Multiprocessor Real-Time Scheduling with **Hierarchical Processor Affinities**

Vincenzo Bonifaci
IASI–CNR

Björn Brandenburg
MPI-SWS

Gianlorenzo D'Angelo
Gran Sasso Science Institute

Alberto Marchetti-Spaccamela
Sapienza Università di Roma

This Paper

Setting

- Real-time scheduling with **restricted processor affinities**
  ➔ *each task may run only on certain processors*

This Paper

Setting

- Real-time scheduling with **restricted processor affinities**
  ➔ *each task may run only on certain processors*

Contributions

- Identify ***hierarchical (or laminar) affinities***
  ➔ *as a special case of great practical relevance*
- Non-obvious **online scheduling algorithm**
  ➔ *with **improved runtime complexity***
- Performance characterization:
  1. **speed-up** bounds for *clustered* and *bi-level* affinities
  2. prototype **implementation in LITMUS^{RT}** and **overhead evaluation** on 24-core Xeon multicore platform

Background

Processor Affinity

- interface to ***restrict the set of processors*** on which a task may be scheduled
- widely available in multiprocessor (real-time) OSs

Linux: **`sched_setaffinity()`**

FreeBSD: **`cpuset_setaffinity()`**

Windows: **`SetThreadAffinityMask()`**

QNX: **`ThreadCtl(_NTO_TCTL_RUNMASK)`**

VxWorks: **`taskCpuAffinitySet()`**

Arbitrary Processor Affinity (APA) Scheduling (*Gujarati et al., 2013*)

- first analysis of processor affinity in real-time systems
- the usual sporadic task model: $\boldsymbol{C_i, D_i, T_i}$
- set of (identical) processors $\boldsymbol{\Pi}_1\ldots\boldsymbol{\Pi}_m$
- plus an ***arbitrary per-task affinity set***

$$\alpha_i \subseteq \{\Pi_1, \ldots, \Pi_m\}$$

Strong vs. Weak APA Scheduling
(*Gujarati et al., 2014*)

weak APA invariant

a job is **backlogged** only if all processors in its affinity execute jobs of **equal or higher priority**

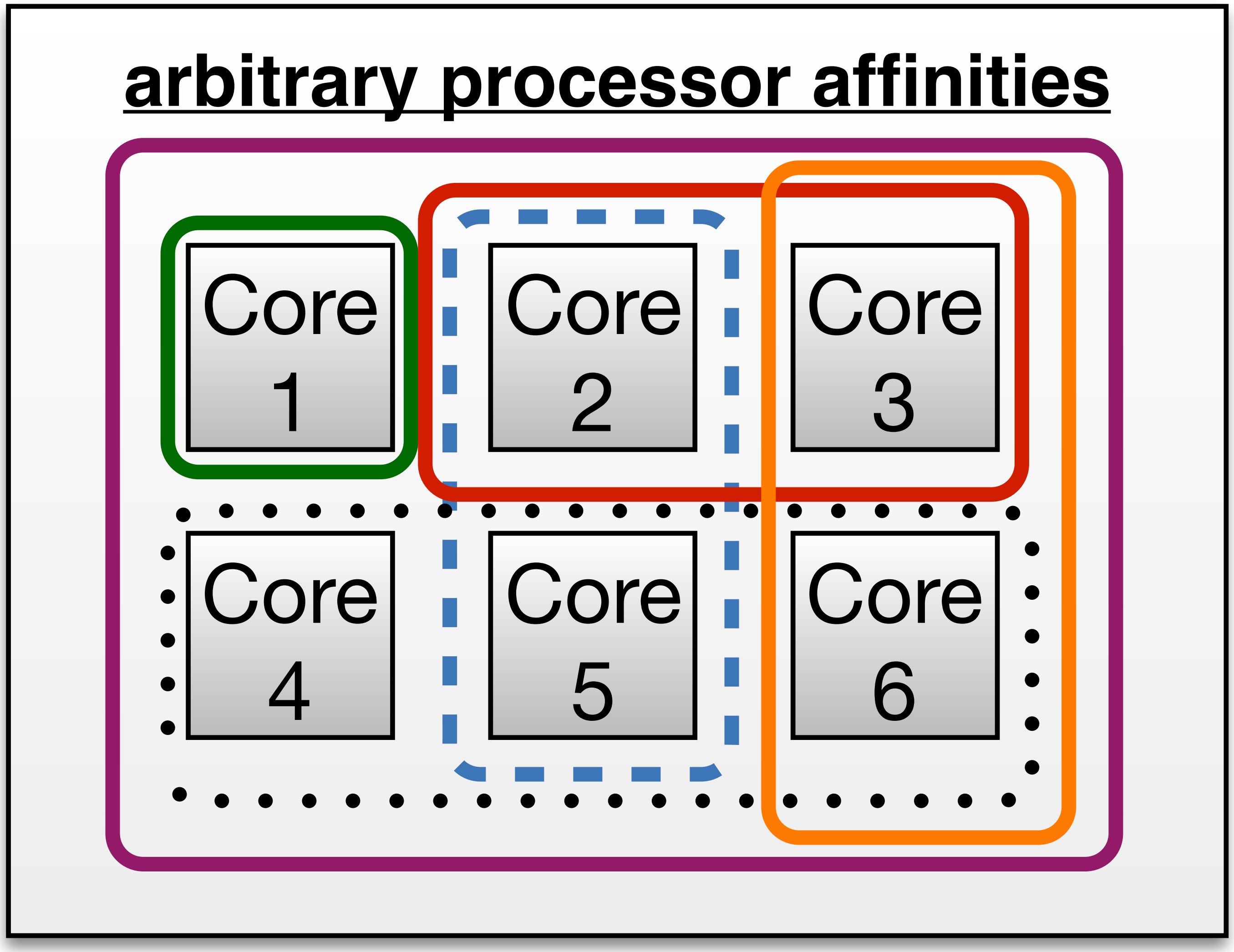
strong APA invariant

weak invariant + **no way to "re-arrange" higher-priority jobs** to free up a core for a backlogged job

- Linux, QNX, etc.
- easier to implement

- better schedulability
- this paper

Arbitrary Affinities: Difficult Scheduling Problem



- difficult to analyze
- difficult to schedule at runtime

Basic Operations

*Job **Arrival**: preemption necessary?*

- for **each core** in affinity, check if new job can be placed
- **weak APA**: only by preempting lower-priority tasks
- **strong APA**: also by ***shifting*** higher-priority tasks to **other cores**

→ $O(m^2)$

n…number of tasks

m…number of cores

Basic Operations

*Job **Arrival**: preemption necessary?*

- for **each core** in affinity, check if new job can be placed
- **weak APA**: only by preempting lower-priority tasks
- **strong APA**: also by ***shifting*** higher-priority tasks to **other cores**

→ $O(m^2)$

*Job **Departure**: schedule backlogged job?*

- for **each backlogged job**, check if freed processor can be used
- **weak APA**: only if freed processor is in affinity set
- **strong APA**: also by ***shifting*** higher-priority tasks to **other cores**

→ $O(nm)$

n…number of tasks

m…number of cores

Prior Strong APA Scheduling Results

	Strong APA (*Gujarati et al., 2014*)
Job arrival cost	$O(m^2)$
Job departure cost	$O(nm)$
Speed-up bound	—
Implemented in OS?	—
Schedulability test	sufficient

Difficult to improve the general case. (combinatorial structure)

But what if we rule out pathological combinations?

n…number of tasks

m…number of cores

Hierarchical Processor Affinities (HPA)

Arbitrary Processor Affinities?

Why do users typically restrict processor affinities?

Arbitrary Processor Affinities?

Why do users typically restrict processor affinities?

- **cache affinity**: e.g., stay on same core / pair of cores / socket to maintain L1 / L2 / L3 affinity, respectively

Arbitrary Processor Affinities?

Why do users typically restrict processor affinities?

- **cache affinity**: e.g., stay on same core / pair of cores / socket to maintain L1 / L2 / L3 affinity, respectively
- **interrupt throughput**: e.g., execute network-facing daemon on same core that processes network interrupts

Arbitrary Processor Affinities?

Why do users typically restrict processor affinities?

- **cache affinity**: e.g., stay on same core / pair of cores / socket to maintain L1 / L2 / L3 affinity, respectively
- **interrupt throughput**: e.g., execute network-facing daemon on same core that processes network interrupts
- **interrupt isolation**: e.g., execute only on half of cores that do not handle device interrupts

Arbitrary Processor Affinities?

Why do users typically restrict processor affinities?

- **cache affinity**: e.g., stay on same core / pair of cores / socket to maintain L1 / L2 / L3 affinity, respectively
- **interrupt throughput**: e.g., execute network-facing daemon on same core that processes network interrupts
- **interrupt isolation**: e.g., execute only on half of cores that do not handle device interrupts
- **security isolation**: e.g., avoid micro-architectural timing channels by forcing sensitive and less trusted tasks to run on separate cores

Arbitrary Processor Affinities?

Why do users typically restrict processor affinities?

- **cache affinity**: e.g., stay on same core / pair of cores / socket to maintain L1 / L2 / L3 affinity, respectively
- **interrupt throughput**: e.g., execute network-facing daemon on same core that processes network interrupts
- **interrupt isolation**: e.g., execute only on half of cores that do not handle device interrupts
- **security isolation**: e.g., avoid micro-architectural timing channels by forcing sensitive and less trusted tasks to run on separate cores

All resulting affinities naturally exhibit structure.
They are ***not completely arbitrary!***

Natural Affinity Structure

- **Goal: isolation**
  ➔ system sliced into differently sized "compartments"
  ➔ affinities do not overlap (**complete exclusion**)

- **Goal: cache affinity**
  ➔ affinities reflect **memory hierarchy**
  ➔ smaller affinities part of larger affinities (**full inclusion**)

- **Goal: sequencing of tasks (partial partitioning)**
  ➔ singleton affinities

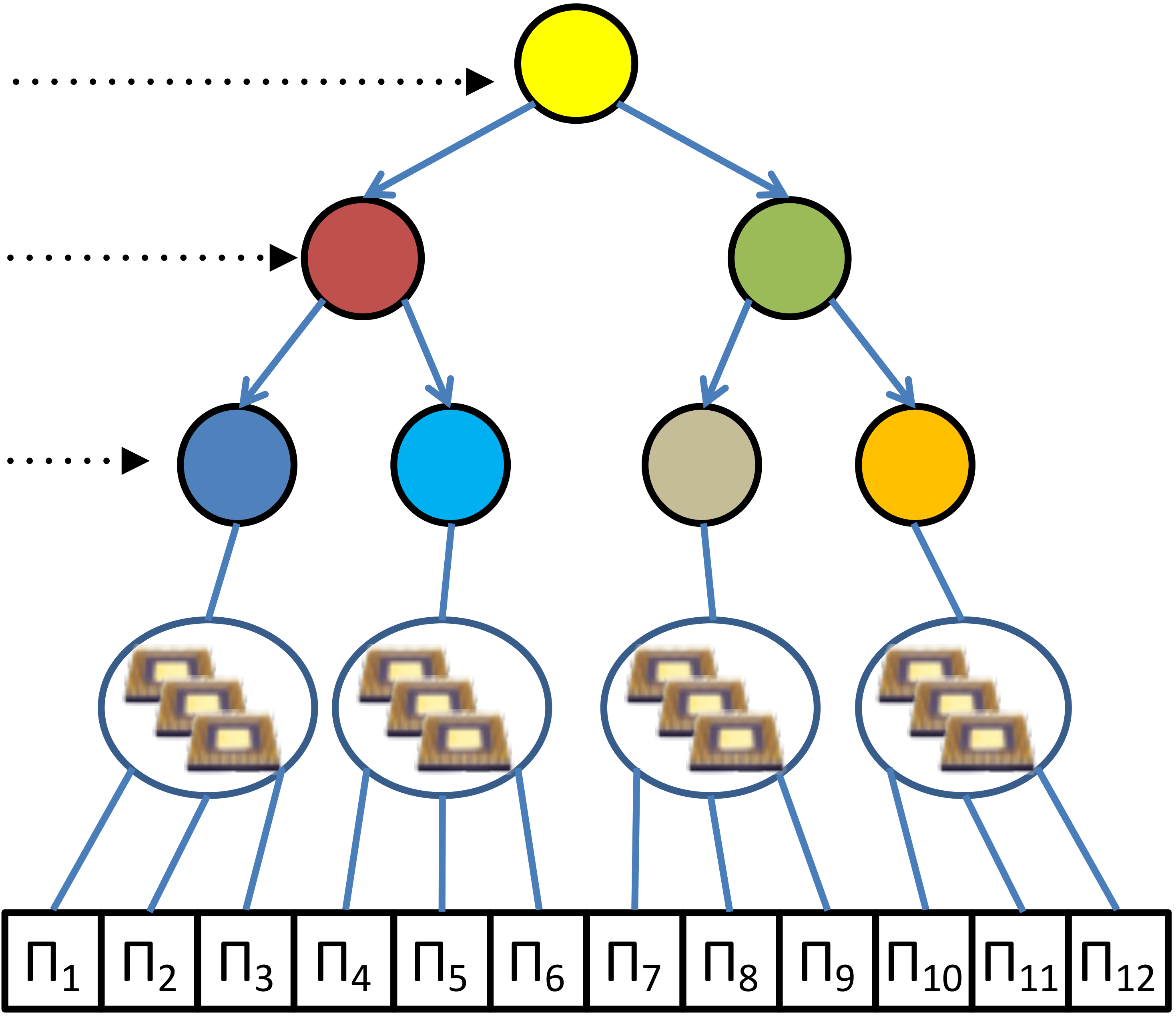
- **Goal: average-case response-time improvements**
  ➔ global (or at least very large) affinities

Hierarchical (or Laminar) Processor Affinities (HPA)

- **Laminar family** of affinity sets (*tree-like structure*)
- For any two jobs *i* and *j*, either:

$\alpha_i \subseteq \alpha_j$ or $\alpha_j \subseteq \alpha_i$ or $\alpha_j \cap \alpha_i = \emptyset$

Example HPA Inclusion Tree



all cores
(global affinity)

two sockets
(e.g., shared L3)

half of socket
(e.g., shared L2)

Overview of Results

	Strong APA (*Gujarati et al., 2014*)	**Strong HPA** (*this paper*)
Job arrival cost	$O(m^2)$	$O(m)$
Job departure cost	$O(nm)$	$O(\log n + m^2)$
Speed-up bound	—	2.415 (bi-level + EDF) 3.562 (clustered + EDF)
Implemented in OS?	—	LITMUS^{RT}
Schedulability test	sufficient	— *[prior APA test applies]*

n…number of tasks

m…number of cores

An Efficient **Strong** **HPA** Scheduler

Insight: Separate Job *Selection* from Job *Placement*

- **Job selection** (or admission): determine the set of jobs that should receive processor service
  - at most ***m***, but subject to affinity constraints.
- **Job placement**: map set of selected jobs to processors, while respecting
  - all affinity constraints and
  - the strong APA invariant.

Algorithms in Paper

- Algorithms 1 & 2: ***conceptual*** scheduling algorithm
  ➔ proof of *strong APA invariant*, but bad complexity

- Algorithms 3–5: ***runtime*** scheduling algorithm
  ➔ same schedule, but better complexity

- Algorithm 6: ***locality-aware*** assignment algorithm
  ➔ avoids some migrations, but worse complexity
  ➔ better suited for kernel-level implementation

Algorithms in Paper

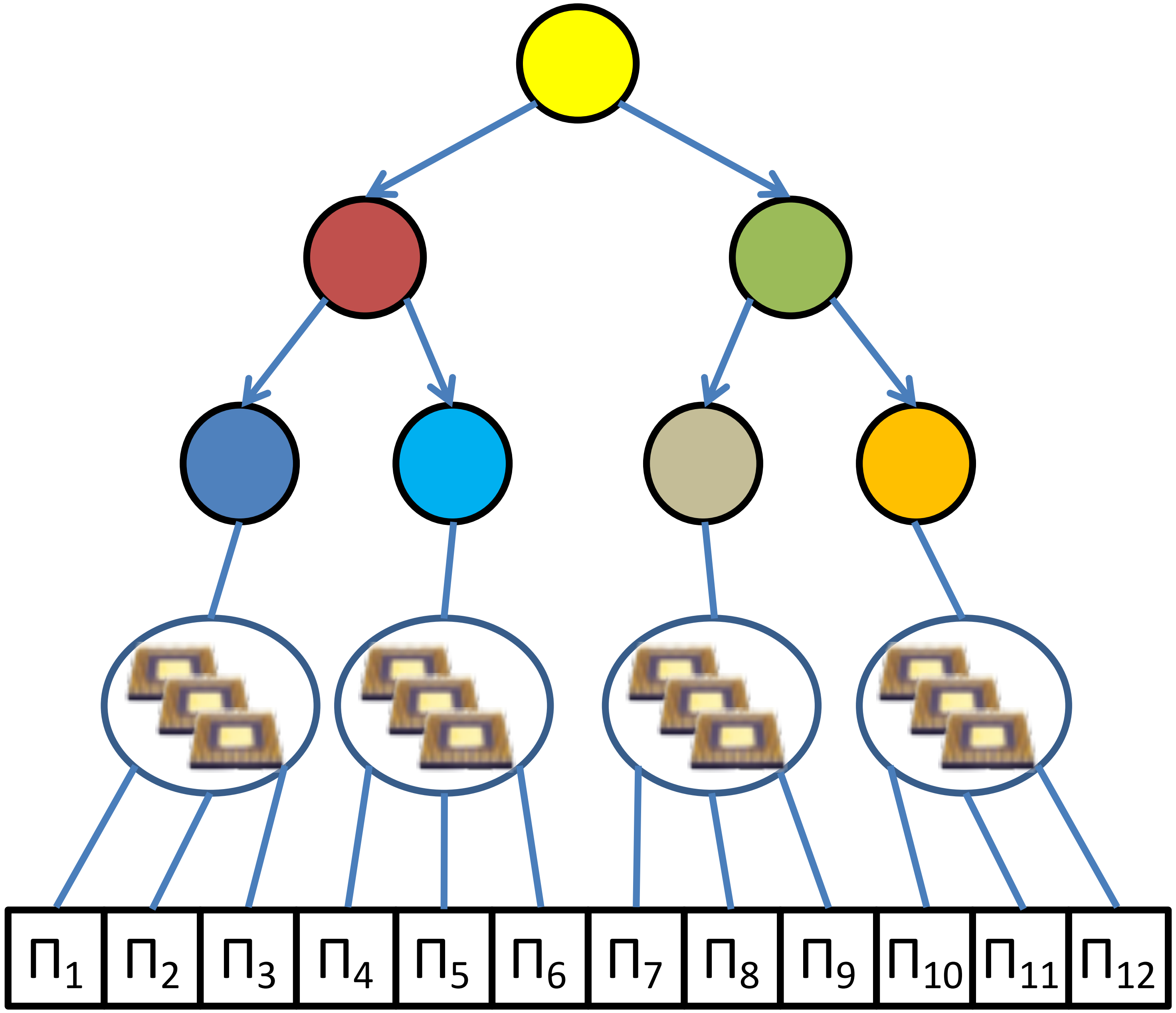
- Algorithms 1 & 2: ***conceptual*** scheduling algorithm
  ➔ proof of *strong APA invariant*, but bad complexity

- Algorithms 3–5: ***runtime*** scheduling algorithm
  ➔ same schedule, but better complexity
  [this talk]

- Algorithm 6: ***locality-aware*** assignment algorithm
  ➔ avoids some migrations, but worse complexity
  ➔ better suited for kernel-level implementation

Insight: Maintain State for each Distinct Affinity Set

- **don’t** have per-processor run-queues (Linux, etc.)
- **don’t** have just a single run queue
- **instead**, associate state with each <u>distinct affinity</u> (*affinity tree node*)

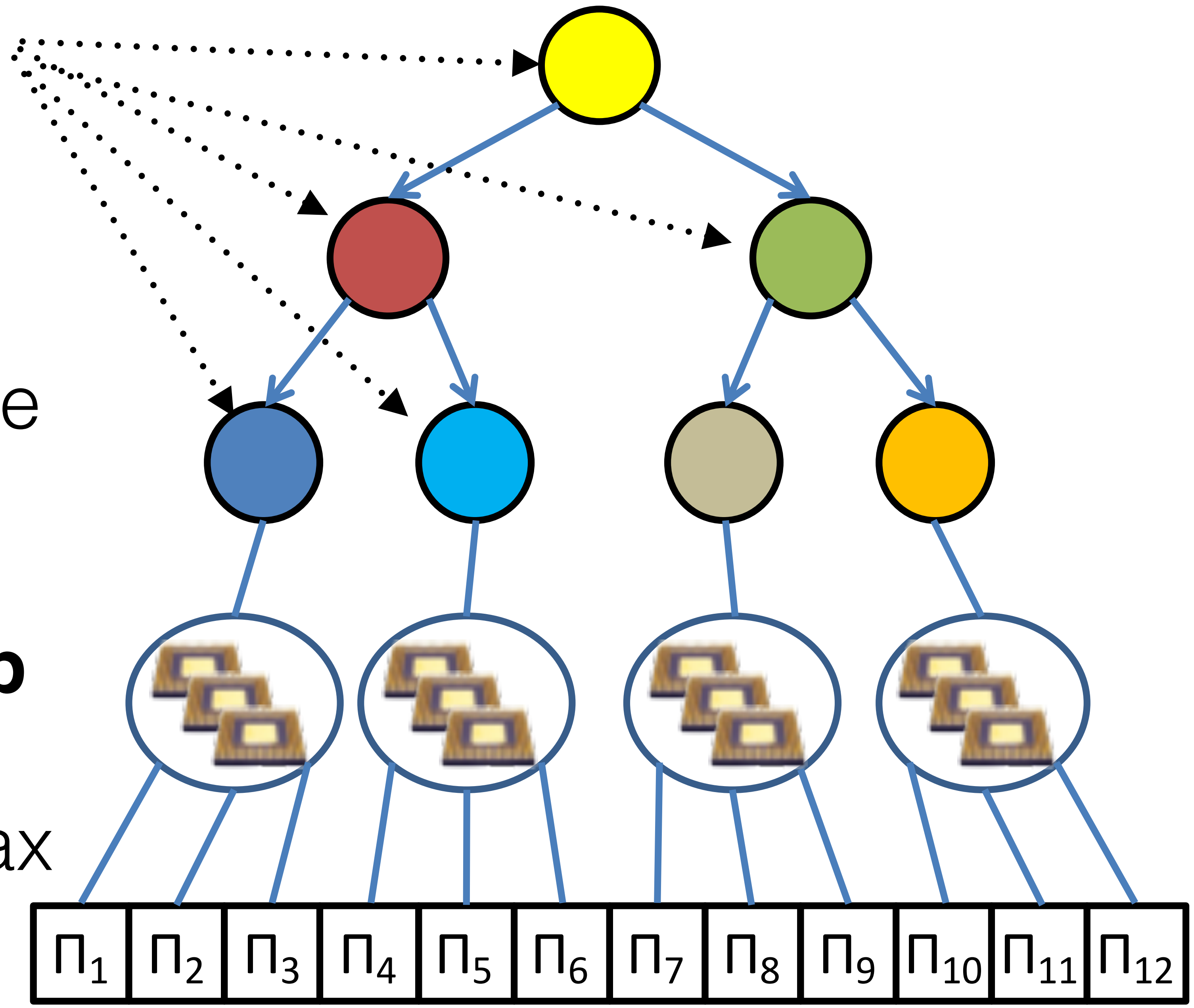


Data Structures

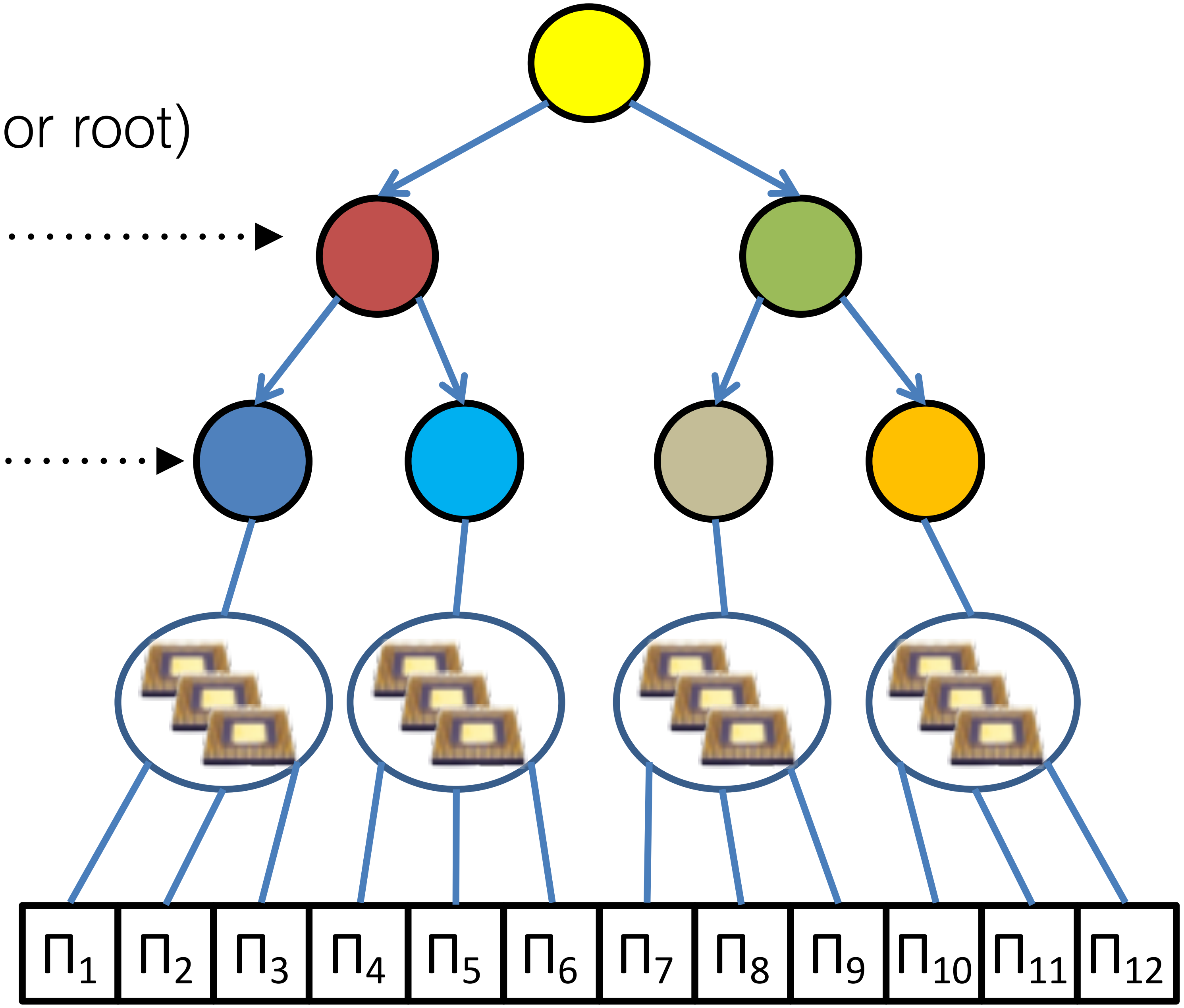
For each distinct affinity

- **doubly linked list** of *scheduled* jobs
  ➔ O(1) Insert, Remove
  ➔ O(n) FindMax

- **strict Fibonacci heap** of *backlogged* jobs
  ➔ O(1) Insert, FindMax
  ➔ O(log n) Remove



Job Arrival Step 1: Find Beta



first “full” affinity on path to root (or root)

β

α_i

affinity of arriving job

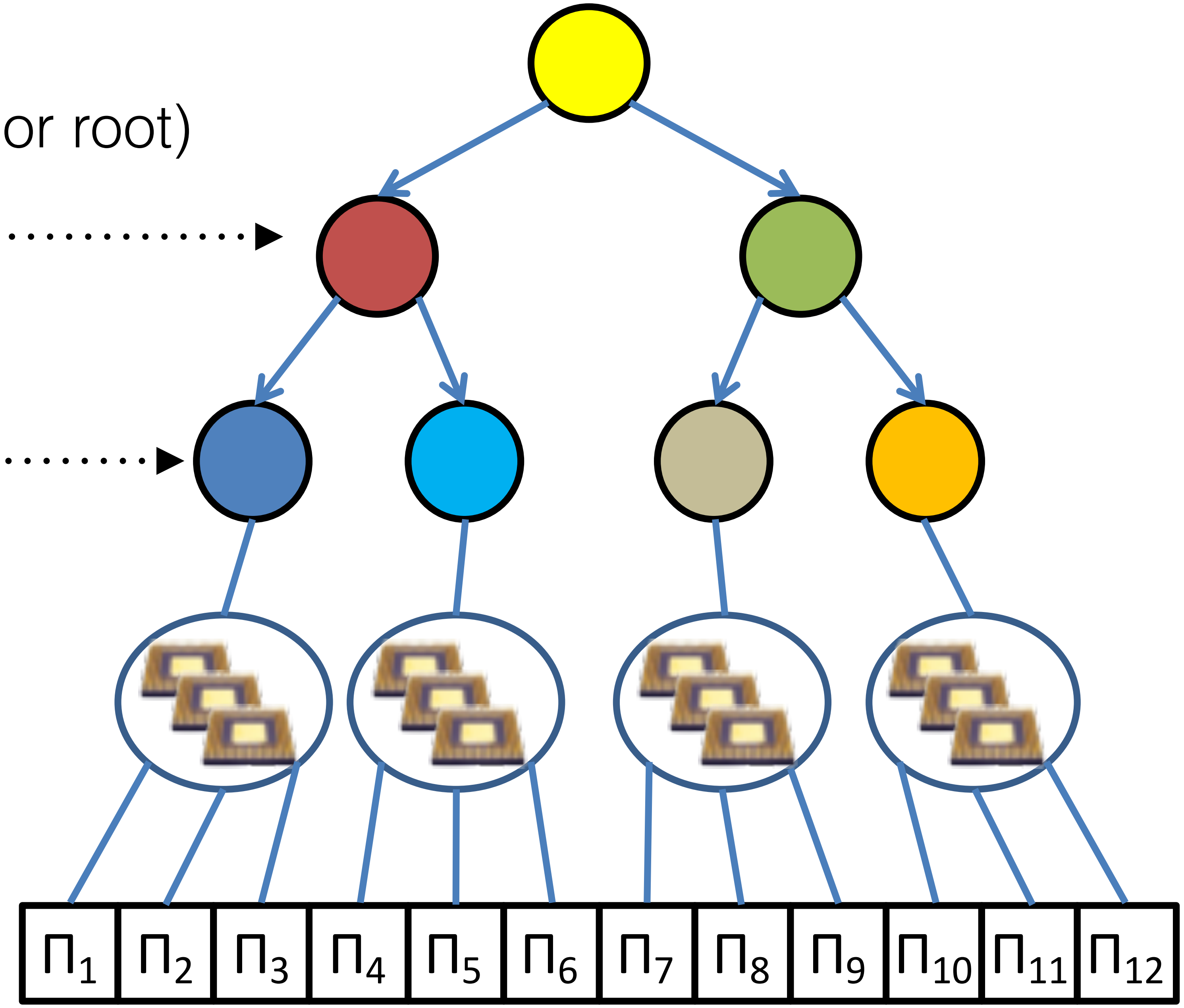
full: #scheduled jobs (list) = #processors in affinity

first “full” affinity on path to root (or root)

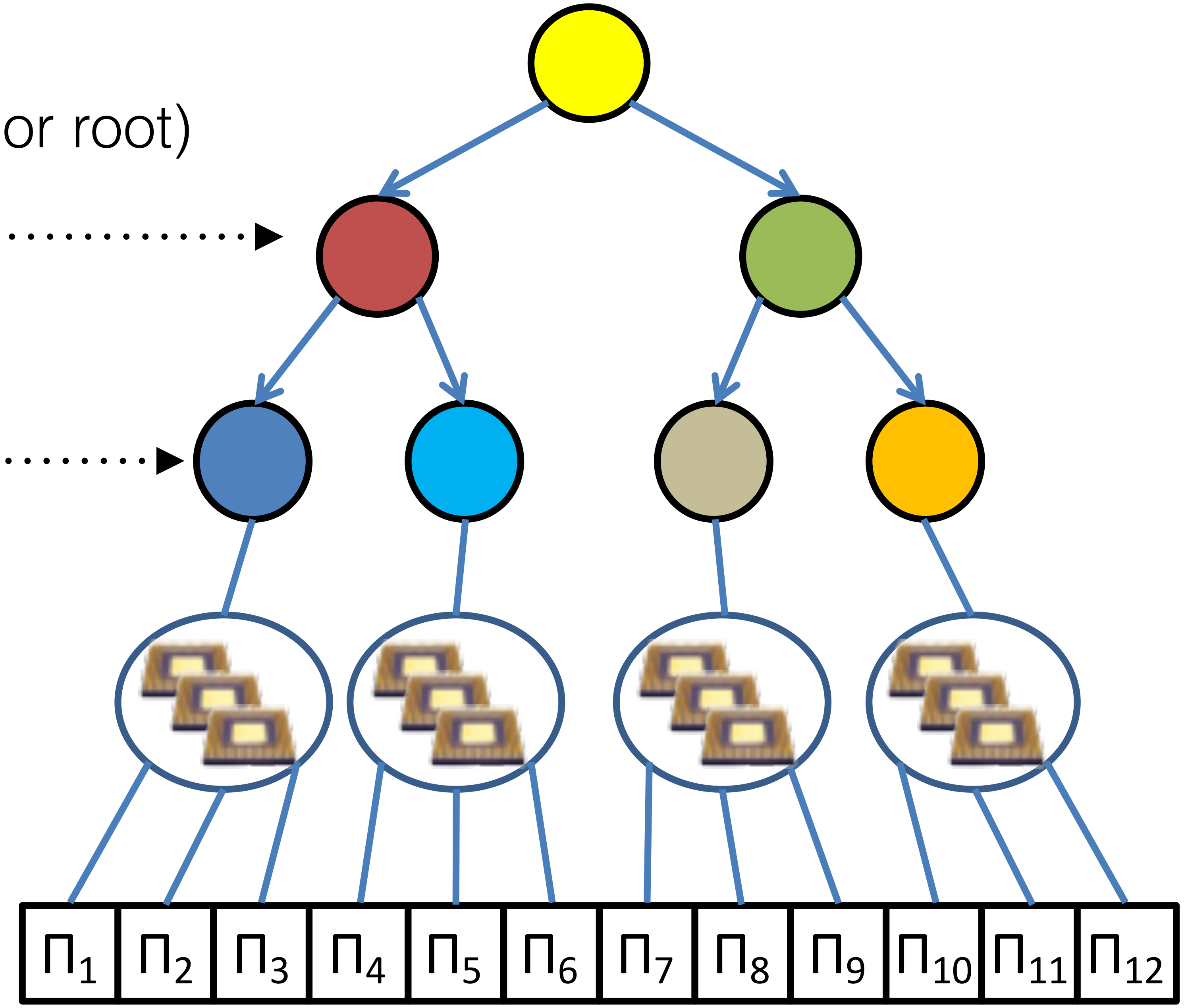
β

α_i

affinity of arriving job



Job Arrival Step 2: Walk Up the Tree and Insert into Lists



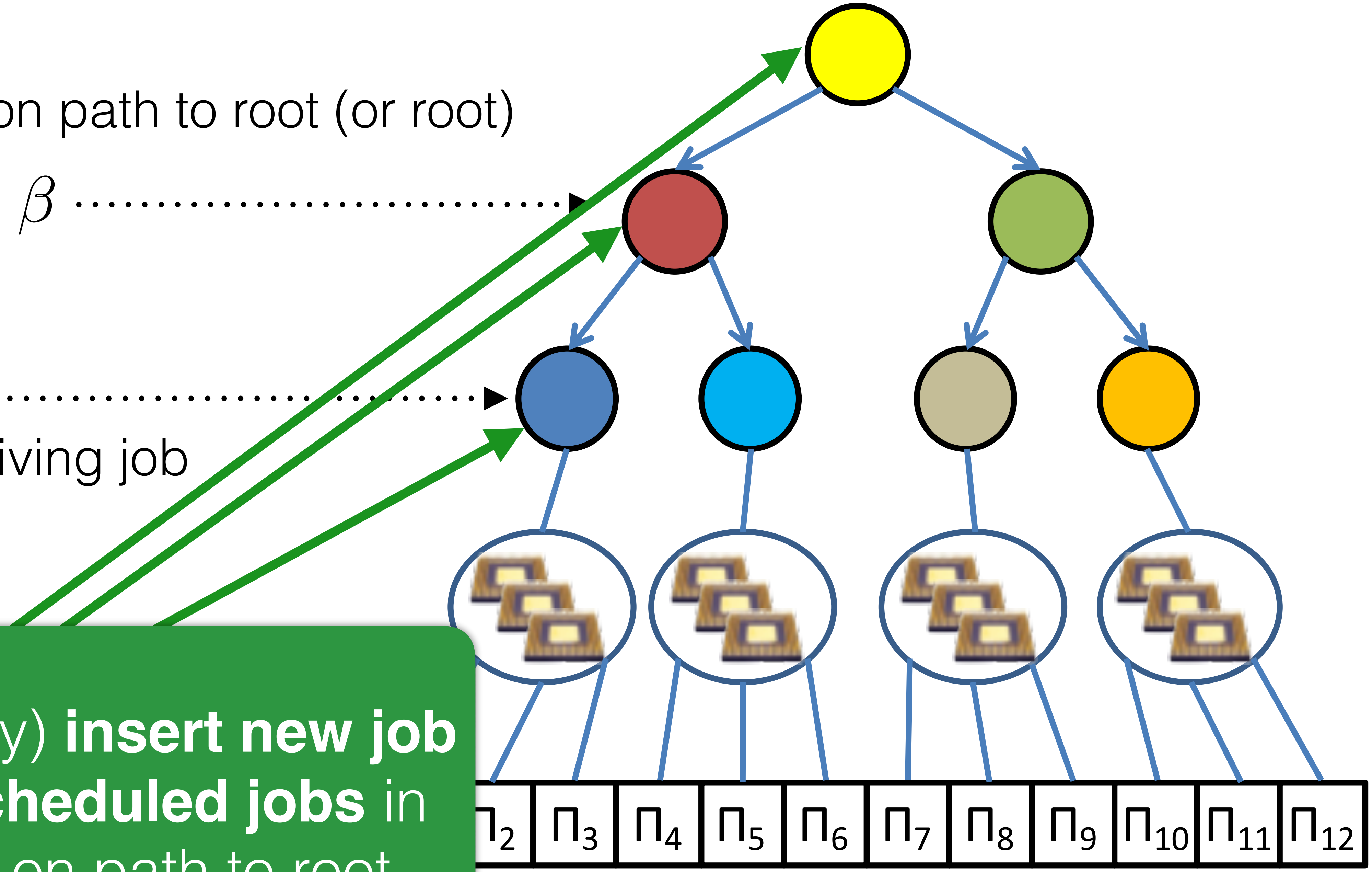
first “full” affinity on path to root (or root)

β

α_i

affinity of arriving job

Job Arrival Step 2: Walk Up the Tree and Insert into Lists



first "full" affinity on path to root (or root)

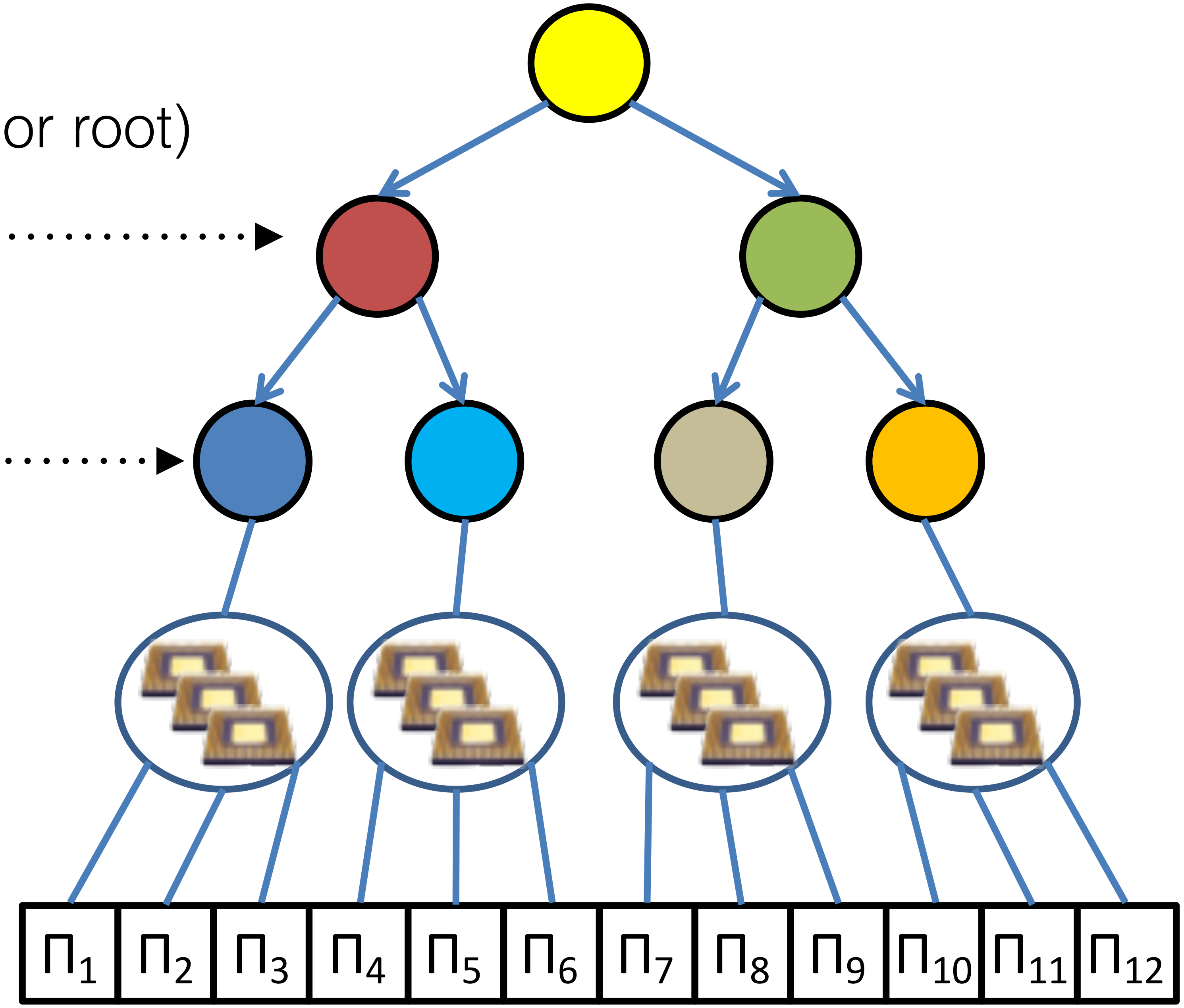
β

α_i

affinity of arriving job

(unconditionally) **insert new job into list of scheduled jobs** in each affinity on path to root

Job Arrival Step 3: Find Lowest-Priority Job in Beta Affinity's List



first “full” affinity on path to root (or root)

β

α_i

affinity of arriving job

Search list of “scheduled” jobs to **find lowest-priority job**

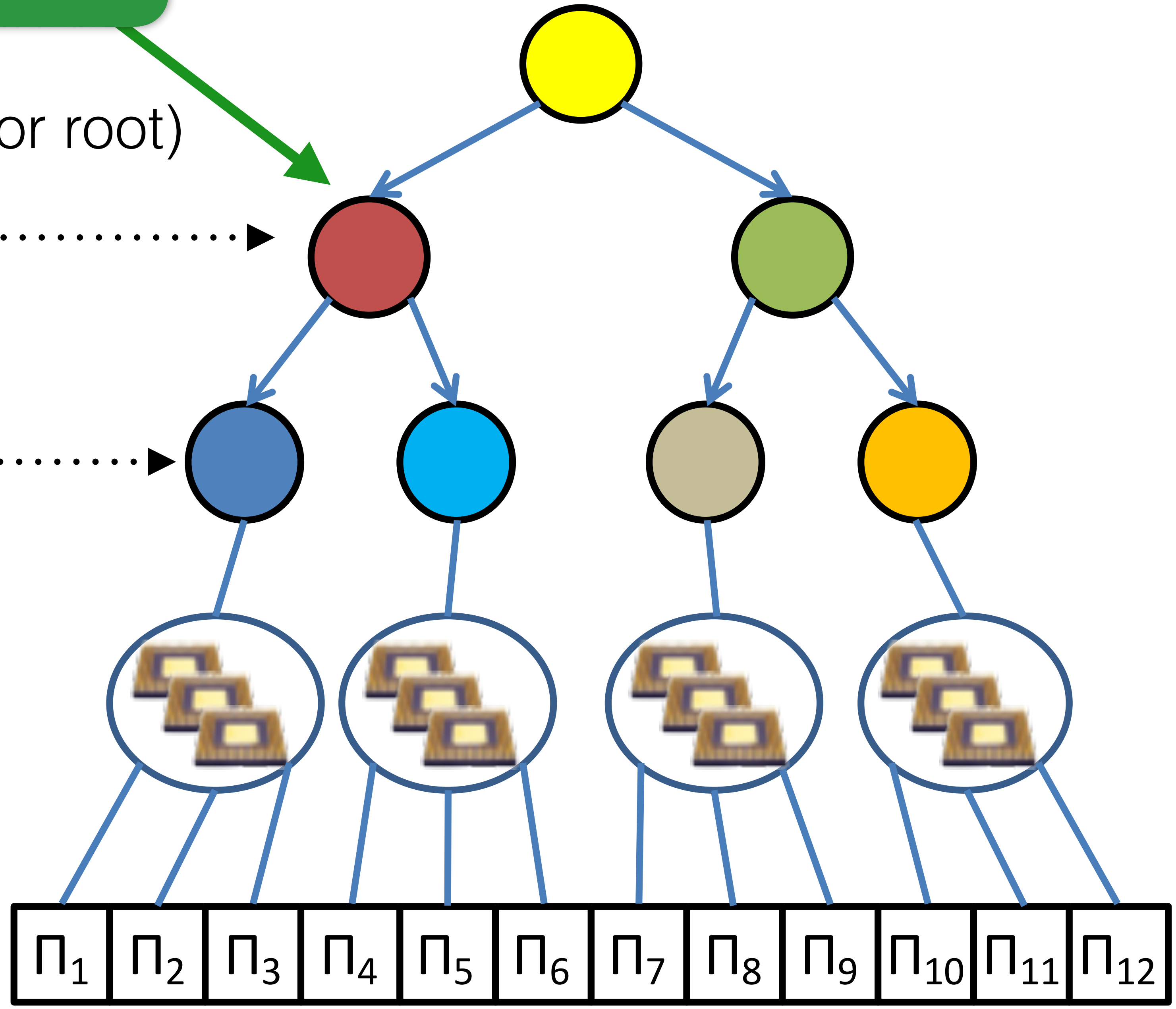
3: Find Lowest- eta Affinity’s List

first “full” affinity on path to root (or root)

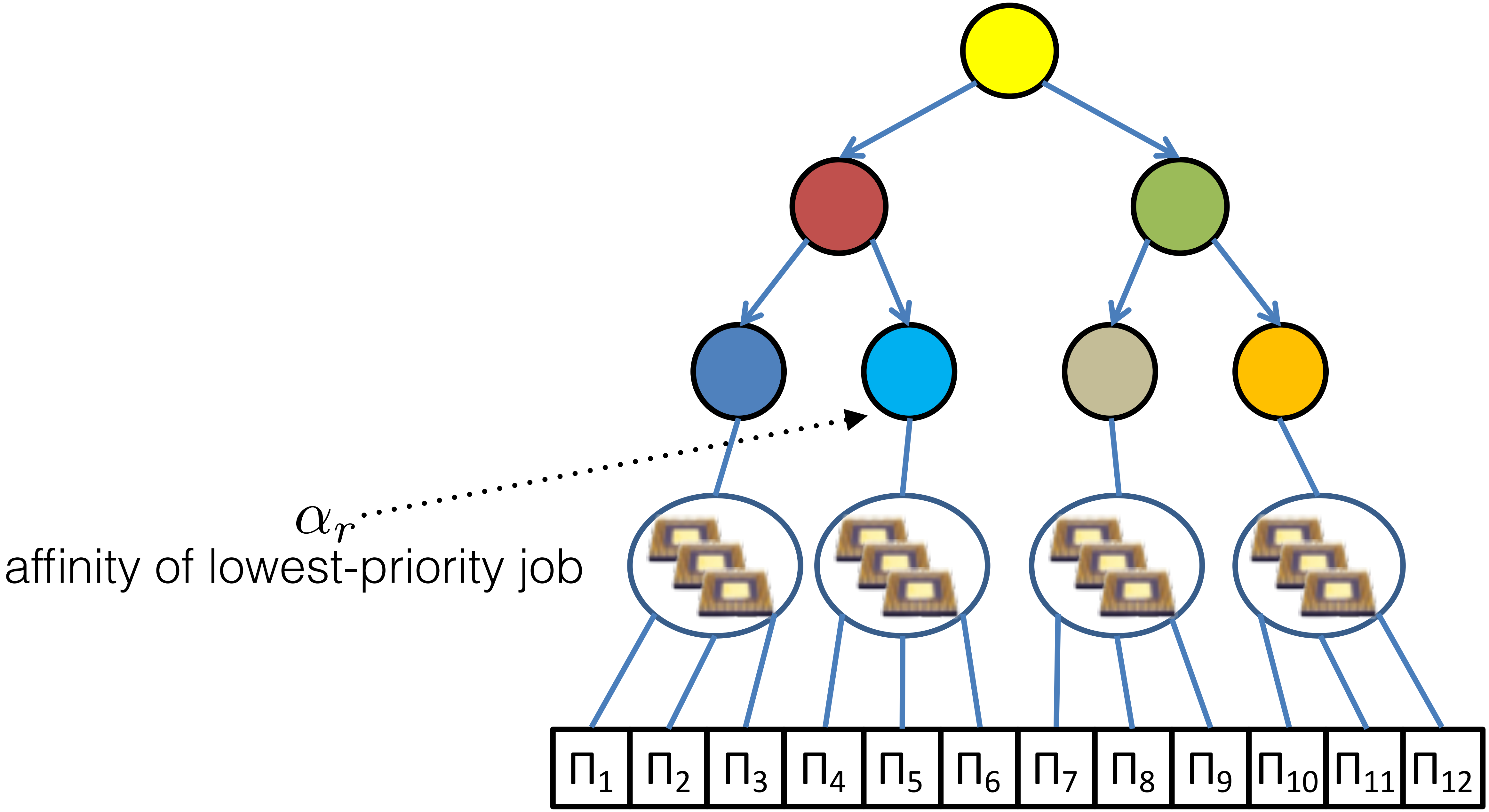
β

α_i

affinity of arriving job



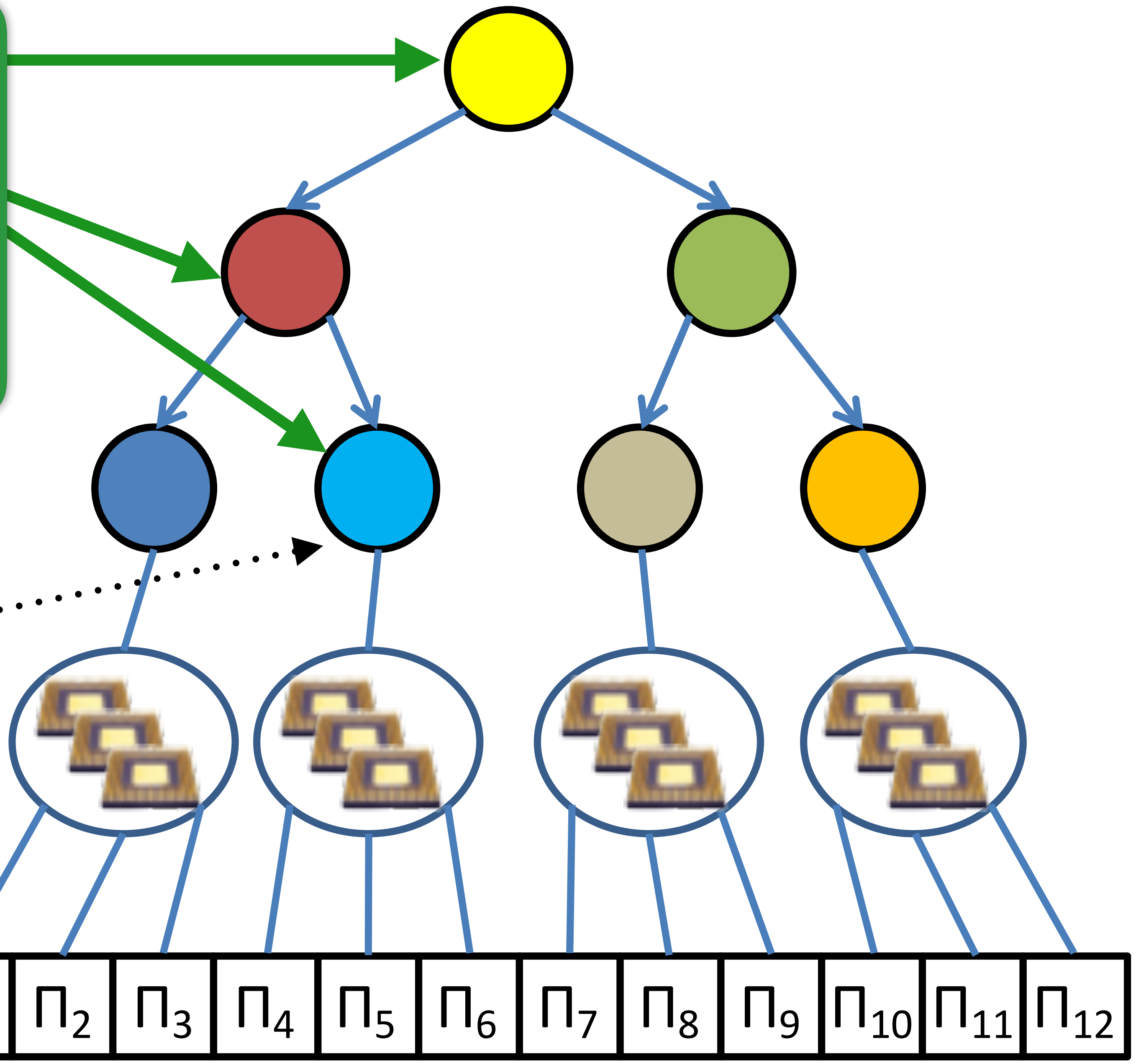
Job Arrival Step 4: Clean Up Lists along Path to Root



α_r

affinity of lowest-priority job

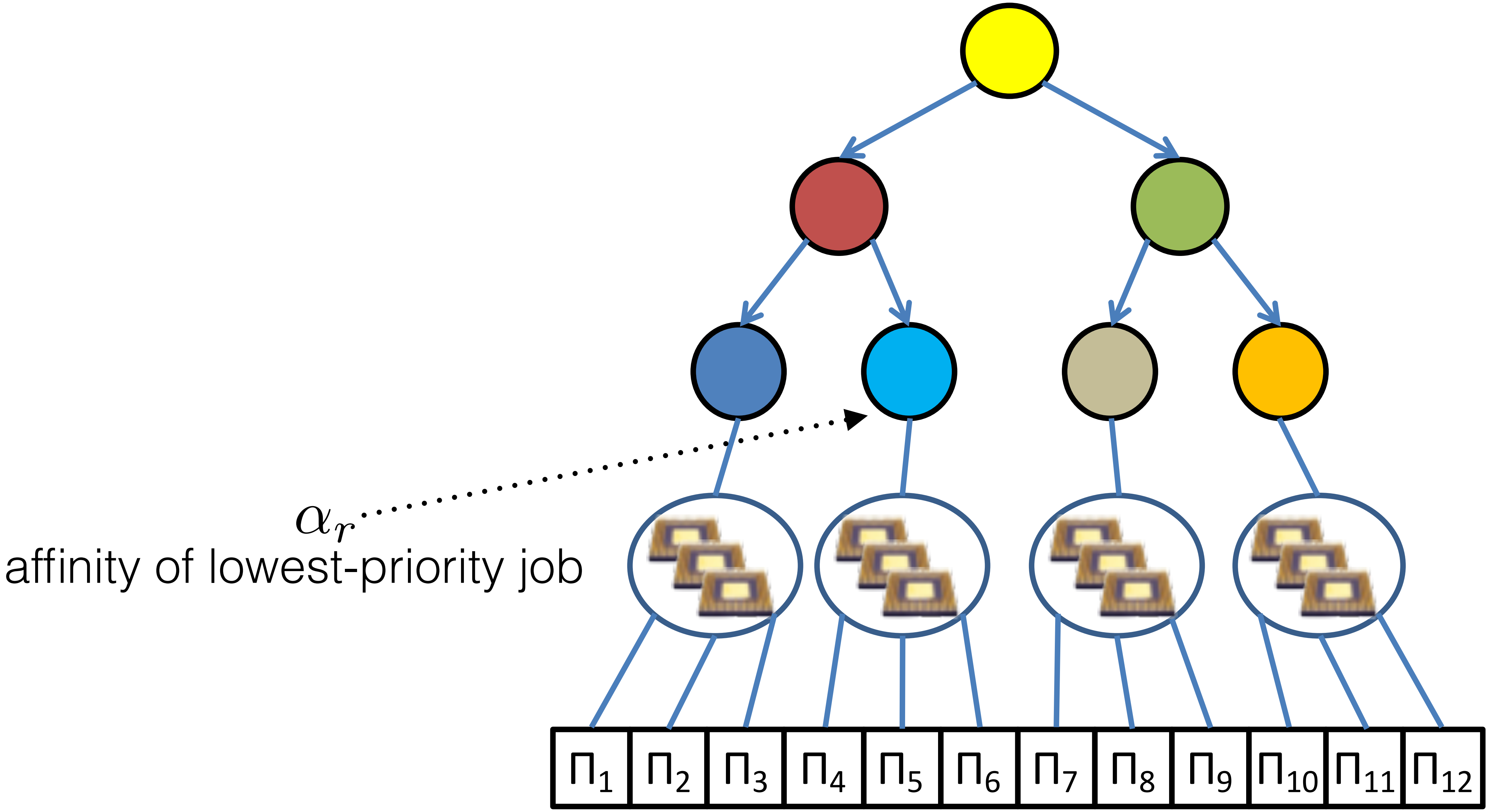
Job Arrival Step 4: Clean Up Lists along Path to Root



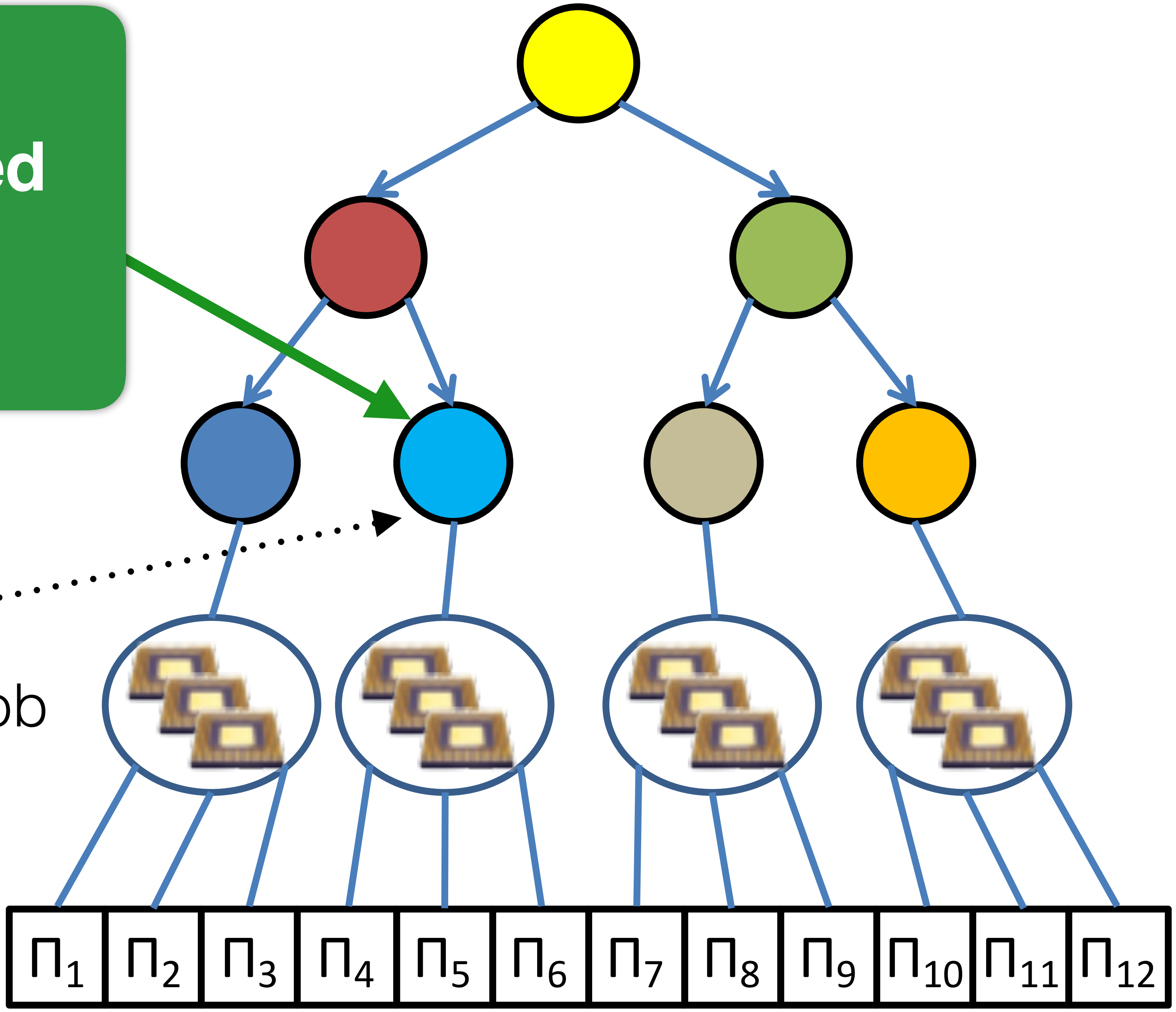
remove from list in each affinity on path to root, thereby ensuring that ***#scheduled ≤ #cores***

α_r
affinity of lowest-priority job

Job Arrival Step 5: Add to Heap of Backlogged Jobs



Job Arrival Step 5: Add to Heap of Backlogged Jobs



add to heap of backlogged jobs (only in own affinity)

α_r
affinity of lowest-priority job

Complexity of Job Selection upon Arrival: $O(m)$

m…number of cores

Complexity of Job Selection upon Arrival: ***O(m)***

1. Find beta affinity, first “full” affinity: **O(*height of tree*) = O(*m*)**

m…number of cores

Complexity of Job Selection upon Arrival: ***O(m)***

1. Find beta affinity, first “full” affinity: **O(*height of tree*) = O(*m*)**

2. Walk up the tree and insert new job into doubly-linked lists: **O(*height of tree*) = O(*m*)**

m…number of cores

Complexity of Job Selection upon Arrival: ***O(m)***

1. Find beta affinity, first “full” affinity: **O(*height of tree*) = O(*m*)**

2. Walk up the tree and insert new job into doubly-linked lists: **O(*height of tree*) = O(*m*)**

3. Find lowest-priority job in beta affinity’s list of scheduled jobs: **O(*length of list*) = O(*size of affinity*) = O(*m*)**

m…number of cores

Complexity of Job Selection upon Arrival: ***O(m)***

1. Find beta affinity, first “full” affinity: **O(*height of tree*) = O(*m*)**

2. Walk up the tree and insert new job into doubly-linked lists: **O(*height of tree*) = O(*m*)**

3. Find lowest-priority job in beta affinity’s list of scheduled jobs: **O(*length of list*) = O(*size of affinity*) = O(*m*)**

4. Walk up the tree and remove lowest-priority job from doubly-linked lists: **O(*height of tree*) = O(*m*)**

m…number of cores

Complexity of Job Selection upon Arrival: ***O(m)***

1. Find beta affinity, first “full” affinity: **O(*height of tree*) = O(*m*)**

2. Walk up the tree and insert new job into doubly-linked lists: **O(*height of tree*) = O(*m*)**

3. Find lowest-priority job in beta affinity’s list of scheduled jobs: **O(*length of list*) = O(*size of affinity*) = O(*m*)**

4. Walk up the tree and remove lowest-priority job from doubly-linked lists: **O(*height of tree*) = O(*m*)**

5. Add to strict Fibonacci heap of backlogged jobs: **O(1)**

m…number of cores

Job Arrival Part 2: Placing the Set of Selected Jobs: $\boldsymbol{O(m)}$

Job Arrival Part 2: Placing the Set of Selected Jobs: ***O(m)***

1. Make a list for each ***leaf node*** in the affinity tree, containing the **free processors** in the affinity: **O(*m*)**

Job Arrival Part 2: Placing the Set of Selected Jobs: ***O*(*m*)**

1. Make a list for each ***leaf node*** in the affinity tree, containing the **free processors** in the affinity: **O(*m*)**

2. copy the **list of scheduled jobs** from the **root node**: **O(*m*)**

Job Arrival Part 2: Placing the Set of Selected Jobs: $O(m)$

1. Make a list for each ***leaf node*** in the affinity tree, containing the **free processors** in the affinity: $O(m)$

2. copy the **list of scheduled jobs** from the **root node**: $O(m)$

3. **sort** the list of jobs by **increasing affinity level**
   (= decreasing distance to root node): $O(m)$ **— counting sort**

Job Arrival Part 2: Placing the Set of Selected Jobs: ***O(m)***

1. Make a list for each ***leaf node*** in the affinity tree, containing the **free processors** in the affinity: **O(*m*)**

2. copy the **list of scheduled jobs** from the **root node**: **O(*m*)**

3. **sort** the list of jobs by **increasing affinity level**
   (= decreasing distance to root node): **O(*m*) — counting sort**

4. for each job (bottom-up):
   - assign to **first core** in job affinity's free processor list and remove core from list: **O(*m*)**
   - when **moving up a level**, concatenate the processor lists of all child nodes and assign to parent node: **O(*number of distinct affinities*) = O(*m*)**

Insight: Reuse Job Arrival Procedure for "Cleanup" After Job Departure

Insight: Reuse Job Arrival Procedure for "Cleanup" After Job Departure

- **Problem**: restoring the strong APA invariant after a job departure is not trivial.

    - The **next job to be scheduled** could come from **any** of the affinity nodes in the tree.

Insight: Reuse Job Arrival Procedure for "Cleanup" After Job Departure

- **Problem**: restoring the strong APA invariant after a job departure is not trivial.

  - The **next job to be scheduled** could come from **any** of the affinity nodes in the tree.

- **Solution**: *simulate* a job **arrival** of the **highest-priority backlogged** job for each distinct affinity

➔ $O(\underline{\log n} + \underline{m}^2)$

Insight: Reuse Job Arrival Procedure for "Cleanup" After Job Departure

- **Problem**: restoring the strong APA invariant after a job departure is not trivial.
  - The **next job to be scheduled** could come from **any** of the affinity nodes in the tree.
- **Solution**: *simulate* a job **arrival** of the **highest-priority backlogged** job for each distinct affinity

➔ $\boldsymbol{O(\underline{\log n} + \underline{m}^2)}$

remove from strict Fibonacci heap

run **O(*m*)** arrival procedure for each of **O(*m*)** distinct affinities

Speed-Up Bounds

Speed-up bound *X* for algorithm *A*

If a task set is schedulable **under *any* policy** on ***m*** **unit-speed processors**, then it is also schedulable under ***A*** with ***m*** ***processors of speed X.***

- quantifiable relation to system **optimality**
- a way to structure the space of non-optimal algorithms
- the lower the speed-up bound, the better

First Speed-Up Results for Real-Time Scheduling with Affinity Restrictions

Considered special cases:

- job priorities determined with **EDF**

and either

- **bi-level** affinities or
- **clustered** affinities.

Bi-Level Affinities

- each task is assigned either
  - a **global** affinity (can use all cores) or
  - a **singleton** affinity (can use only one specific core)

Bi-Level Affinities

- each task is assigned either
  - a **global** affinity (can use all cores) or
  - a **singleton** affinity (can use only one specific core)

HPA-EDF + Bi-Level Affinities

required speed-up s: **s < 2.415**

Bi-Level Aff

Context

speed-up bound
of global EDF is **2**

- each task is assigned either
  - a **global** affinity (can use all cores) or
  - a **singleton** affinity (can use only one specific core)

HPA-EDF + Bi-Level Affinities

required speed-up s: **$s < 2.415$**

Clustered Affinities

- each task is assigned either
    - a **global** affinity (can use all cores) or
    - a **clustered** affinity (can use only subset of cores)
- all **clusters are mutually disjoint**

Clustered Affinities

- each task is assigned either
    - a **global** affinity (can use all cores) or
    - a **clustered** affinity (can use only subset of cores)
- all **clusters are mutually disjoint**

HPA-EDF + Clustered Affinities

required speed-up ***s***: ***s*** **< 3.562**

ECRTS * Artifact * AE * Evaluated *
Consistent * Complete * Well Documented * Easy to Reuse *

Implementation in

LITMUS^{RT}

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

www.litmus-rt.org

LITMUS^{RT}

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

- real-time extension of the Linux kernel (*currently, Linux 4.1*)
- continuously maintained since 2006
- makes it ~~**easy**~~ **easier** to implement and evaluate (multiprocessor) real-time resource management policies on real hardware
- relevant highlights: built-in global **migration support** and **overhead tracing infrastructure**

THE UNIVERSITY
of NORTH CAROLINA
at CHAPEL HILL

[2006–2011]

Max
Planck
Institute
for
Software Systems

[2011–]

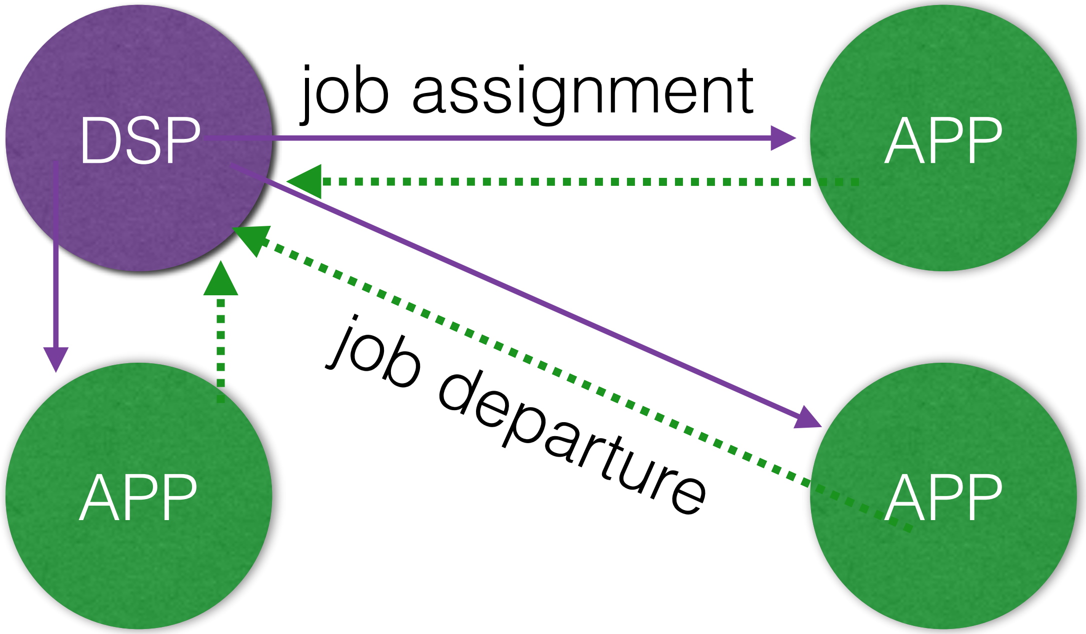
Evaluation Questions

- Can you actually run the proposed HPA scheduler **in a real OS kernel**?
- What **practical tweaks** are required?
- Isn't this algorithm prohibitively expensive in terms of **actual runtime overheads**?

Baseline

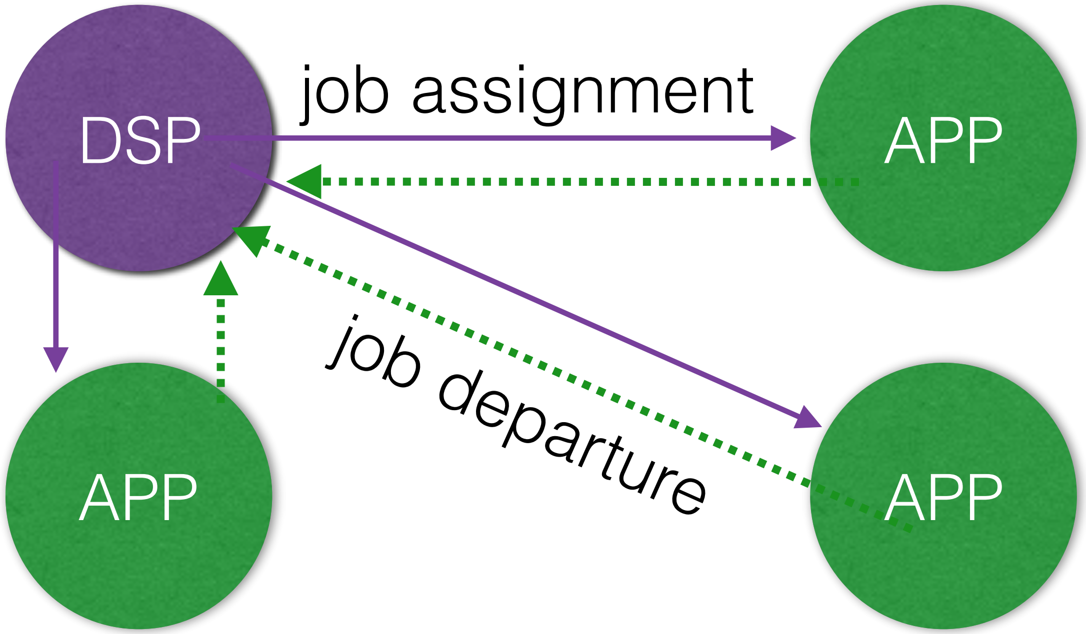
- **HPA-FP** (HPA + fixed priority) implemented on top of Cerqueira et al.'s **message-passing-based** **global scheduler** [RTAS'14].

- **Basic idea**
  ➔ one *designated scheduling processor* (**DSP**)
  ➔ **DSP** makes *all* scheduling decisions (for all cores)
  ➔ *application processors* send job state changes via messages
  ➔ simple **dispatcher** enacts scheduling decisions on app procs.



Baseline

- **HPA-FP** (HPA + fixed priority) implemented on top of Cerqueira et al.'s **message-passing-based** **global scheduler** [RTAS'14].
- **Basic idea**
  ➔ one *designated scheduling processor* (**DSP**)
  ➔ **DSP** makes *all* scheduling decisions (for all cores)
  ➔ *application processors* send job state changes via messages
  ➔ simple **dispatcher** enacts scheduling decisions on app procs.



- **no locking of scheduler state**
- **no cache-line bouncing**
- **better scalability [max. overheads]**

Practical Tweaks

Practical Tweaks

- **Affinity tree** is explicitly stored in kernel and **dynamically extended** as tasks are admitted

Practical Tweaks

- **Affinity tree** is explicitly stored in kernel and **dynamically extended** as tasks are admitted

- Message-passing-based design removes the need to synchronize tree traversals (big plus!).

Practical Tweaks

- **Affinity tree** is explicitly stored in kernel and **dynamically extended** as tasks are admitted

- Message-passing-based design removes the need to synchronize tree traversals (big plus!).

- **Strict Fibonacci heaps** are complicated & slow
  ➔ use standard ***priority bitmap + linked lists***
  ➔ ***effectively*** **O(1)** for fixed #priorities

Practical Tweaks

- **Affinity tree** is explicitly stored in kernel and **dynamically extended** as tasks are admitted

- Message-passing-based design removes the need to synchronize tree traversals (big plus!).

- **Strict Fibonacci heaps** are complicated & slow
  ➔ use standard ***priority bitmap + linked lists***
  ➔ ***effectively* O(1)** for fixed #priorities

- **Locality-aware task mapping** to avoid needless migrations (Algorithm 6)
  ➔ implemented with sets (=bit operations)
  ➔ ***effectively* O(1)** for fixed, small #cores

Platform & Workloads

Platform

- Xeon E7 8857, two sockets, 12 cores each (***m = 24***)
- private **L1** and **L2** (32 KiB and 256 KiB, resp.)
- shared **L3** (30 MiB) per socket

Workload

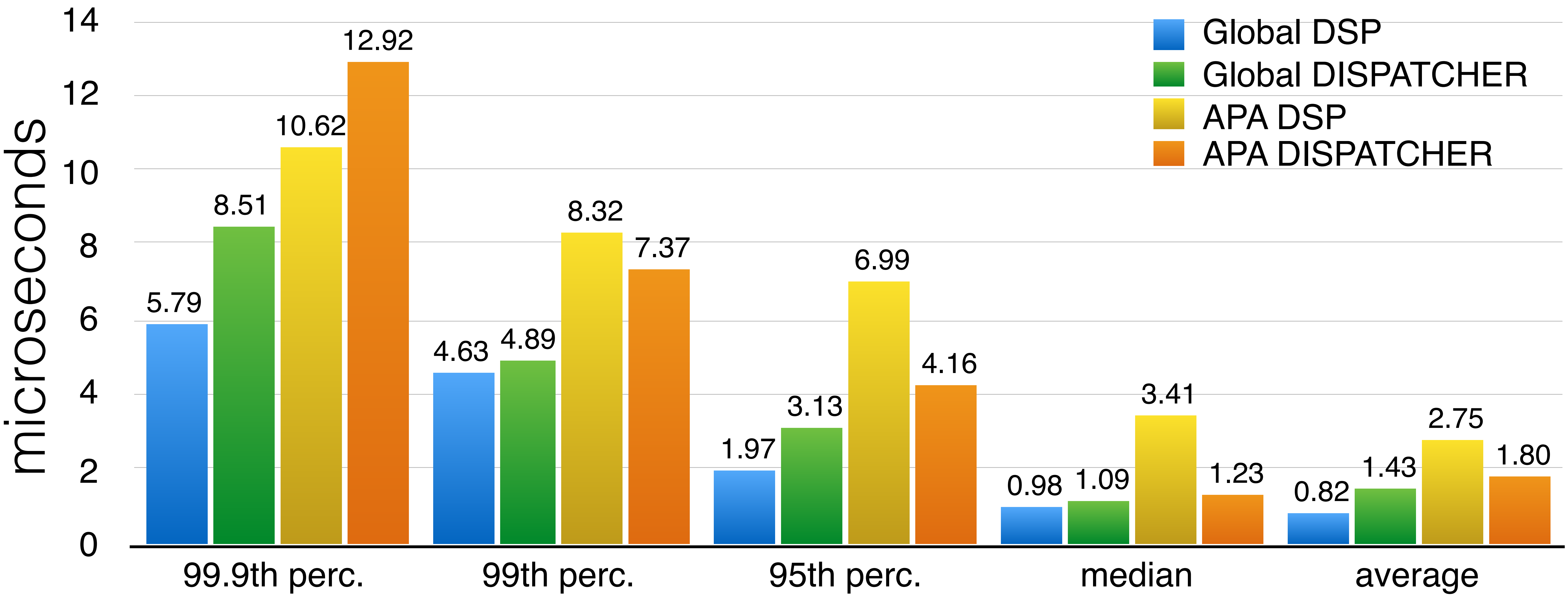
- 75%/85% utilization
- execution costs: Emberson et al. (2010)
- log-uniform periods **1ms** to **1000ms**
- 2***m*** to 10***m*** tasks (48 to 240)
- three affinity levels: ***global, socket, partitioned***
- rate-monotonic priorities

Experiments

- 150 task sets per scheduler
- 60 seconds per task set
- traced **scheduler overheads** with Feather-Trace
- 34 GiB trace data
- extracted 700,000,000 valid samples

Results Overview

- substantially increased costs (~ *1.5x to ~3.5x*), but still in a feasible range (*a few microseconds*)



Evaluation Questions

- Can you actually run the proposed HPA scheduler **in a real OS kernel**?
- What **practical tweaks** are required?
- Isn't this algorithm prohibitively expensive in terms of **actual runtime overheads**?

Evaluation Questions

- Can you actually run the proposed HPA scheduler **in a real OS kernel**?

➔ ***Yes!***

- What **practical tweaks** are required?

- Isn't this algorithm prohibitively expensive in terms of **actual runtime overheads**?

Evaluation Questions

- Can you actually run the proposed HPA scheduler **in a real OS kernel**?

➔ *Yes!*

- What **practical tweaks** are required?

➔ *locality-aware assignment* *and* *simpler queues*

- Isn't this algorithm prohibitively expensive in terms of **actual runtime overheads**?

Evaluation Questions

- Can you actually run the proposed HPA scheduler **in a real OS kernel**?

➔ ***Yes!***

- What **practical tweaks** are required?

➔ ***locality-aware assignment and simpler queues***

- Isn't this algorithm prohibitively expensive in terms of **actual runtime overheads**?

➔ ***more costly, but not prohibitively so***

Concluding Remarks

Summary

Summary

Hierarchical processor affinities are an important special case.

Summary

Hierarchical processor affinities are an important special case.

The laminar affinity structure **allows for a much more efficient online scheduler.**

Summary

Hierarchical processor affinities are an important special case.

The laminar affinity structure **allows for a much more efficient online scheduler.**

first speed-up result for real-time scheduling with restricted processor affinities

Summary

Hierarchical processor affinities are an important special case.

The laminar affinity structure **allows for a much more efficient online scheduler.**

first speed-up result for real-time scheduling with restricted processor affinities

first implementation of a strong APA scheduler in a real OS kernel

Some Open Questions

- A more **efficient** ***weak*** **HPA** scheduler?
- **Speed-up** bounds for more **general cases**?
- More **accurate schedulability tests** for strong and weak HPA scheduling?
- Is there some **interesting class of affinities** between arbitrary and hierarchical?

APA > **?**PA > HPA

LITMUS^{RT}

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

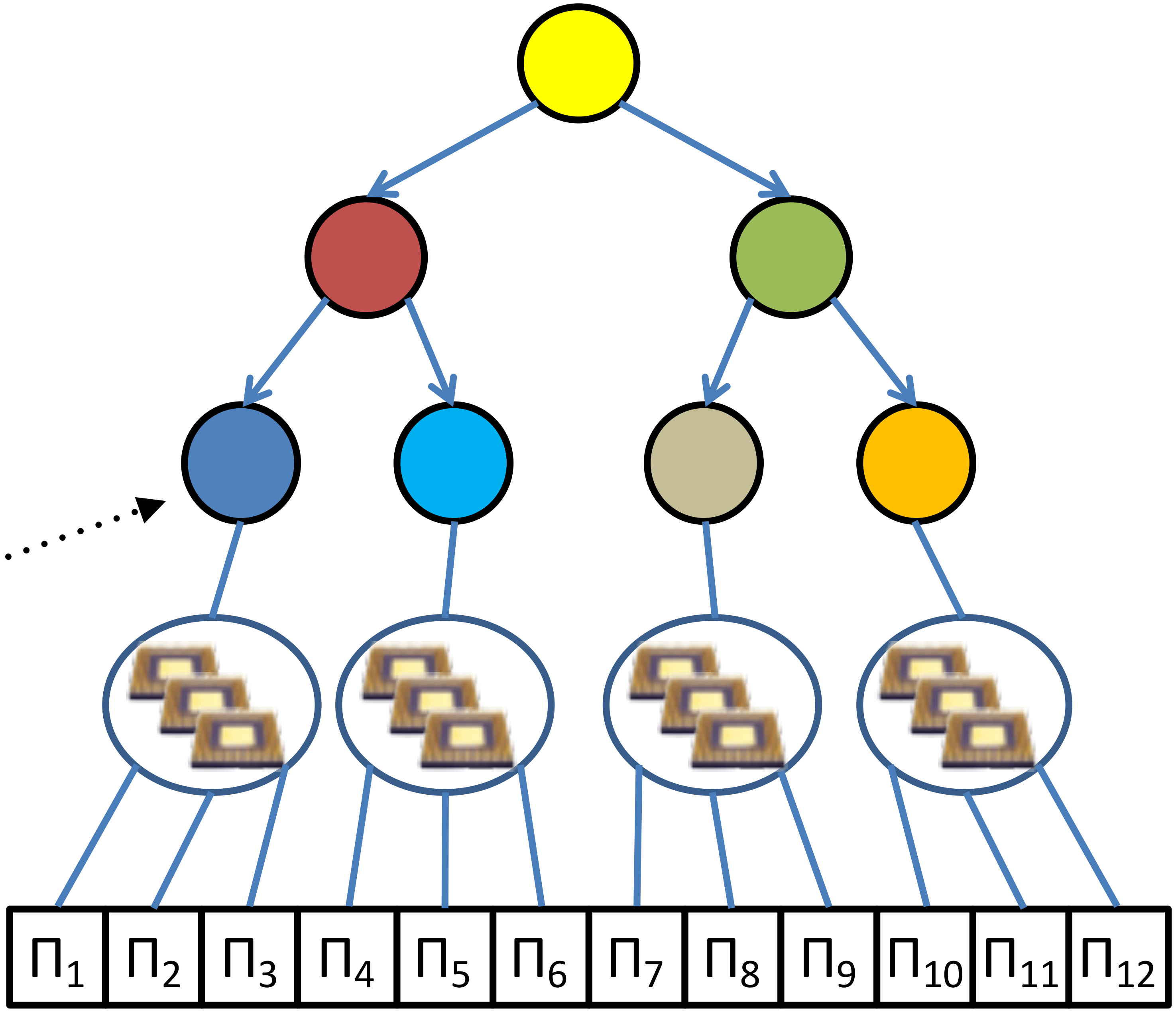
- New release **2016.1**
  ➔ framework for proper **reservation-based scheduling**

- A new tutorial: **Getting Started with LITMUS^{RT}**
  ➔ http://www.litmus-rt.org/tutor16/

- Detailed **artifact evaluation** instructions
  ➔ how to run our HPA scheduler
  ➔ how to collect and process data
  ➔ https://www.mpi-sws.org/~bbb/papers/ae/ecrts16/laminar-apa.html

Appendix

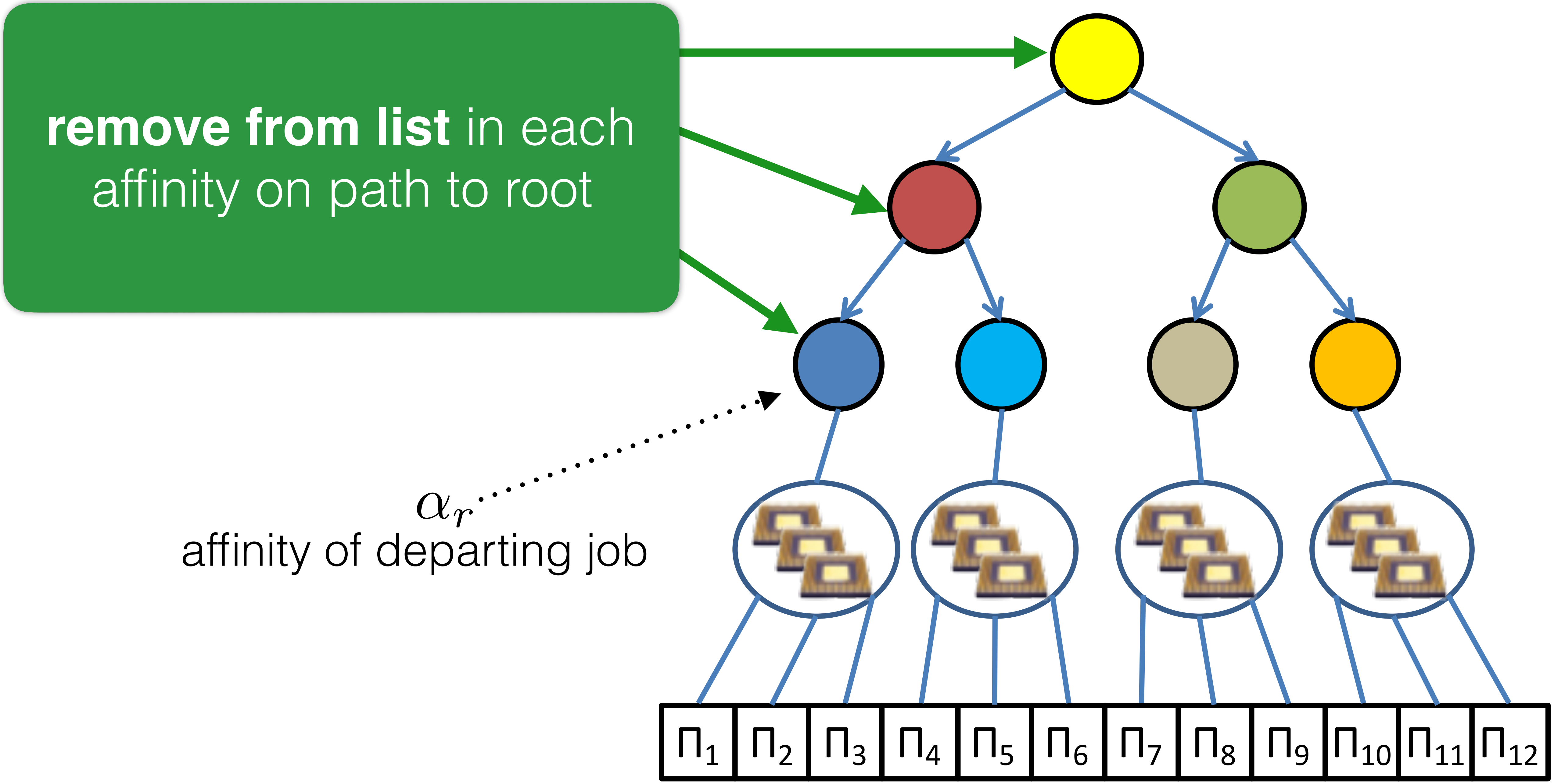
Job Departure Step 1: Remove from Lists



α_r

affinity of departing job

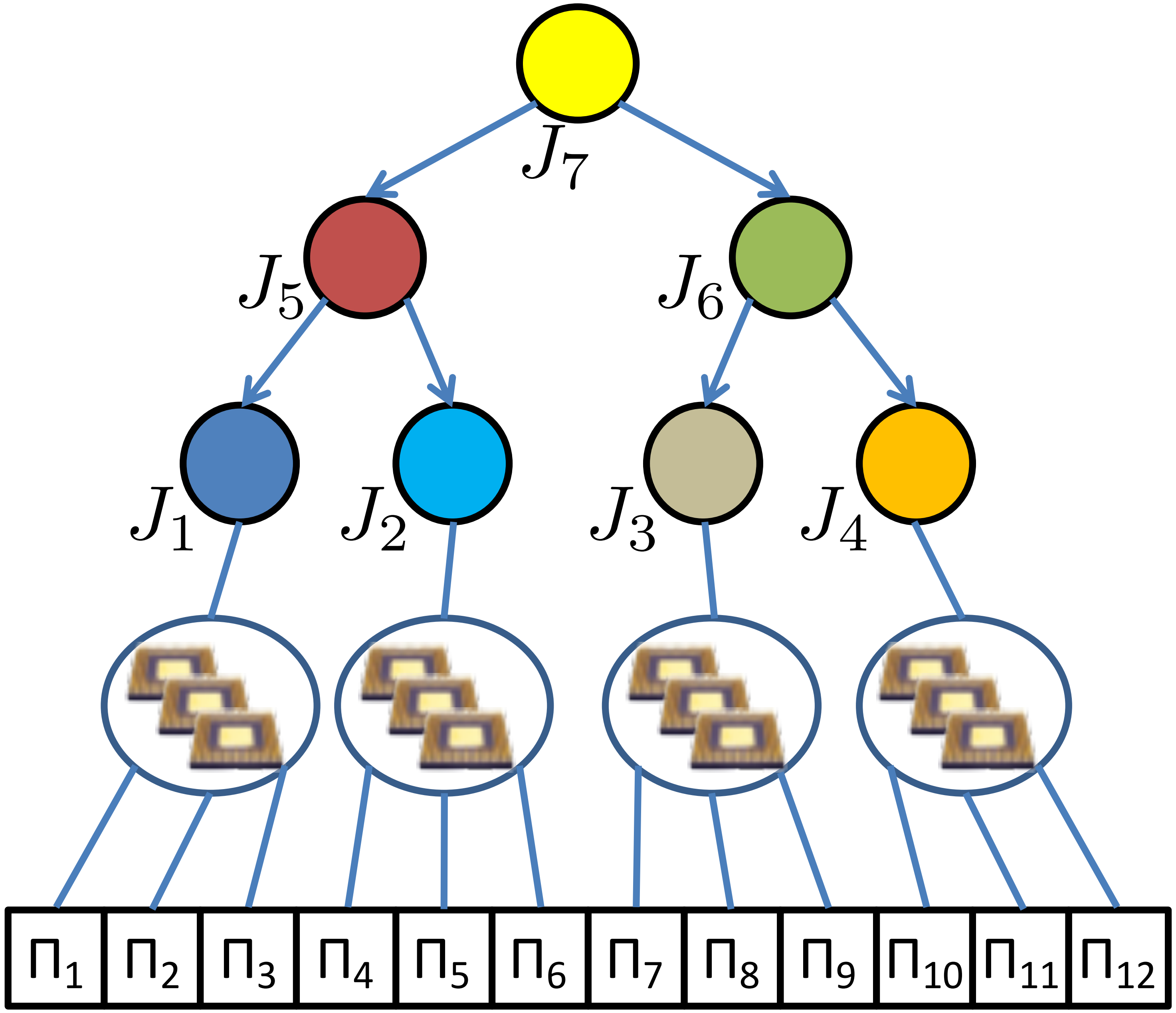
Job Departure Step 1: Remove from Lists



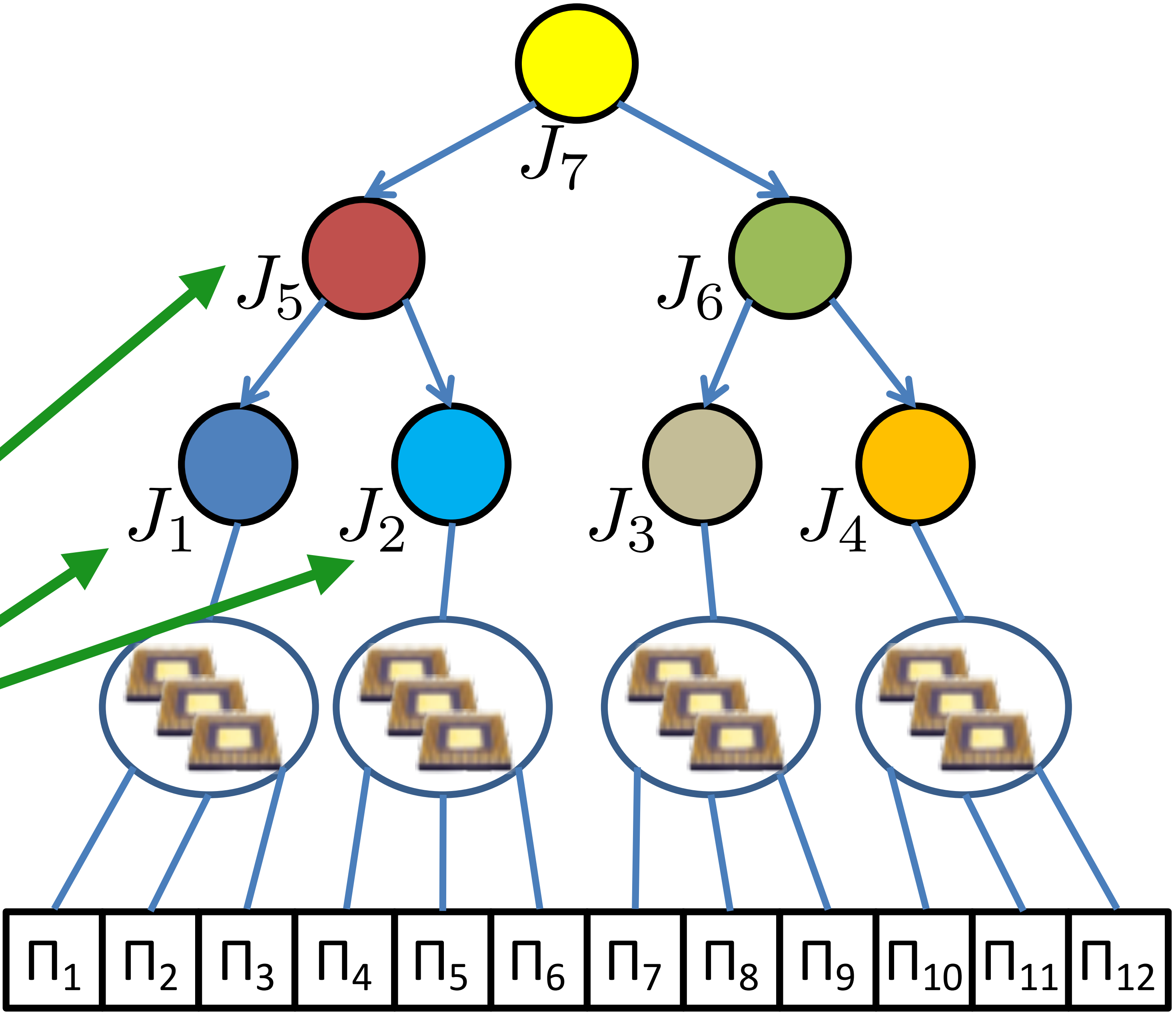
remove from list in each affinity on path to root

α_r
affinity of departing job

Job Departure Step 2: Find Max in each Affinity

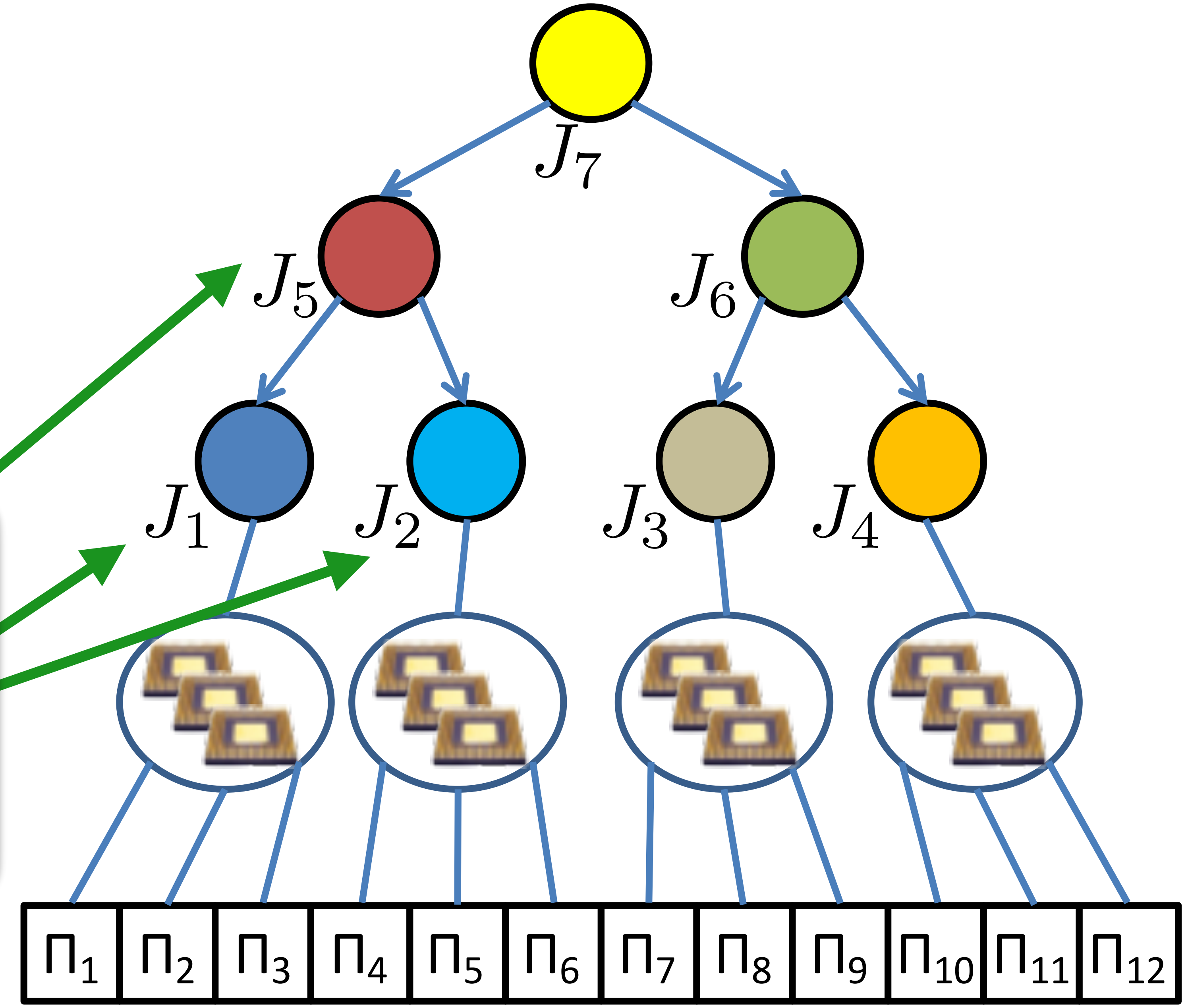


Job Departure Step 2: Find Max in each Affinity



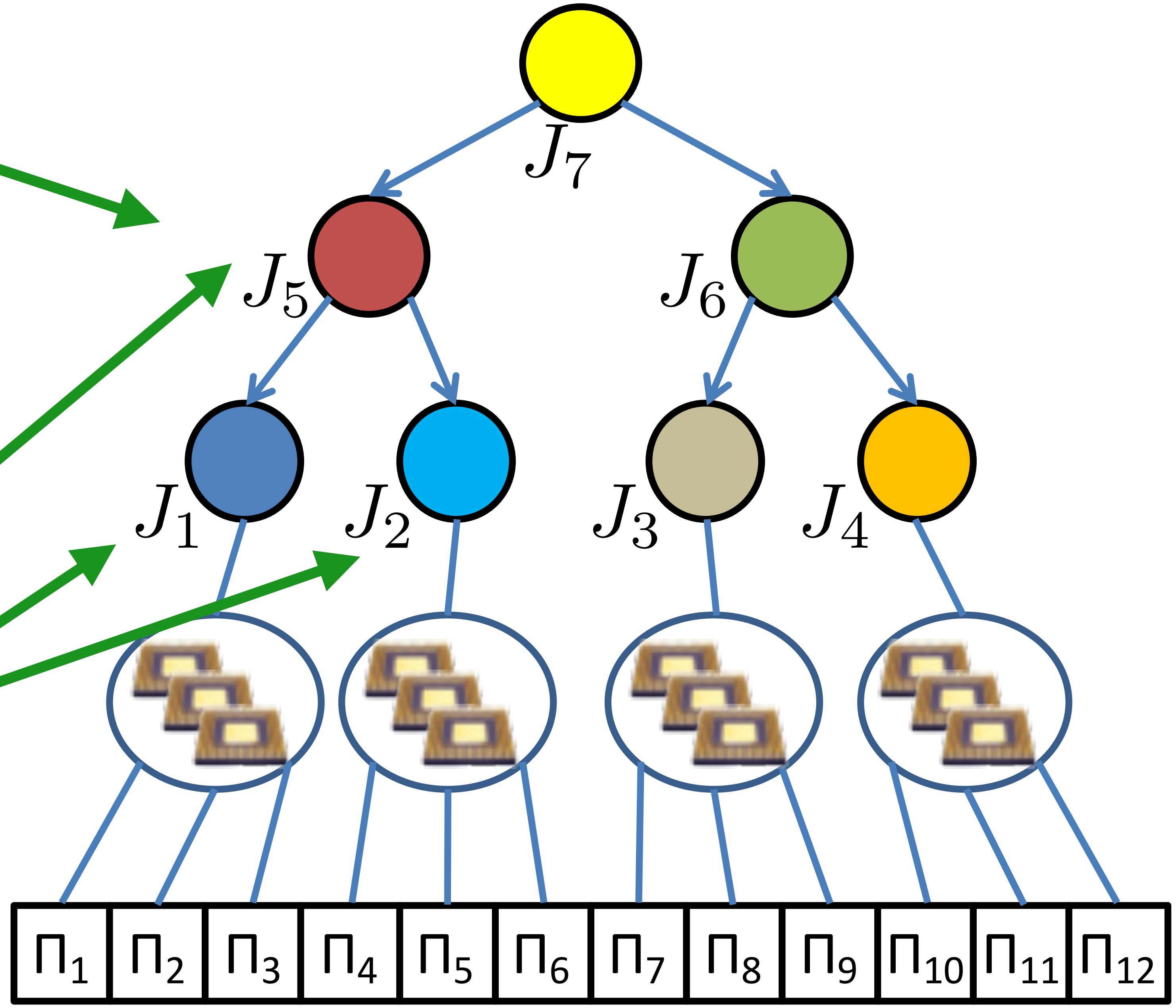
find **highest-priority backlogged job** in each distinct affinity (Fibonacci Heap)

Job Departure Step 3: Simulate Arrivals



find **highest-priority backlogged job** in each distinct affinity (Fibonacci Heap)

Job Departure Step 3: Simulate Arrivals

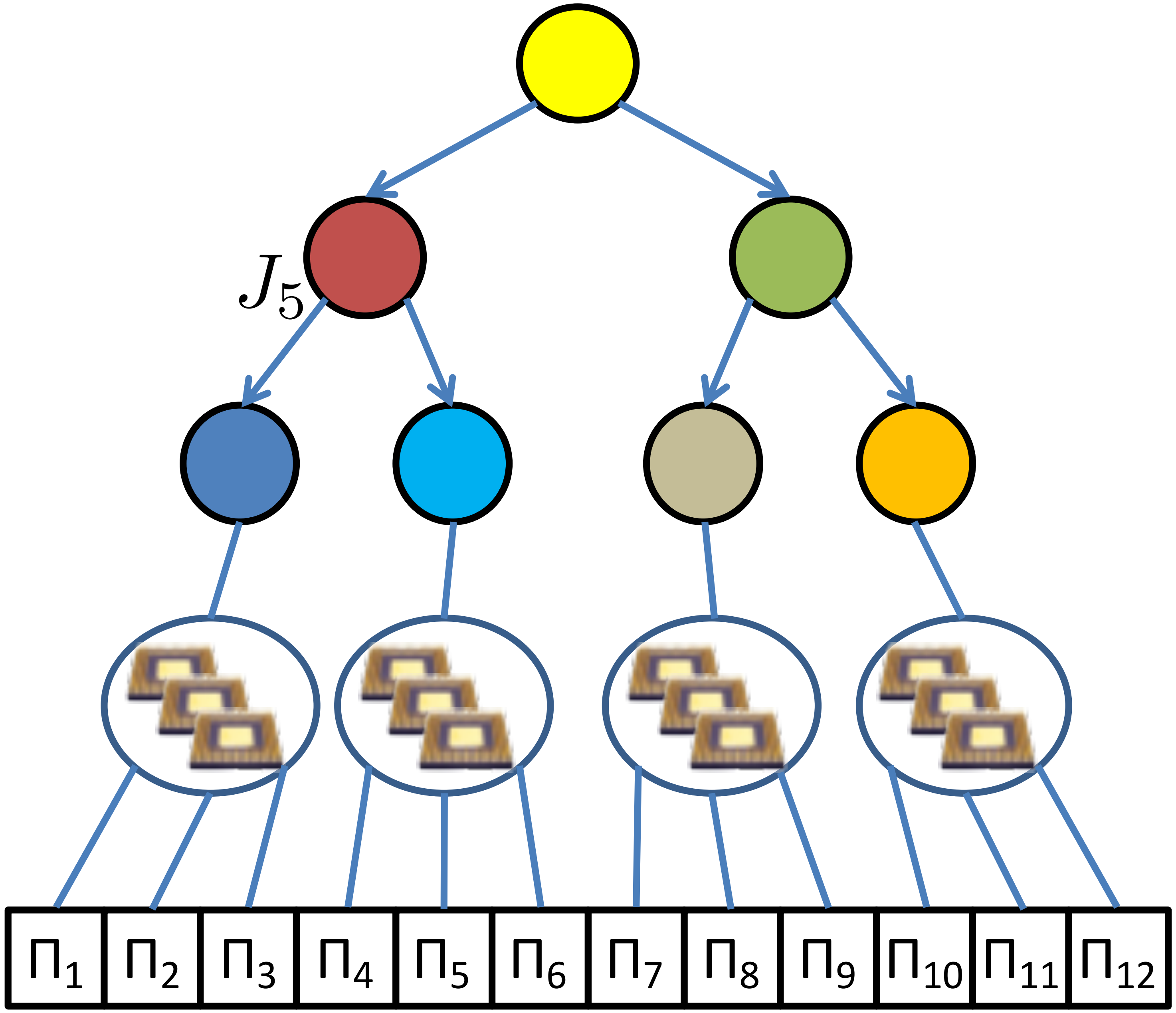


run arrival procedure for each such job (in any order)

[*but don't modify backlogged heap*]

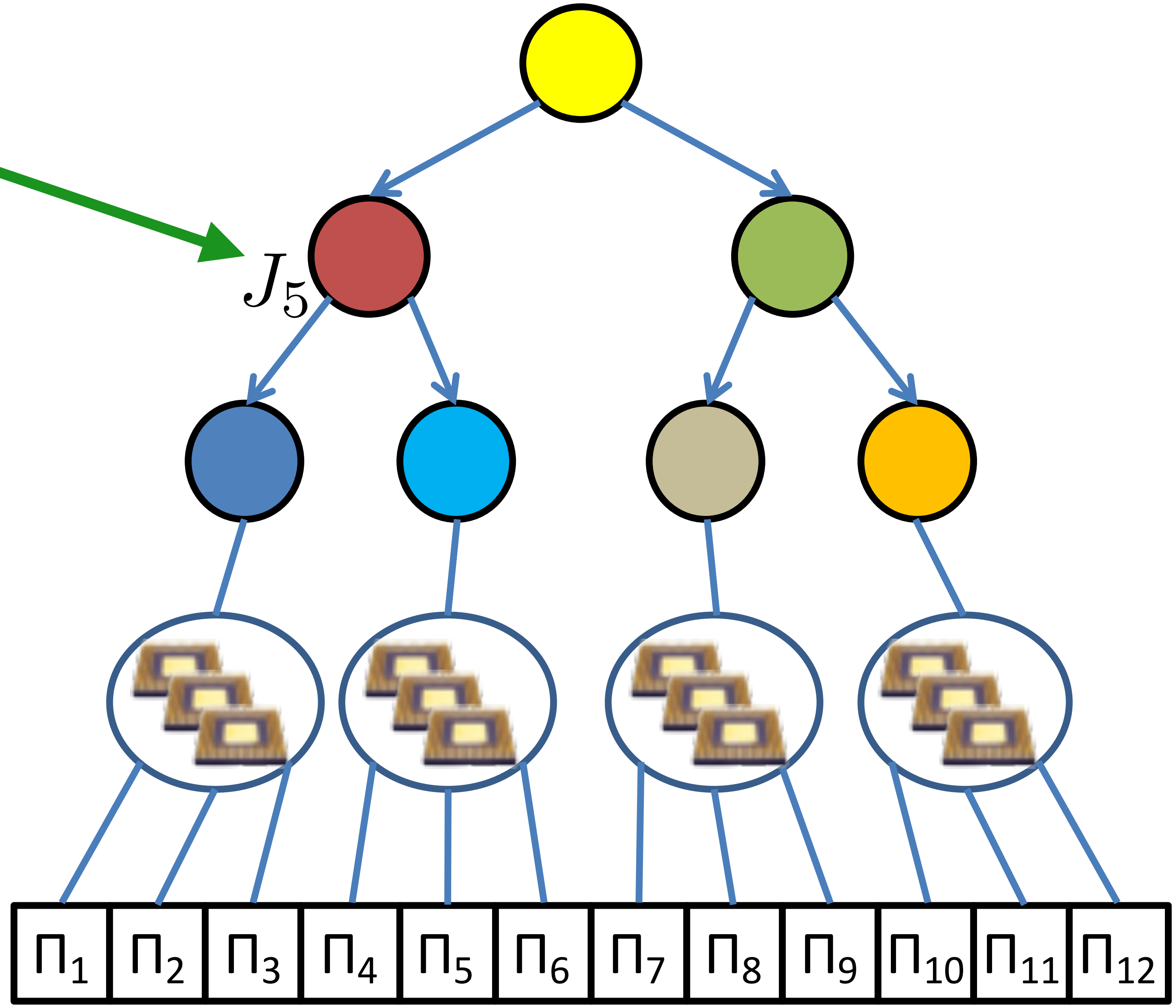
find **highest-priority backlogged job** in each distinct affinity (Fibonacci Heap)

Job Departure Step 4: Remove from Backlogged Heap

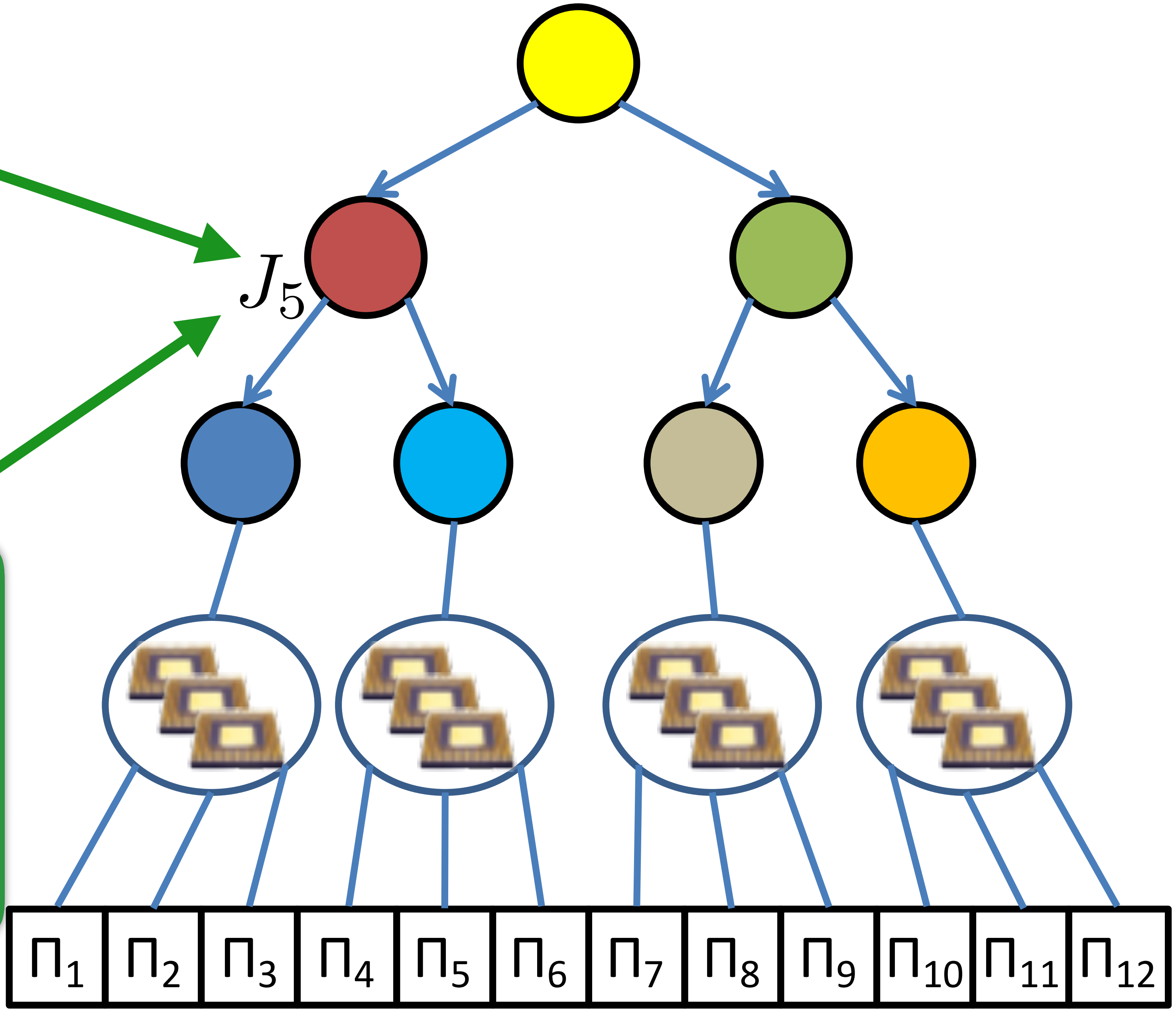


Job Departure Step 4: Remove from Backlogged Heap

at most **one job** will effectively be **added to list of scheduled jobs**



Job Departure Step 4: Remove from Backlogged Heap



at most **one job** will effectively be **added to list of scheduled jobs**

remove this job from the heap of **backlogged jobs**

Complexity of Job Departure: $\boldsymbol{O(\log n + m^2)}$

n…number of tasks

m…number of cores

Complexity of Job Departure: $O(\log n + m^2)$

1. Walk up the tree and remove departing job from lists: **O(*height of tree*) = O(*m*)**

n…number of tasks

m…number of cores

Complexity of Job Departure: $O(\log n + m^2)$

1. Walk up the tree and remove departing job from lists: **O(*height of tree*) = O(*m*)**

2. Find highest-priority backlogged job in each affinity: **O(*#distinct affinities*) = O(*m*)**

n…number of tasks

m…number of cores

Complexity of Job Departure: $O(\log n + m^2)$

1. Walk up the tree and remove departing job from lists: **O(*height of tree*) = O(*m*)**

2. Find highest-priority backlogged job in each affinity: **O(*#distinct affinities*) = O(*m*)**

3. Simulate arrivals: **O(*#distinct affinities x m*) = O(m^2)**

n…number of tasks

m…number of cores

Complexity of Job Departure: $O(\log n + m^2)$

1. Walk up the tree and remove departing job from lists: **O(*height of tree*) = O(*m*)**

2. Find highest-priority backlogged job in each affinity: **O(*#distinct affinities*) = O(*m*)**

3. Simulate arrivals: **O(*#distinct affinities x m*) = O(m^2)**

4. Remove from backlogged: **O(*log n*)**

n…number of tasks

m…number of cores