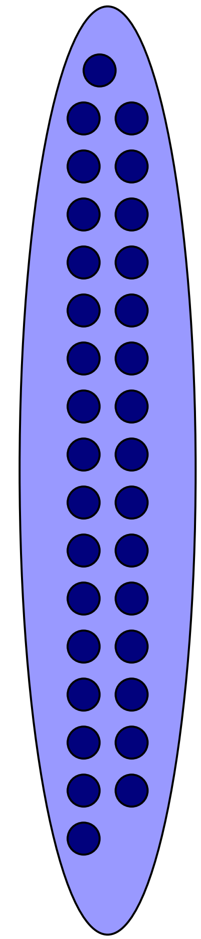
Dynamic Processor Allocation for Adaptively Parallel Jobs

What is the problem?

```
[kunal@ygg ~]$ ./strassen --nproc 4
```

```
[sidsen@ygg ~]$ ./nfib –nproc 32
```

```
[bradley@ygg ~]$ ./nfib --nproc 16
```



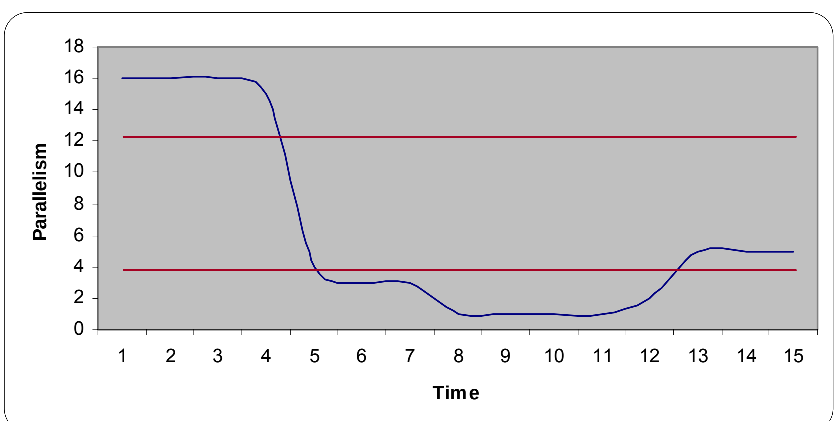
Allocate the processors fairly and efficiently

Why so Dynamic Scheduling?

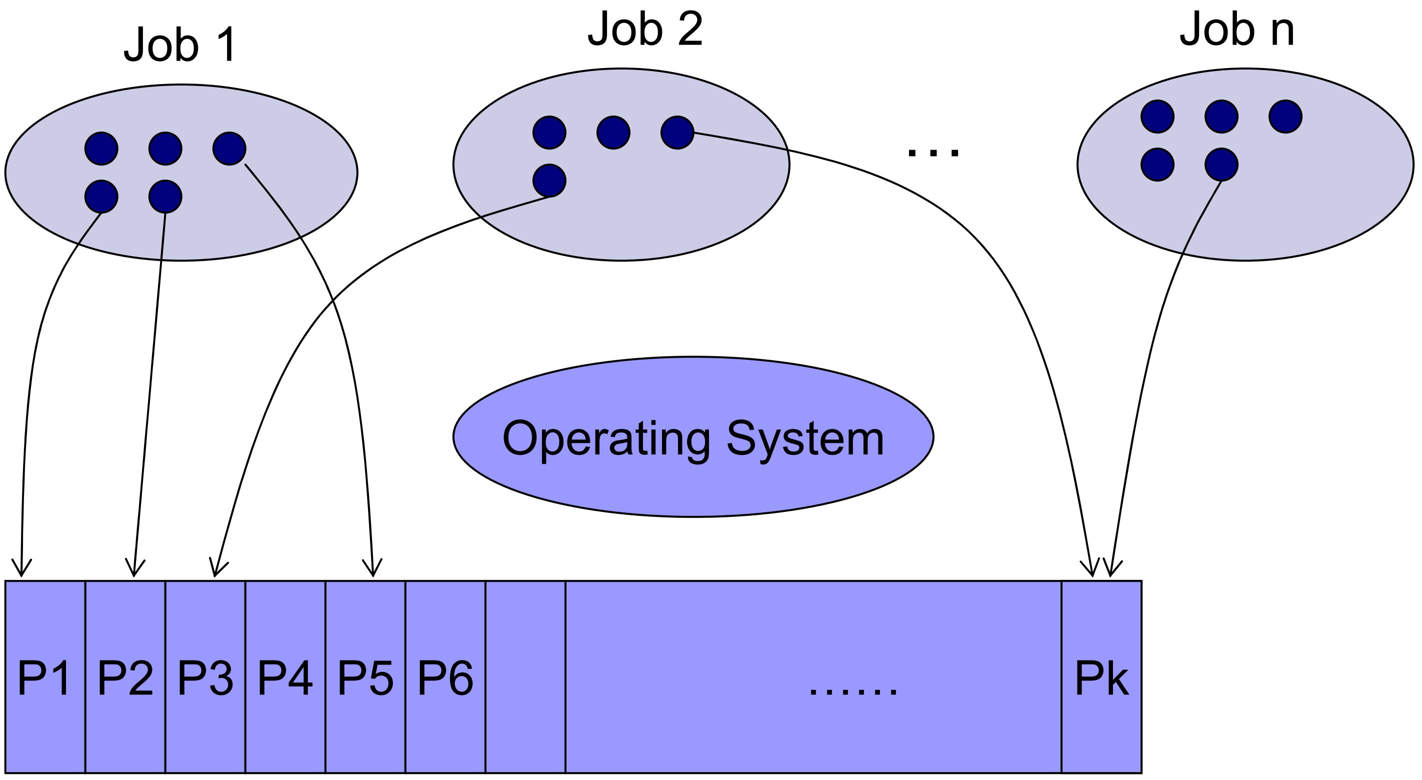
- Considers all the jobs in the system.
- Programmer doesn't have to specify the number of processors.

```
[kunal@ygg ~]$ ./strassen    --nproc 4
```

- Parallelism can change during execution.



Allocation vs. Scheduling



Terminology

- The parallelism of a job is dynamic
  - *adaptively parallel jobs*—jobs for which the number of processors that can be used without waste varies during execution.
- At any given time, each job *j* has a
  - *desire*—the maximum number of efficiently usable processors, or the parallelism of the job (d_j).
  - *allocation*—the number of processors allotted to the job (a_j).

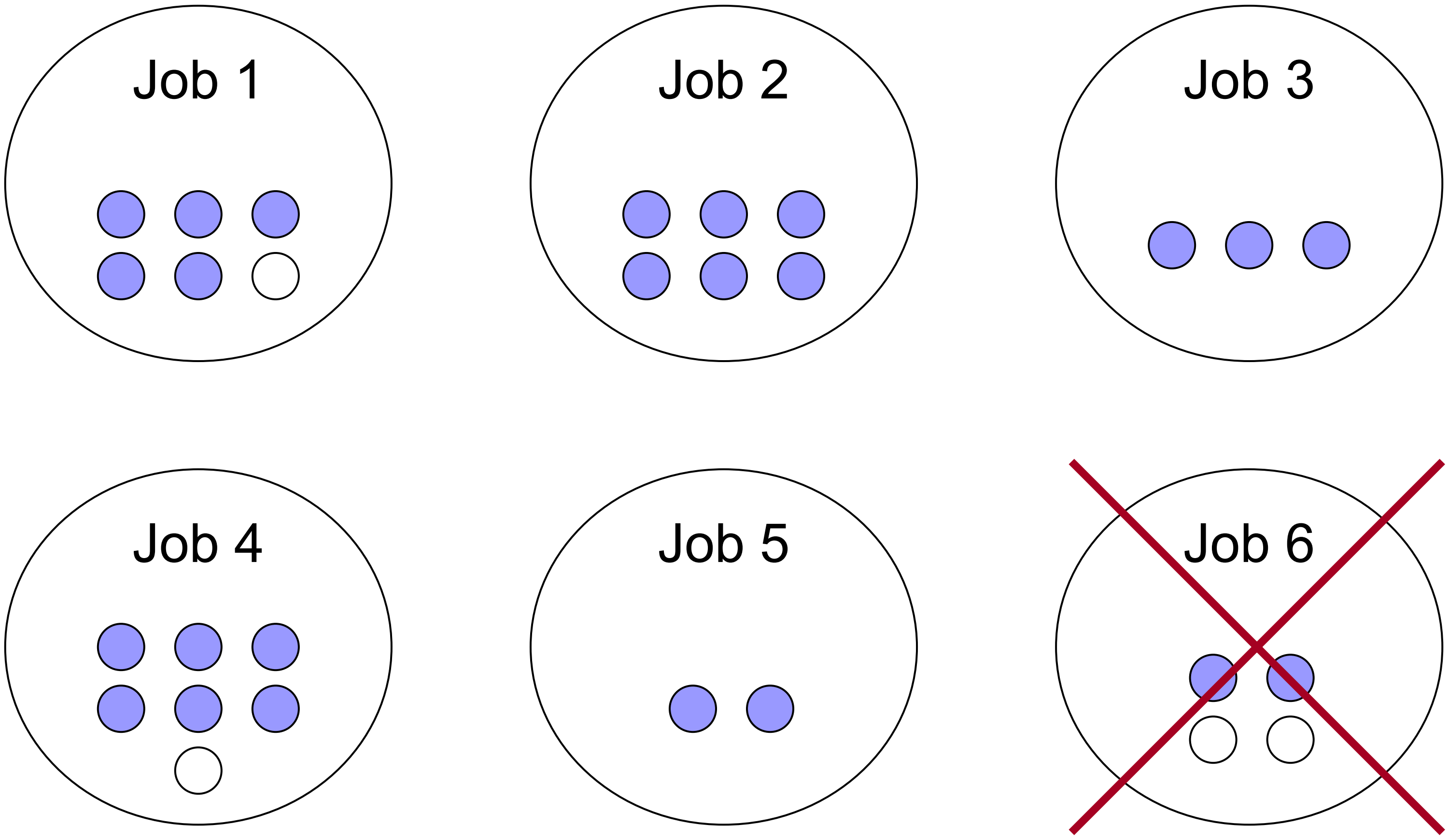
Terminology

- We want to allocate processors to jobs in a way that is
  - *fair*—whenever a job receives fewer processors than it desires, all other jobs receive at most one more processor than this job received.
    - $a_j < d_j \Rightarrow (a_j + 1)$ is a max
  - *efficient*—no job receives more processors than it desires, and we use as many processors as possible.
    - $\forall j\ a_j \leq d_j$
    - $\exists j\ a_j < d_j \Rightarrow$ there are no free processors

Overall Goal

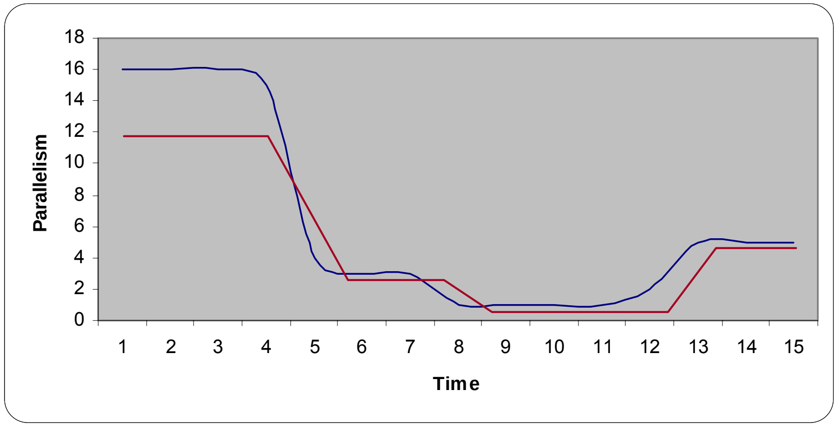
Design and implement a *fair* and *efficient* dynamic processor *allocation* system for *adaptively parallel jobs*.

Example: Fair and Efficient Allocation

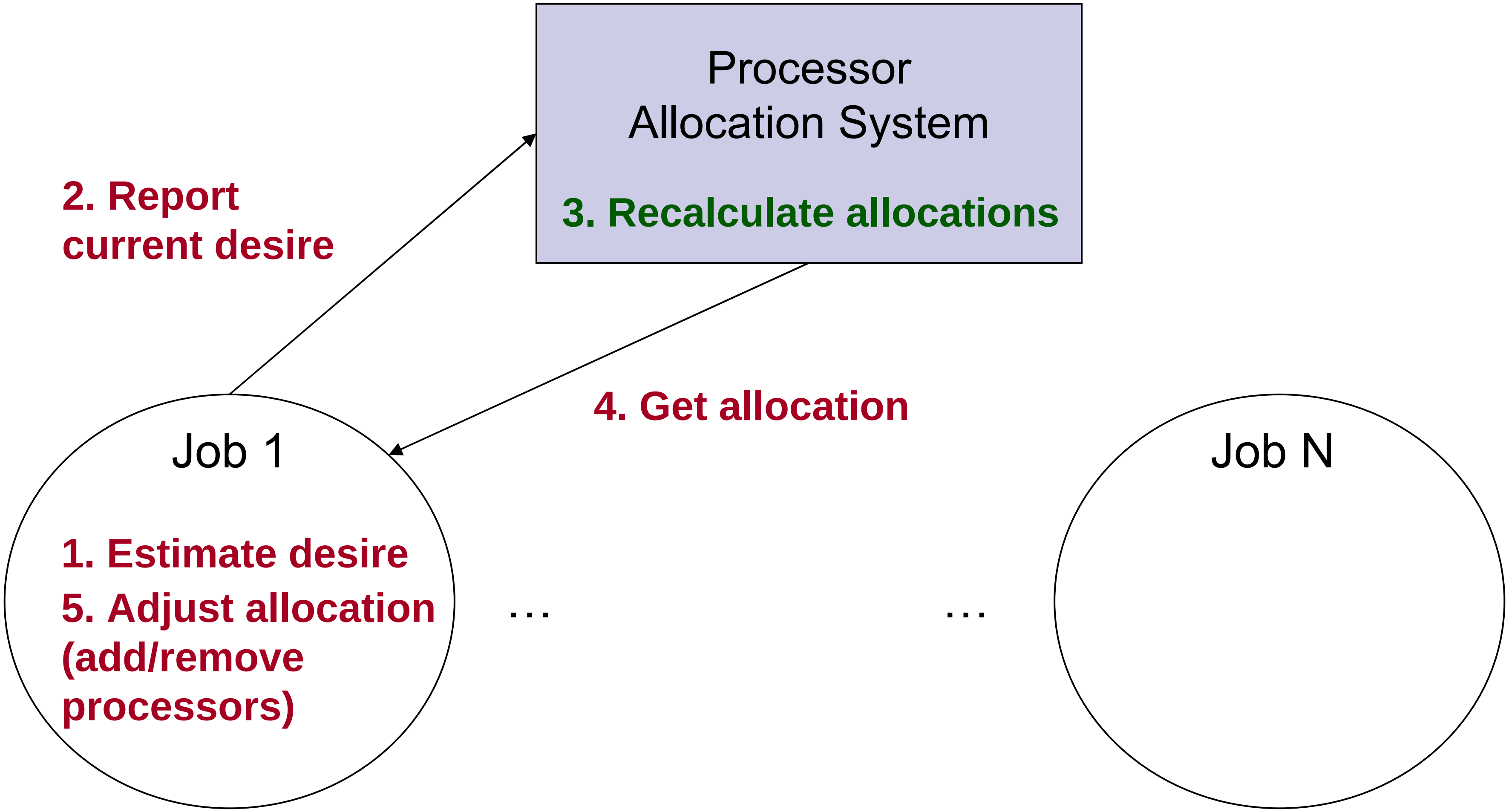


Assumptions

- All jobs are Cilk jobs.
- Jobs can enter and leave the system at will.
- All jobs are mutually trusting, in that they will
  - stay within the bounds of their allocations.
  - communicate their desires honestly.
- Each job has at least one processor.
- Jobs have some amount of time to reach their allocations.

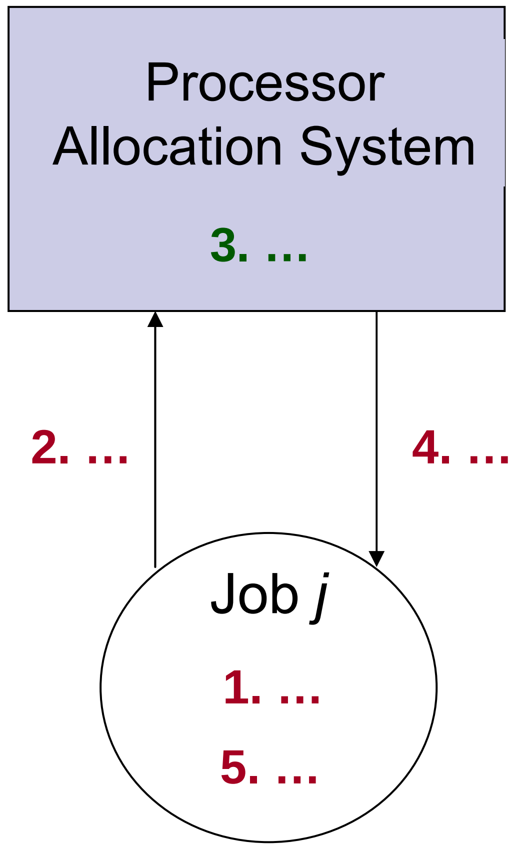


High-Level Sequence of Events



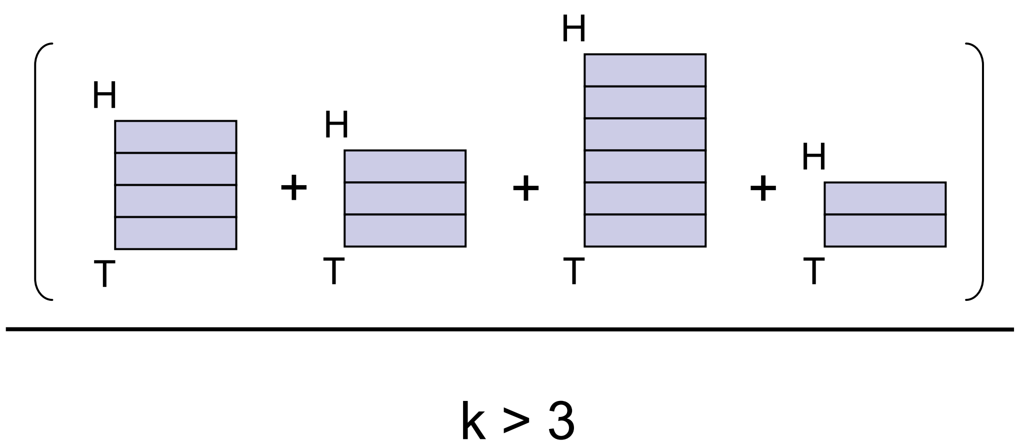
Main Algorithms

- (1, 2) Dynamically estimate the current desire of a job.
  - ☐ Steal rate (Bin Song)
  - ☑ Number of threads in ready deque
- (3) Dynamically determine the allotment for each job such that the resulting allocation is fair and efficient.
  - ☐ SRLBA algorithm (Bin Song)
  - ☑ Global allocation algorithm
- (4, 5) Converge to the granted allocation by increasing/decreasing number of processors in use.
  - ☑ While work-stealing?
  - ☑ Periodically by a background thread?

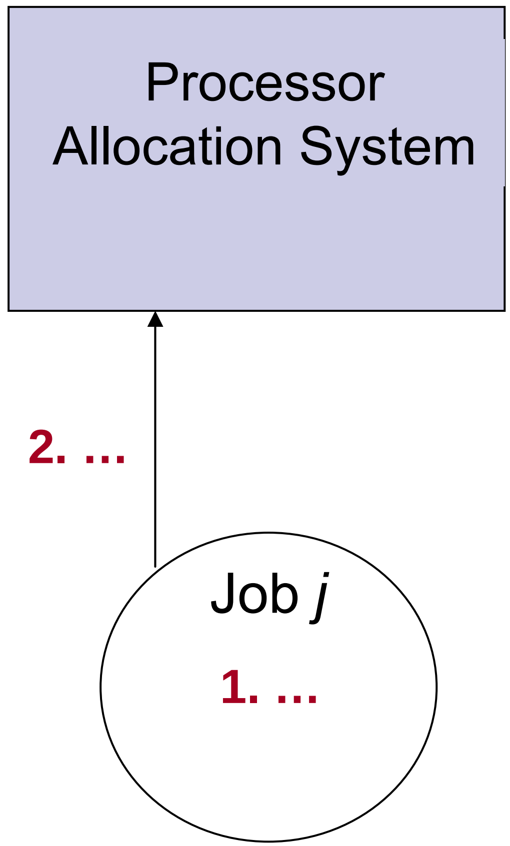


Desire Estimation

- (1) Estimate processor desire d_j: add up the number of threads in the ready deques of each processor and divide by a constant.

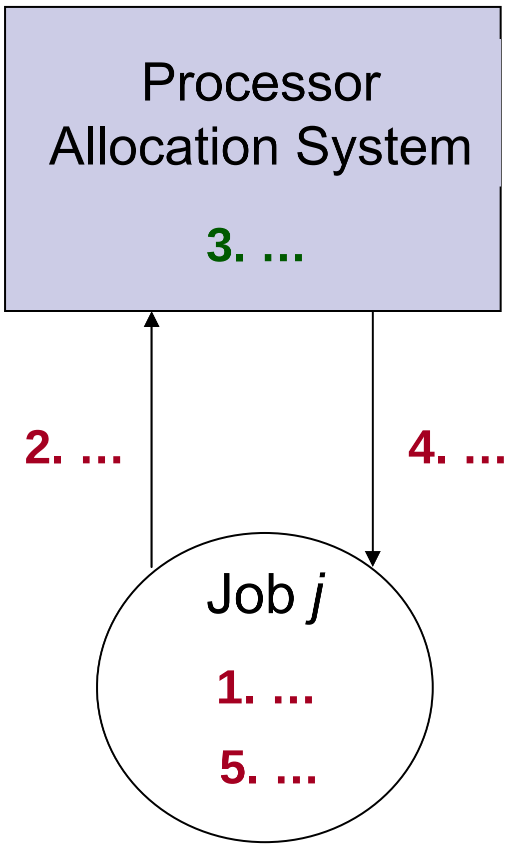


- (2) Report the desire to the processor allocation system.



Adjusting the Allocation

- (4) Get the allocation a_{new}.
- (5) Adjust the allocation.
  - If $a_{new} < a_{old}$, remove $(a_{old} - a_{new})$ processors
  - If $a_{new} > a_{old}$, add $(a_{new} - a_{old})$ processors

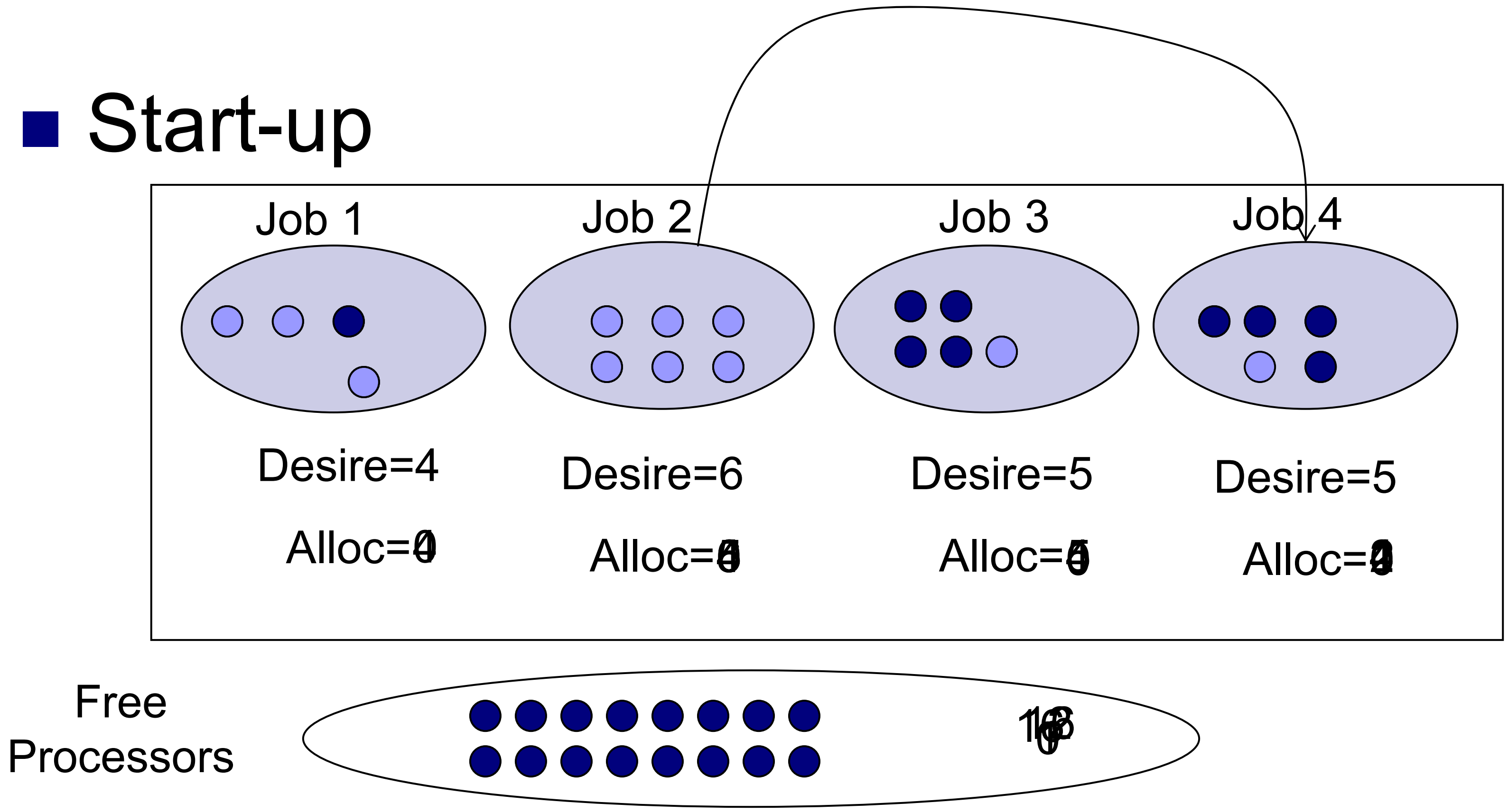


Implementation Details

- Adding up the number of threads in the ready deques
  - ☒ While work-stealing — Too late!
  - ☑ Periodically by a background thread
- Removing processors
  - ☑ While work-stealing
  - ☒ Periodically by a background thread — Complicated
- Adding processors
  - ☒ While work-stealing — Bad idea
  - ☑ Periodically by a background thread

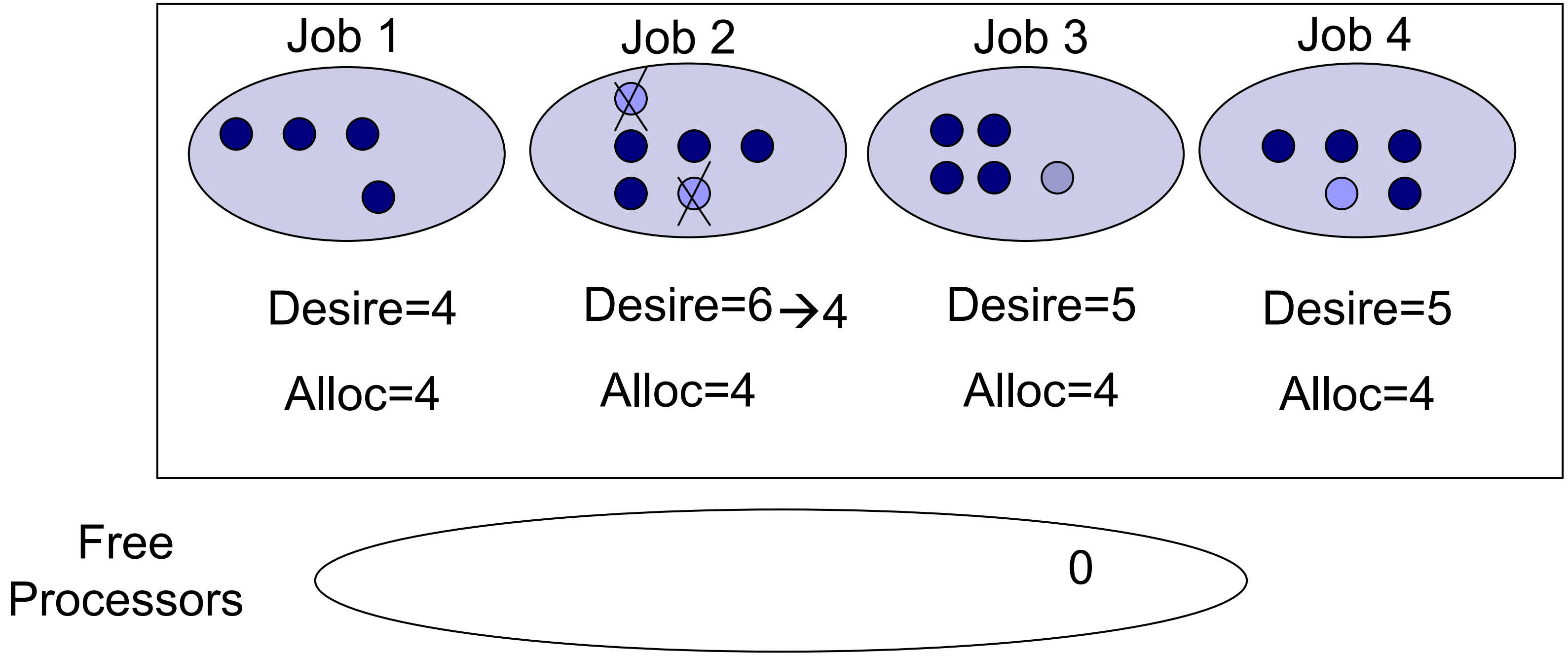
Processor Allocation

- Start-up



Processor Allocation

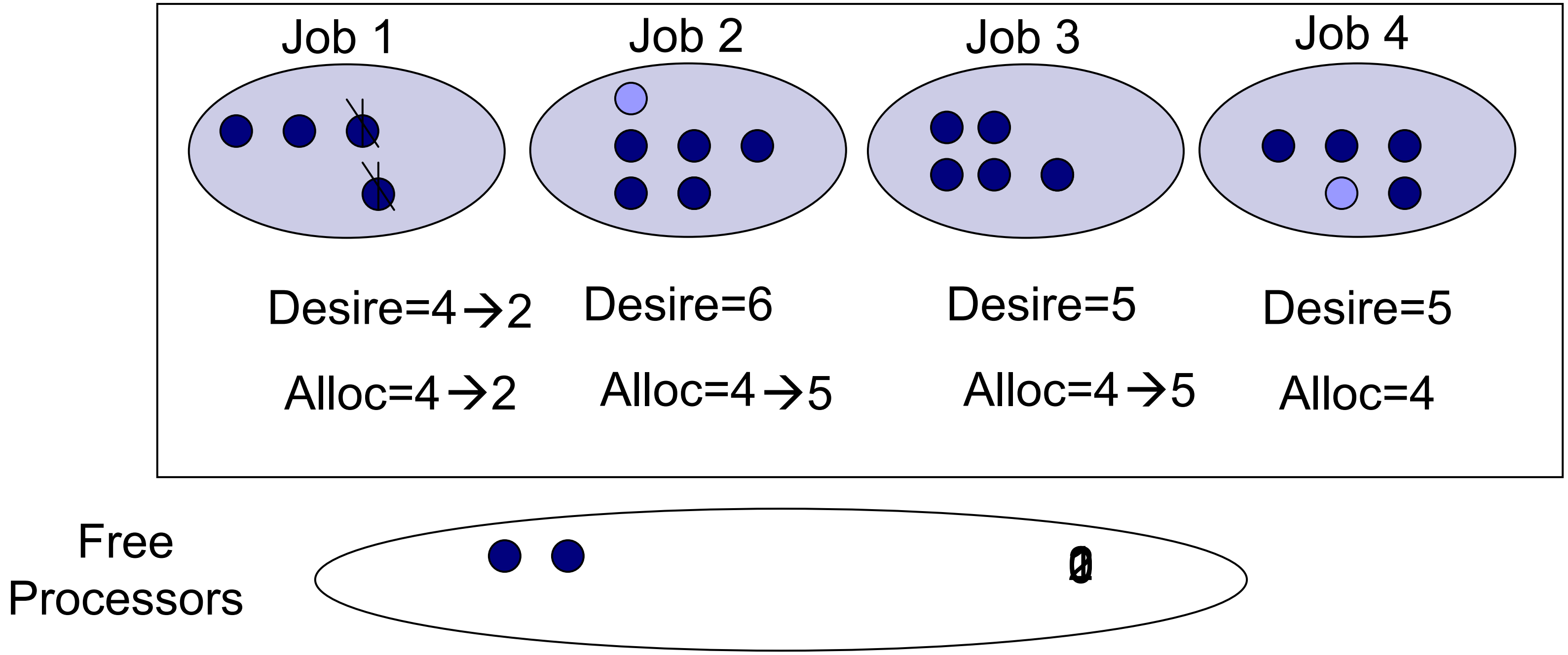
- Job 2 *decreases* desire.



No Reallocation !!

Processor Allocation

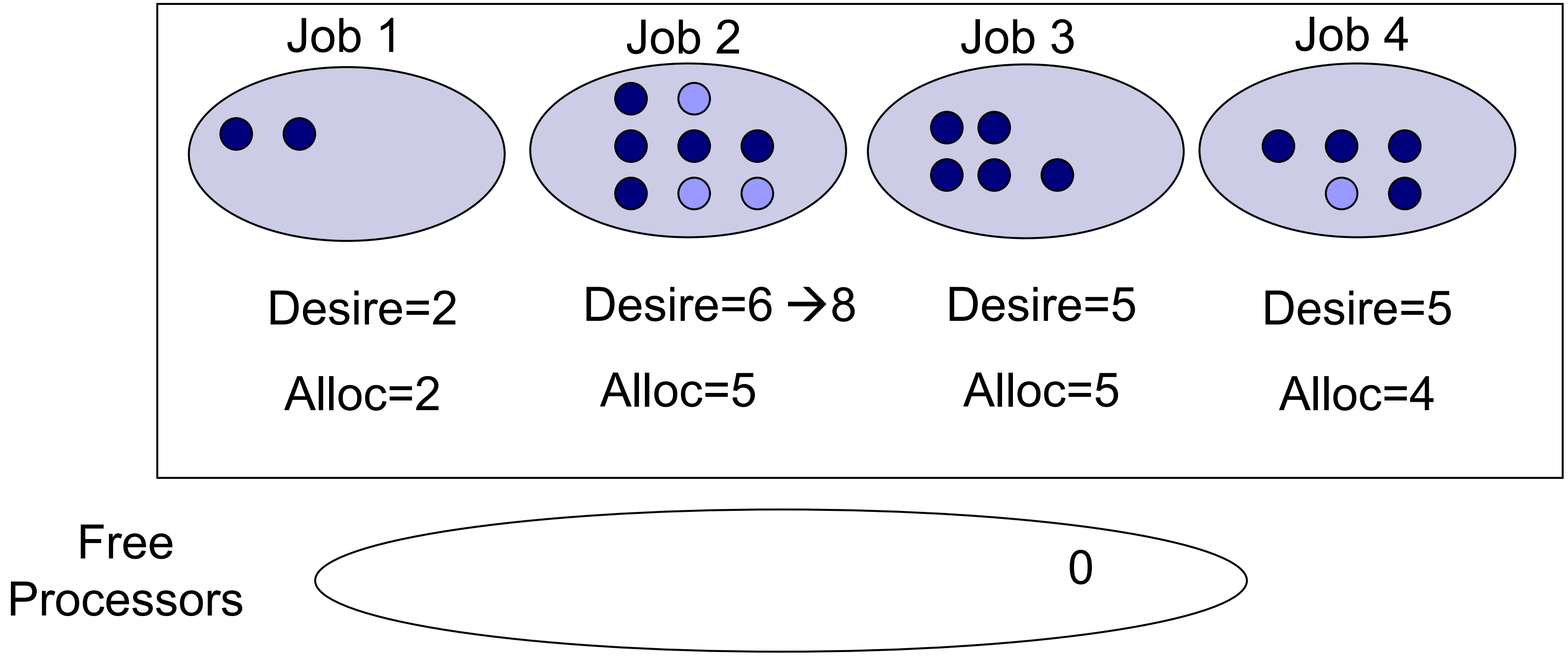
- Job 1 *decreases* desire.



Reallocate !!

Processor Allocation

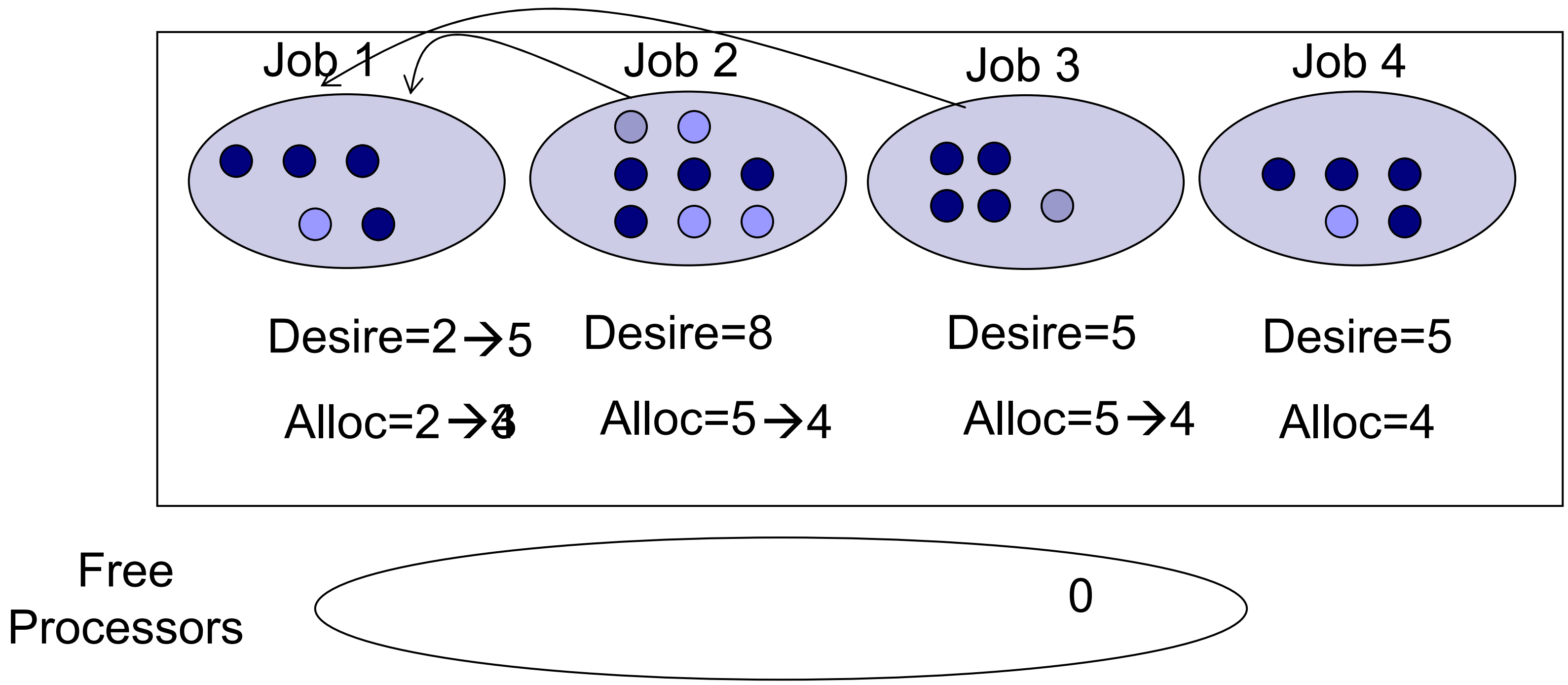
- Job 2 *Increases* desire.



No Reallocation !!

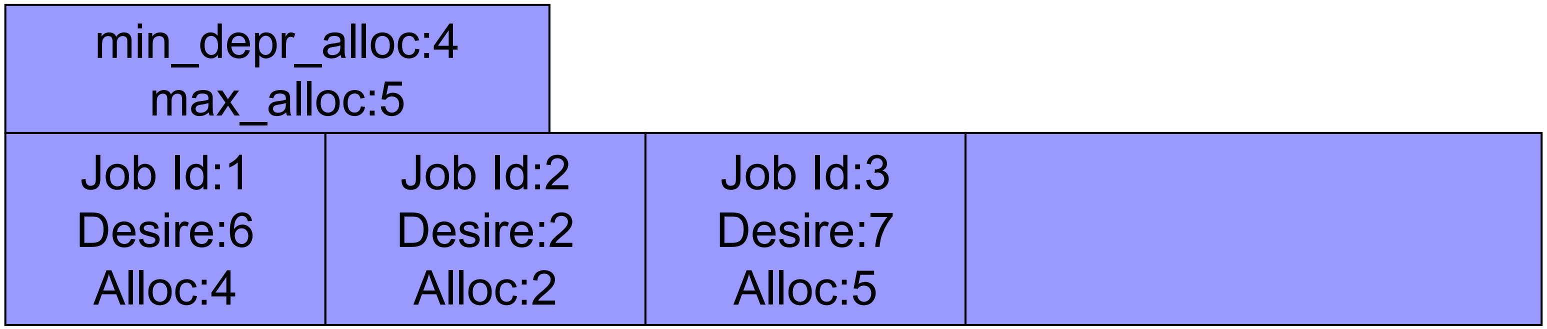
Processor Allocation

- Job 1 *Increases* desire.



Reallocate !!

Implementation Details



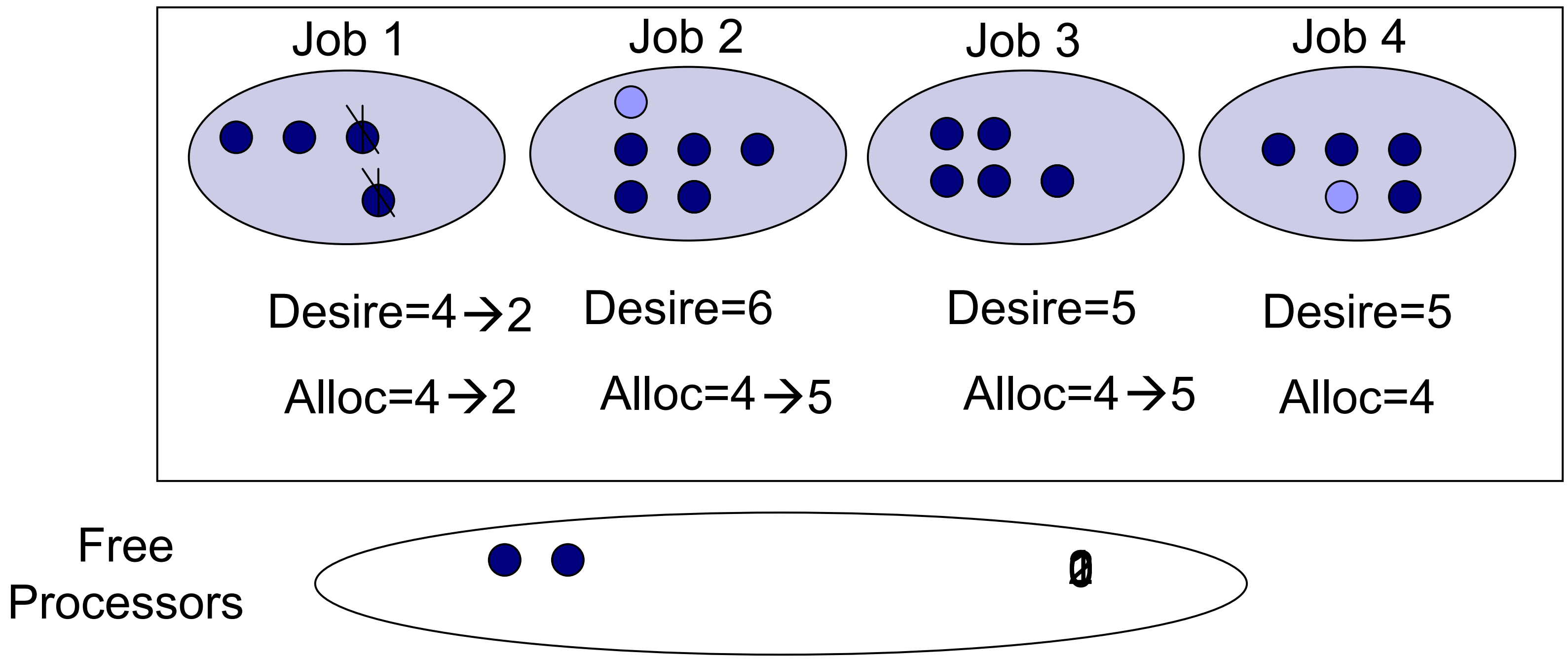
- When desire of job j *decreases*: if (*new_desire<alloc*)
  - take processors from *j* and give to jobs having *min_depr_alloc*.

Processor Allocation

mda=4

ma=5

- Job 1 *decreases* desire.



Implementation

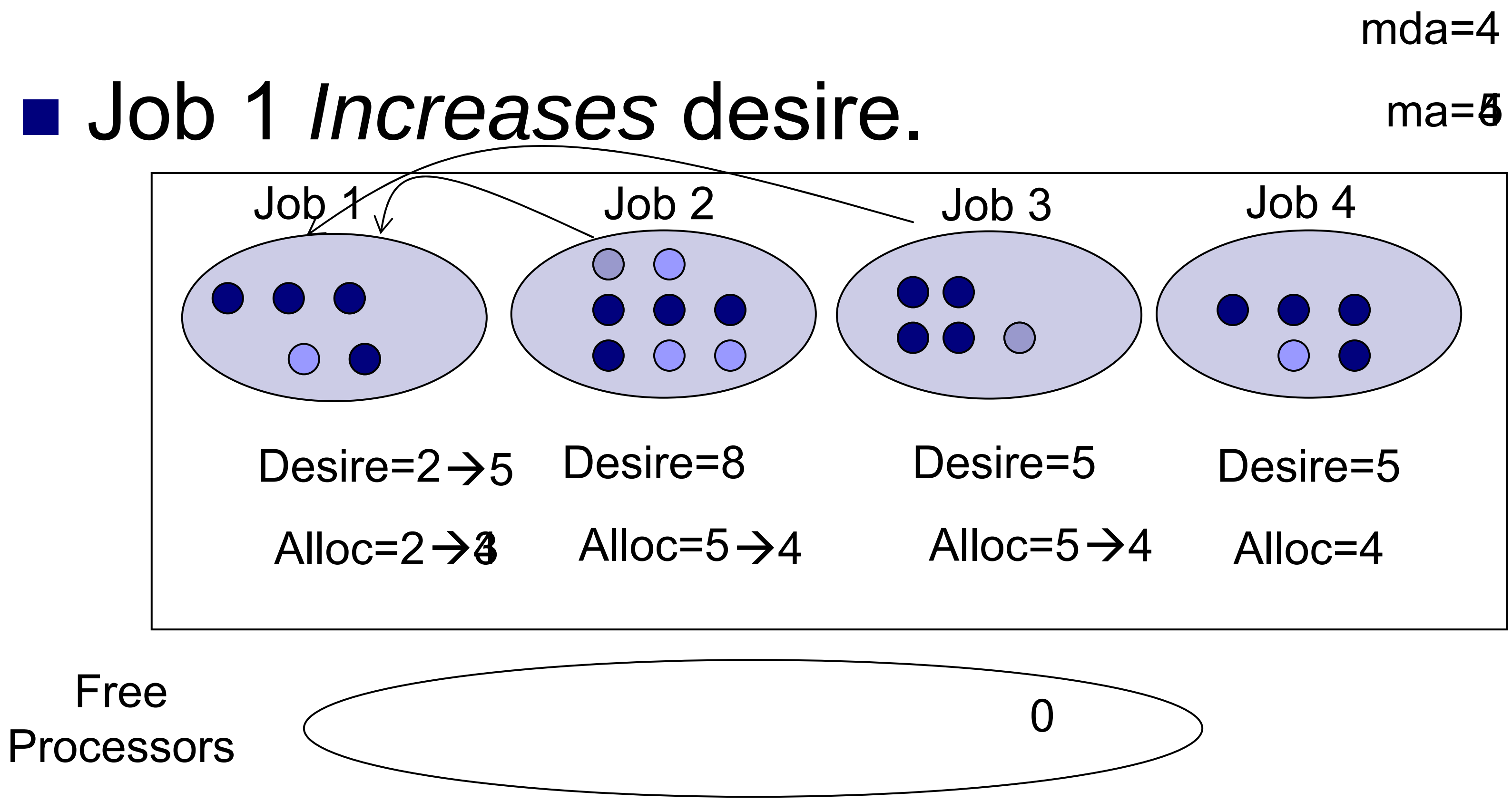
min_depr_alloc:4
max_alloc:5

Job Id:1	Job Id:2	Job Id:3	
Desire:6	Desire:2	Desire:7	
Alloc:4	Alloc:2	Alloc:5	

- When desire of job j *decreases*: if (*new_desire<alloc*)
  - take processors from *j* and give to jobs having *min_depr_alloc*.
- When desire of job j *increases*: if (alloc<*mda*)
  - take processors from jobs having *max_alloc* and give them to *j* until j reaches *min_depr_alloc* or *new_desire*.

Processor Allocation

- Job 1 *Increases* desire.



Experiments

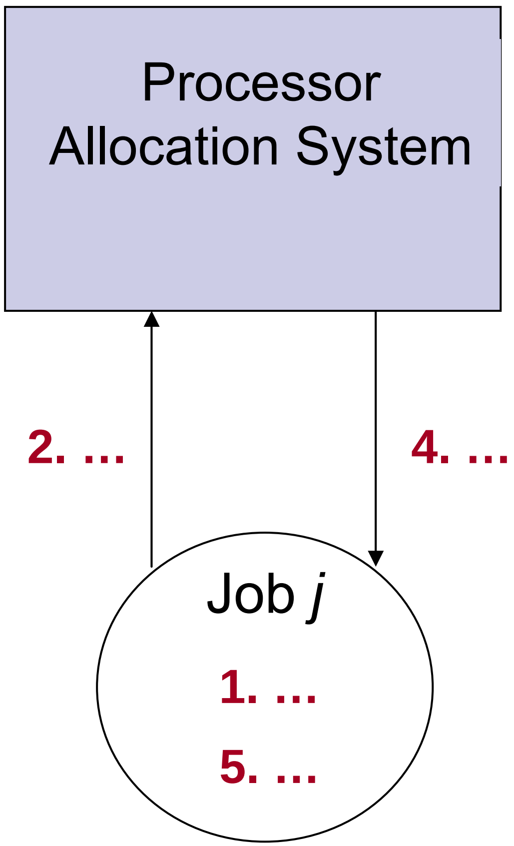
- Correctness: Does it work?
- Effectiveness: Are there cases where it is better than the static allocation?
- Responsiveness: How long does it take the jobs to reach their allocation?

Conclusions

- The desire estimation and processor allocation algorithms are simple and easy to implement.
- We'll see how well they do in practice once we've performed the experiments.
- There are many ways of improving the algorithms and in many cases it is not clear what we should do.

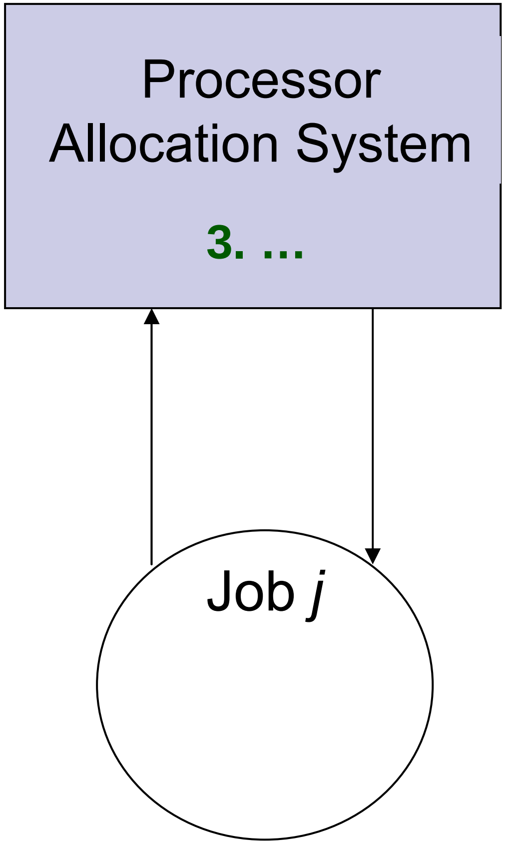
Job Tasks (Extensions)

- Incorporate heuristics on steal-rate (Bin Song's idea).
- Remove processors in the background thread, not while work stealing.
  - Need a mechanism for putting processors with pending work to sleep
  - When adding processors, wake up processors with pending work first



Processor Allocation System (Extensions)

- Use a sorted data structure for job entries.
  - Sort by desires
  - Sort by allocations
  - Group jobs:
    - Desires satisfied ($a_j = d_j$)
    - Minimum deprived allocation (a_j = min_depr_alloc)
    - Maximum allocation (a_j = max_alloc)
- Need fast inserts/deletes and fast sequential walk.



Processor Allocation System (Extensions)

- Rethink definitions of fairness and efficiency.
  - Incorporate histories of processor usage for each job
  - Implement a mechanism for assigning different priorities to users or jobs
- Move the processor allocation system into the kernel.
  - Jobs still report desires since they know best
  - How to group the jobs?
    - Make classes of jobs (Cilk, Emacs, etc.)
    - Group by user (sidsen, kunal, etc.)

Questions?