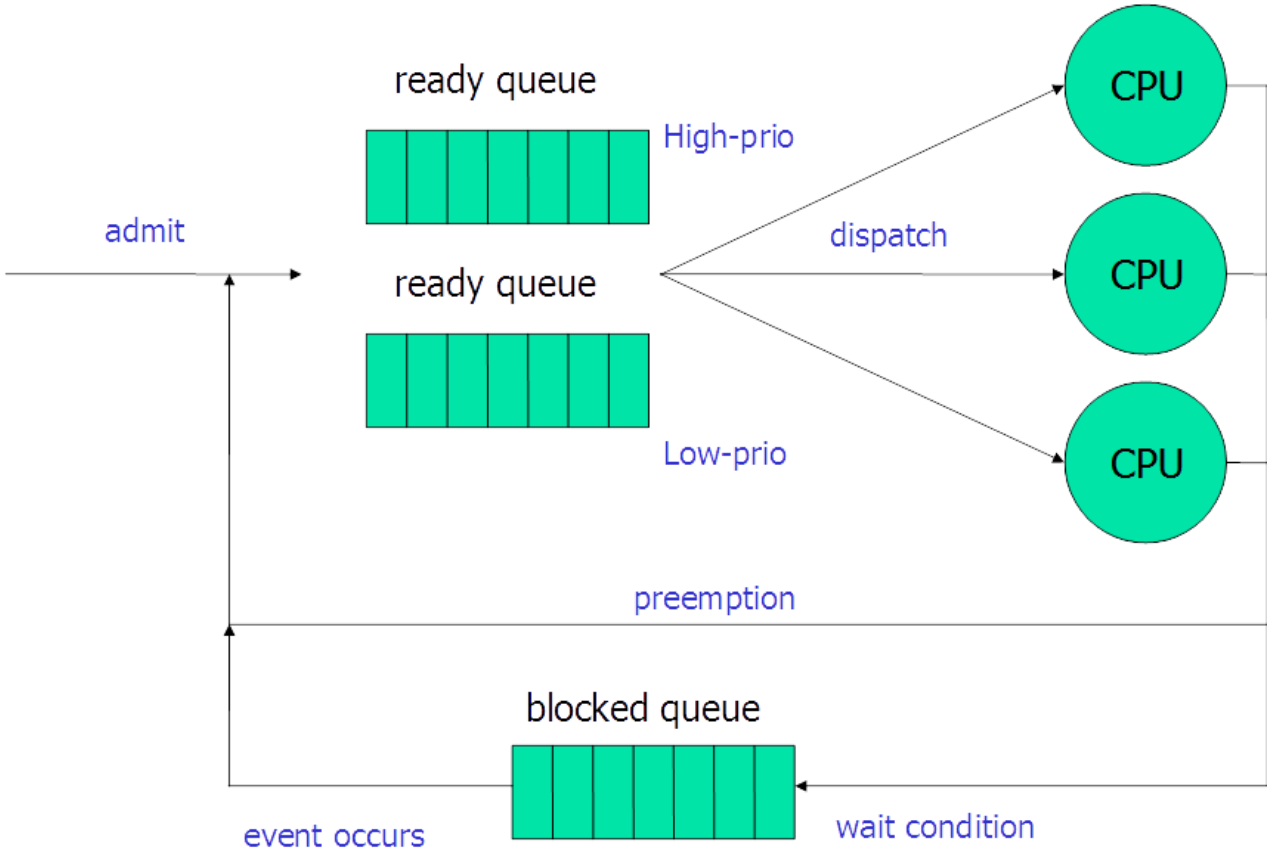
OS thread queues

- ✓ Multi-processor with migration



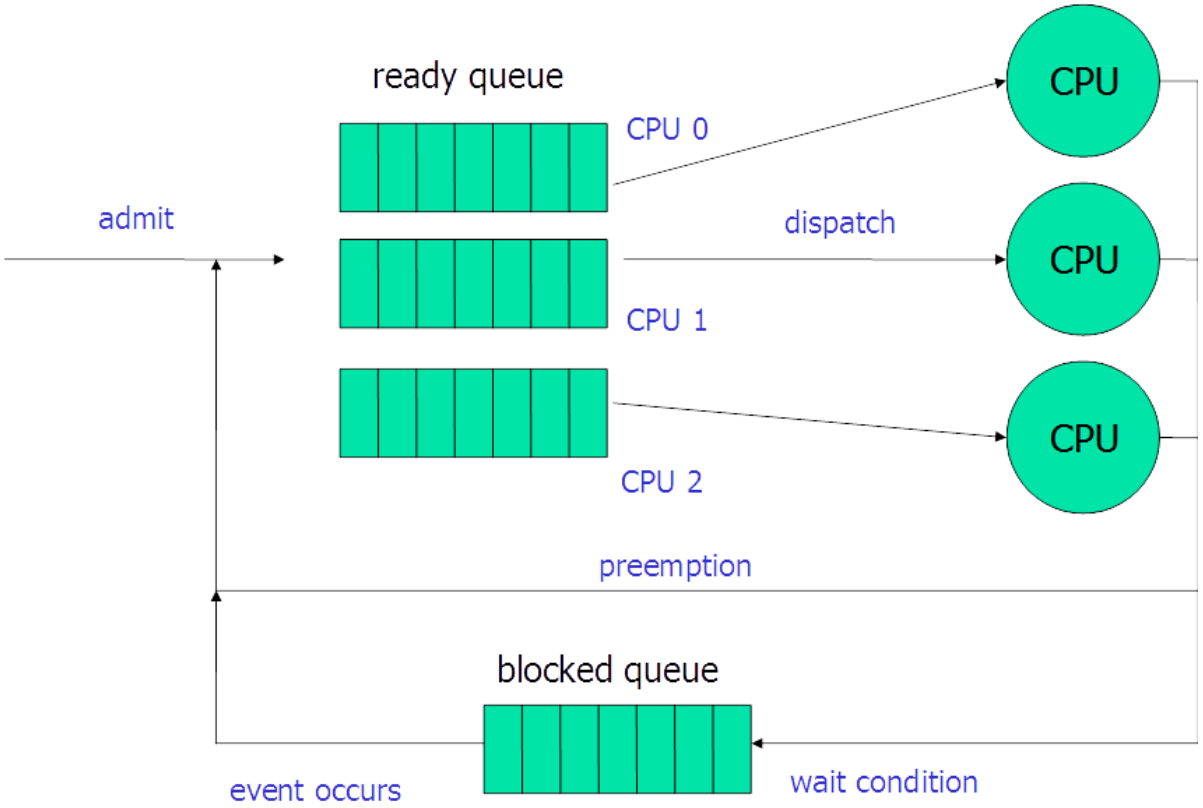
OS thread queues

- ✓ Multi-processor with priorities



OS thread queues

- ✓ Multi-processor with dedicated queues



Linux scheduling policies - recall

- ✓ `FIFO`
  - High vs. Low priority
- ✓ `Round Robin`
- ✓ `CFS`
  - User-space, non real-time
- ✓ `BFS`
  - Non-real-time
- ✓ `sched_deadline`
  - Molto promettente
  - Real-time
  - Complesso da configurare

Thanks Francesco Bellei

Pthreads scheduling

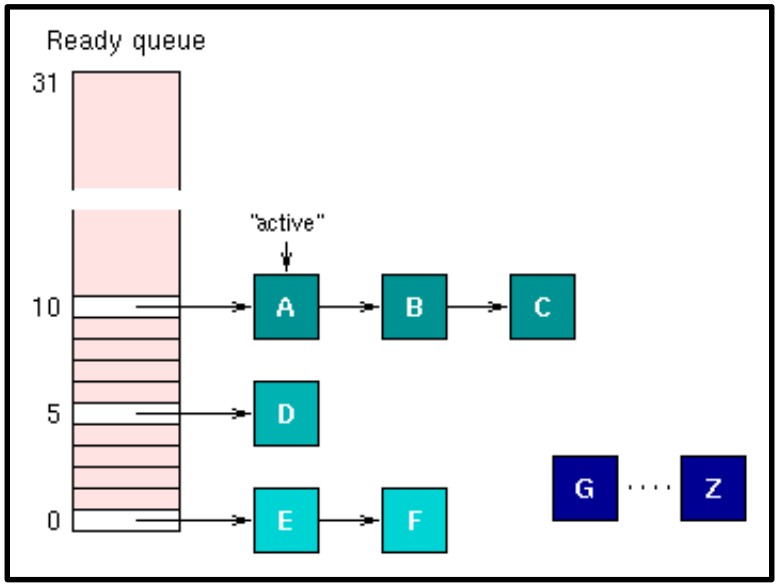
- The scheduling policy can either be SCHED_FIFO or SCHED_RR.
  - *"FIFO is a first come first serve policy. RR is a round robin policy that might preempt threads. But again, the policy only effects threads that have the same priority."*

- Realtime Process Scheduling
  - It is also possible to do realtime process scheduling. `sched_setscheduler()` is used to set the process scheduling parameters.

Priority + FIFO scheduler

Thanks Francesco Bellei

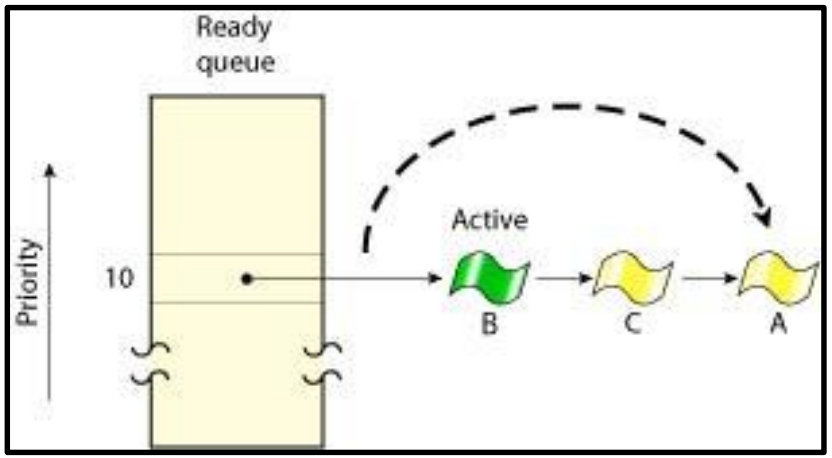
- First-in first-out (FIFO) scheduling
  - Every thread has a **priority**
  - *"When multiple threads have the same priority level, they run to completion in FIFO order."*



Priority + Round Robin scheduler

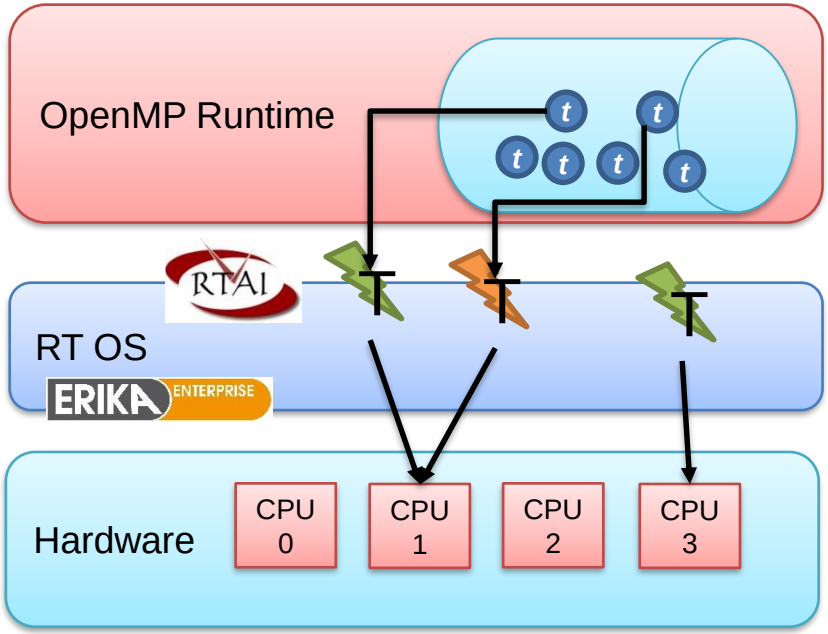
- ✓ Round-robin (RR) scheduling.
  - Every thread has a **priority**
  - ..meaning a guaranteed core BW
  - Similar to FIFO, but w/guaranteed bandwidth

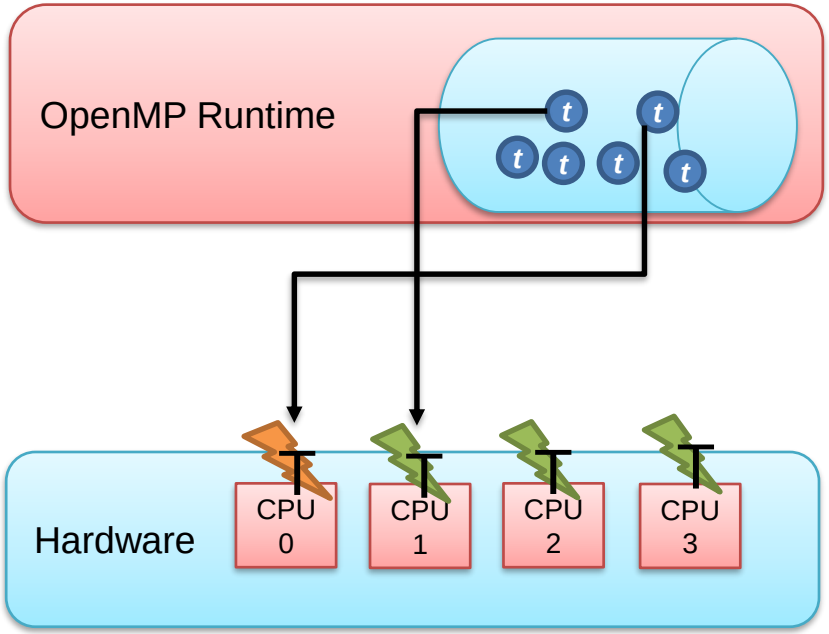
Thanks Francesco Bellei



Real-time system

- ✓ Requirements: worst-case/guaranteed performance
  - Leverage on a Real-Time OS that has advanced FIFO/Priority policies for Thread-to-core mapping
  - Or..simplest solution: remove thread-to-os mapping
    - Persistent threads





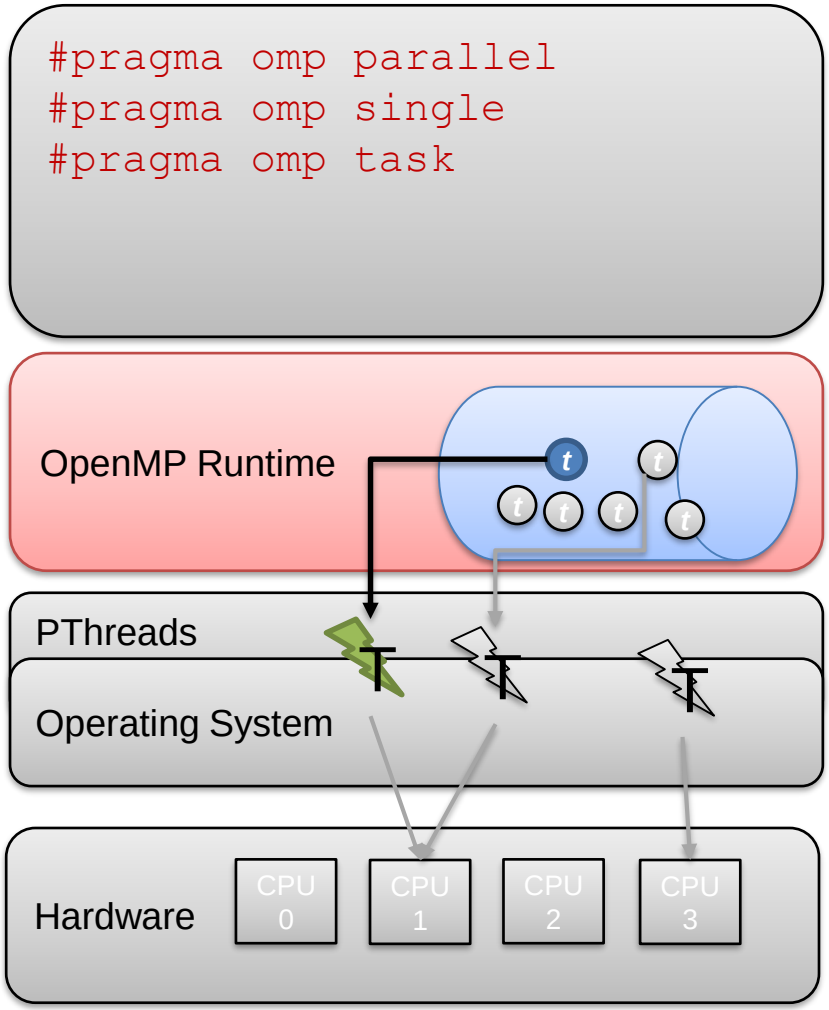
UNIMORE

UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Task scheduling on multi-/many-cores

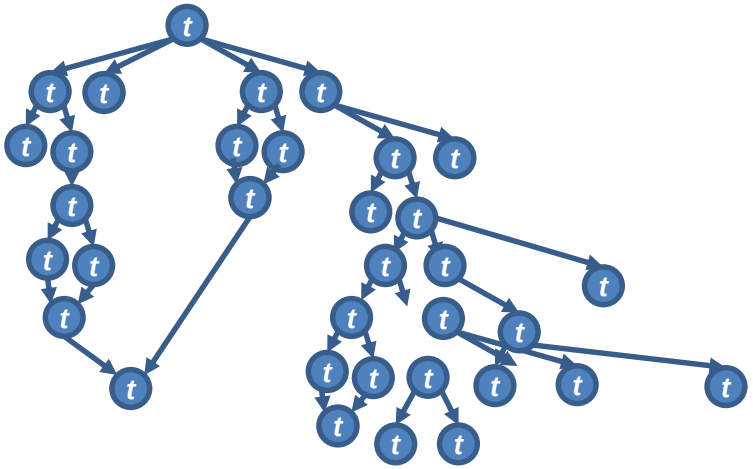
Task scheduling

- ✓ OMP runtime schedules tasks to threads
  - Typically, every idle threads requests work
  - "Enhanced" FIFO manner
  - In case of `taskwait/group`, might want to execute children first
- ✓ Scheduling policy is at Runtime level!
- ✓ Runtime also manages task dependencies



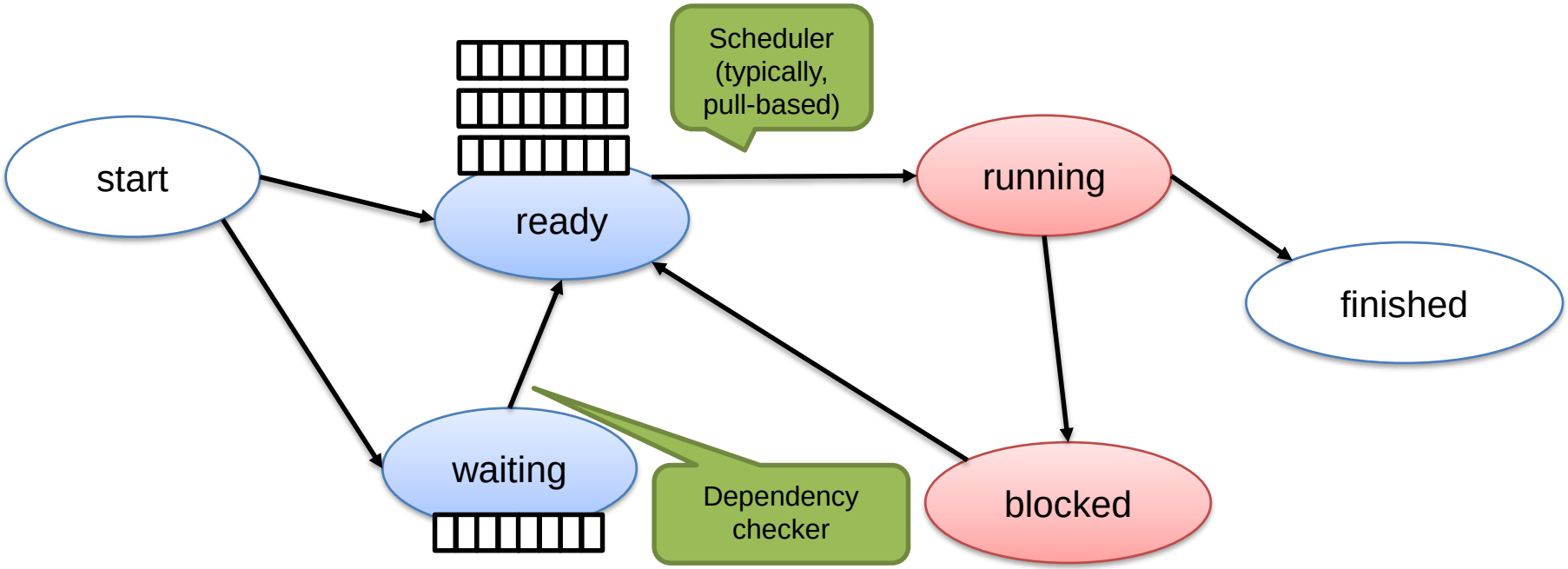
Possible implementation

- ✓ Remember that we also need to manage dependencies



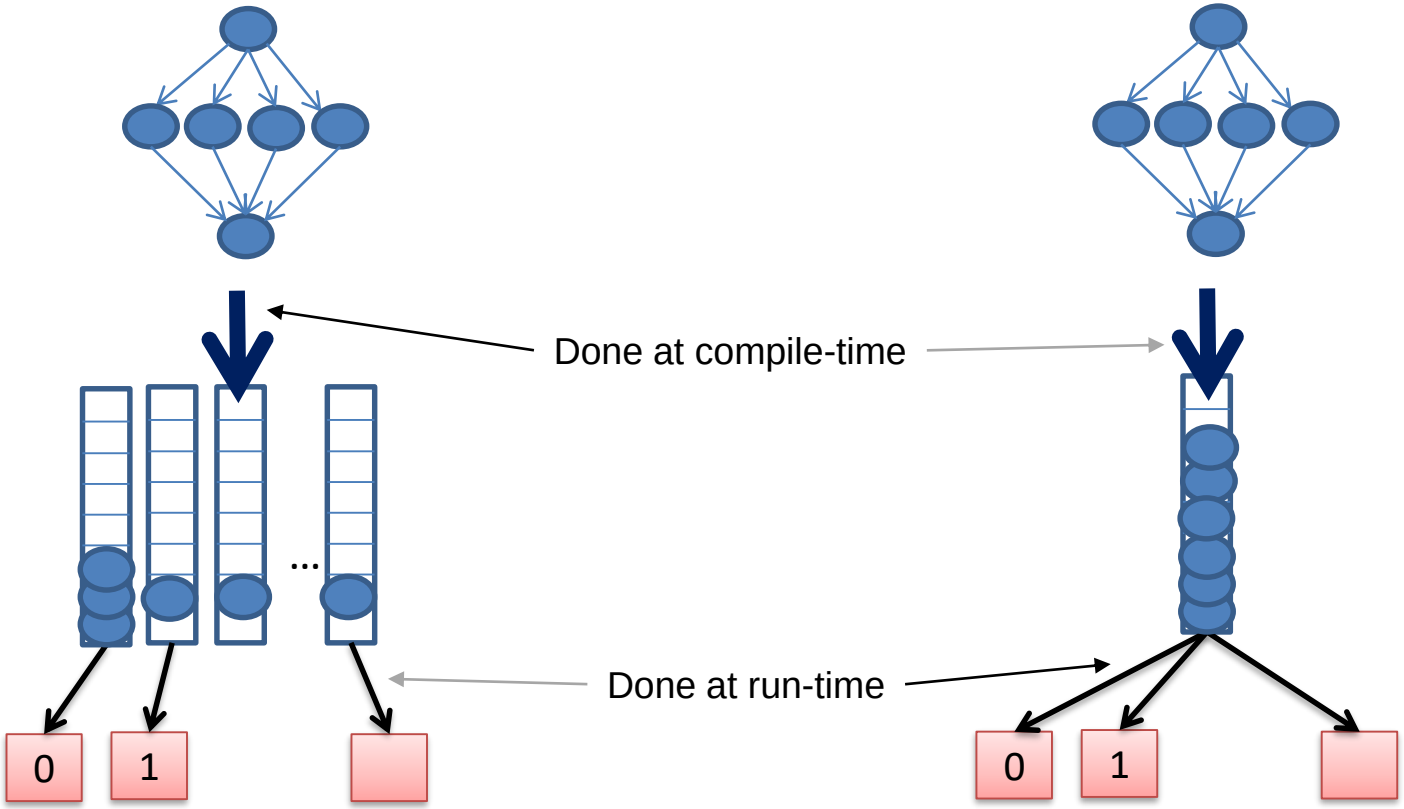
Two levels of queues

- ✓ Similar to Unix threads status
- ✓ One for the "waiting" tasks, e.g., whose dependencies has not yet been satisfied
- ✓ One (or more) for the "ready" tasks, that could potentially execute



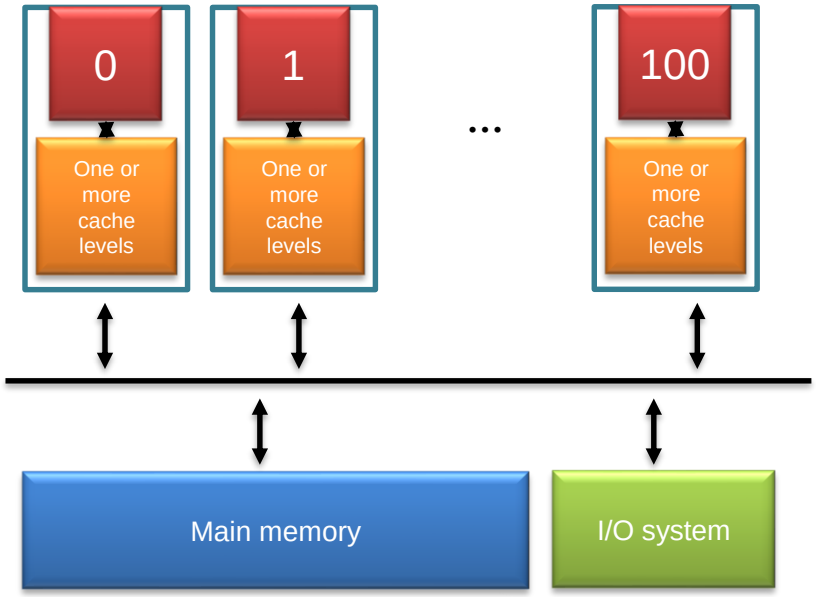
Static vs. dynamic scheduling

- ✓ Static/partitioned
  - – Guaranteed performance
  - – Worst avg performance
  - – Real-time/critical systems
- ✓ Dynamic/global
  - – Best perf. thanks to work balancing
  - – Poor guaranteed performance
  - – Linux



Scaling to many-core

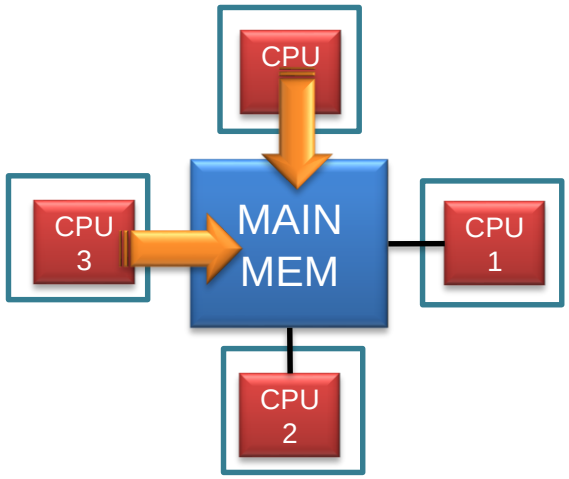
- ✓ For "physical" reasons, it is not possible to build a "flat" system made of more than 20-30 cores
- ✓ Architectural scaling *via* core clustering (e.g., GPUs)



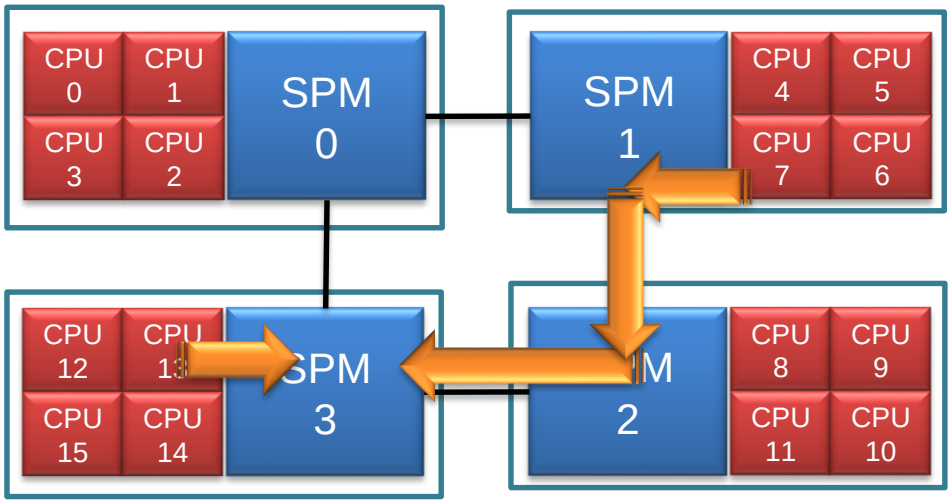
UMA vs. NUMA

- ✓ Shared mem: every thread can access every memory item
  - (Not considering security issues…)
- ✓ Uniform Memory Access (UMA) vs Non-Uniform Memory Access (NUMA)
  - Different access time for accessing different memory spaces

UMA



NUMA

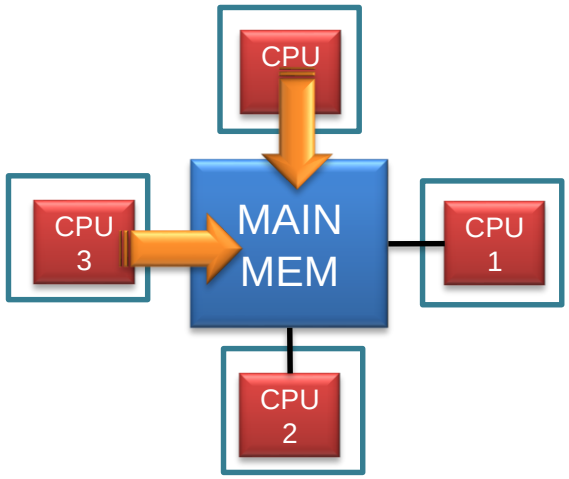


UMA vs. NUMA

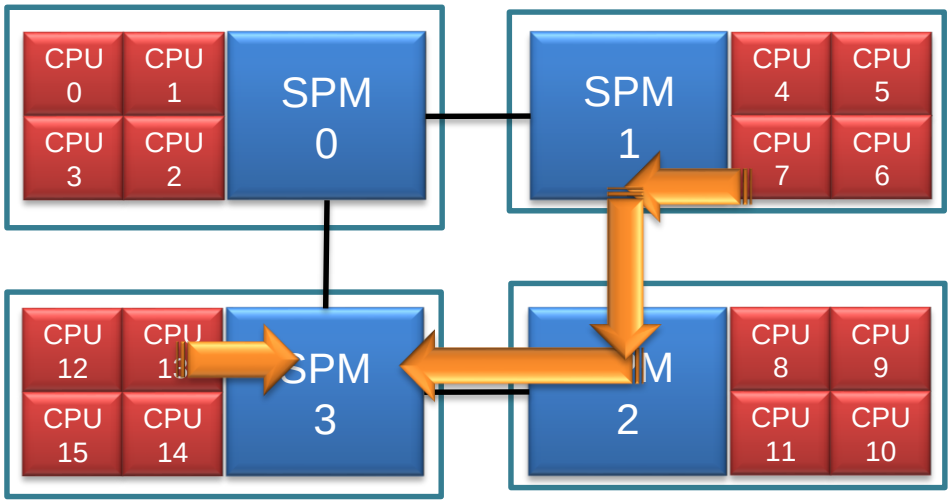
- ✓ Share
  - – (N
- ✓ Unifo ess (NUMA)
  - – Di

	MEM0	MEM1	MEM2	MEM3
CPU0…3	0 clock	10 clock	20 clock	10 clock
CPU4…7	10 clock	0 clock	10 clock	20 clock
CPU8…11	20 clock	10 clock	0 clock	10 clock
CPU12..15	10 clock	20 clock	10 clock	00 clock

UMA

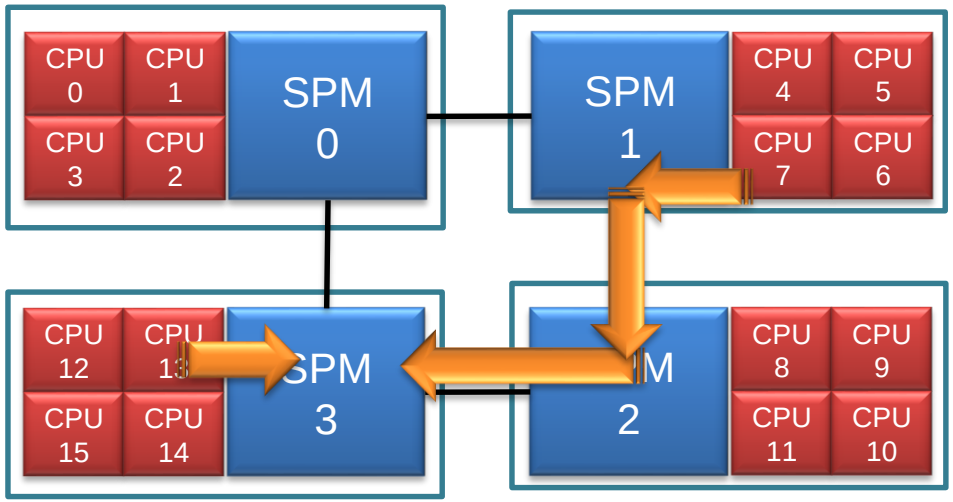


NUMA



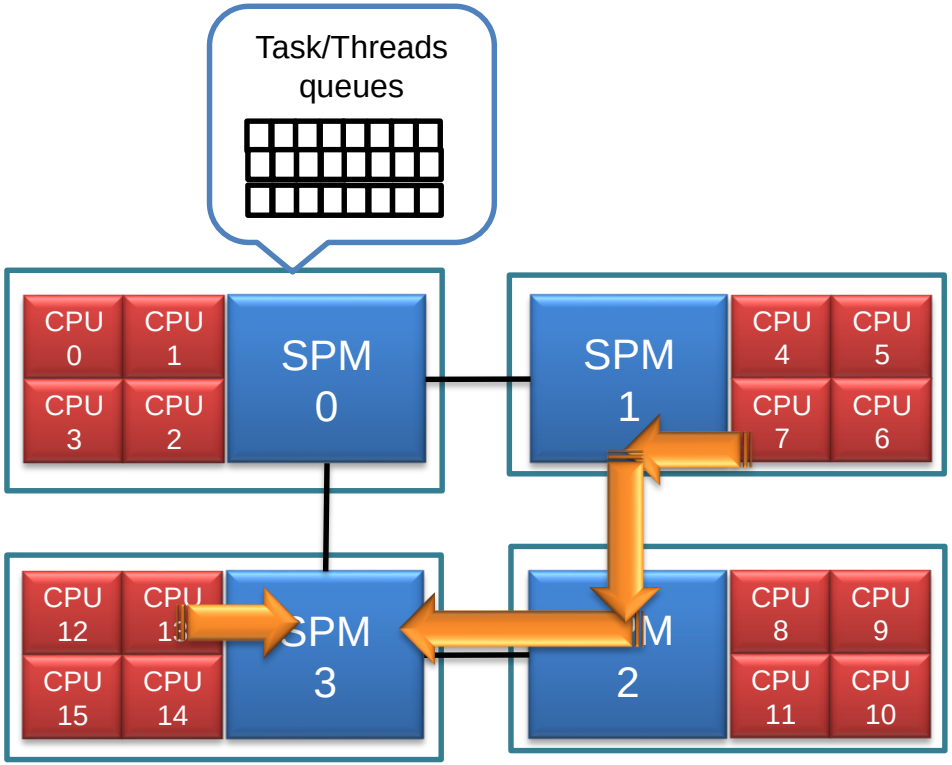
Scheduling on clustered many-cores

- NUMA architectures mean that we should schedule tasks(/threads as close as possible to data!
  - Or..the other way around: put (map) data where we know threads will be
  - Co-scheduling problem!



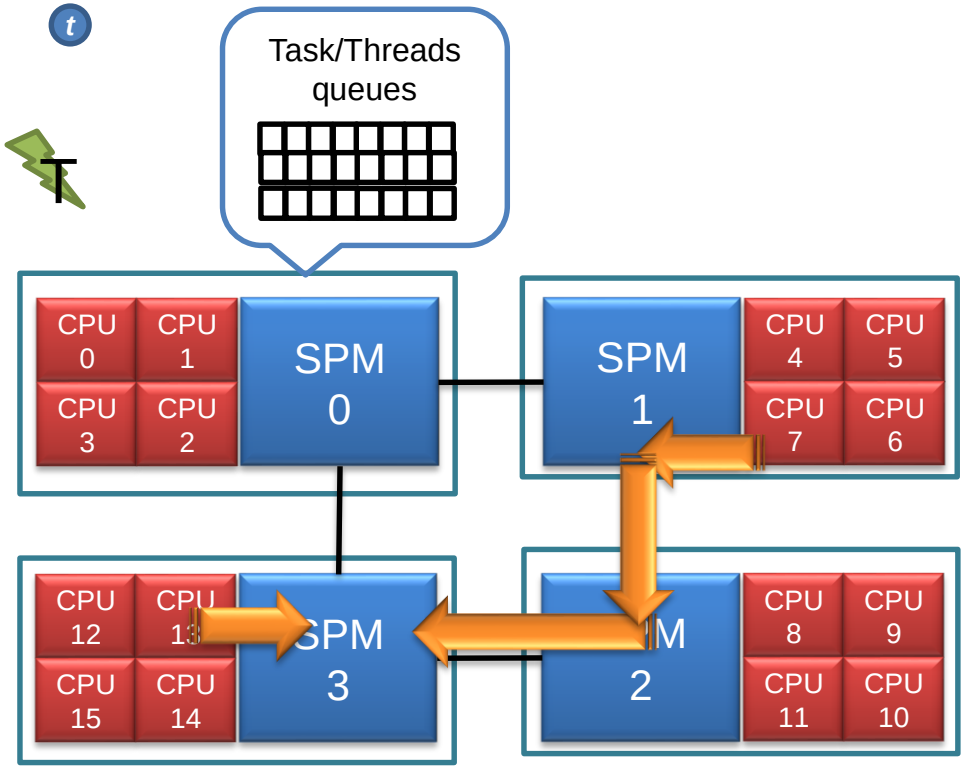
Scheduling on clustered many-cores

- NUMA architectures mean that we should schedule tasks(/threads as close as possible to data!
  - Or..the other way around: put (map) data where we know threads will be
  - Co-scheduling problem!



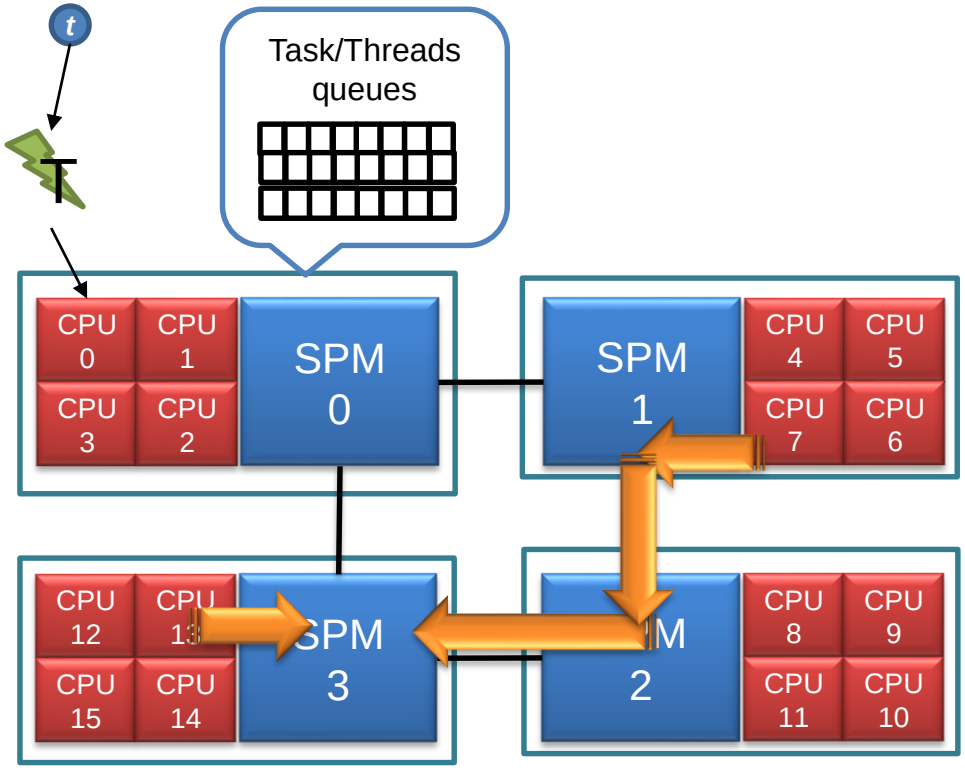
Scheduling on clustered many-cores

- ✓ NUMA architectures mean that we should schedule tasks(/threads as close as possible to data!
  - Or..the other way around: put (map) data where we know threads will be
  - Co-scheduling problem!



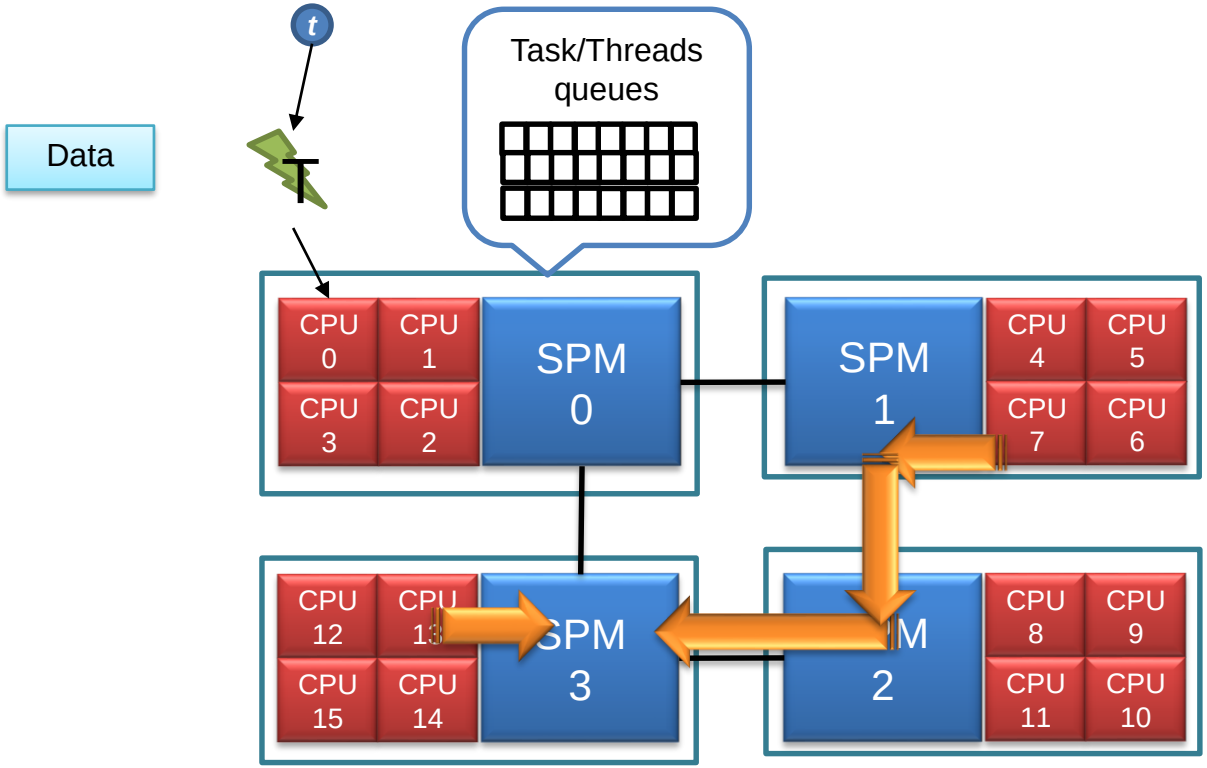
Scheduling on clustered many-cores

- NUMA architectures mean that we should schedule tasks(/threads as close as possible to data!
  - Or..the other way around: put (map) data where we know threads will be
  - Co-scheduling problem!



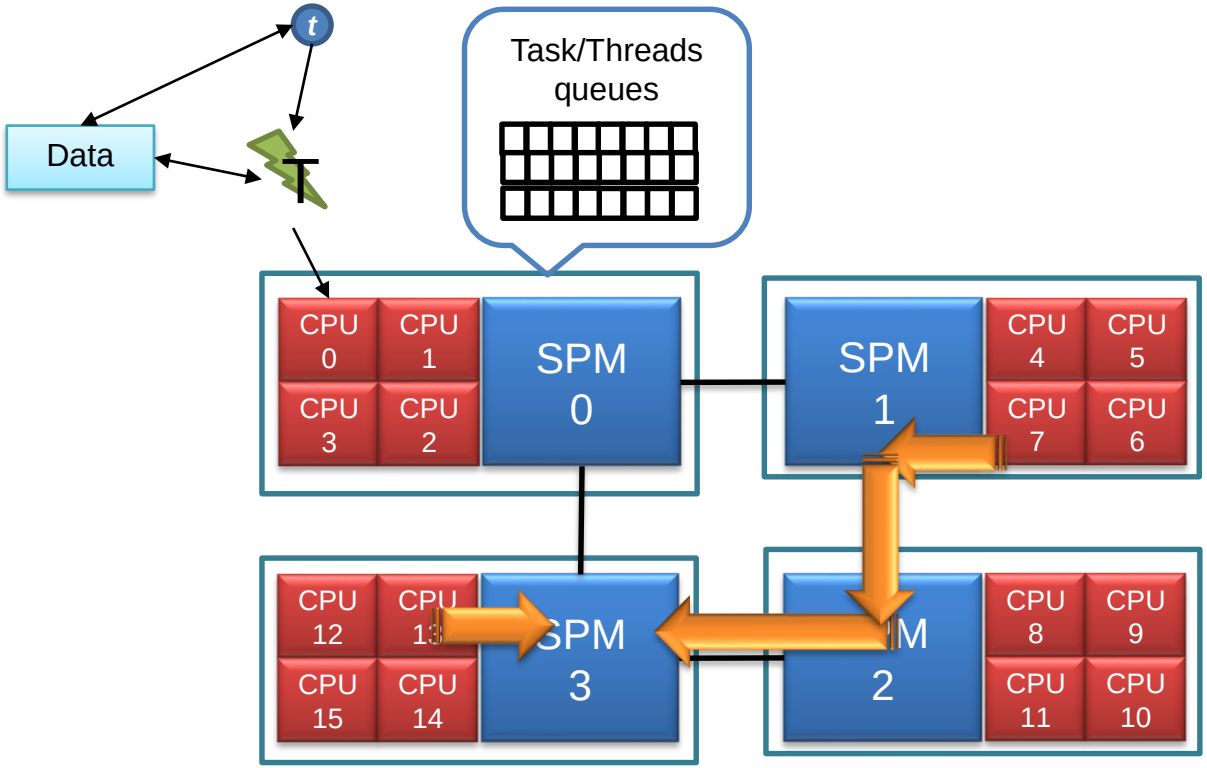
Scheduling on clustered many-cores

- NUMA architectures mean that we should schedule tasks(/threads as close as possible to data!
  - Or..the other way around: put (map) data where we know threads will be
  - Co-scheduling problem!



Scheduling on clustered many-cores

- NUMA architectures mean that we should schedule tasks(/threads as close as possible to data!
  - Or..the other way around: put (map) data where we know threads will be
  - Co-scheduling problem!



References

- "Calcolo parallelo" website
  - http://hipert.unimore.it/people/paolob/pub/Calcolo_Parallelo/
- My contacts
  - paolo.burgio@unimore.it
  - http://hipert.mat.unimore.it/people/paolob/
- OpenMP
  - http://www.openmp.org
  - https://computing.llnl.gov/tutorials/openMP/