Scheduling in Linux – Part 1

Acknowledgement

The example of O(1) scheduler is borrowed from the slides of the same course offered by Prof. Sandip Chakraborty in earlier years (very slight changes done)

The materials for some of the other slides are borrowed from the same source

Types of Tasks

- Interactive
  - Requires fast response time
  - May not require CPU for long durations, but when it needs the CPU, it should be given asap
    - So requires CPU in small bursts
  - Ex: GUI tasks, Word processing, making a powerpoint slide,…
- Batch
  - Requires long CPU times, but response time is not very important
  - Throughput is more important
  - Ex: scientific computations, …
- Real Time
  - Required to be completed within some time

Goals of a Scheduler

- Real time tasks should have higher priority over other tasks
- A higher priority job should run as soon as possible
- Lower priority jobs should not be starved by higher priority jobs
- Interactive tasks should have fast response time and should not be preempted while running
- Context switches should be reduced

Process Priorities in Linux

- 0-99: real time tasks
  - Higher value means higher priority
- 100 – 139: non-real time tasks
  - Actually processes get a nice value between -20 to +19
    - -20 maps to 100
    - +19 maps to 139
    - Higher nice value means lower priority (you are being "nice" to other processes ☺)
  - Default nice value of a process is 0
    - Maps to 120
- Thus, complete internal range of priority values of Linux is 0-139

Scheduling Classes

- Every process is attached to a scheduling class
- Five scheduling classes (in order of lower to higher priority)
  - *Idle* (/kernel/sched/idle.c)
  - *Fair* (/kernel/sched/fair.c)
  - *Real time* (/kernel/sched/rt.c)
  - *Deadline* (/kernel/sched/deadline.c)
- A task in *rt* class will always preempt a task in *fair* class, which will always preempt the *idle* task etc.
- There is also a *stop* class in the list of scheduling classes for use in stopping the cpu for some specific cases (highest priority class)

Scheduling Policies

- Every class has one or more policies associated with it
  - For *idle* class
    - SCHED_IDLE
      - For some very low priority background processes
  - For *fair* class
    - SCHED_OTHER/SCHED_NORMAL
    - SCHED_BATCH
  - For *real time* class
    - SCHED_FIFO
    - SCHED_RR
  - For *deadline* class
    - SCHED_DEADLINE

- Scheduling policies have associated algorithms
  - Ex: for fair class, SCHED_NORMAL policy, completely fair scheduler (CFS) is the algorithm

- We will study SCHED_NORMAL only in detail
- Will come back and talk about the other scheduling classes and policies a bit at the end

Older Linux Schedulers

Genesis (1991)

- Kernel version 0.01
- A single queue of runnable processes, default is 32 process
- The scheduler iterates over the entire queue to select a task to run
  - Check if any alarm is raised for a task, if yes, mark for processing
    - Also move the tasks from waiting to running state if alarm raised
  - Find the task with the largest unused timeslice and schedule it
  - If no such process
    - Assign all processes new timeslice values based on priority
      - Higher priority gets larger timeslice
    - Schedule the one with the largest timeslice
- Very simple, but O(n)
  - Did not scale as systems became more powerful and complex

```c
void schedule(void) {
int i,next,c;
struct task_struct ** p;

/* check alarm, wake up any interruptible tasks
that have got a signal */
for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
  if (*p) {
     if ((*p)->alarm && (*p)->alarm < jiffies) {
        (*p)->signal |= (1<<(SIGALRM-1));
        (*p)->alarm = 0;
     }
     if ((*p)->signal && (*p)->state==TASK_INTERRUPTIBLE)
        (*p)->state=TASK_RUNNING;
 }

```

```
   /* this is the scheduler proper: */
   while (1) {
      c = -1;
      next = 0;
      i = NR_TASKS;
      p = &task[NR_TASKS];
      while (--i) {
         if (!*--p)
            continue;
         if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
            c = (*p)->counter, next = i;
      }
      if (c) break;
      for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
         if (*p)
            (*p)->counter = ((*p)->counter >> 1) + (*p)->priority;
   }
   switch_to(next);
}
```

- From comments in Genesis schedule() function ☺

"*'schedule()' is the scheduler function. This is GOOD CODE! There probably won't be any reason to change this, as it should work well in all circumstances (ie gives IO-bound processes good response etc)…*".

O(N) Scheduler

- From Kernel versions 2.4 onwards, till before 2.6
- Similar to the Genesis scheduler
- Main change is in the metric used for selecting the next process – *Goodness* of a process
- *Goodness* of a process is calculated as the number of clock-ticks a task had left plus some weight based on the task's priority; returns integer values
  - -1000: Never select this task to run
  - positive number: The goodness value, larger the better
  - +1000: A real time process

- No preemption of running process
  - So a real time task coming cannot preempt a simple user process
- Has the same problem of scalability
  - Needs to loop through all processes
  - Goodness computations were costly
  - Single global queue does not scale for multiprocessors

O(1) Scheduler

- Introduced in Kernel Version 2.6.0 (2003)
- Introduced
  - The priority scale (0-139) we discussed and the separation between normal and real time tasks
  - Early preemption: A new runnable task of higher priority can preempt the currently running process of lower priority
  - Dynamic priority for considering interactivity
    - Decided based on recent interactivity (how often the process used the CPU in the past)
  - Separate runqueues for each CPU

- Timeslice given for each process
  - For priority < 120, timeslice $= (140 - \text{priority})*20$ milliseconds
    otherwise, timeslice $= (140 - \text{priority})*5$ milliseconds
- Two sets of queues, Active and Expired
- Each set has multiple queues, one for each priority
  - So total 140 queues in each set
- At any point of time, schedule from the active set
- A process moves to the expired set when if it uses up its timeslice
  - Except in some cases, will discuss

Example

- Reorganize the runqueue data structure

Active Array	Expired Array
[0]	[0]
[1]	[1]
[2]	[2]
…	…
[99]	[99]
[100]	[100]
…	…
[139]	[139]

Example

- Reorganize the runqueue data structure

Active Array

[0]
[1]
[2]
…
[99]
[100]
…
[139]

Timeslice available

Expired Array

[0]
[1]
[2]
…
[99]
[100]
…
[139]

Example

- Reorganize the runqueue data structure

Active Array

[0]
[1]
[2]
…
[99]
[100]
…
[139]

Expired Array

[0]
[1]
[2]
…
[99]
[100]
…
[139]

Timeslice complete

Example

- Reorganize the runqueue data structure

Active Array

[0]
[1] 1
[2]
…
[99]
[100]
…
[139]

Expired Array

[0]
[1]
[2]
…
[99]
[100]
…
[139]

Example

- Reorganize the runqueue data structure

Active Array

[0]
[1] 1
[2]
…
[99] 2
[100]
…
[139]

Expired Array

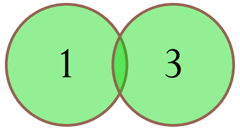
[0]
[1]
[2]
…
[99]
[100]
…
[139]

Example

- Reorganize the runqueue data structure

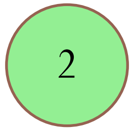
Active Array

[0]

[1] 

[2]

…

[99] 

[100]

…

[139]

Expired Array

[0]

[1]

[2]

…

[99]

[100]

…

[139]

Example

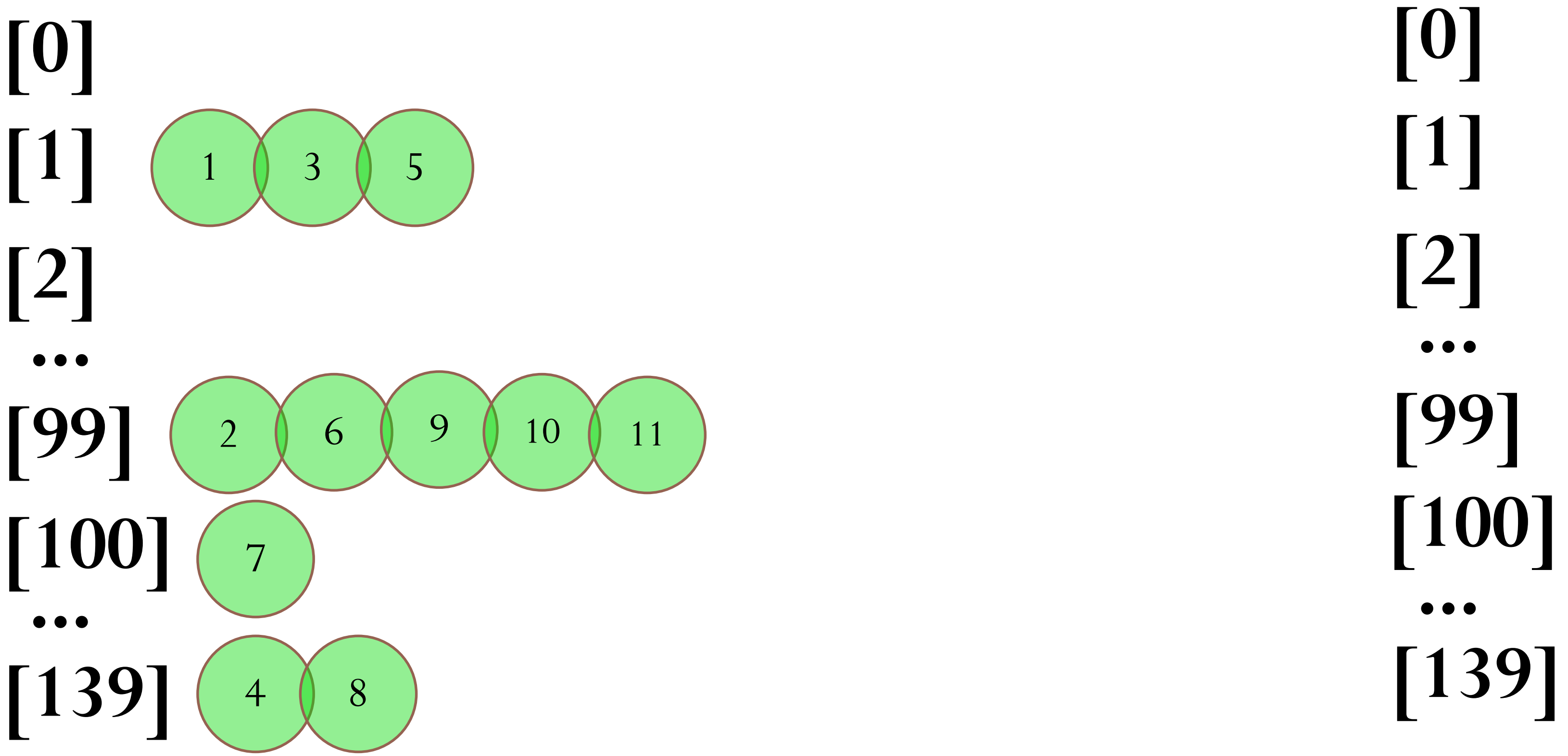
- Reorganize the runqueue data structure

Active Array		Expired Array
[0]		[0]
[1]	1 3	[1]
[2]		[2]
…		…
[99]	2	[99]
[100]		[100]
…		…
[139]	4	[139]

Example

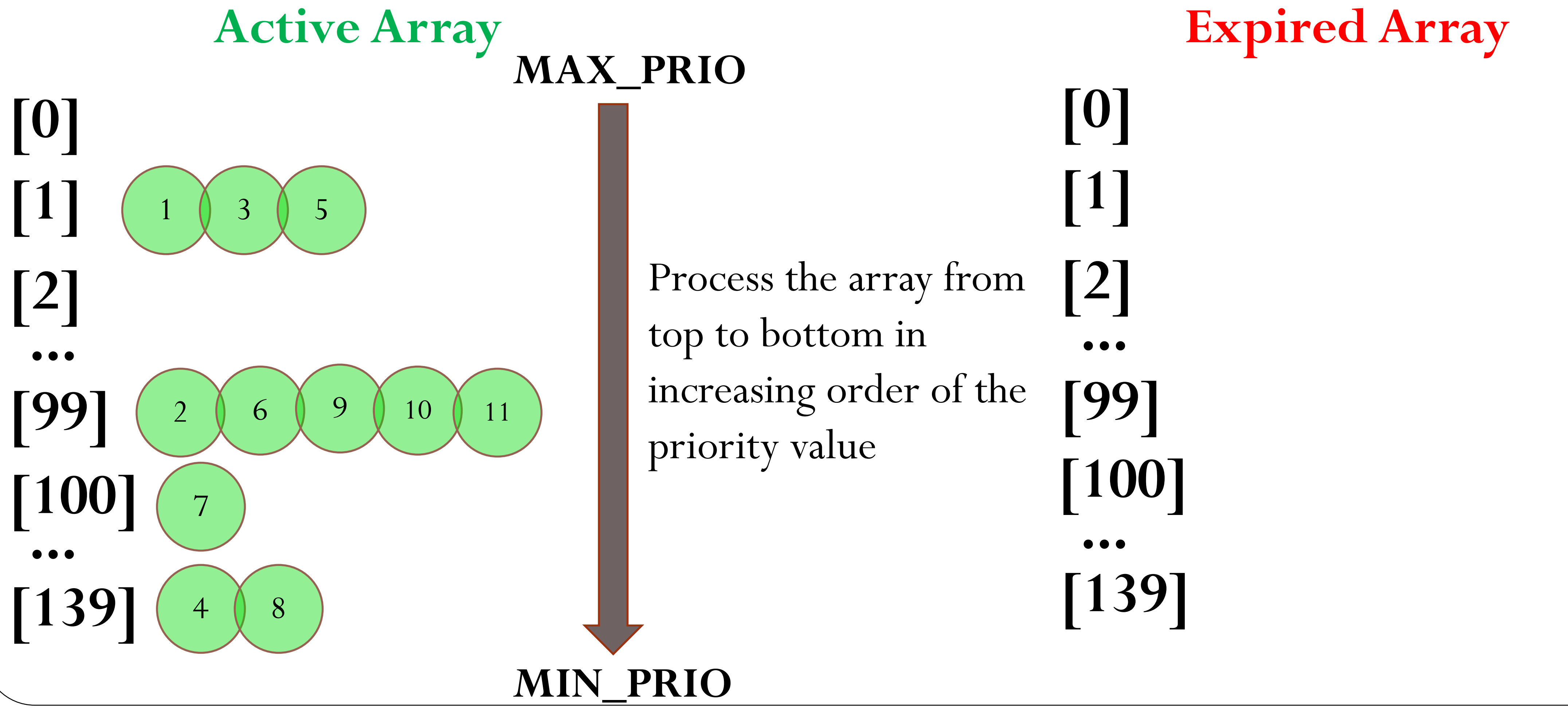
- Reorganize the runqueue data structure

Active Array **Expired Array**



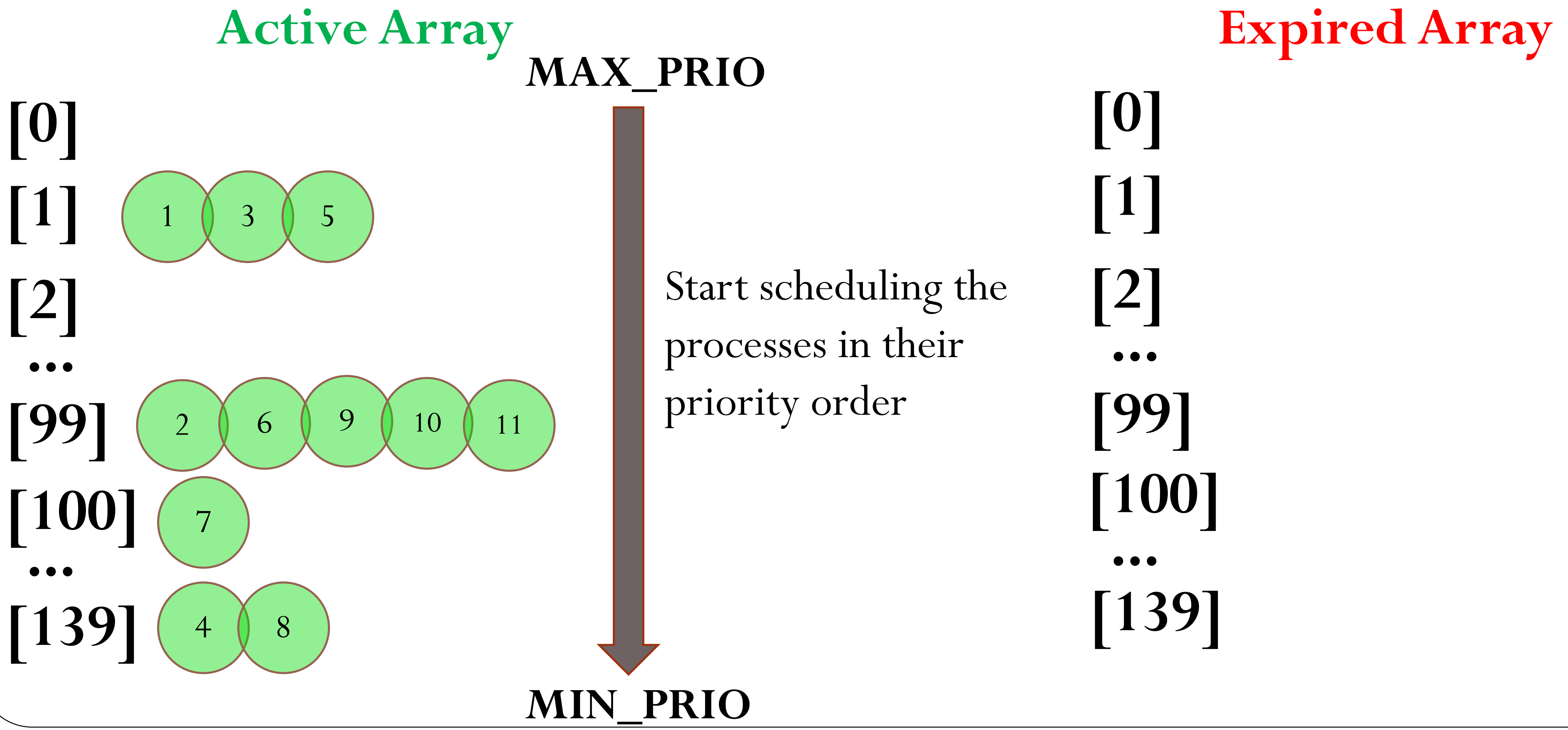
Example

- Reorganize the runqueue data structure



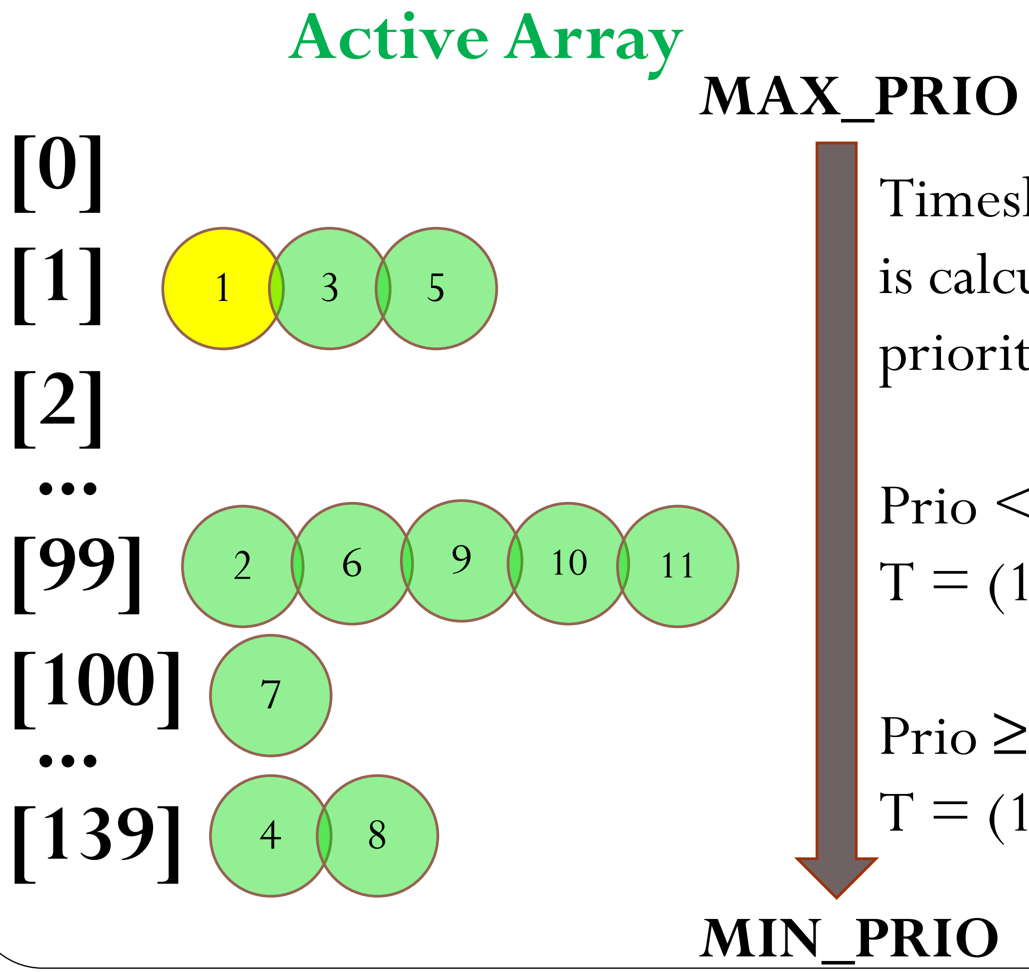
Example

- Reorganize the runqueue data structure



Example

- Reorganize the runqueue data structure



Timeslice for a process is calculated from its priority

Prio < 120
T = (140-Prio)*20

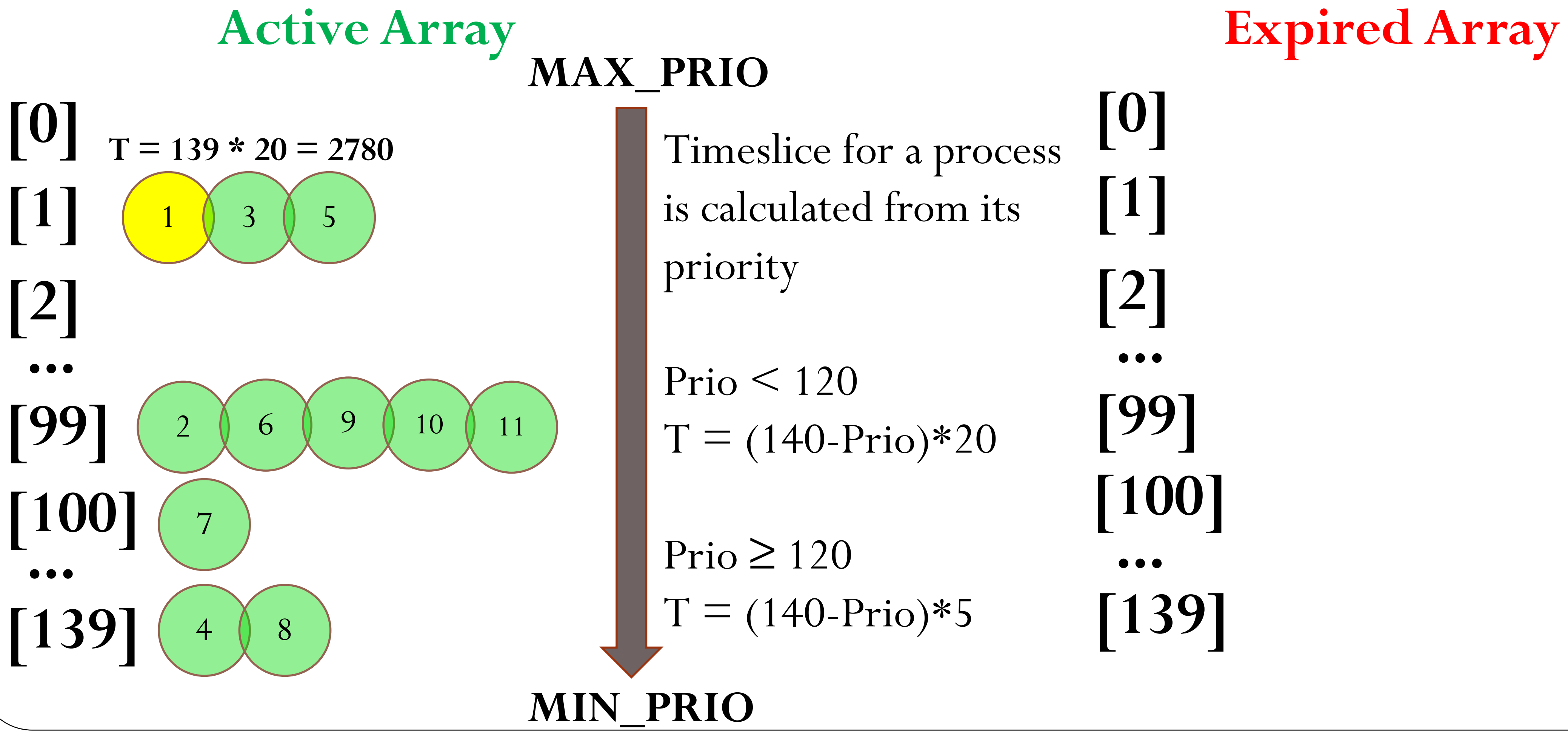
Prio ≥ 120
T = (140-Prio)*5

Expired Array

[0]
[1]
[2]
…
[99]
[100]
…
[139]

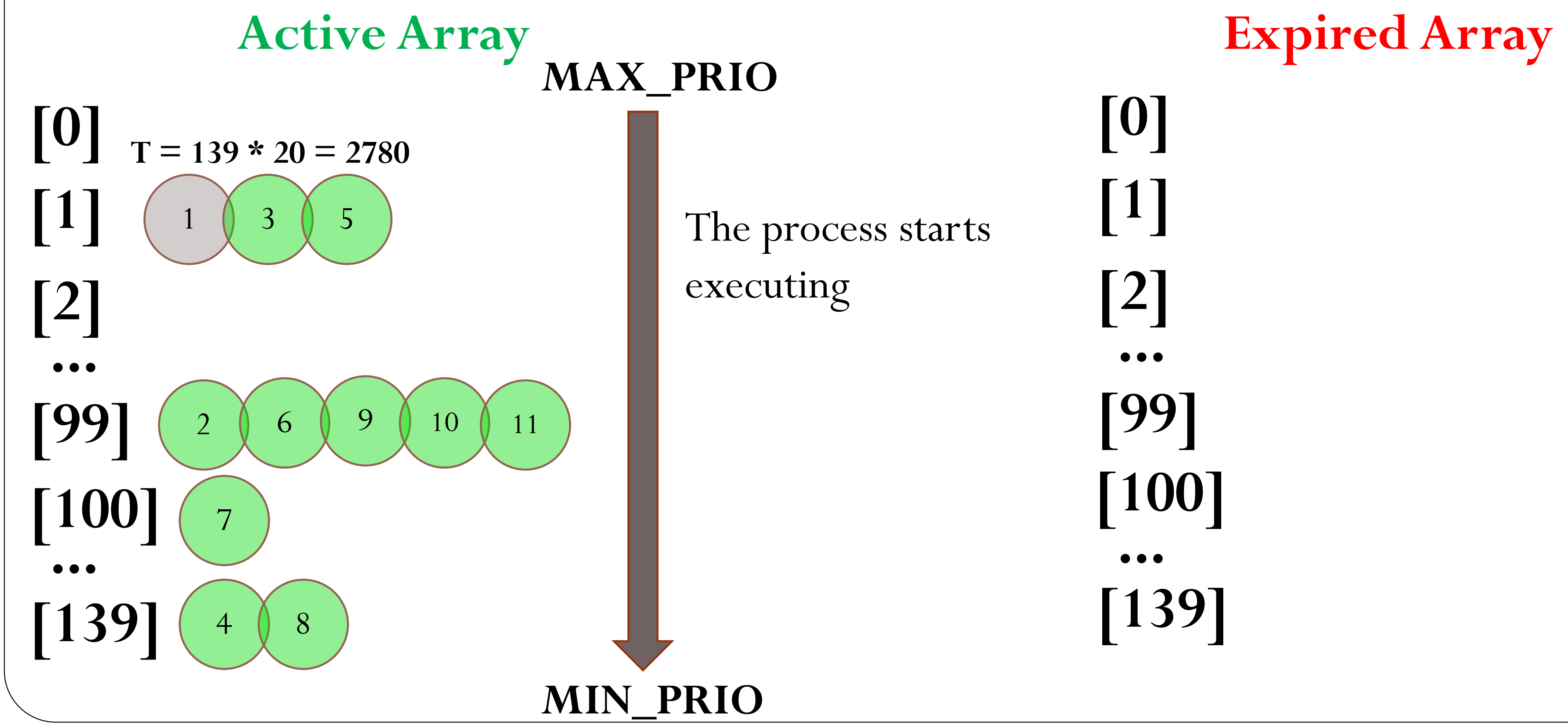
Example

- Reorganize the runqueue data structure



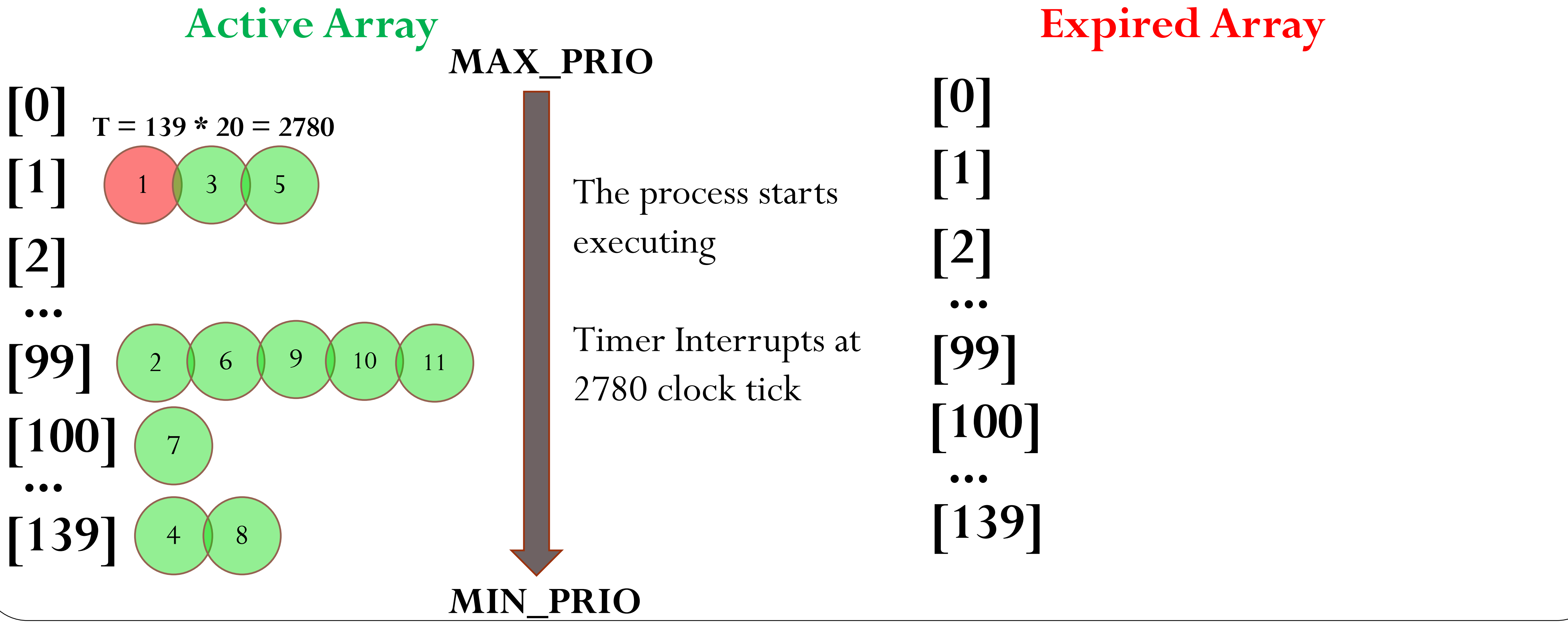
Example

- Reorganize the runqueue data structure



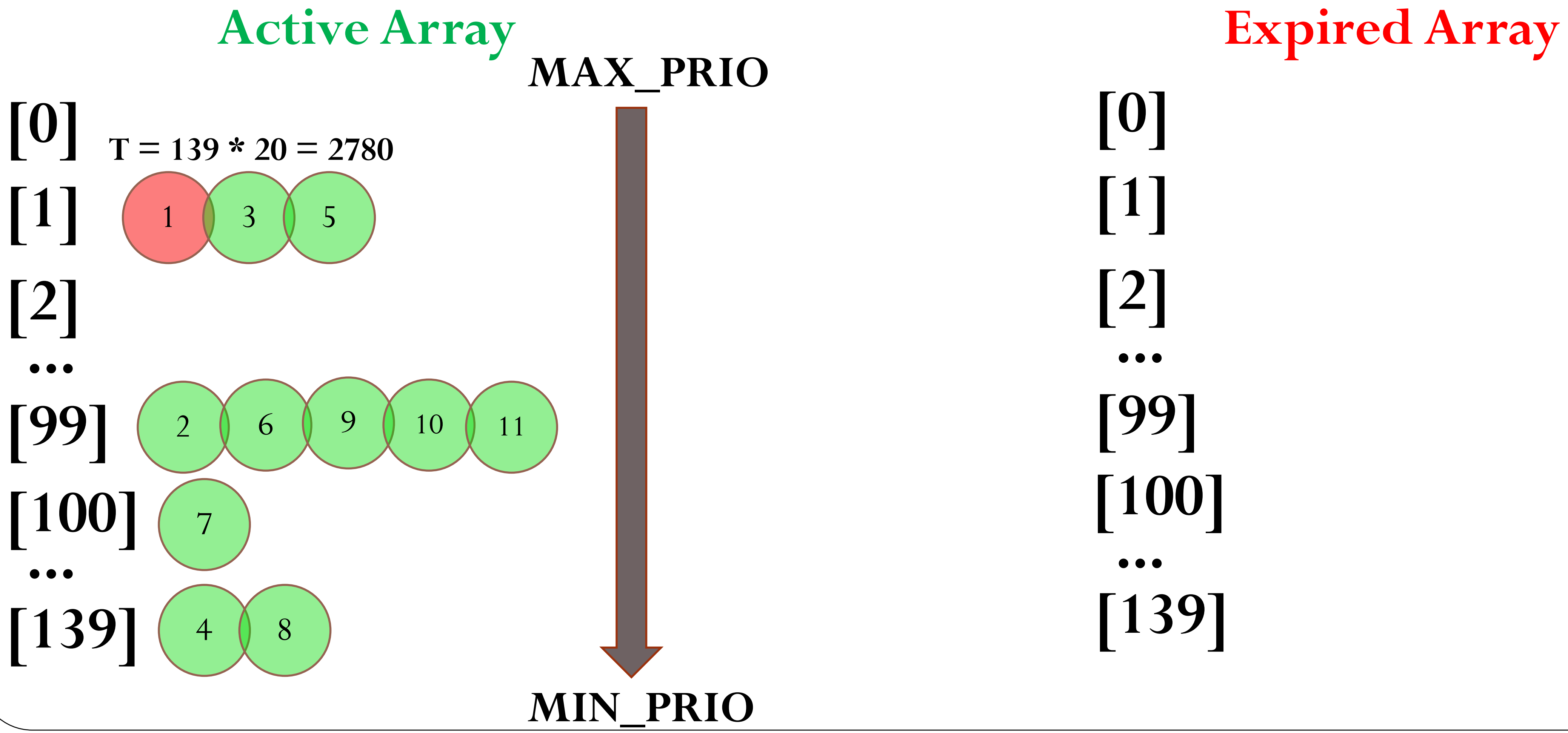
Example

- Reorganize the runqueue data structure



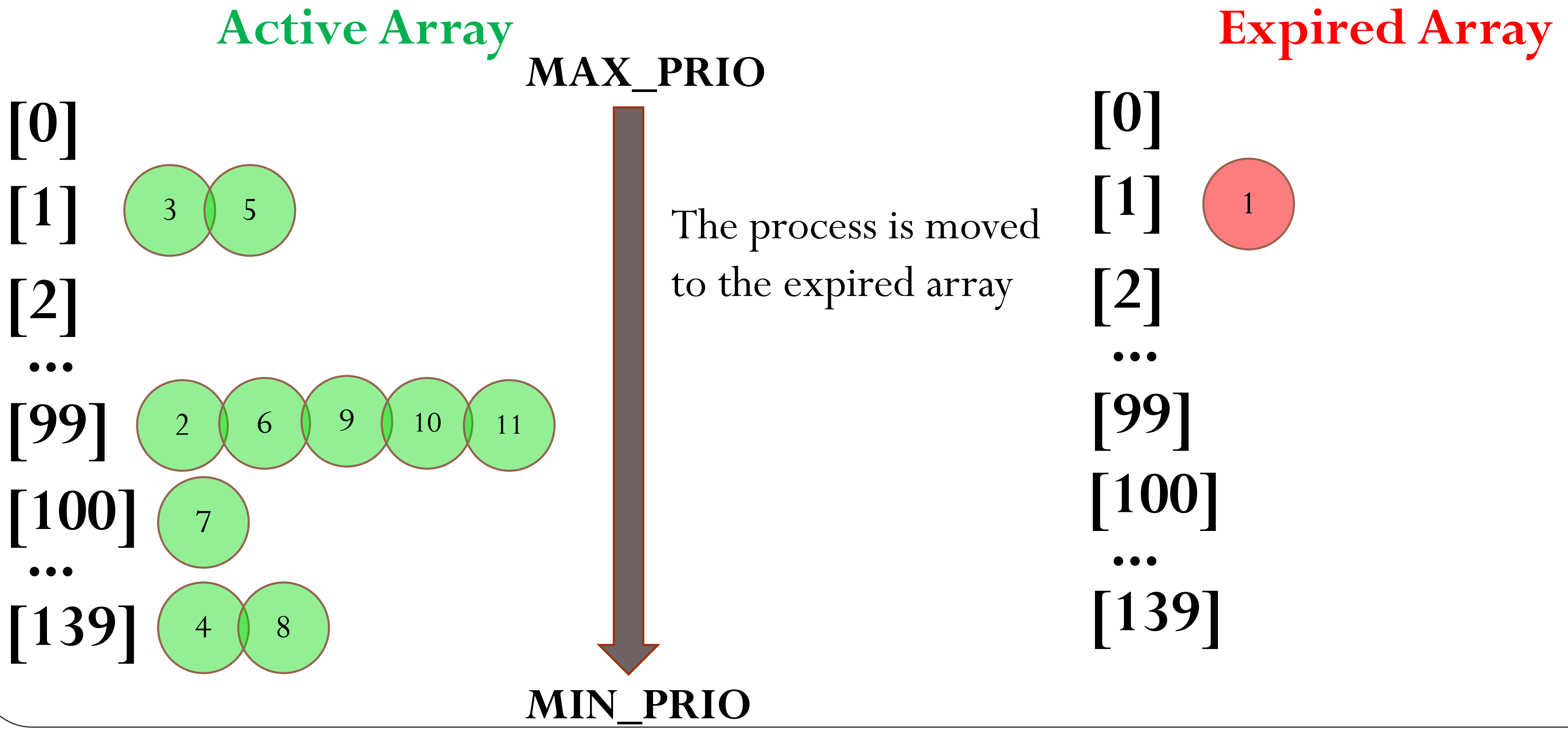
Example

- Reorganize the runqueue data structure



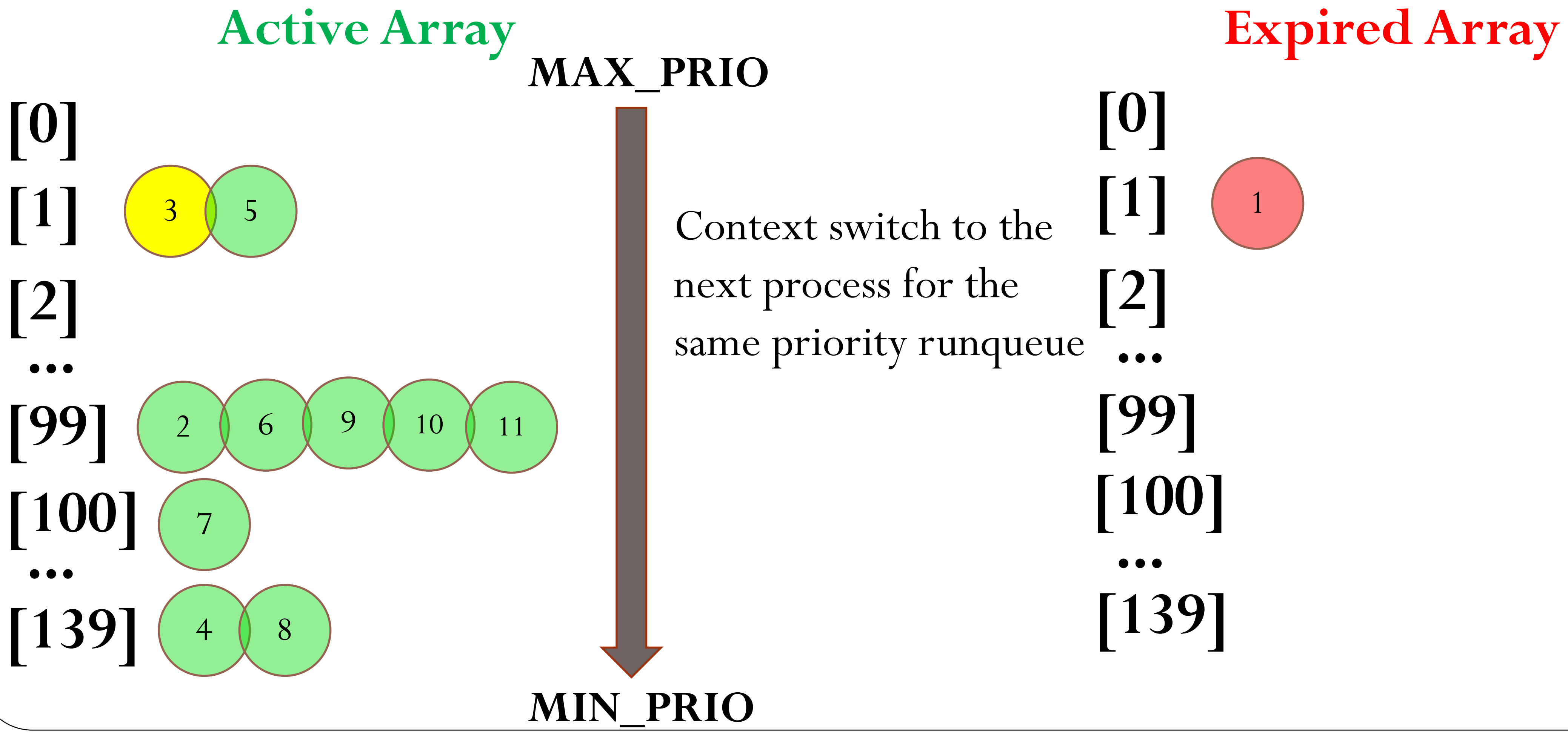
Example

- Reorganize the runqueue data structure



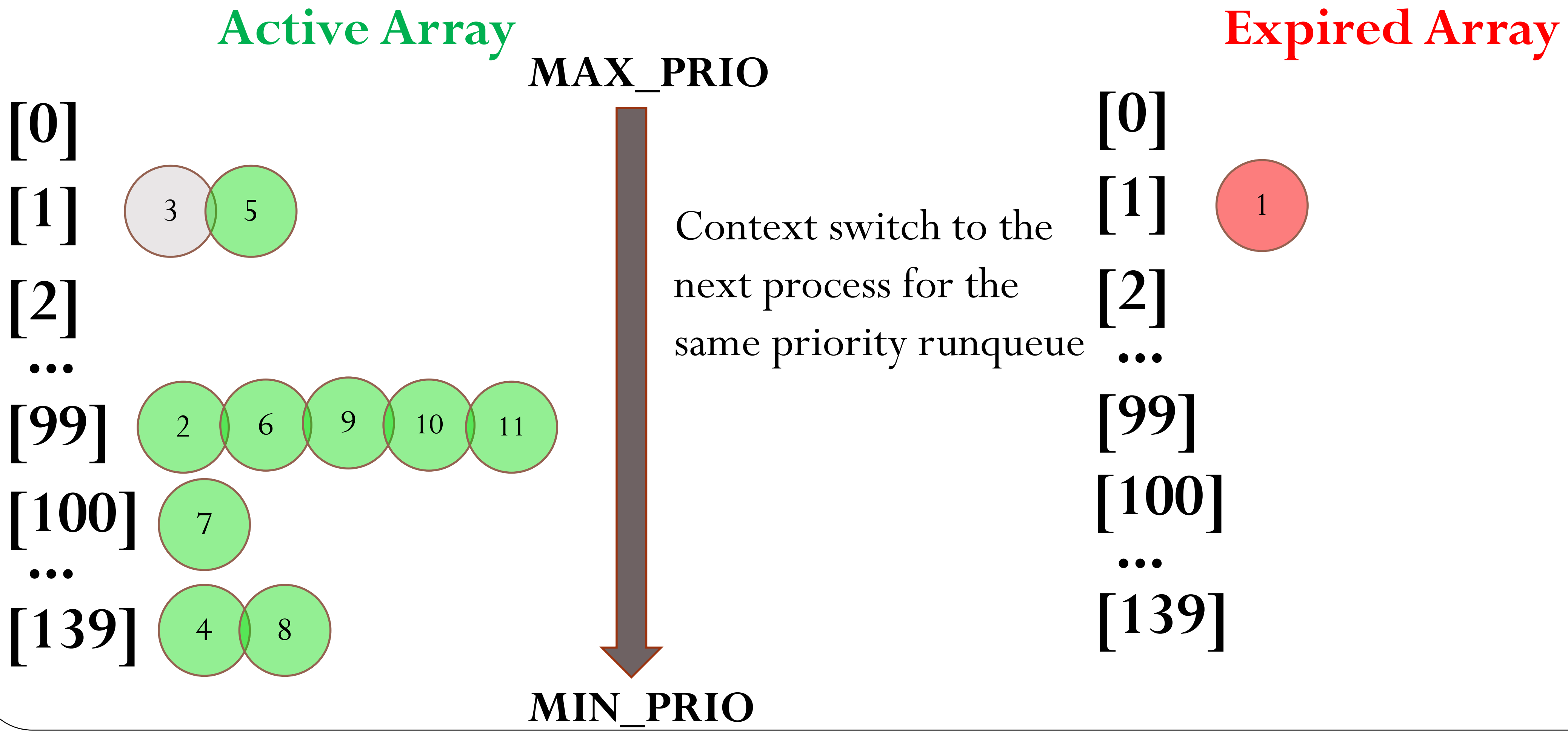
Example

- Reorganize the runqueue data structure



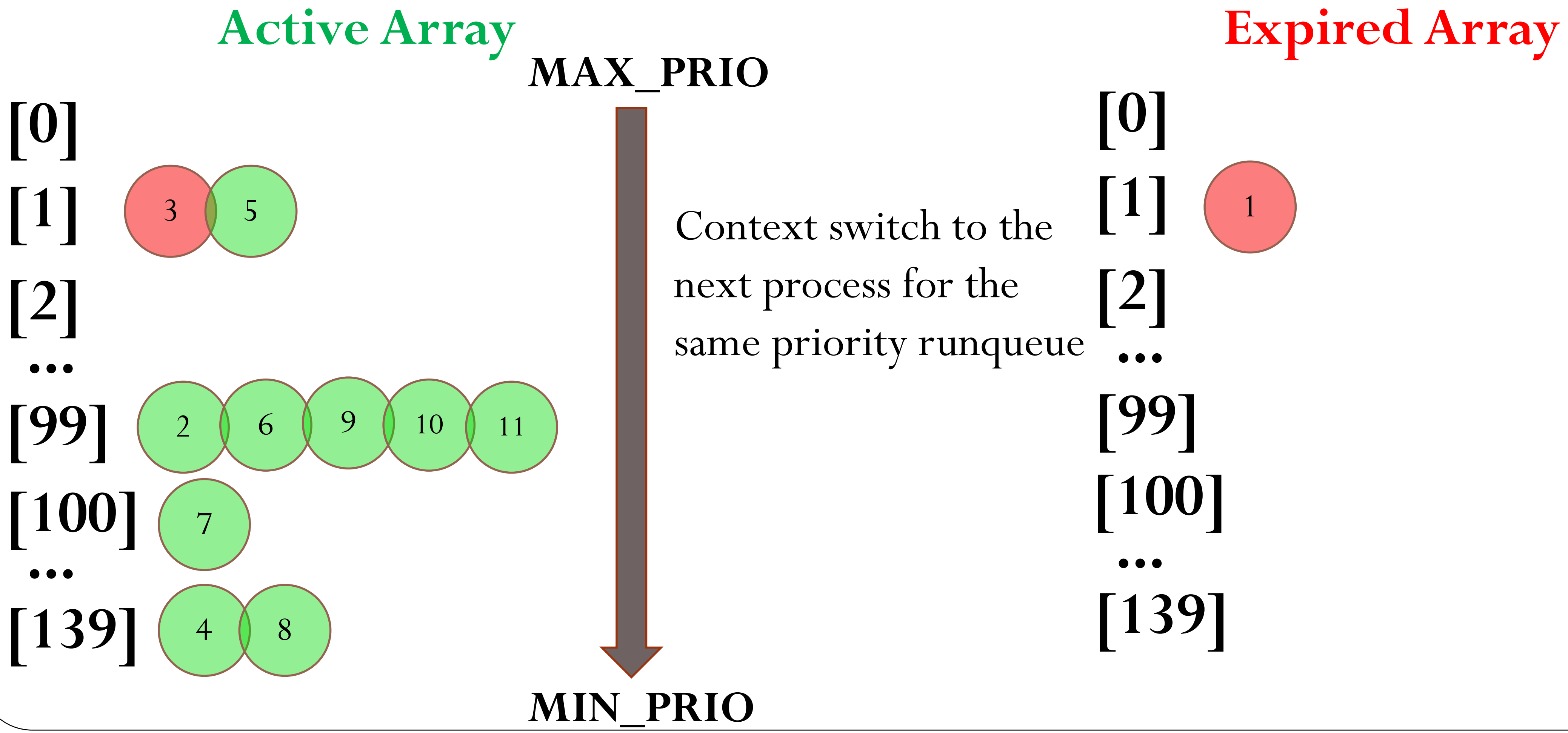
Example

- Reorganize the runqueue data structure



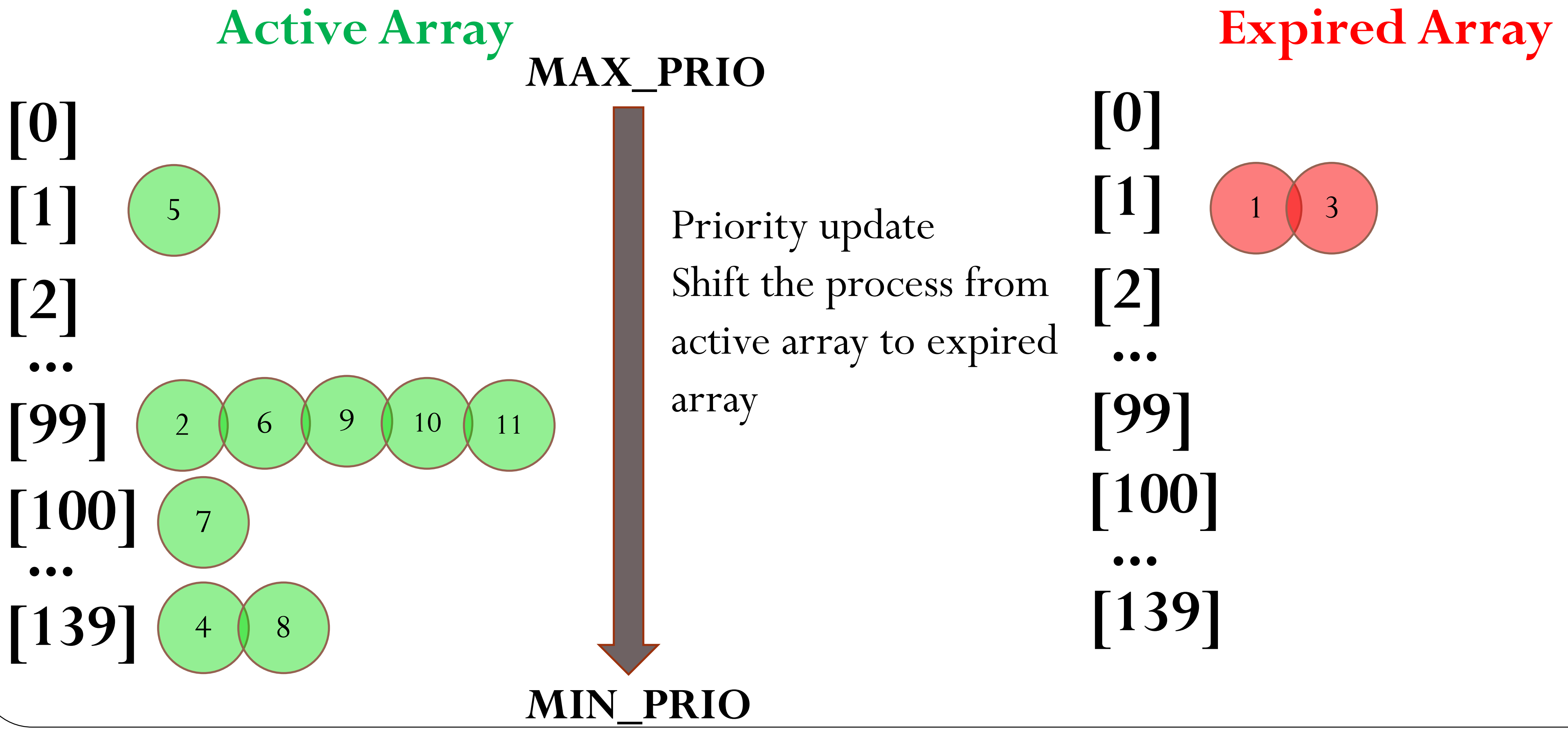
Example

- Reorganize the runqueue data structure



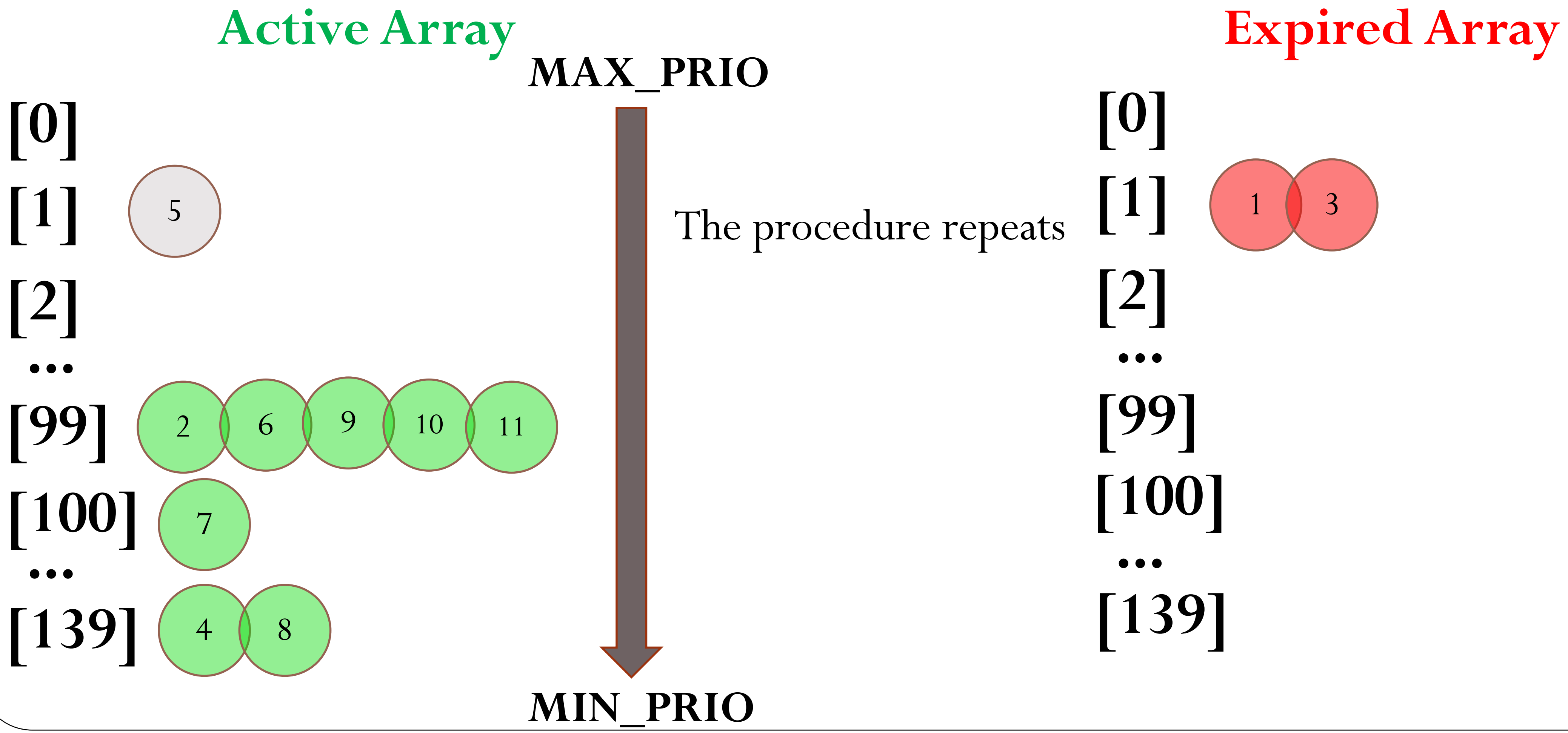
Example

- Reorganize the runqueue data structure



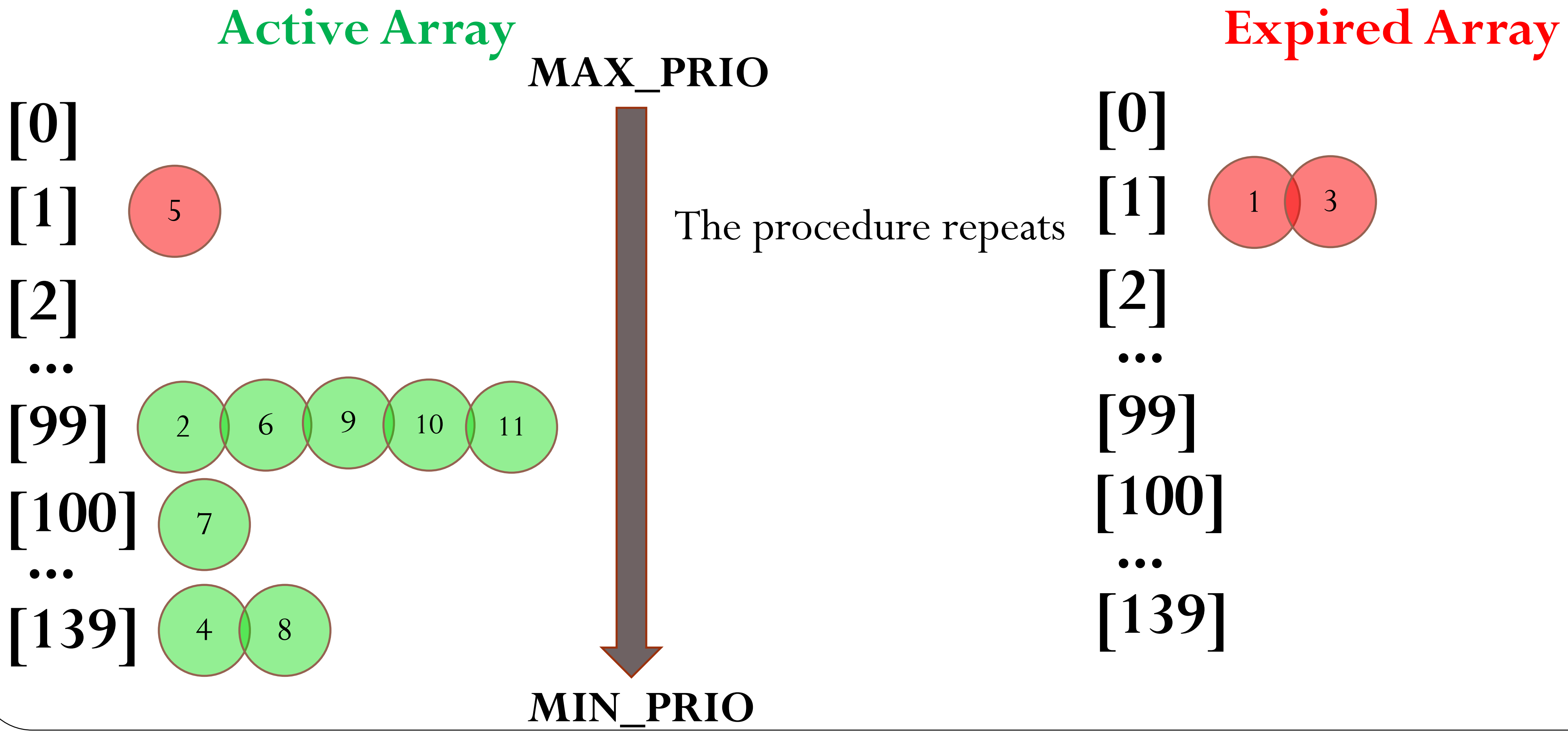
Example

- Reorganize the runqueue data structure



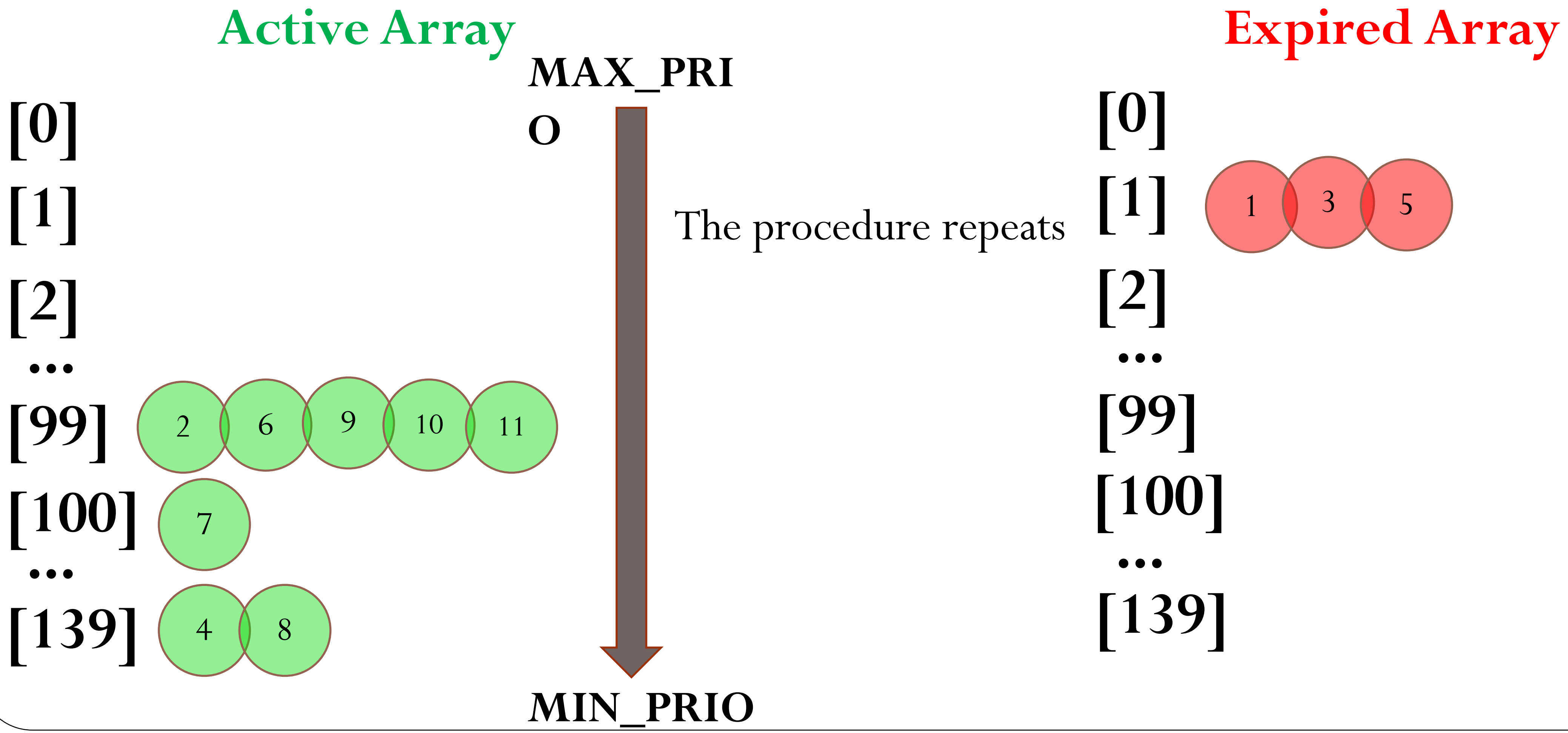
Example

- Reorganize the runqueue data structure



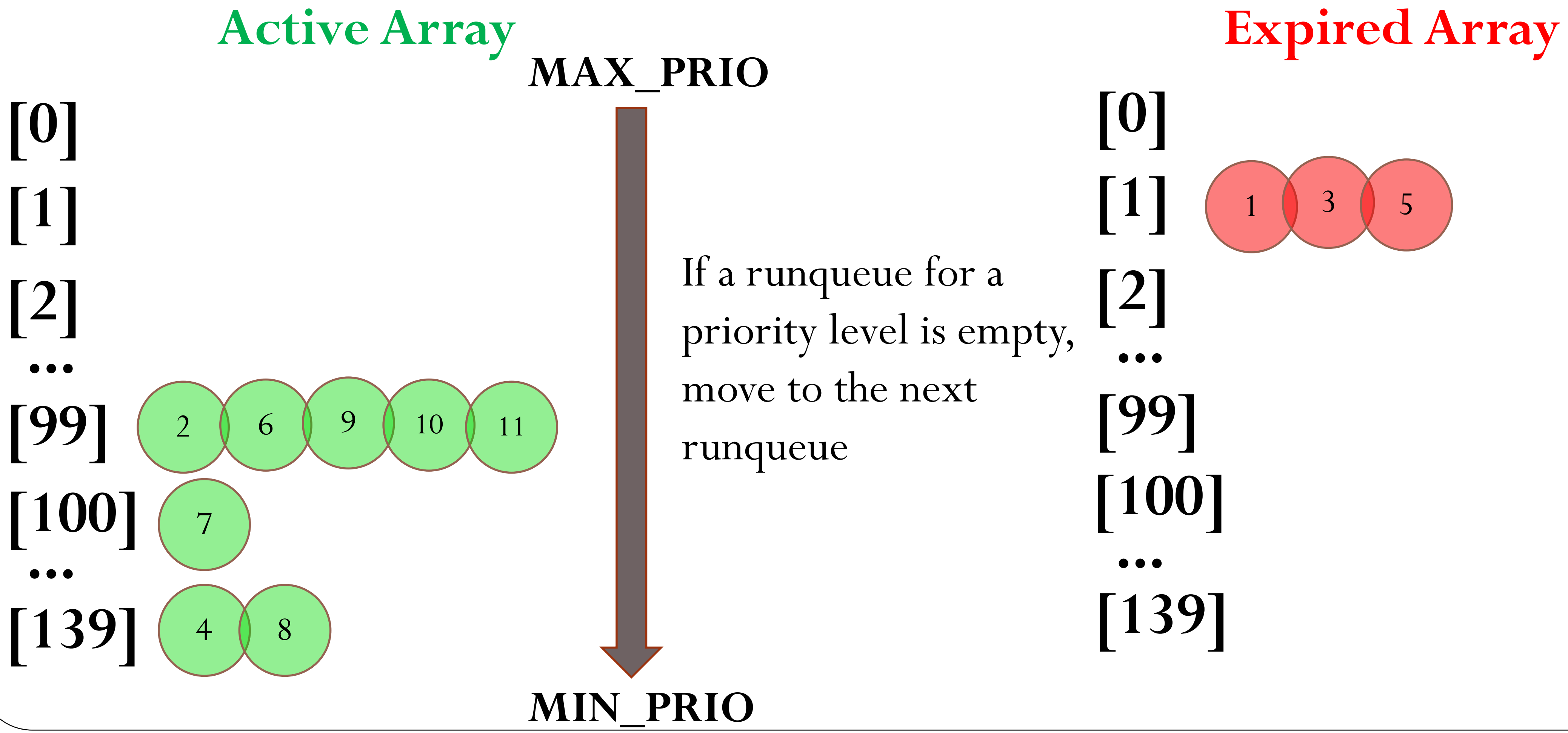
Example

- Reorganize the runqueue data structure



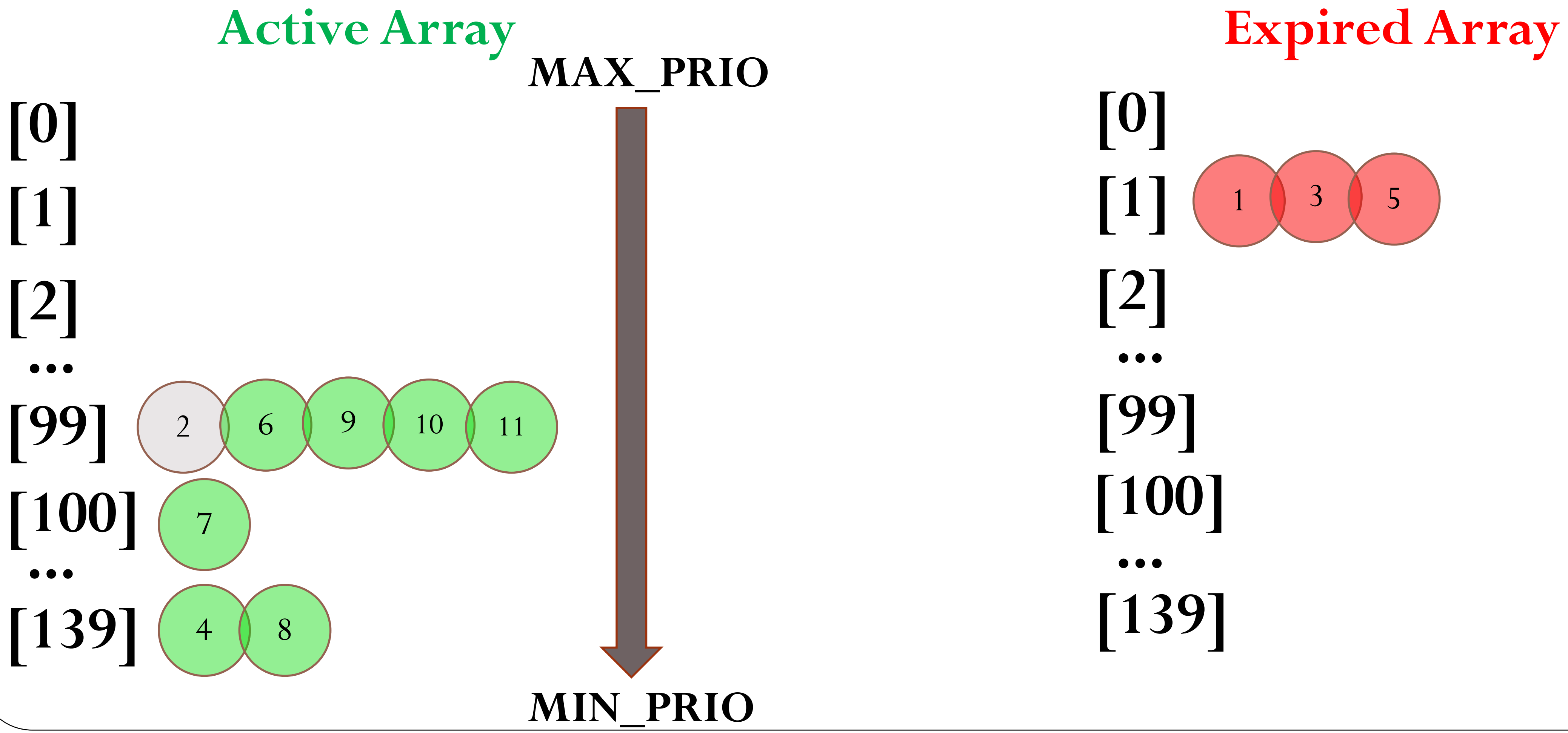
Example

- Reorganize the runqueue data structure



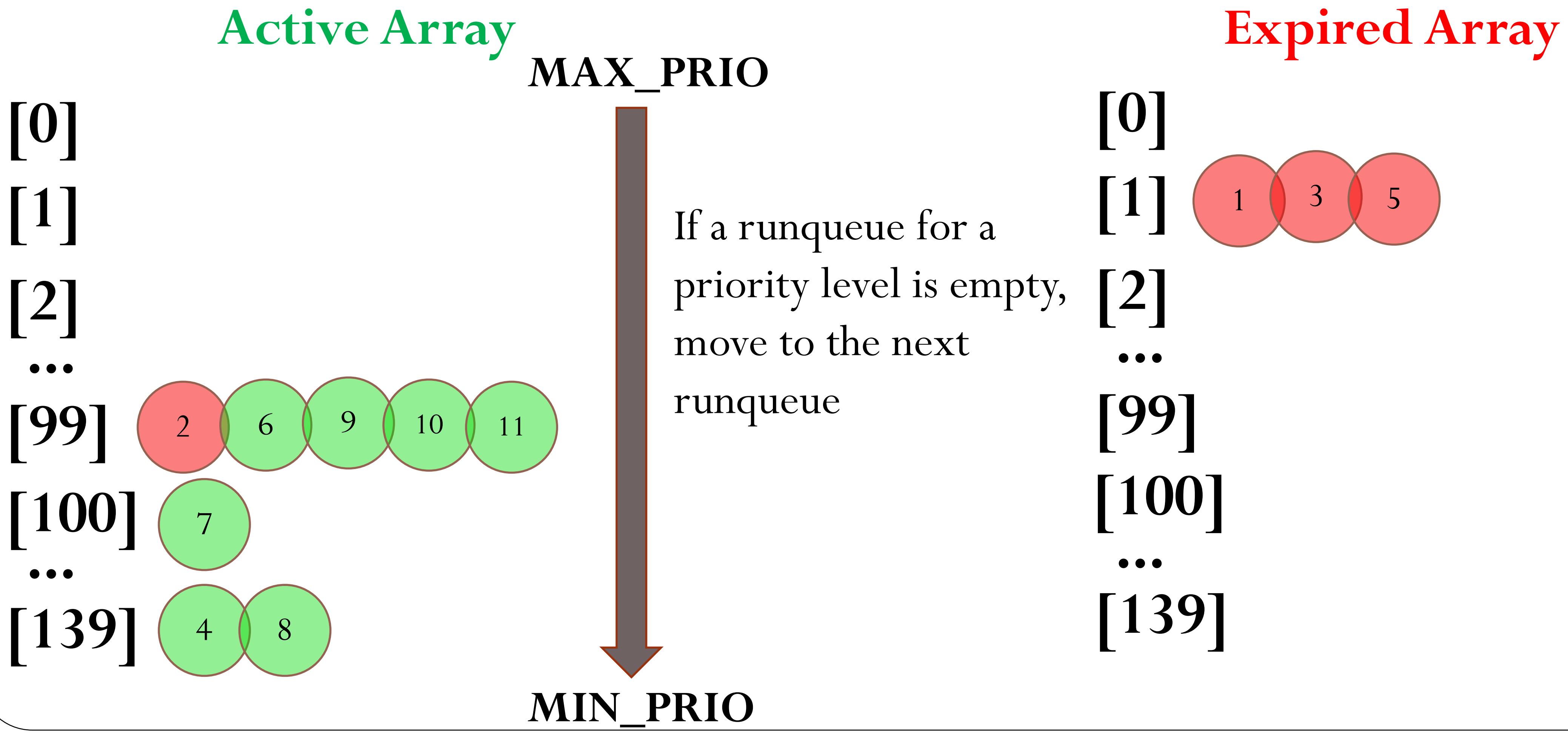
Example

- Reorganize the runqueue data structure



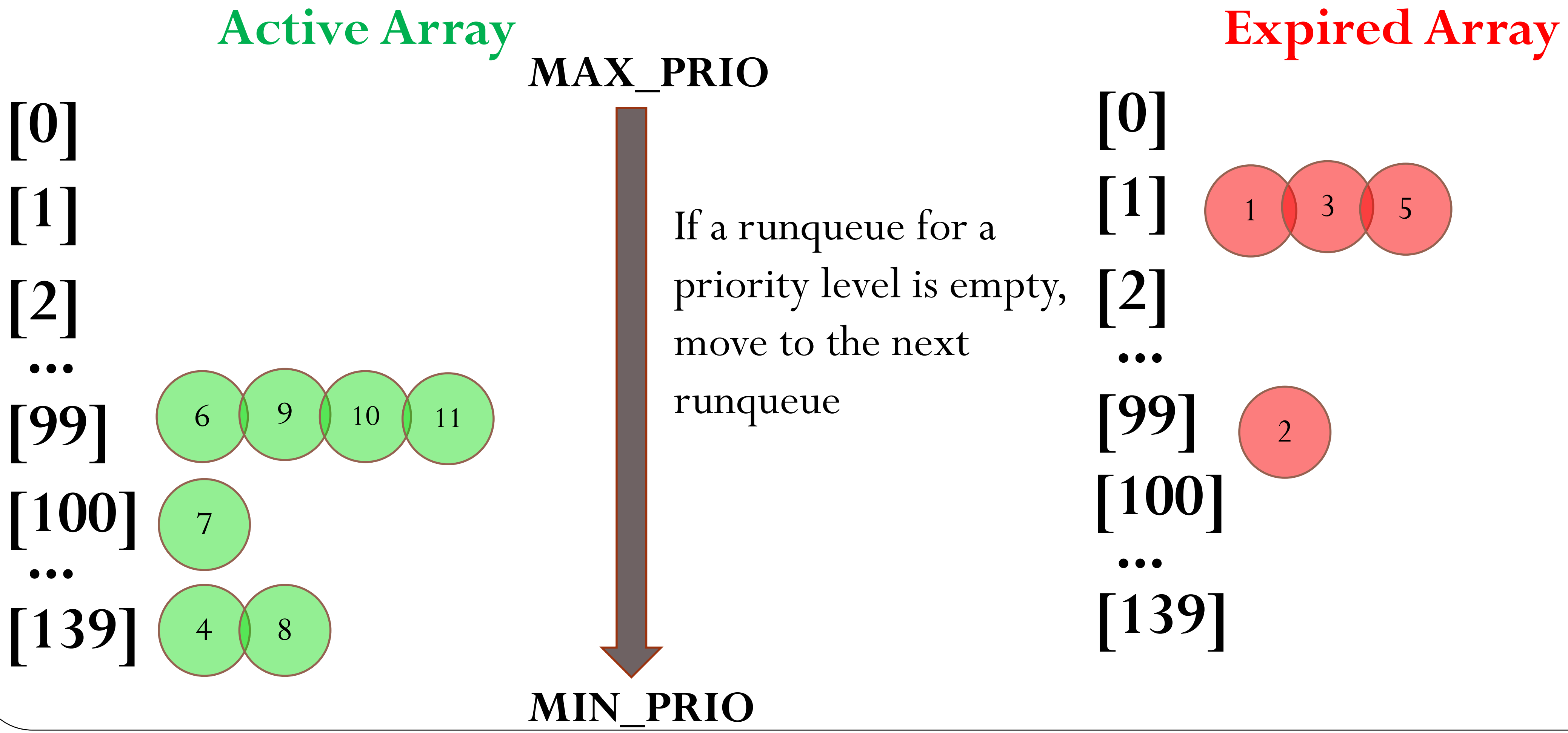
Example

- Reorganize the runqueue data structure



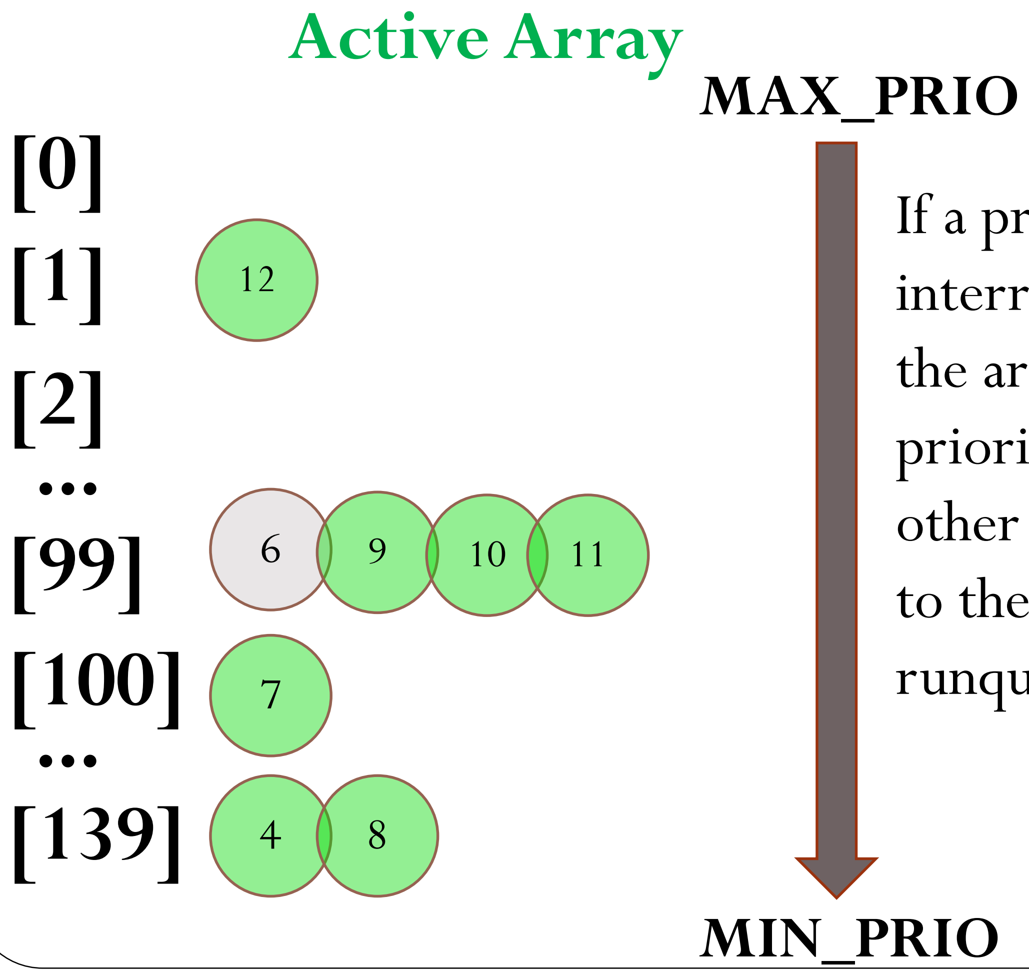
Example

- Reorganize the runqueue data structure

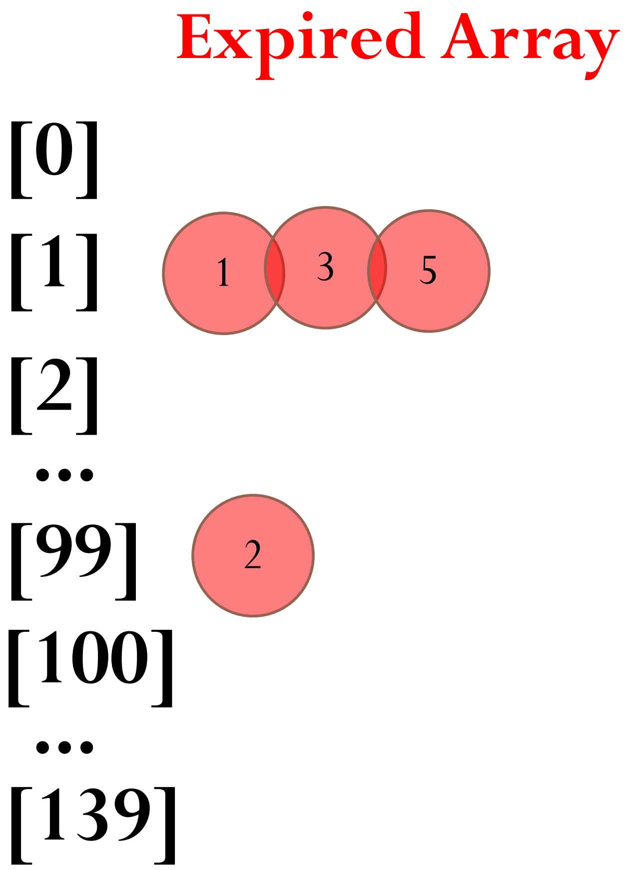


Example

- Reorganize the runqueue data structure

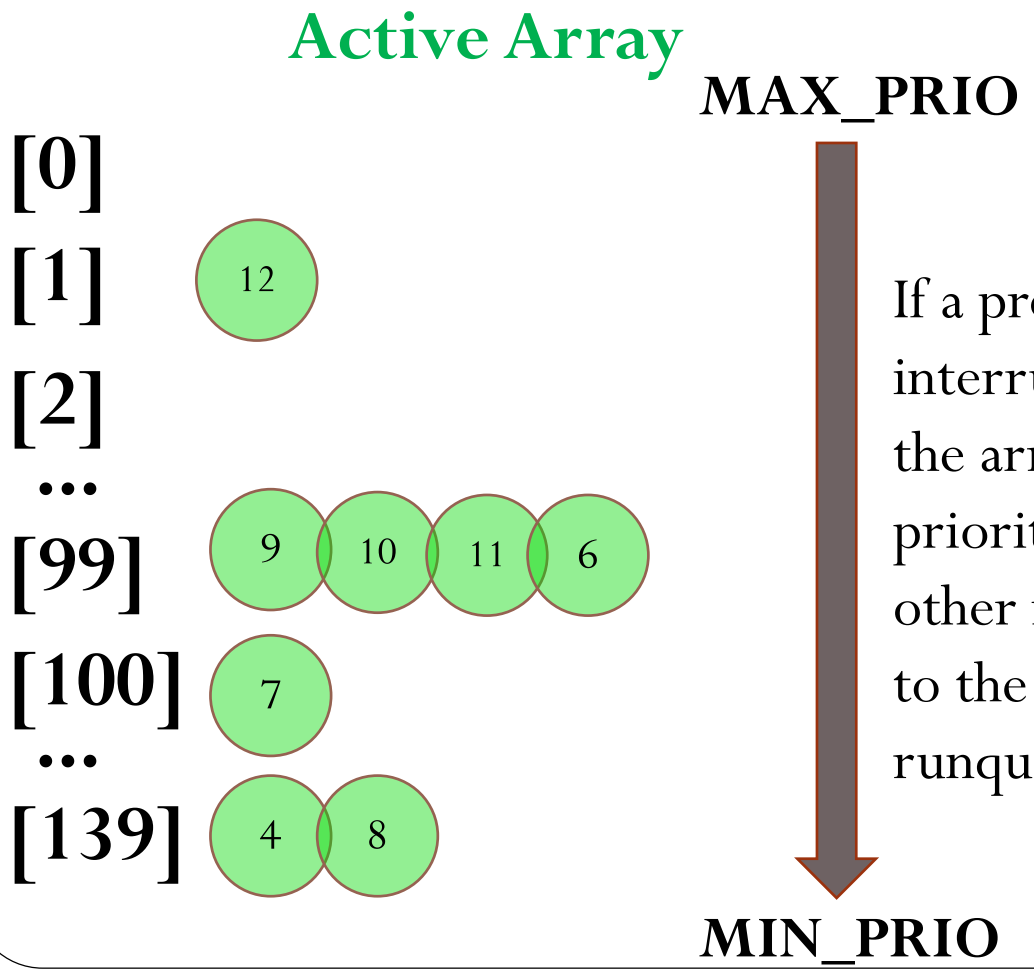


If a process is interrupted because of the arrival of a higher priority process, or for other reason, move it to the end of the runqueue

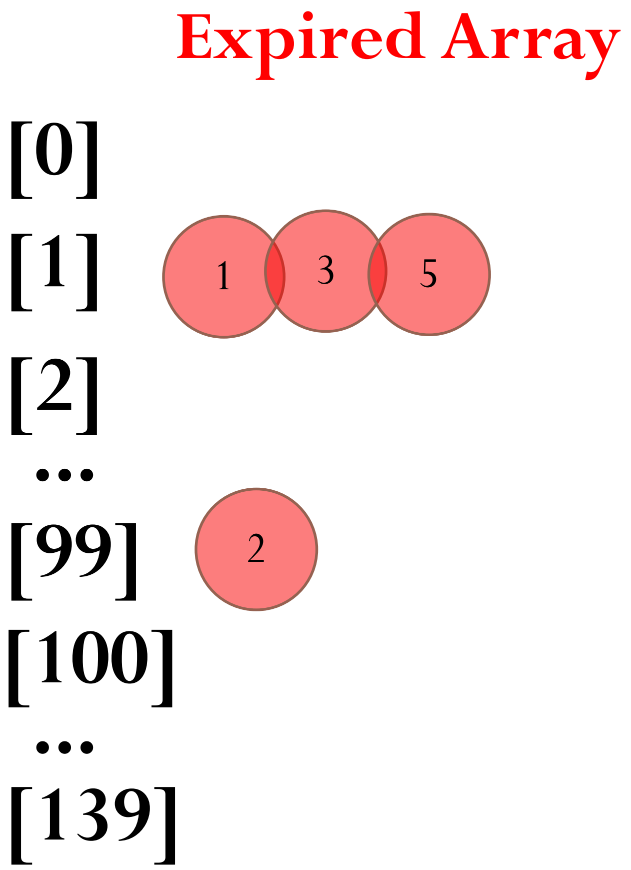


Example

- Reorganize the runqueue data structure

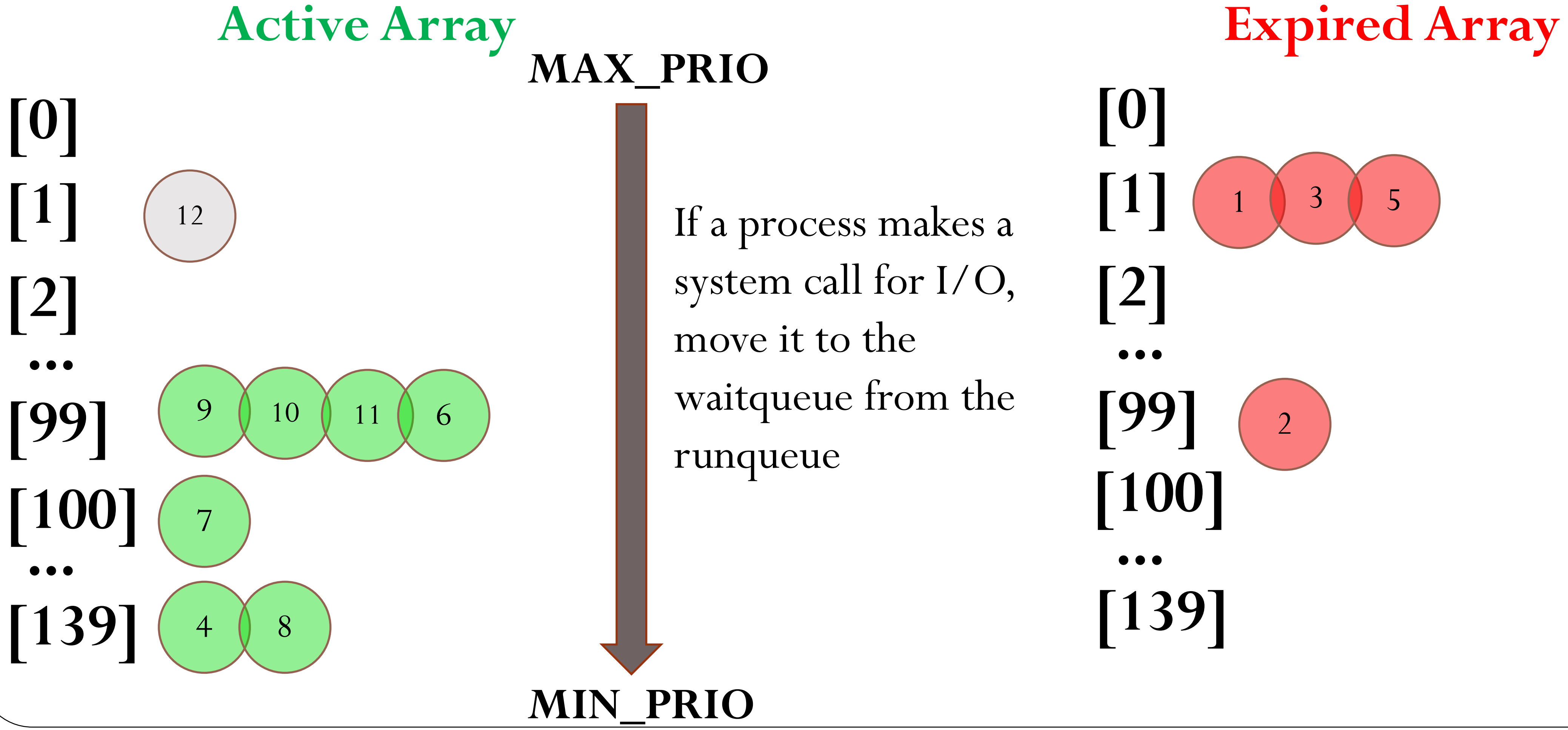


If a process is interrupted because of the arrival of a higher priority process, or for other reason, move it to the end of the runqueue



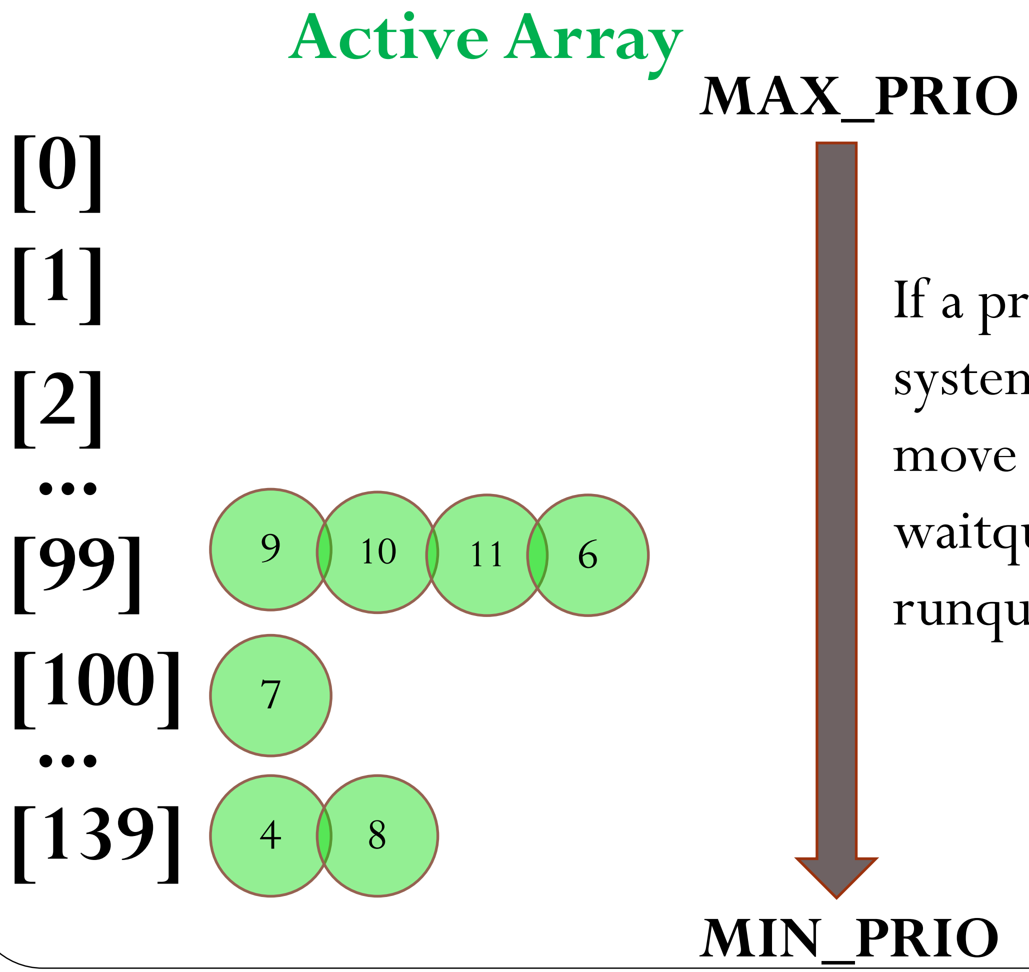
Example

- Reorganize the runqueue data structure

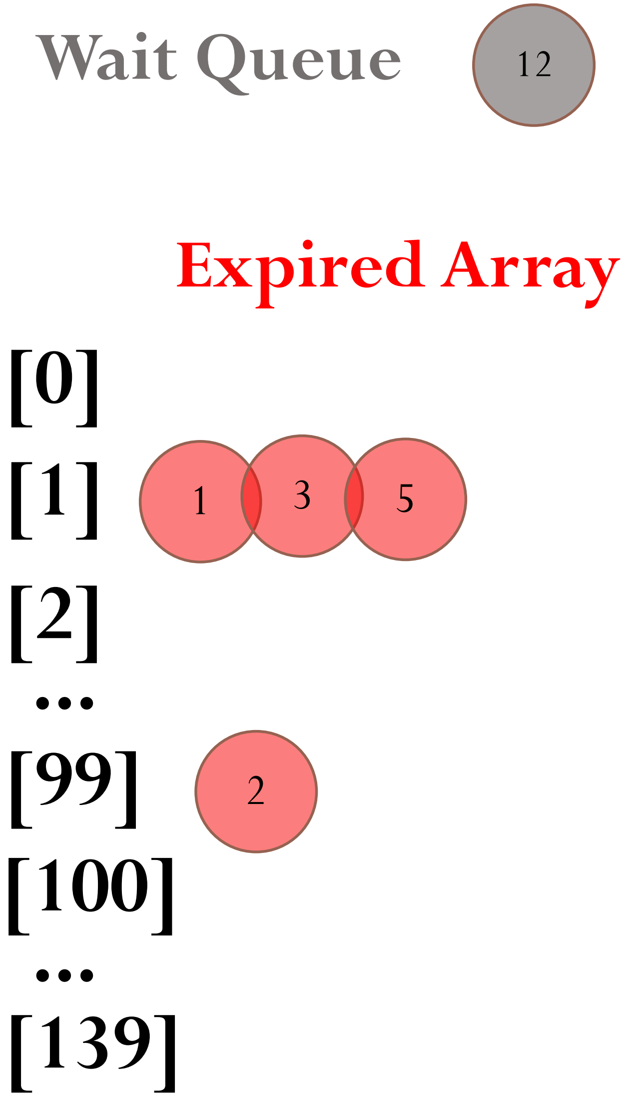


Example

- Reorganize the runqueue data structure

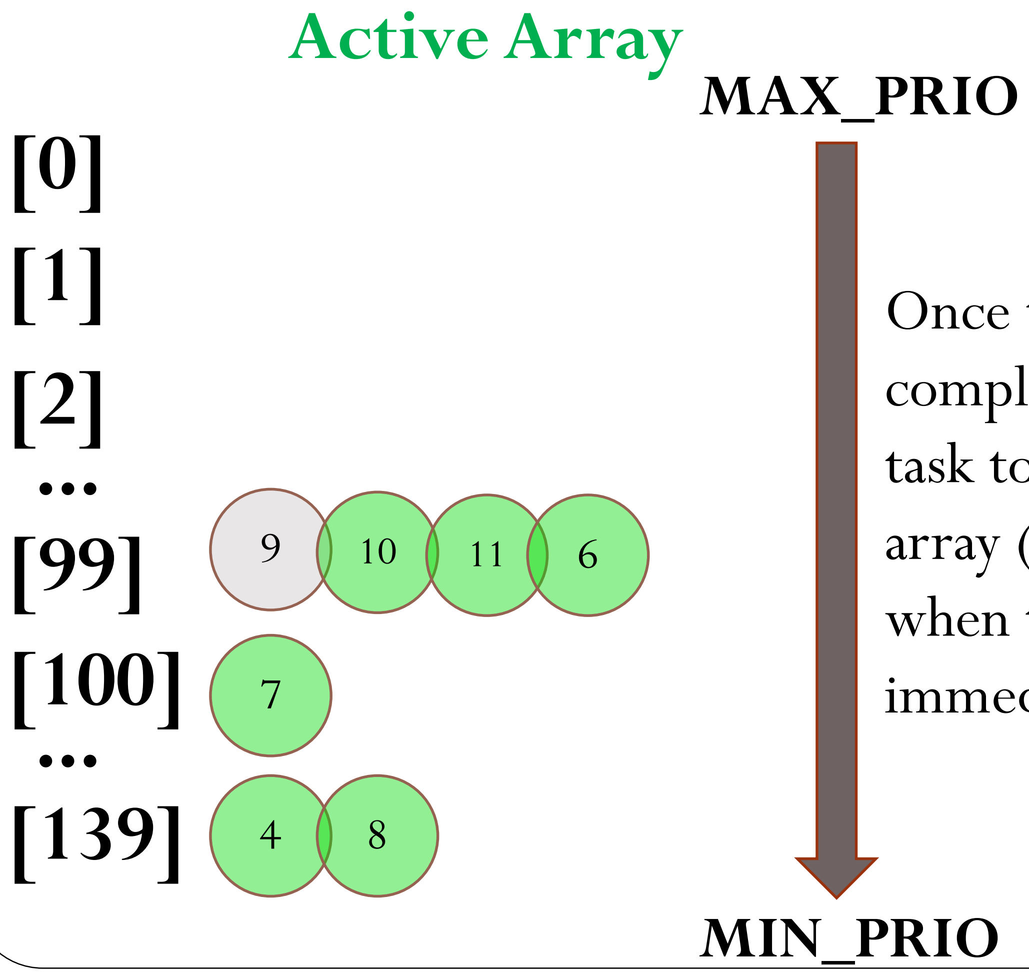


If a process makes a system call for I/O, move it to the waitqueue from the runqueue

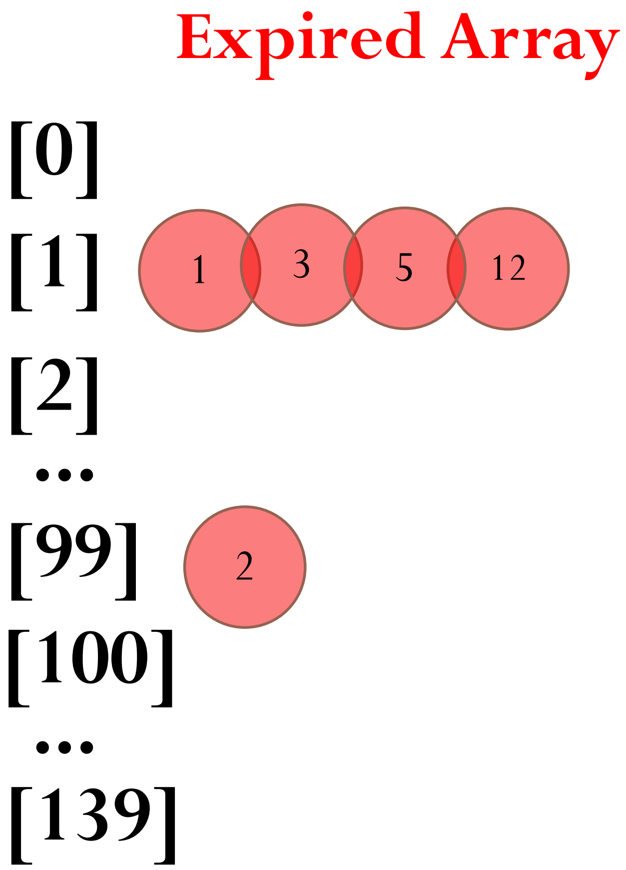


Example

- Reorganize the runqueue data structure

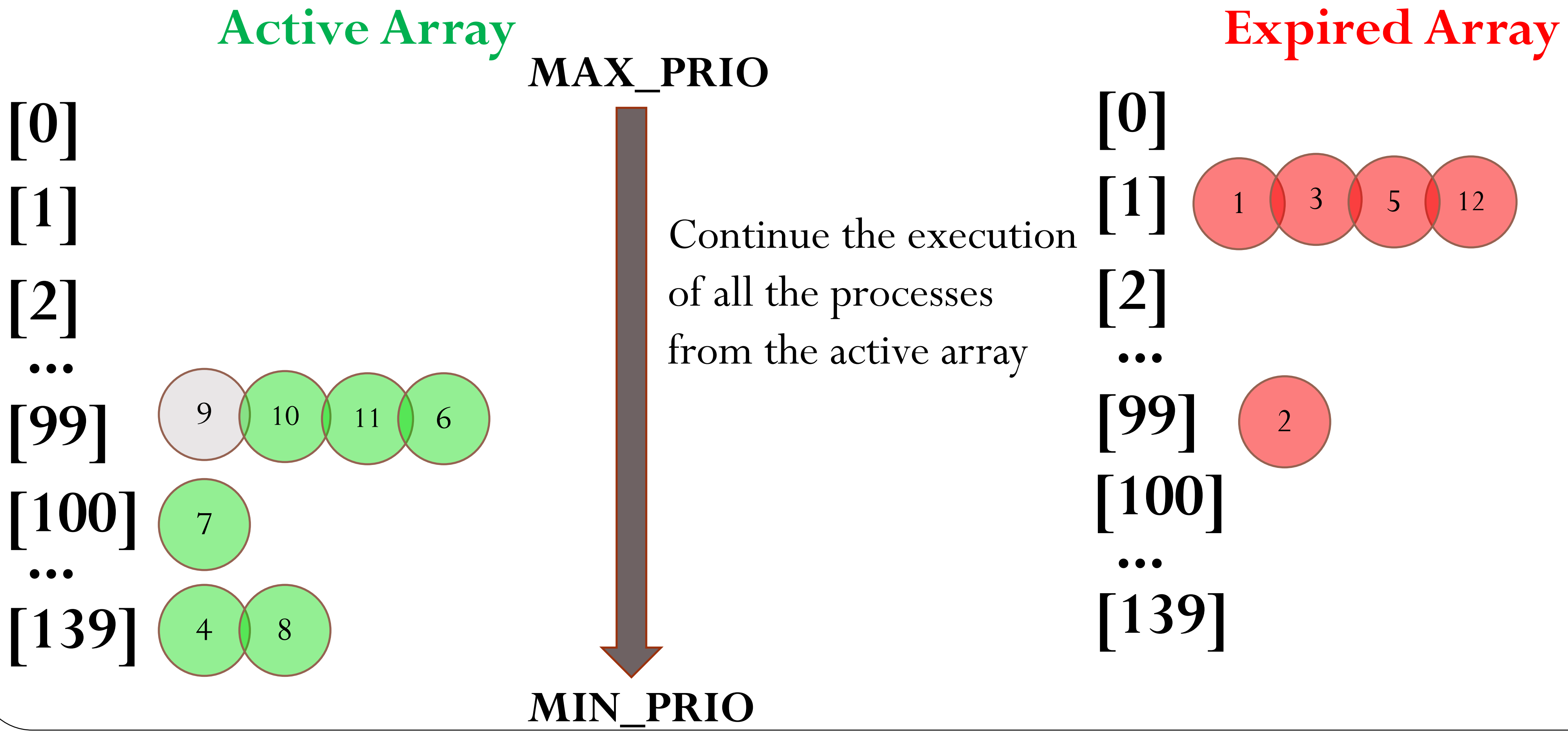


Once the I/O is complete, move the task to the expired array (or active array when the task needs immediate scheduling)



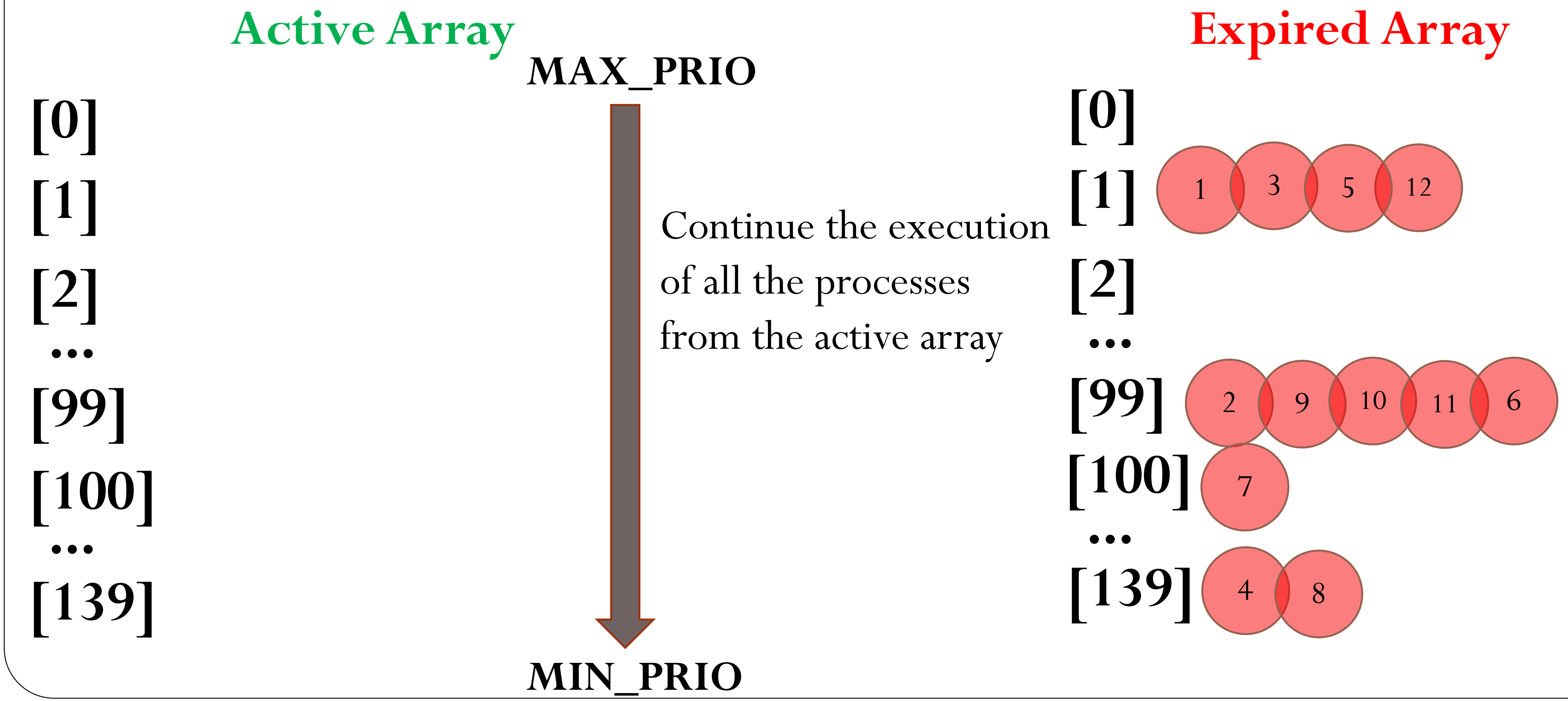
Example

- Reorganize the runqueue data structure



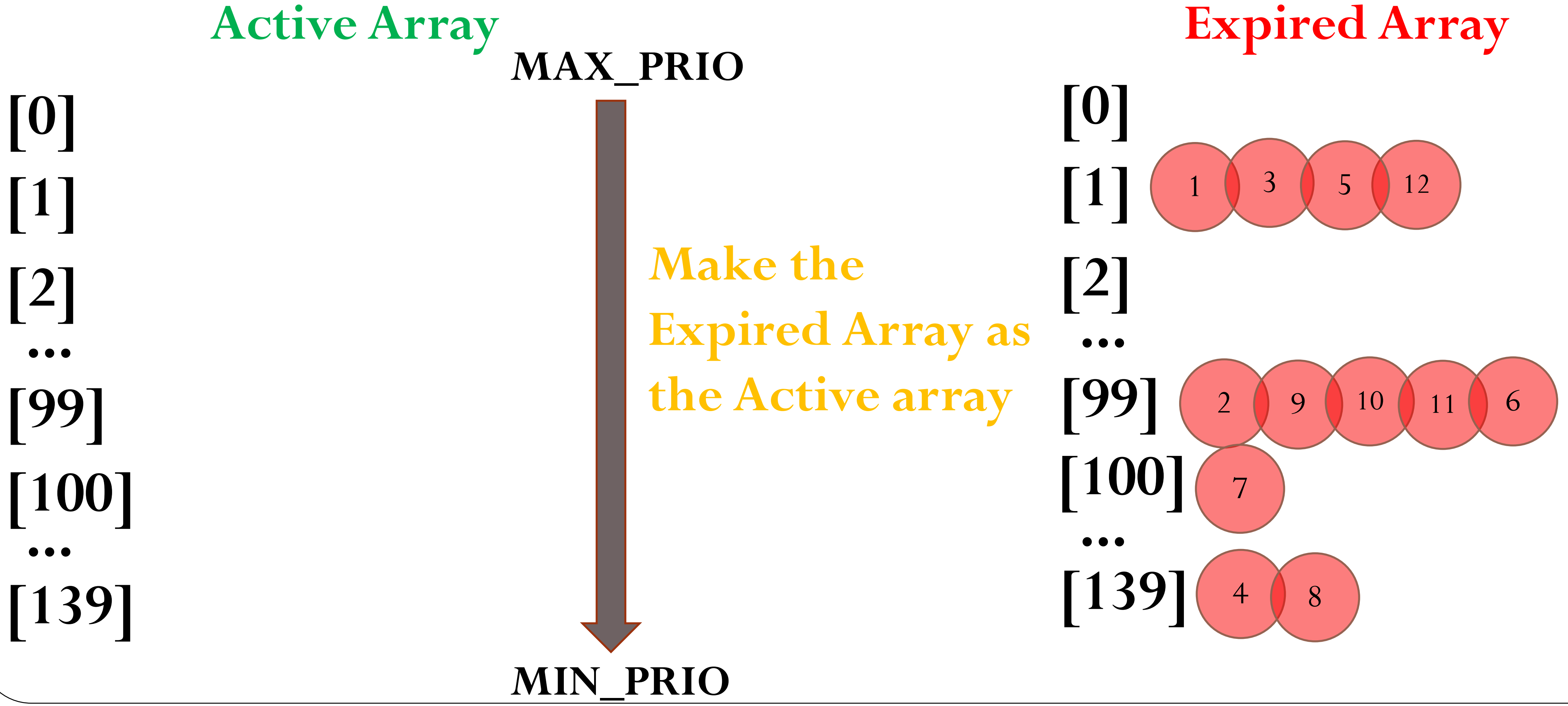
Example

- Reorganize the runqueue data structure



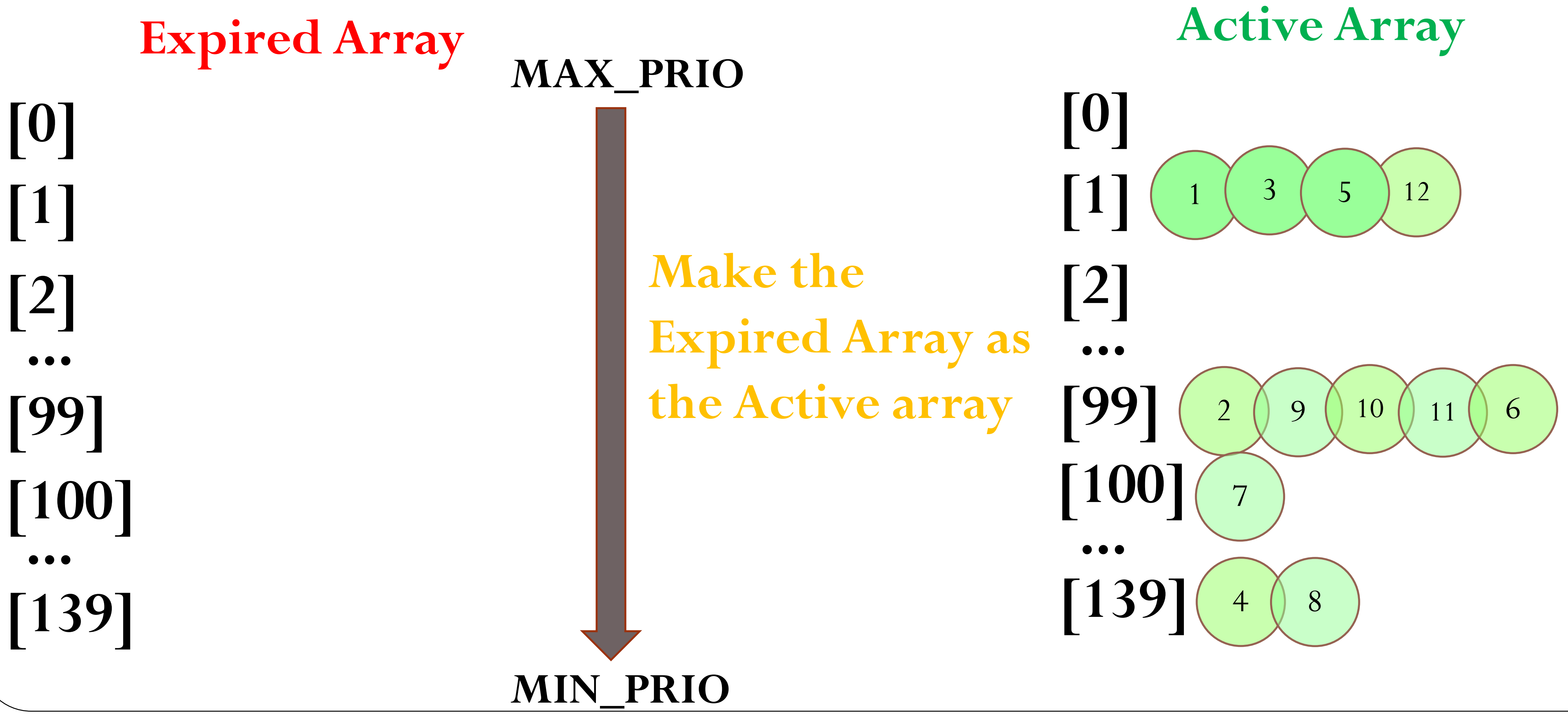
Example

- Reorganize the runqueue data structure



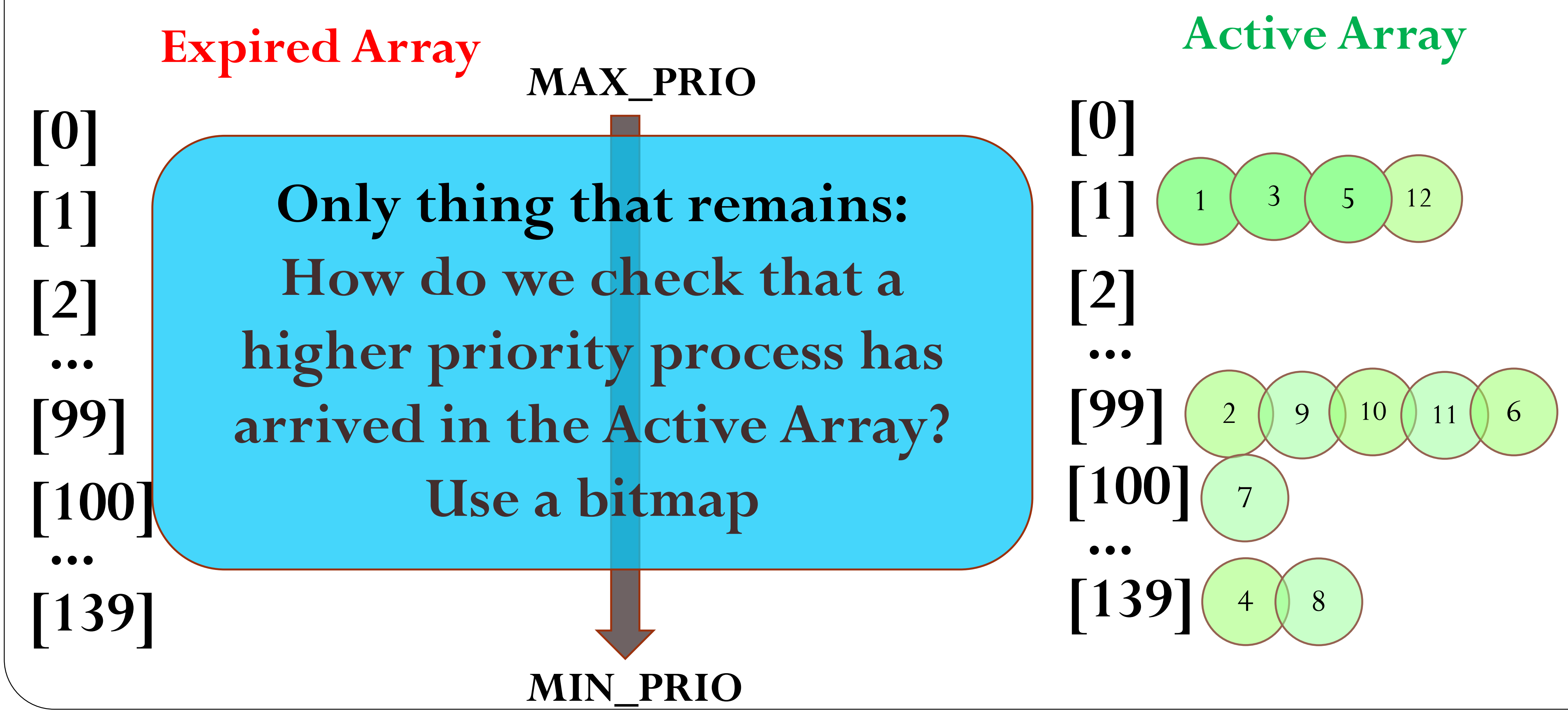
Example

- Reorganize the runqueue data structure



Example

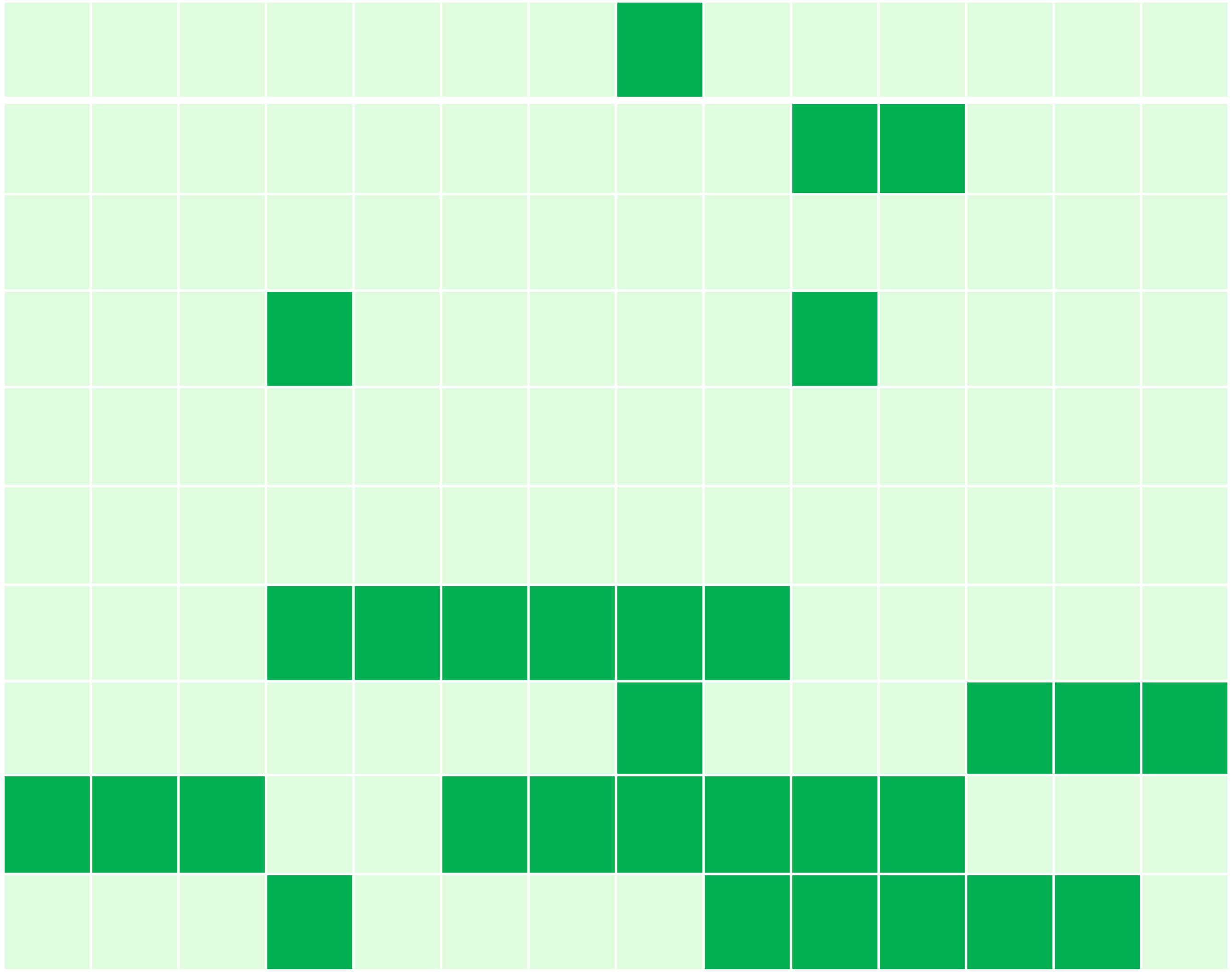
- Reorganize the runqueue data structure



Only thing that remains:

How do we check that a higher priority process has arrived in the Active Array?

Use a bitmap



```
struct prio_array {

    /* number of tasks */
    int nr_active;

    /* priority bitmap */
    unsigned long bitmap[BITMAP_SIZE];

    /* priority queues */
    struct list_head queue[MAX_PRIO];

};
```

Dynamic Priority

- Good thing seen so far
  - Timeslices computed based on priority
  - Fast access to runqueues
- Not so good
  - No distinction between interactive and batch jobs
- *Dynamic priority*: allows this distinction
  - Dynamically increase priority level of interactive jobs
  - Based on average sleep time of a process
    - Sleep time added to a variable when a process wakes up
    - CPU time subtracted from the variable when a process runs
- Dynamic priority is computed only for normal, real time tasks still use static priority only

- Dynamic Priority
  - MAX(100, min(static priority – *bonus* +5, 139))
  - *Bonus* is a value between 0 and 10 set based on average sleep time
    - I/O bound processes sleep more, so should have higher priority when they are runnable with higher bonus value
    - Opposite for CPU-bound

Average sleep time	Bonus
Greater than or equal to 0 but smaller than 100 ms	0
Greater than or equal to 100 ms but smaller than 200 ms	1
Greater than or equal to 200 ms but smaller than 300 ms	2
Greater than or equal to 300 ms but smaller than 400 ms	3
Greater than or equal to 400 ms but smaller than 500 ms	4
Greater than or equal to 500 ms but smaller than 600 ms	5
Greater than or equal to 600 ms but smaller than 700 ms	6
Greater than or equal to 700 ms but smaller than 800 ms	7
Greater than or equal to 800 ms but smaller than 900 ms	8
Greater than or equal to 900 ms but smaller than 1000 ms	9

Has a value between 0 and 10

If bonus < 5, implies less interaction with the user
thus more of a CPU bound process.
The dynamic priority is therefore decreased (toward 139)

If bonus > 5, implies more interaction with the user
thus more of an interactive process.
The dynamic priority is increased (toward 100).

- The runqueues are arranged based on this dynamic priorities actually
- Other optimization
  - Define a process as interactive if (*bonus* – 5 ≥ (static priority)/4 – 28)
  - Add an interactive process back to end of active queue with a fresh quanta when it finishes its quanta
  - But should this not cause starvation to lower level queues?
    - Makes certain checks on the expired queue (what do you think should be checked?)
    - Also, if an interactive task keeps on running, its interactivity will go down
- Note that dynamic priority does not affect the timeslice, that is still based on the static priority

- Why is this called O(1) scheduler?
- Problems with the O(1) scheduler
  - Complex heuristics for interactivity check, did not work well in practice
  - Managing 2 x 140 runqueues is complex
  - Codebase was complex and difficult to debug
- Replaced by Completely Fair Scheduler (CFS) in 2007 (Kernel version 2.6.23)