CS 60038: Advances in Operating Systems Design

Kernel Data Structures

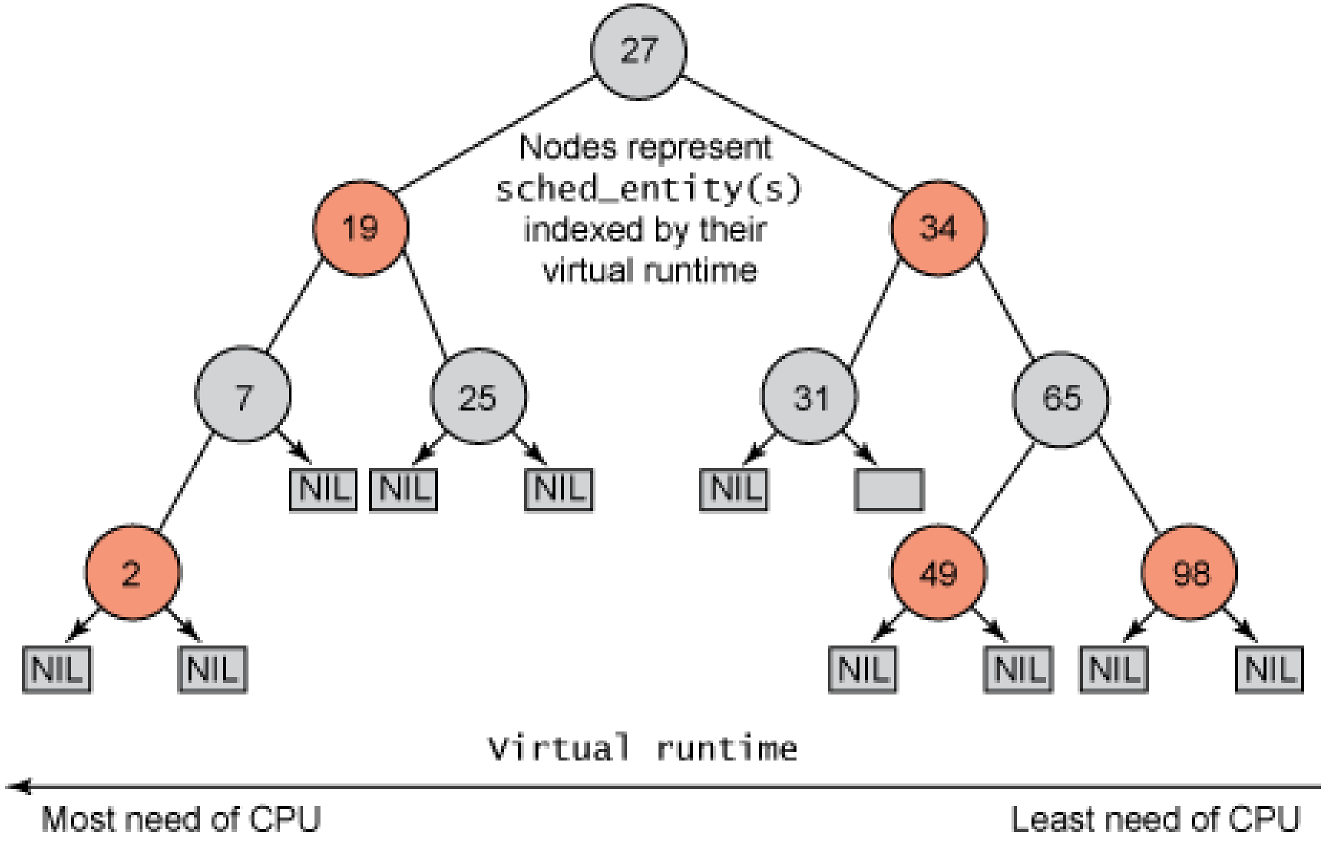
Evolution of Kernel Scheduling

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Sandip Chakraborty
sandipc@cse.iitkgp.ac.in

The Evolution of Linux Process Schedulers



The Genesis Scheduler

- Introduced in 1991 within Kernel 0.01
  - Fairly simple, minimum design
  - Does not aim to support massive multiprocessing systems

The Genesis Scheduler

- Introduced in 1991 within Kernel 0.01
  - Fairly simple, minimum design
  - Does not aim to support massive multiprocessing systems

- A single process queue – the scheduler iterates over the entire queue to select a task to run
  - Queue size is not very long; default NR_TASKS of kernel set to 32

The Genesis Scheduler – The Code

```
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;
}

```

The Genesis Scheduler – The Code

```
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;
 }

```

Iterate over all the tasks in the task array

The Genesis Scheduler – The Code

```
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;
 }

```

Check whether an alarm has been raised before the current jiffies (the interrupt timer counter)

The Genesis Scheduler – The Code

```
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;
 }
```

Raise the corresponding signal and reset the alarm

The Genesis Scheduler – The Code

```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;
 }

```

If the task is waiting, then move it to the running state (ready to run)

The Genesis Scheduler – The Code

```
/* 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;
```

Iterate until a runnable process with largest unused chunk of timeslice is found

The Genesis Scheduler – The Code

```
/* 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;
```

Iterate backwards

The Genesis Scheduler – The Code

```
/* 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;
```

Not a valid task, continue the iteration

The Genesis Scheduler – The Code

```c
/* 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;
```

A runnable task and have the maximum available timeslice

The Genesis Scheduler – The Code

```
/* 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;
```

Come out of the loop if you have seen a process with maximum available timeslice

The Genesis Scheduler – The Code

```
      for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
         if (*p)
            (*p)->counter = ((*p)->counter >> 1) + (*p)->priority;
   }
   switch_to(next);
}
```

Iterate over the task array if all the processes have taken the allocated timeslice

The Genesis Scheduler – The Code

```
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
            if (*p)
                (*p)->counter = ((*p)->counter >> 1) + (*p)->priority;
    }
    switch_to(next);
}
```

Reallocate timeslice equals to its previous timeslice divided by 2 plus its priority

The Genesis Scheduler – The Code

```
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
            if (*p)
                (*p)->counter = ((*p)->counter >> 1) + (*p)->priority;
    }
    switch_to(next);
}
```

Trigger a context switch to execute the next task (task having the maximum available timeslice)

```
        if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
            c = (*p)->counter, next = i;
```

The Genesis Scheduler – The Code

```
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
            if (*p)
                (*p)->counter = ((*p)->counter >> 1) + (*p)->priority;
    }
    switch_to(next);
}
```

Trigger a context switch to execute the next task (task having the maximum available timeslice)

Running complexity?

```
            if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                c = (*p)->counter, next = i;
```

The Genesis Scheduler – The Code

```
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
            if (*p)
                (*p)->counter = ((*p)->counter >> 1) + (*p)->priority;
    }
    switch_to(next);
}
```

Trigger a context switch to execute the next task (task having the maximum available timeslice)

Running complexity?

O(n)

```
            if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                c = (*p)->counter, next = i;
```

O(n) Scheduler – Kernel 2.4

- Conceptually same as the Genesis scheduler
  - Update 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
  - +1000: A real time process

O(n) Scheduler – Kernel 2.4

- Conceptually same as the Genesis scheduler
  - Update 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
  - +1000: A real time process

O(n) Scheduler – Kernel 2.4

- Conceptually same as the Genesis scheduler
  - Update 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
  - +1000: A real time process

- **Cons:** A lot of time goes for goodness calculation

O(1) Scheduler – Kernel 2.6

Introduced a number of new features --

- **Global priority scale** (0-139), separation between real-time (0-99) and normal (100-139) processes based on priority values

O(1) Scheduler – Kernel 2.6

Introduced a number of new features --

- **Global priority scale** (0-139), separation between real-time (0-99) and normal (100-139) processes based on priority values
- **Early preemption:** When a new task enters in the TASK_RUNNING state, it can preempt the currently running process if having lower priority

O(1) Scheduler – Kernel 2.6

Introduced a number of new features --

- **Global priority scale** (0-139), separation between real-time (0-99) and normal (100-139) processes based on priority values
- **Early preemption:** When a new task enters in the TASK_RUNNING state, it can preempt the currently running process if having lower priority
- Static priority for real-time tasks, dynamic priority for normal tasks

O(1) Scheduler – Kernel 2.6

Introduced a number of new features --

- **Global priority scale** (0-139), separation between real-time (0-99) and normal (100-139) processes based on priority values
- **Early preemption:** When a new task enters in the TASK_RUNNING state, it can preempt the currently running process if having lower priority
- Static priority for real-time tasks, dynamic priority for normal tasks
- Dynamic priority for the normal tasks is decided depending on its interactivity in the past – how frequently the process runs CPU-bound instructions

Moving from O(n) to O(1)

- **The Good:** Timeslices are allocated in proportion to the priority of the tasks
  - higher priority tasks get more timeslice to complete the job

Moving from O(n) to O(1)

- **The Good:** Timeslices are allocated in proportion to the priority of the tasks
  - higher priority tasks get more timeslice to complete the job

- **The Bad:** Priorities are not fixed all the time
  - New tasks with higher priority can arrive – preempt the currently running process?
  - Process priority can get updated dynamically (Kernel 2.6)

Moving from O(n) to O(1)

- **The Good:** Timeslices are allocated in proportion to the priority of the tasks
  - higher priority tasks get more timeslice to complete the job
- **The Bad:** Priorities are not fixed all the time
  - New tasks with higher priority can arrive – preempt the currently running process?
  - Process priority can get updated dynamically (Kernel 2.6)
- **The Ugly:** Recompute the amount of timeslices to be allocated to each task after the current epoch is complete – **O(n)**
  - Iterate over the entire runqueue
  - Recompute each task's priority and the timeslice to be allocated

O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure

O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure

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

O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure

Active Array

[0]
[1]
[2]
…
[99]
[100]
…
[139]

Timeslice available

Expired Array

[0]
[1]
[2]
…
[99]
[100]
…
[139]

O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure

Active Array

[0]
[1]
[2]
…
[99]
[100]
…
[139]

Expired Array

[0]
[1]
[2]
…
[99]
[100]
…
[139]

Timeslice complete

O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure

Active Array

[0]
[1] 1
[2]
...
[99]
[100]
...
[139]

Expired Array

[0]
[1]
[2]
...
[99]
[100]
...
[139]

O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure

Active Array

[0]
[1] 1
[2]
...
[99] 2
[100]
...
[139]

Expired Array

[0]
[1]
[2]
...
[99]
[100]
...
[139]

O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure

Active Array

[0]
[1] 1 3
[2]
...
[99] 2
[100]
...
[139]

Expired Array

[0]
[1]
[2]
...
[99]
[100]
...
[139]

O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure

Active Array

[0]
[1] 1 3
[2]
…
[99] 2
[100]
…
[139] 4

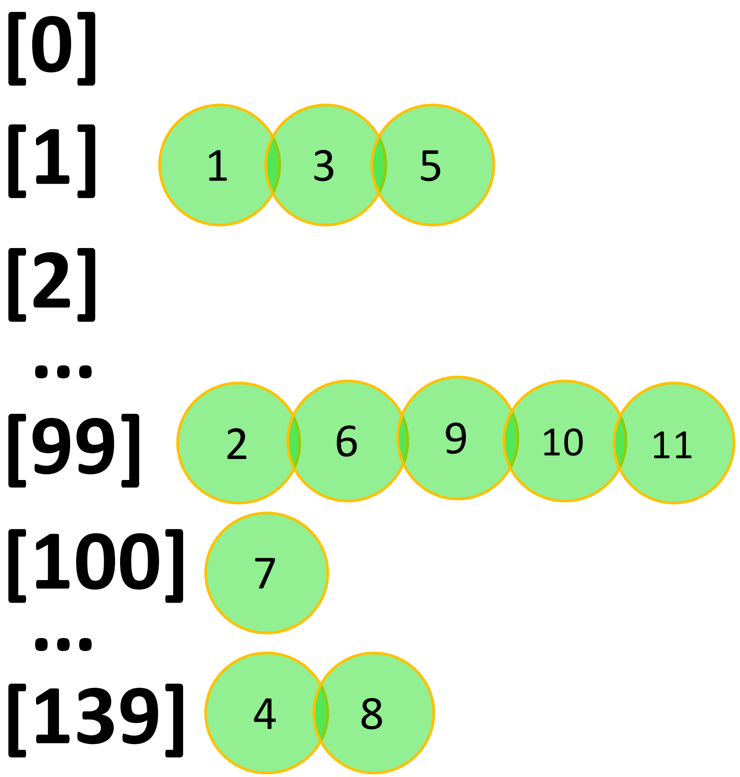
Expired Array

[0]
[1]
[2]
…
[99]
[100]
…
[139]

O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure

Active Array



Expired Array

[0]
[1]
[2]
...
[99]
[100]
...
[139]

O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure

Active Array

MAX_PRIO

[0]

[1] 1 3 5

[2]

...

[99] 2 6 9 10 11

[100] 7

...

[139] 4 8

MIN_PRIO

Process the array from top to bottom in increasing order of the priority value

Expired Array

[0]

[1]

[2]

...

[99]

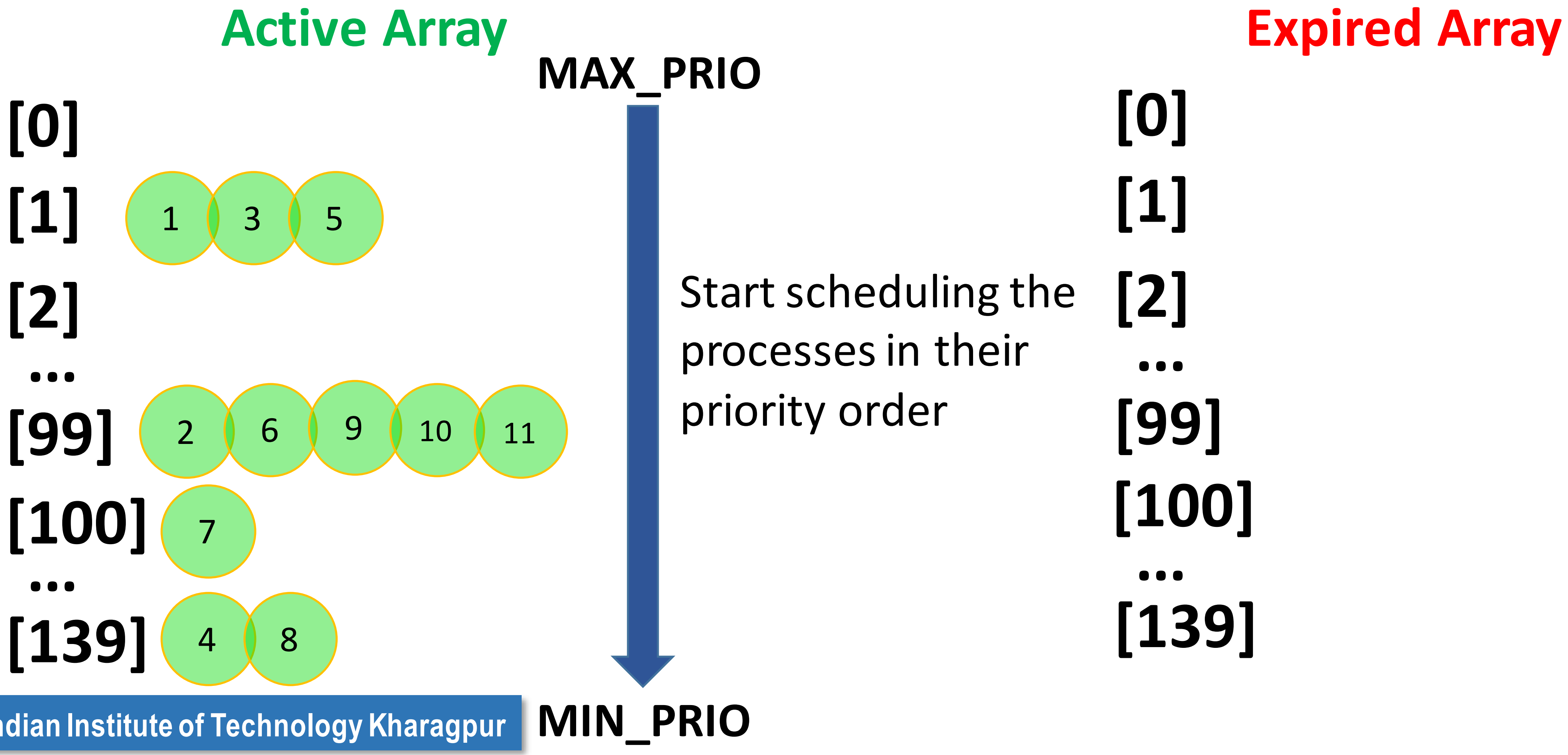
[100]

...

[139]

O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure



O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure

Active Array

[0]
[1] 1 3 5
[2]
...
[99] 2 6 9 10 11
[100] 7
...
[139] 4 8

MAX_PRIO

Timeslice for a process is calculated from its priority

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

Prio ≥ 120
T = (140-Prio)*5

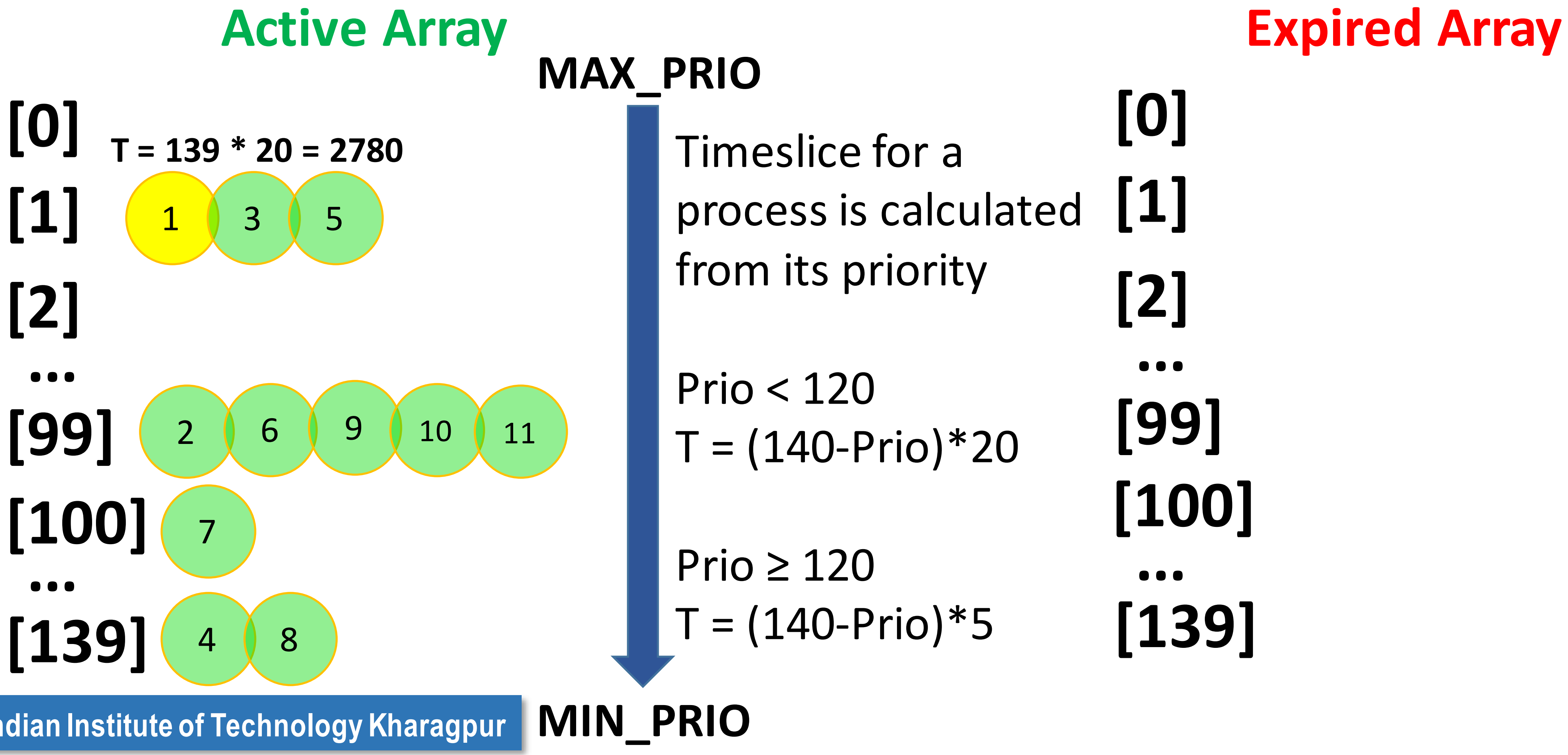
MIN_PRIO

Expired Array

[0]
[1]
[2]
...
[99]
[100]
...
[139]

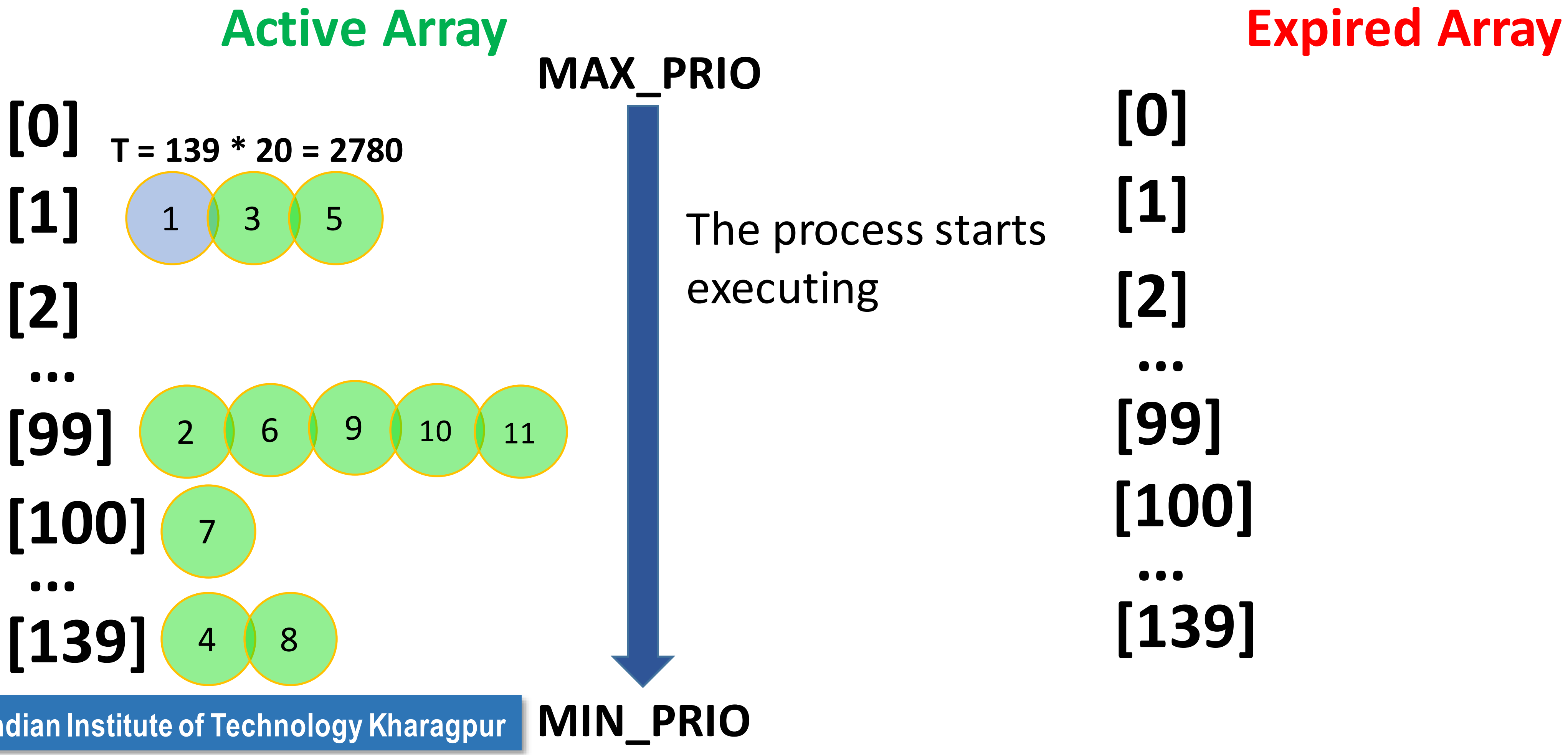
O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure



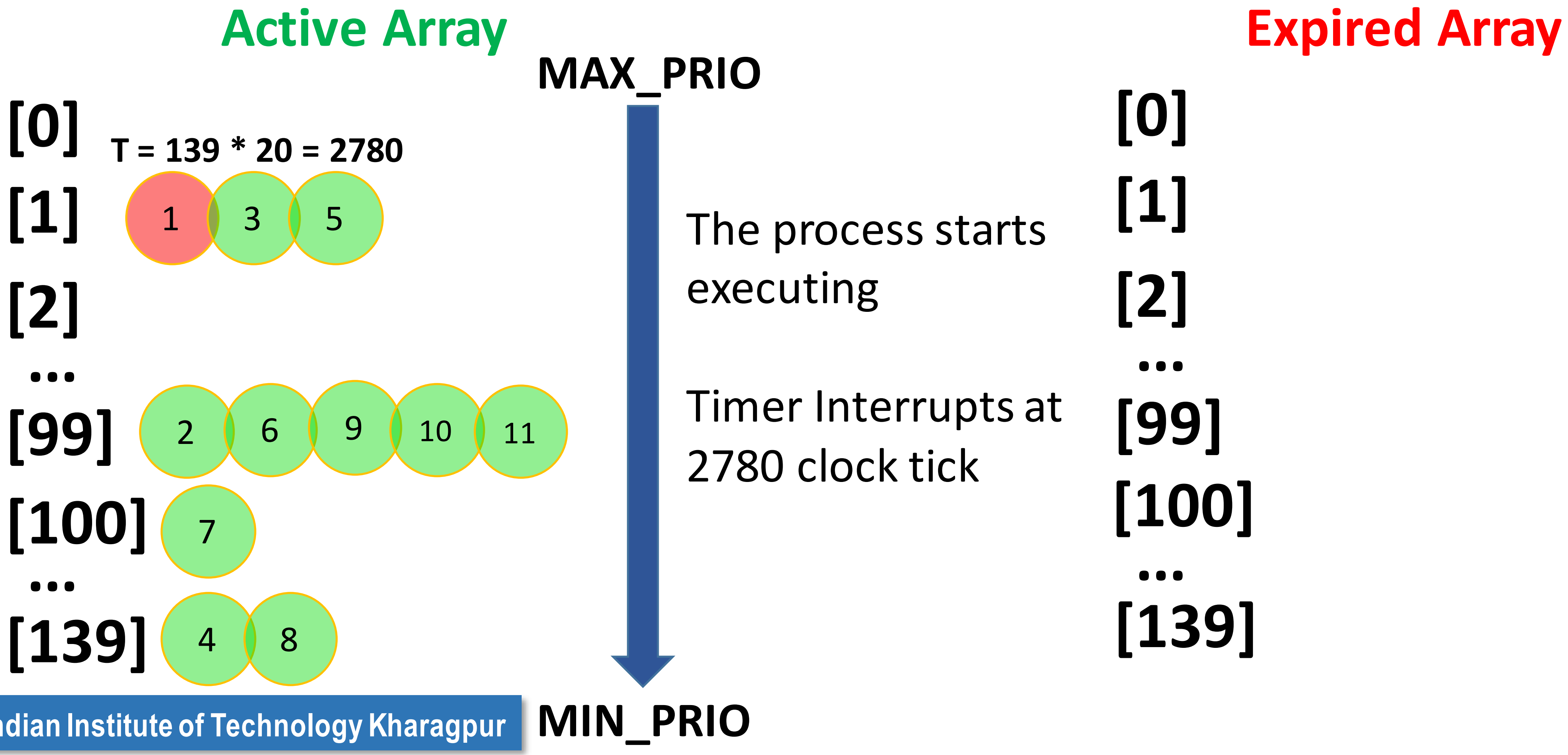
O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure



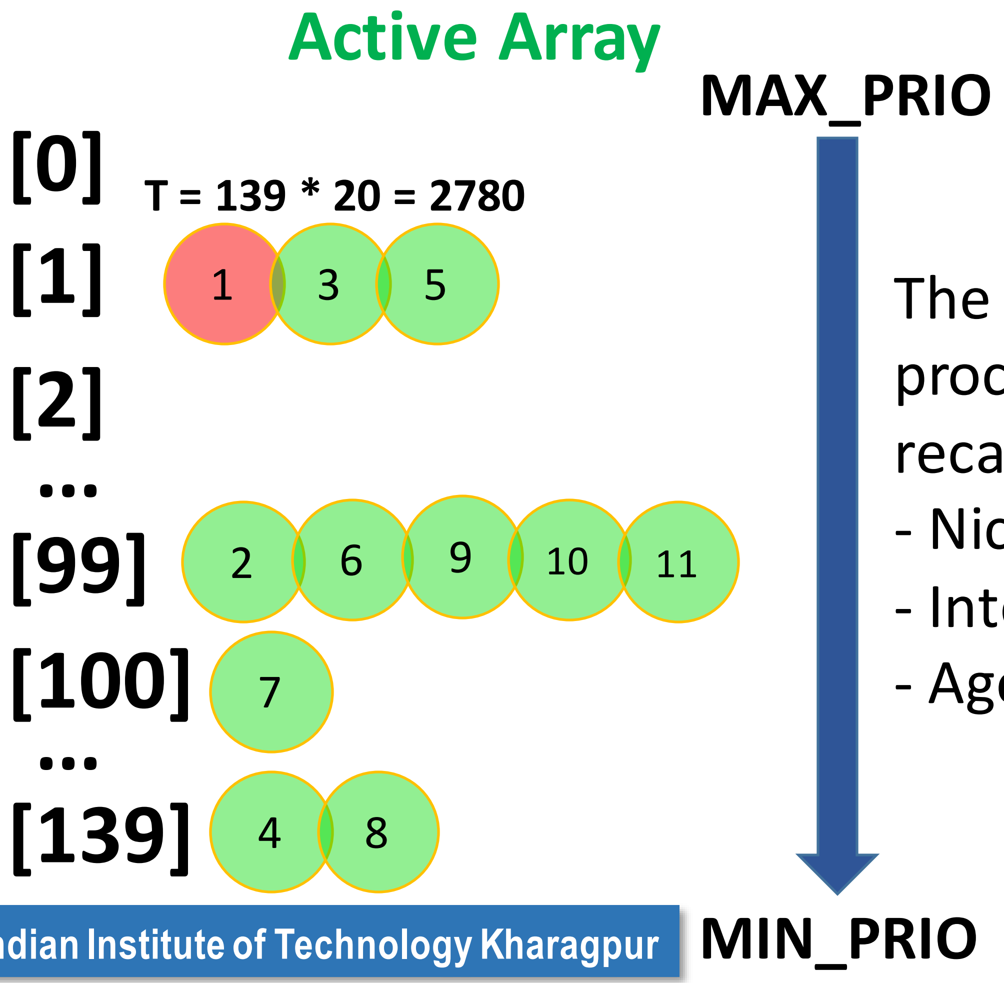
O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure



O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure



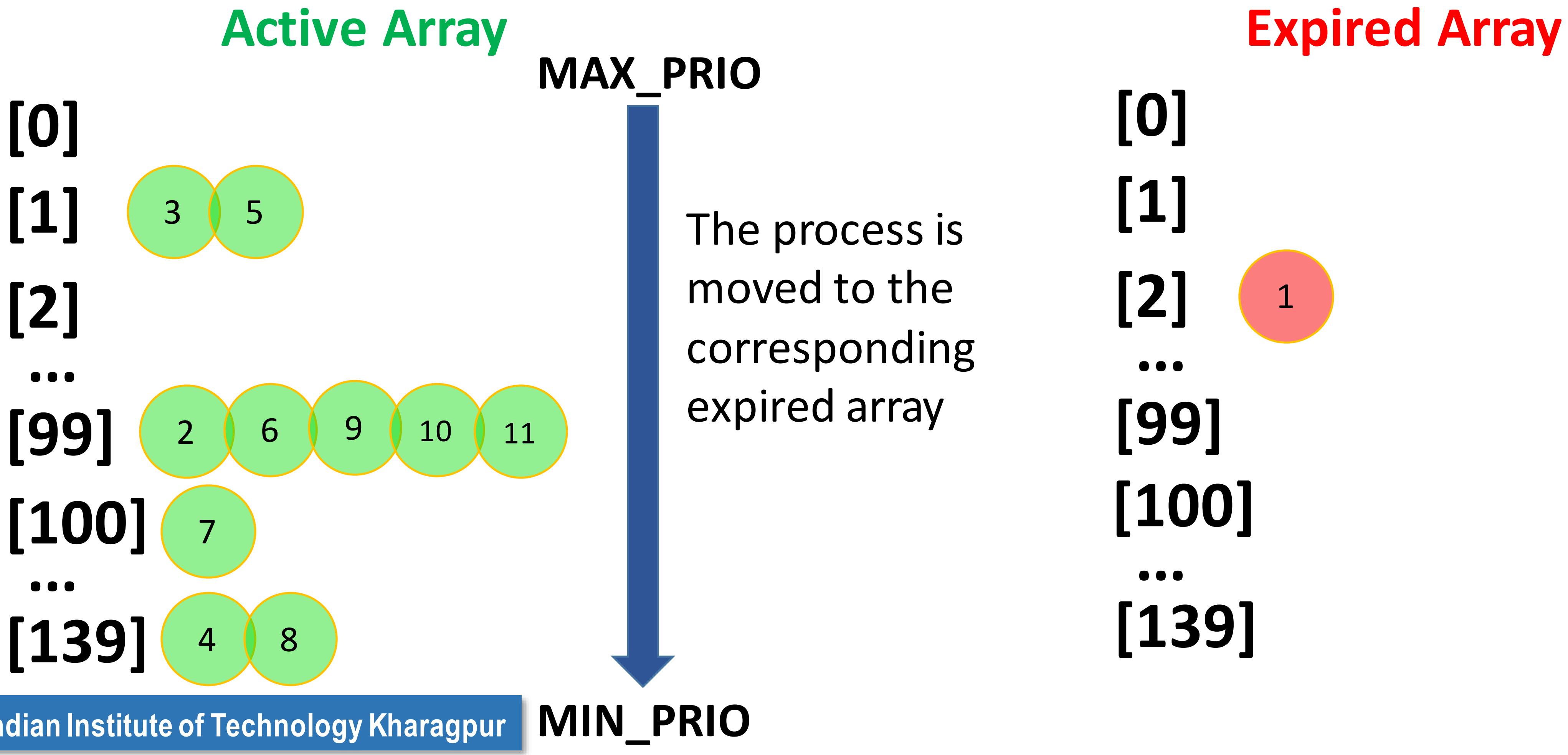
The priority of the process is recalculated
- Niceness
- Interactivity
- Ageing

Expired Array

[0]
[1]
[2]
...
[99]
[100]
...
[139]

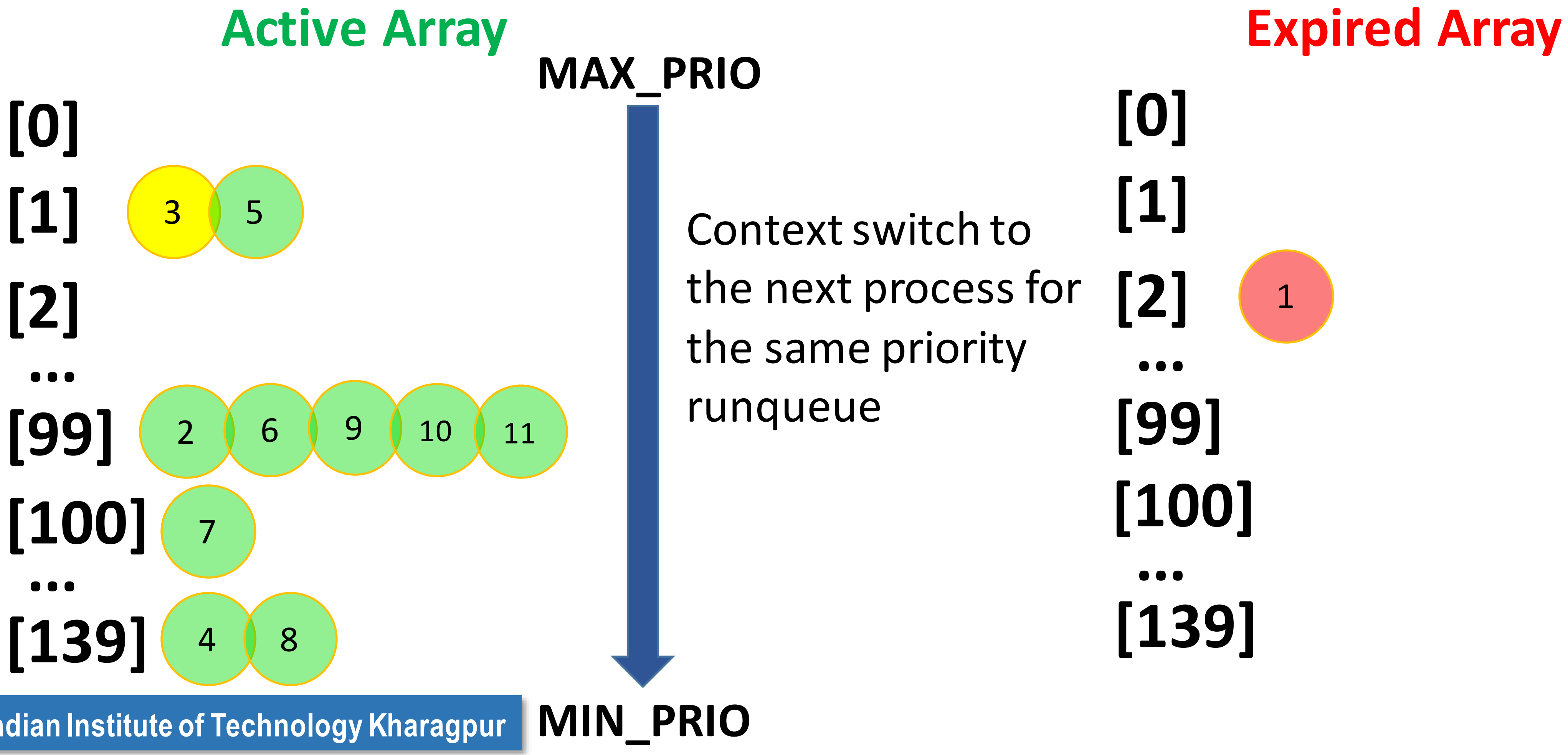
O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure



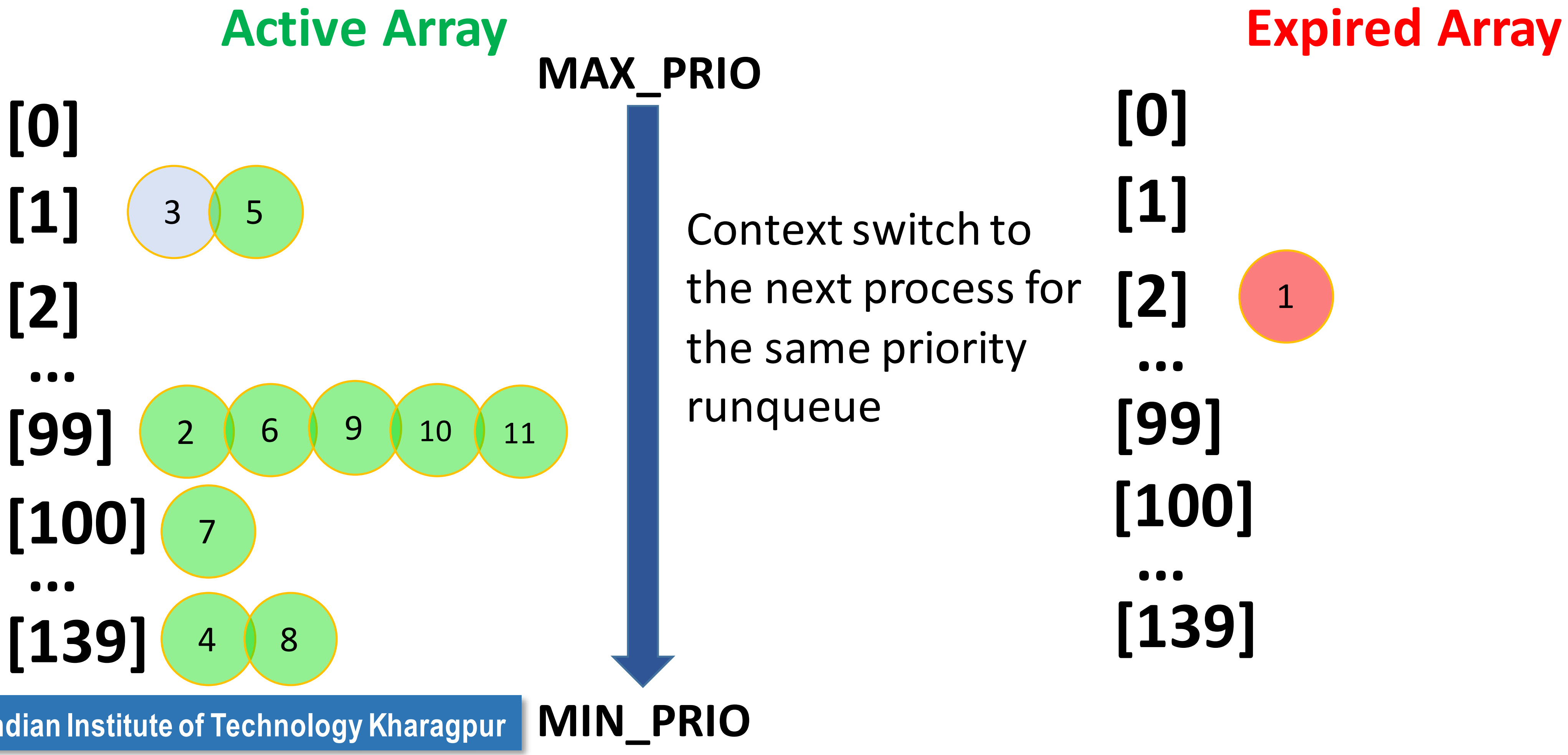
O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure



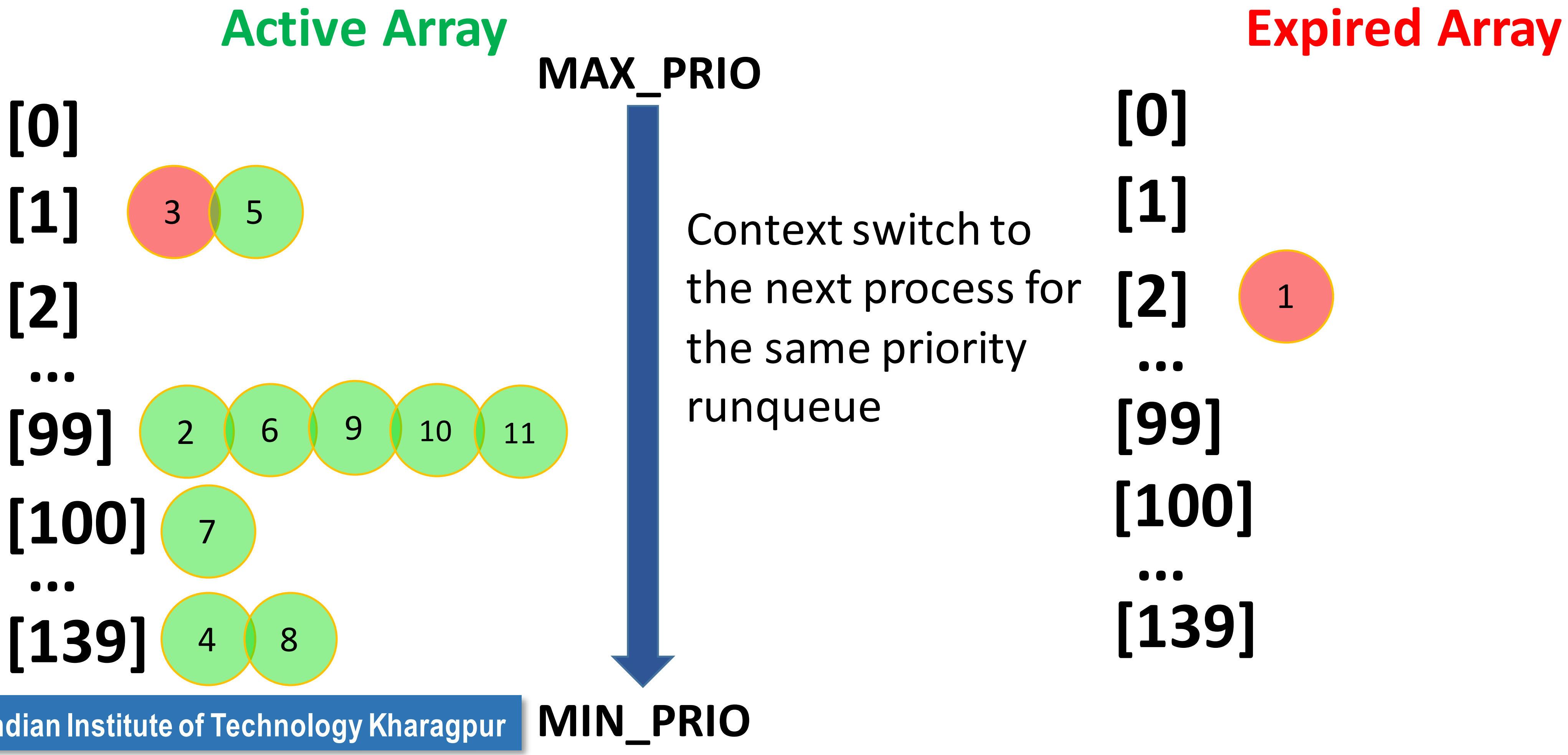
O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure



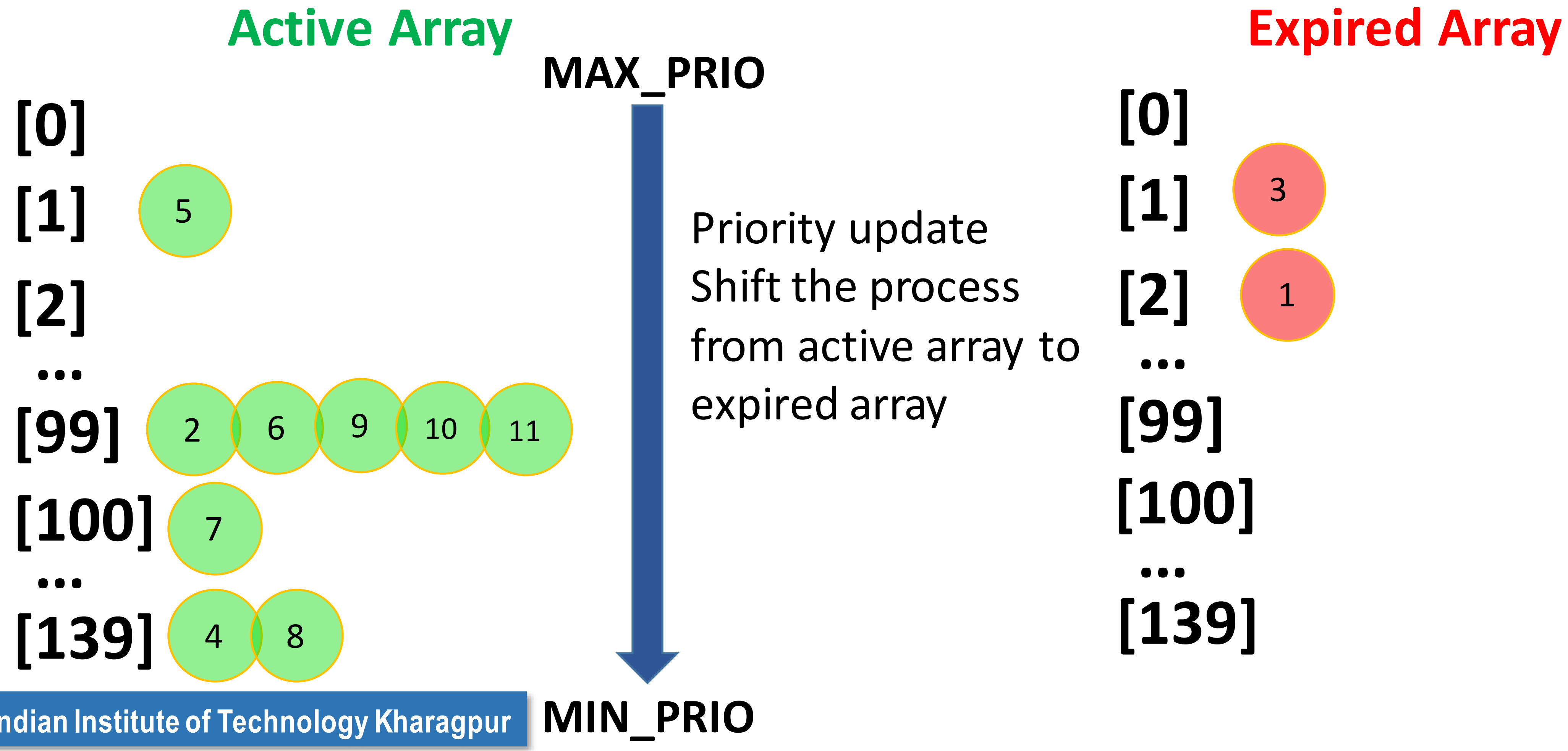
O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure



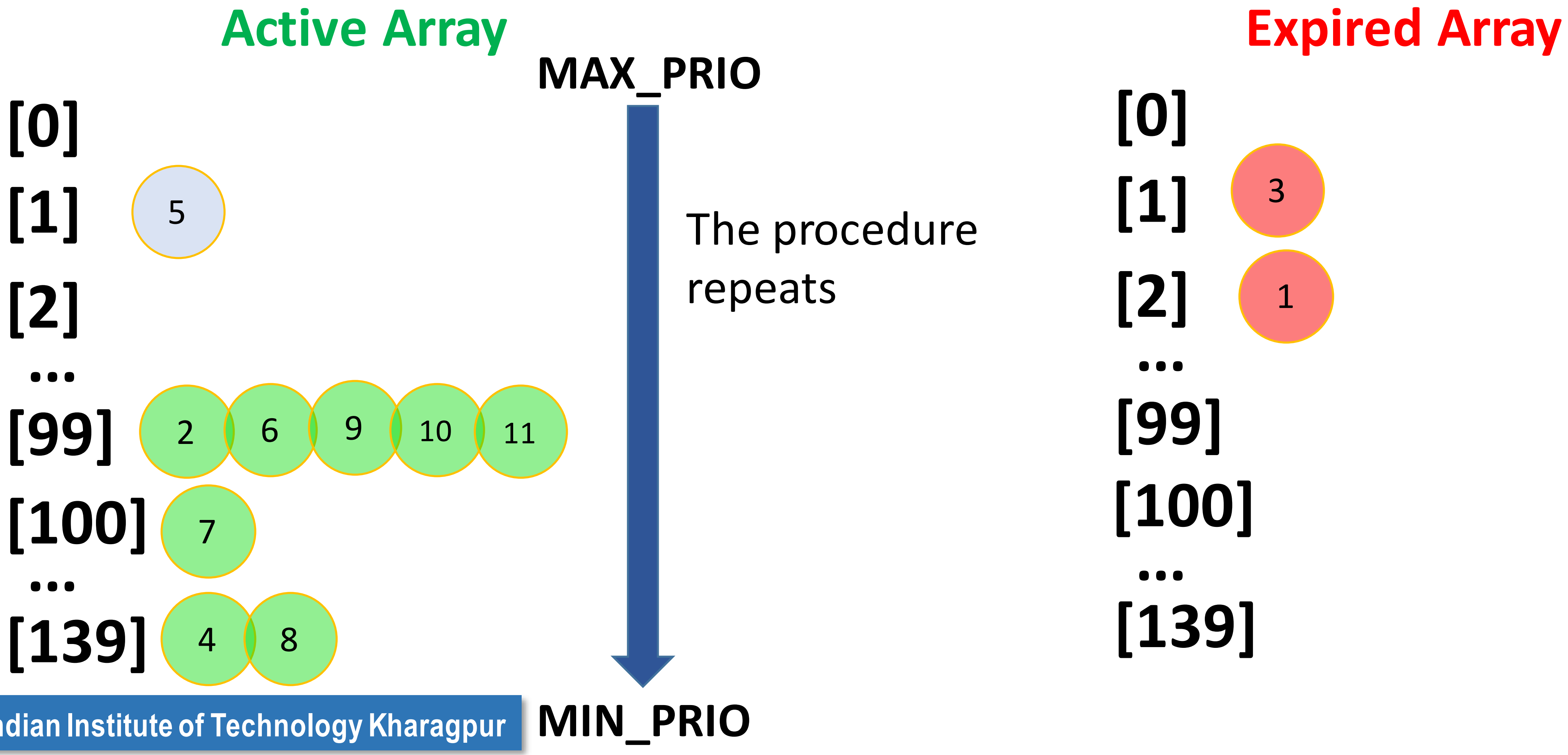
O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure



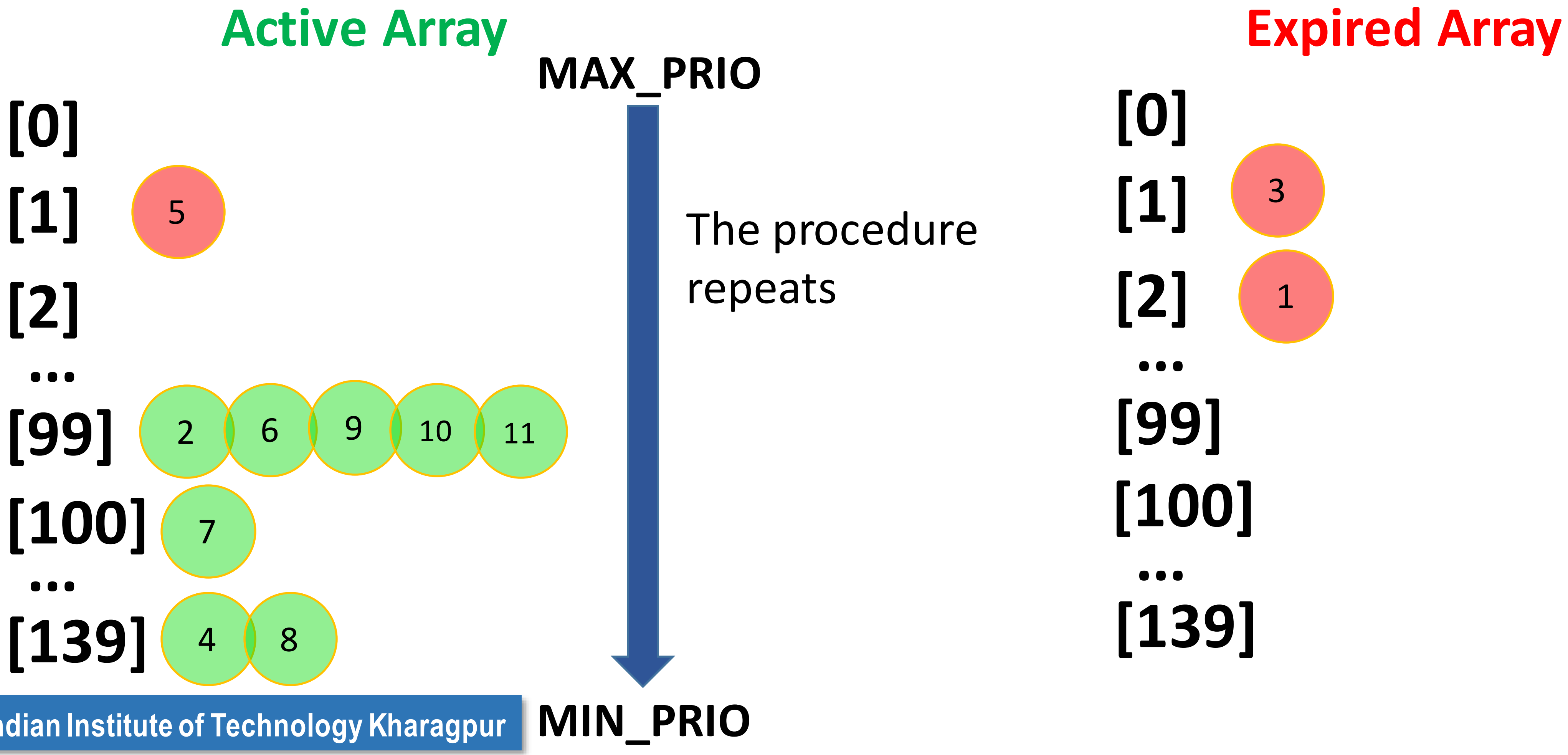
O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure



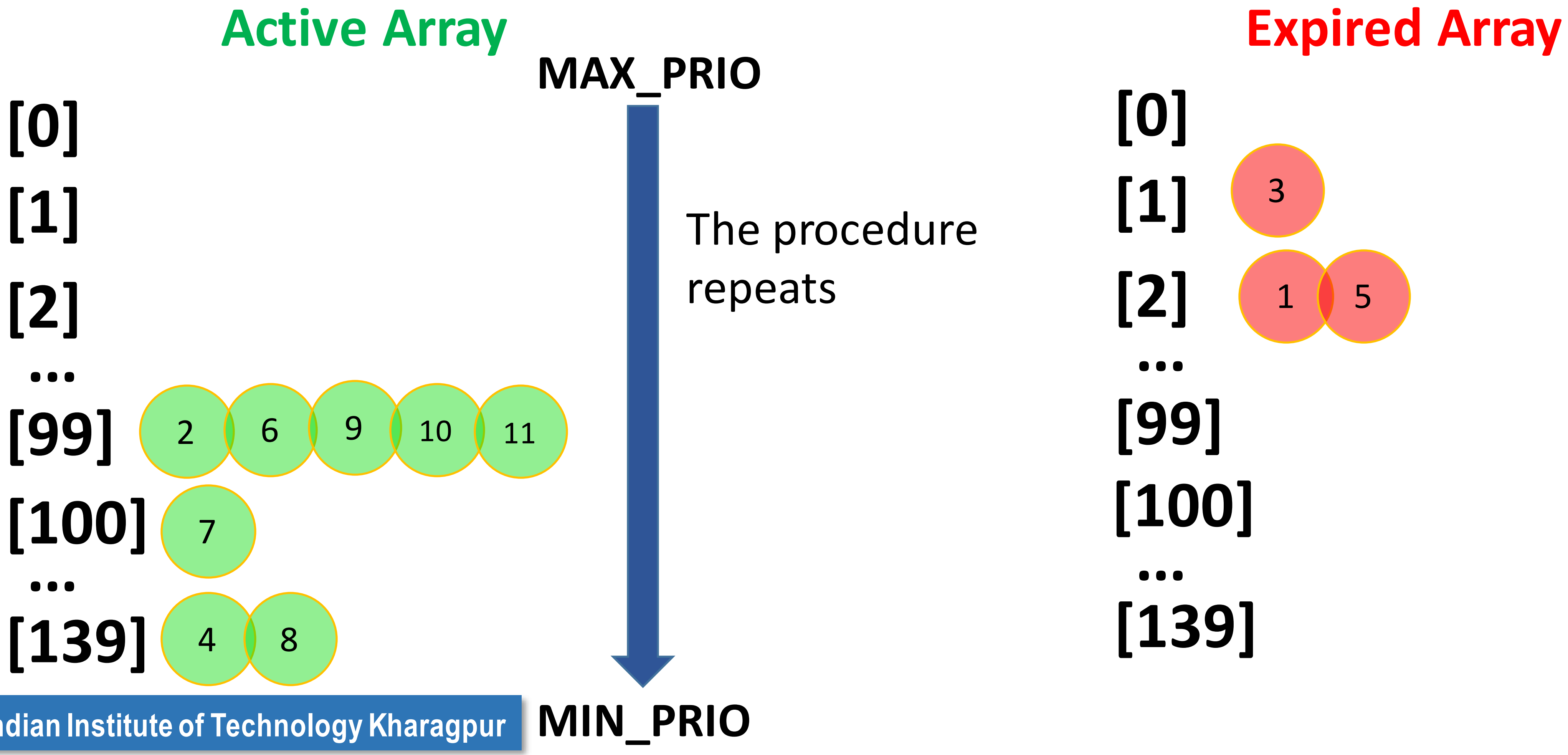
O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure



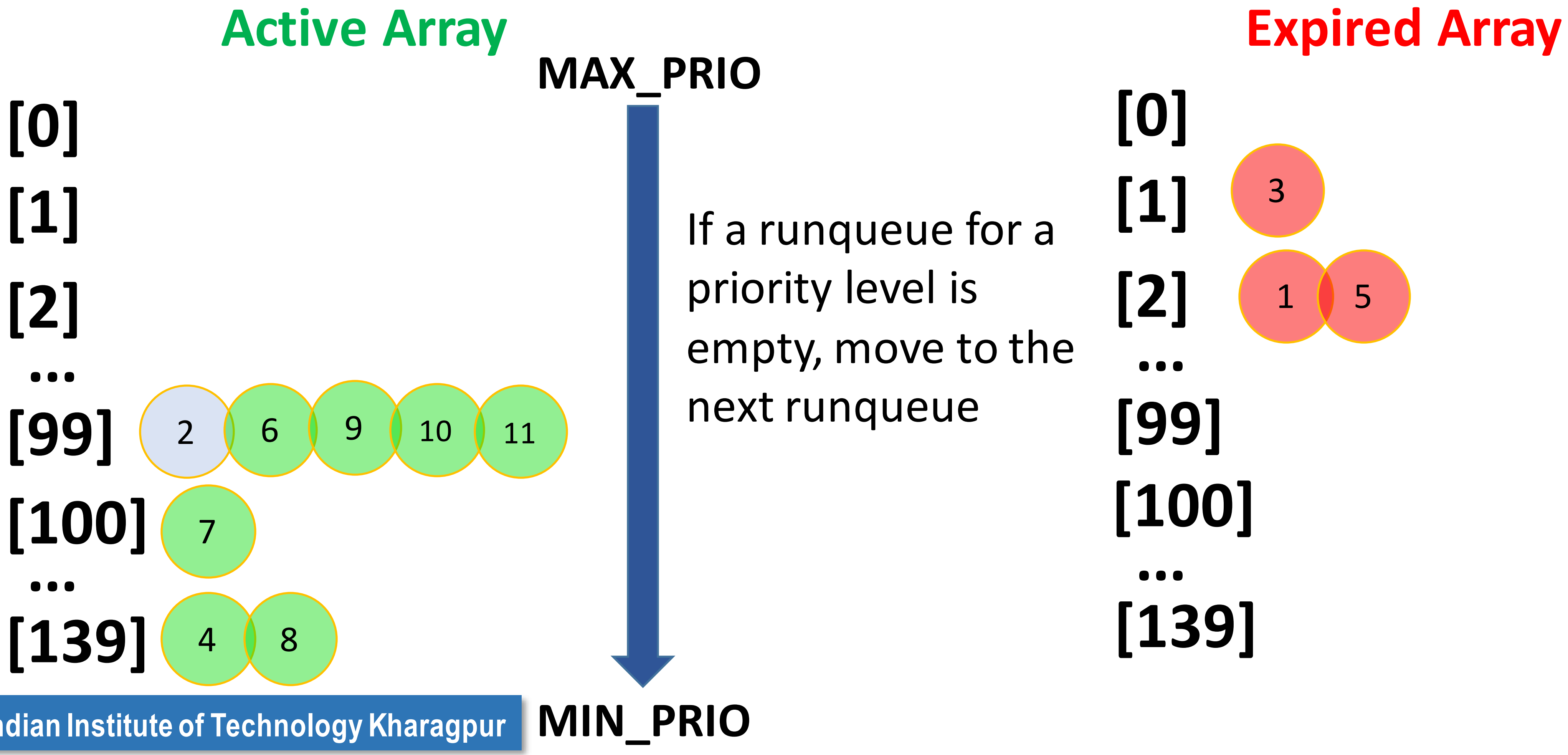
O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure



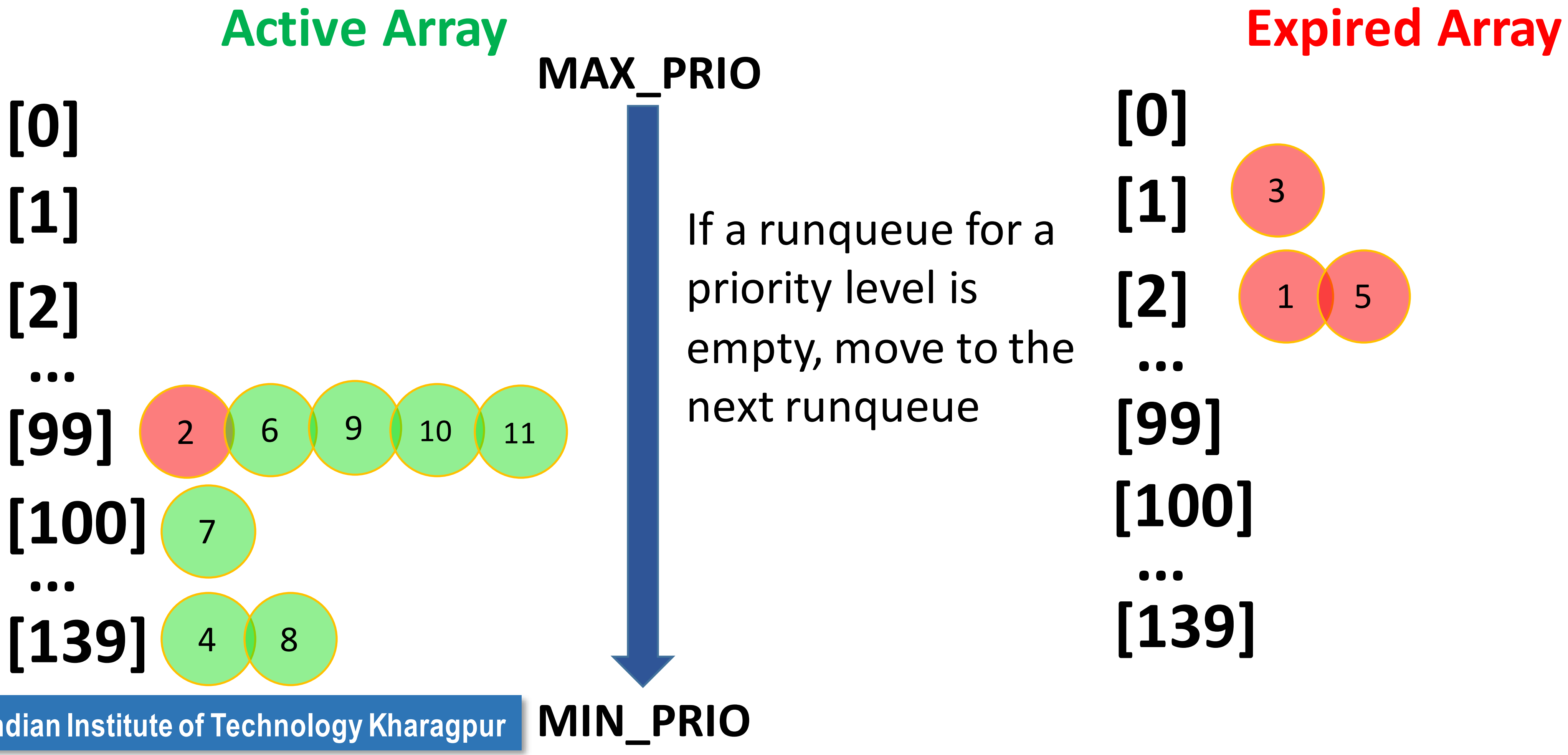
O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure



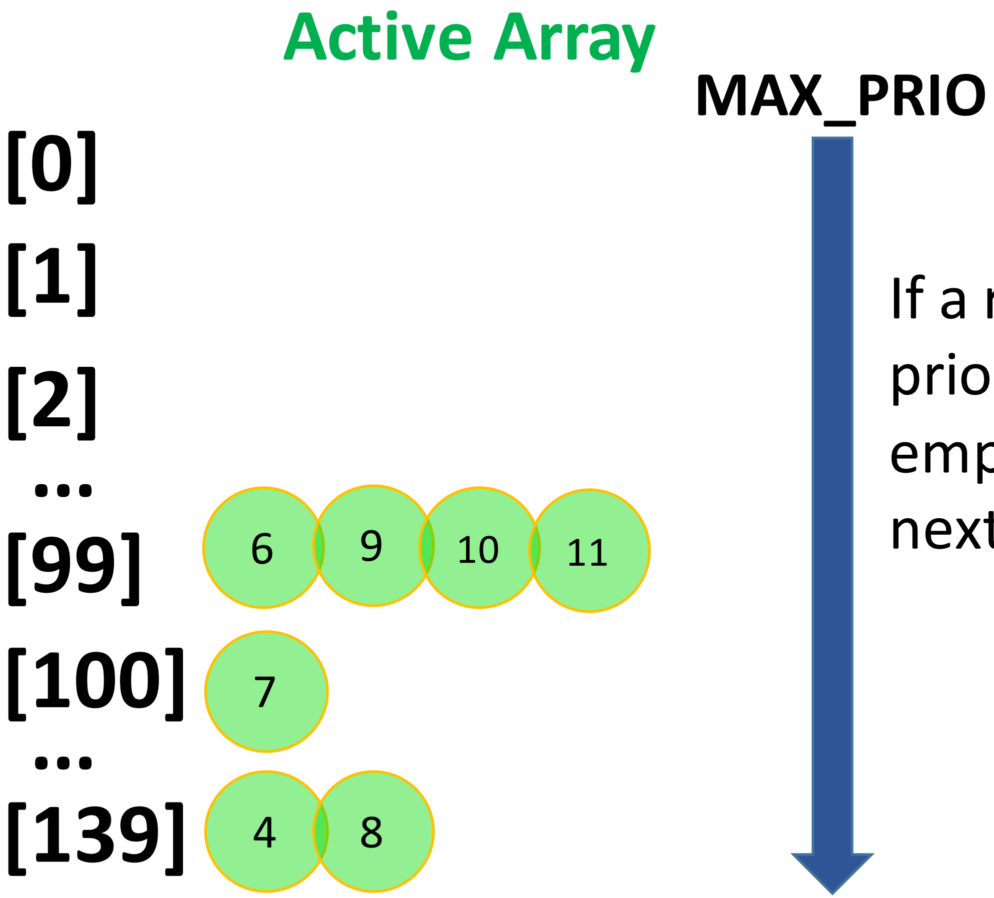
O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure



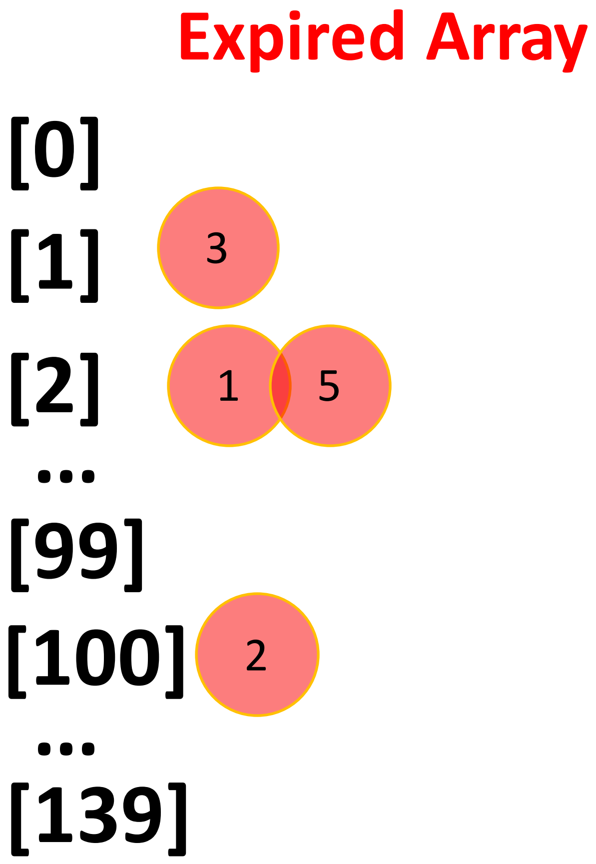
O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure



MIN_PRIO

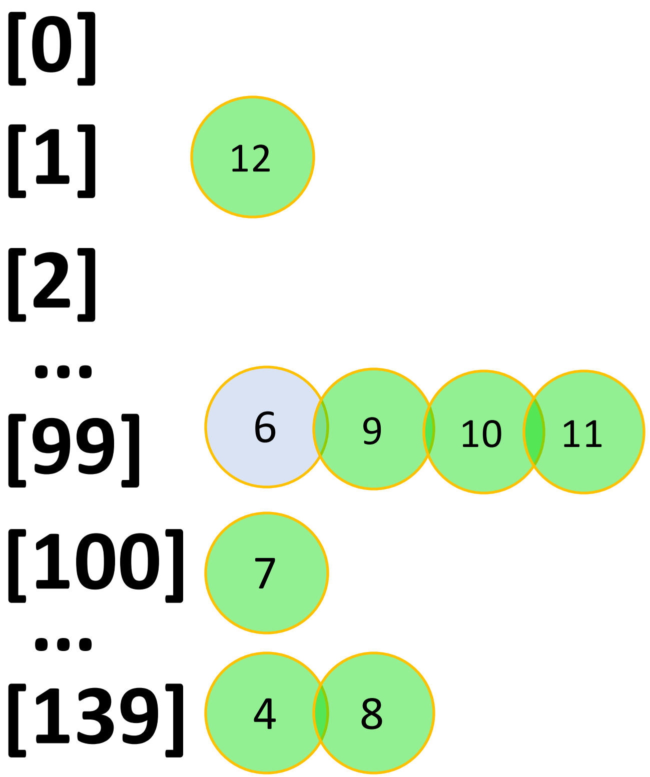
If a runqueue for a priority level is empty, move to the next runqueue



O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure

Active Array

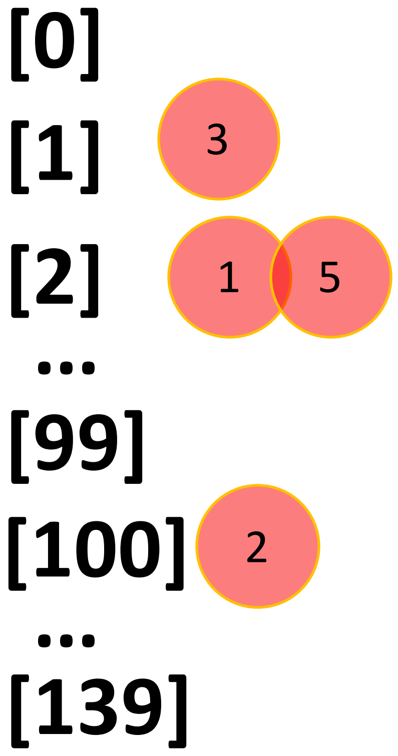


MAX_PRIO

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

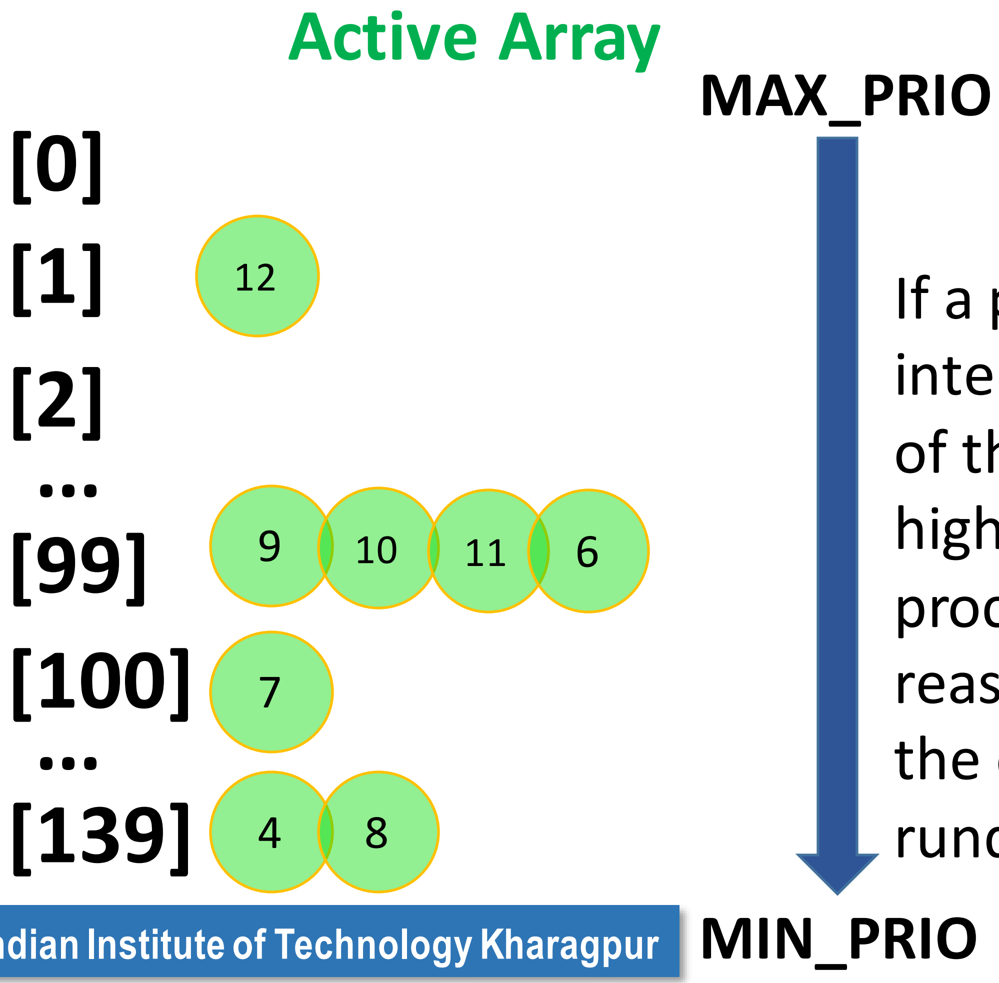
MIN_PRIO

Expired Array

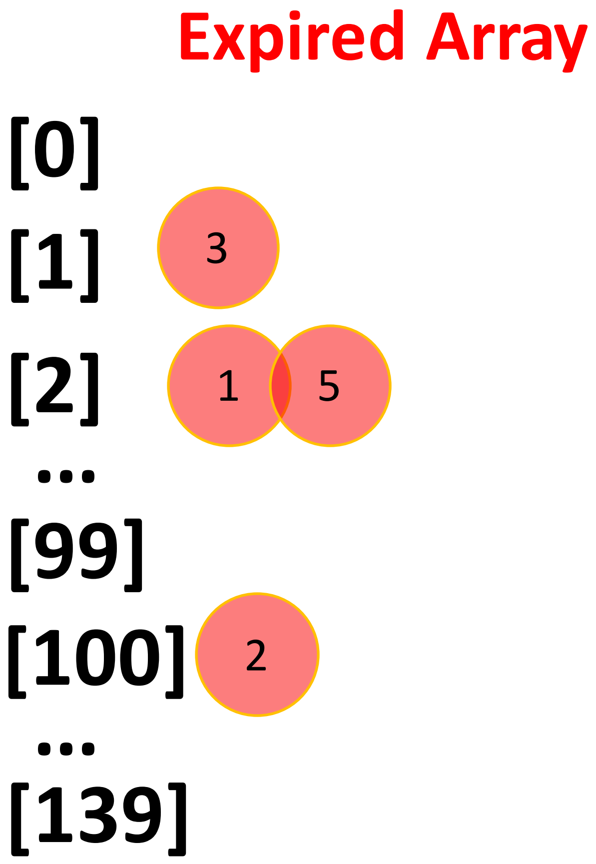


O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure

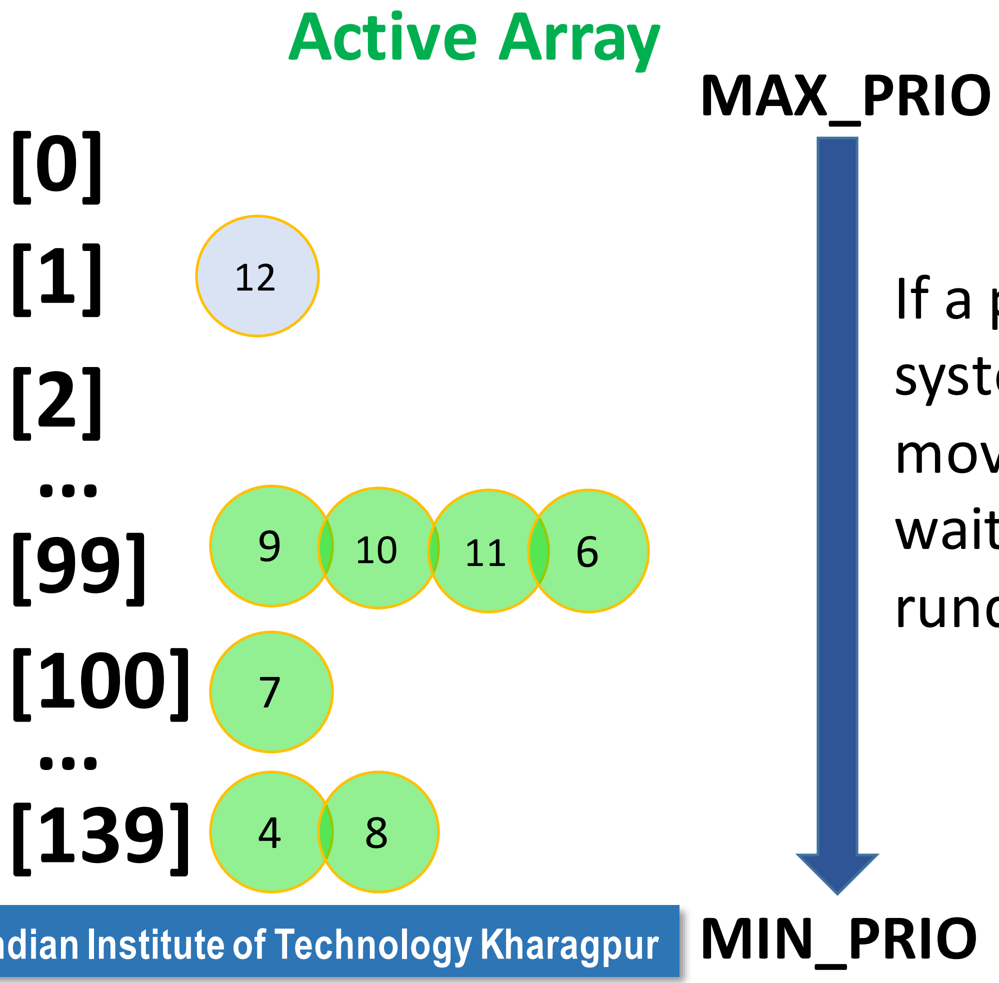


If a process is interrupted because of the arrival of a higher priority process, or for other reason, move it to the end of the runqueue

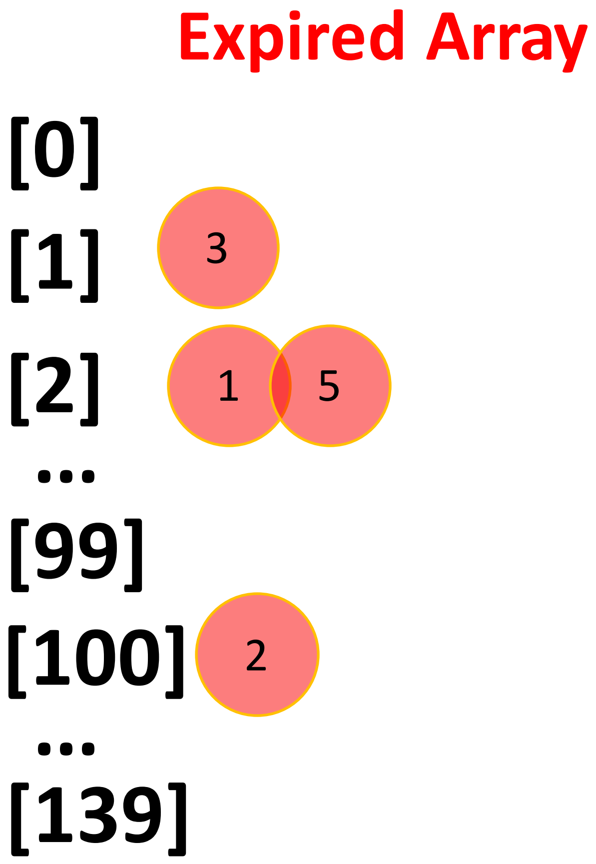


O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure

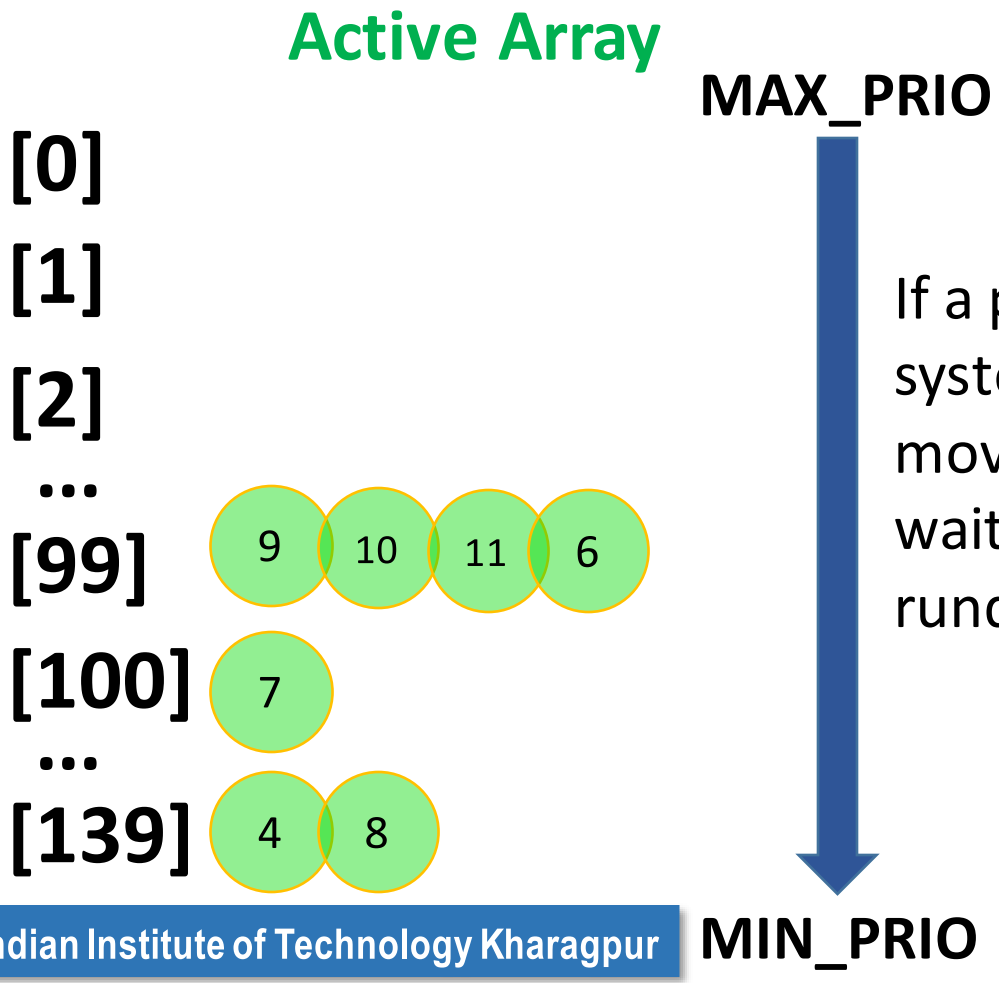


If a process makes a system call for I/O, move it to the waitqueue from the runqueue

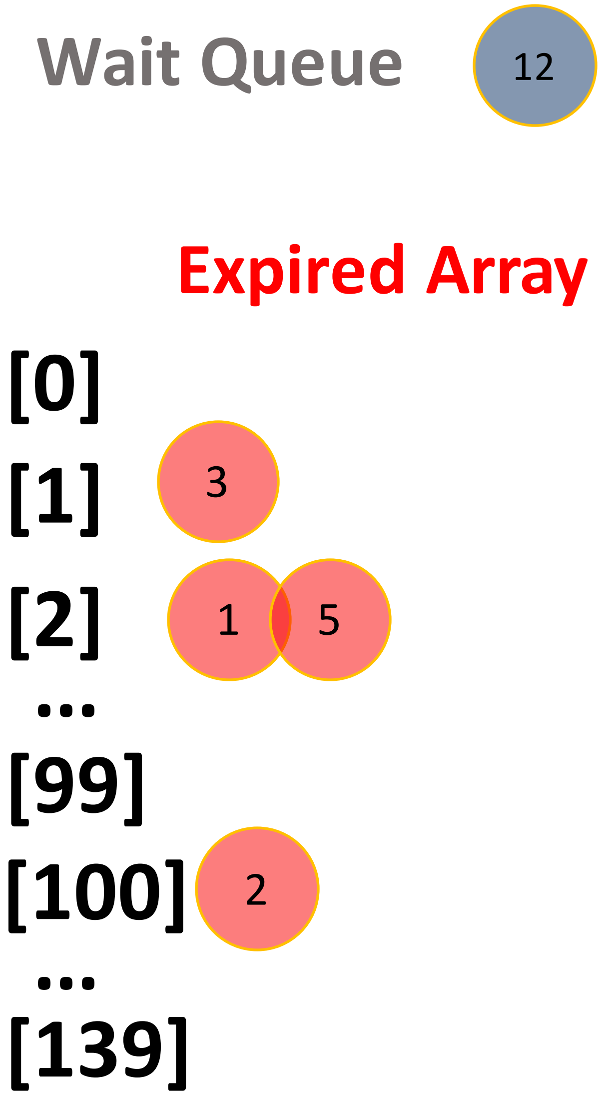


O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure

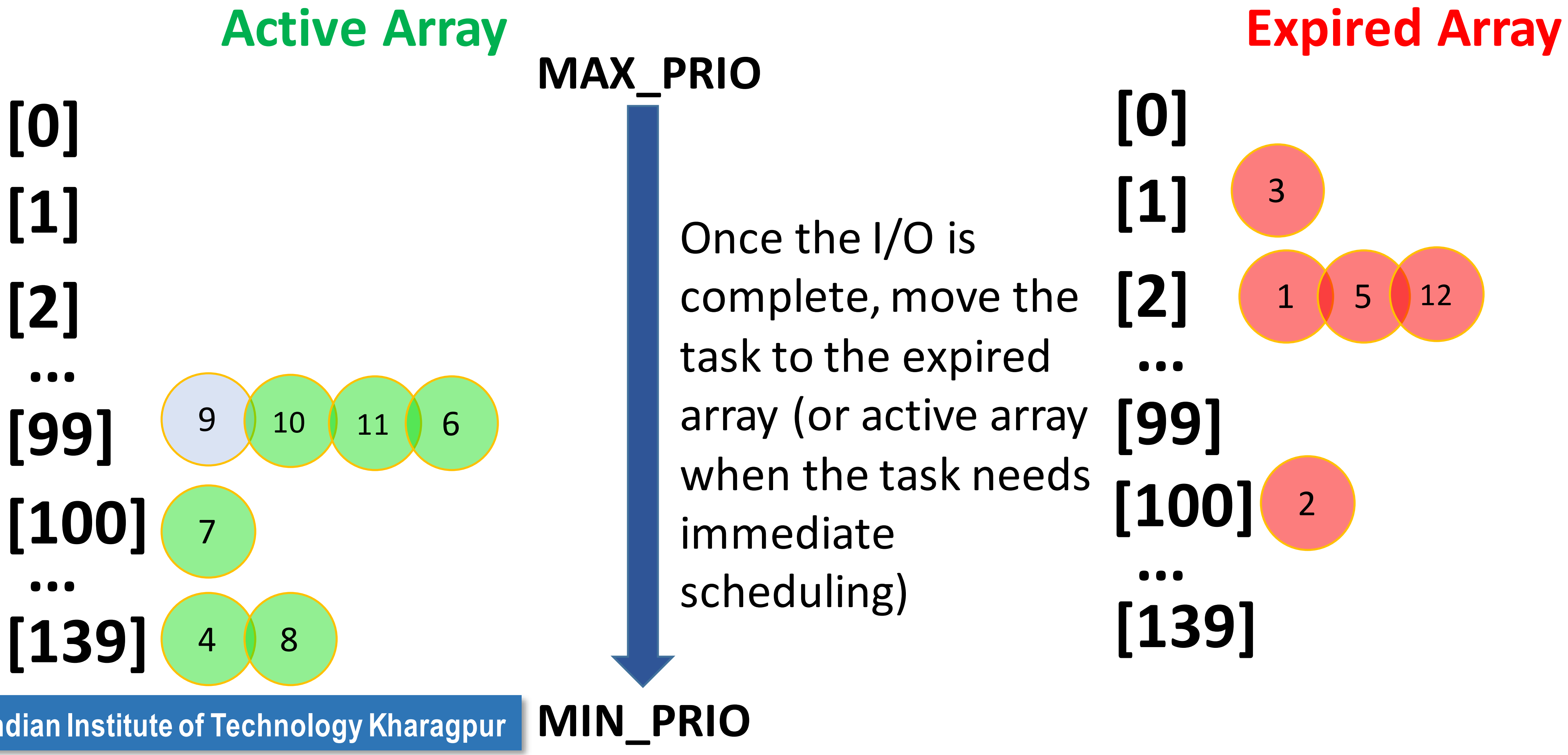


If a process makes a system call for I/O, move it to the waitqueue from the runqueue



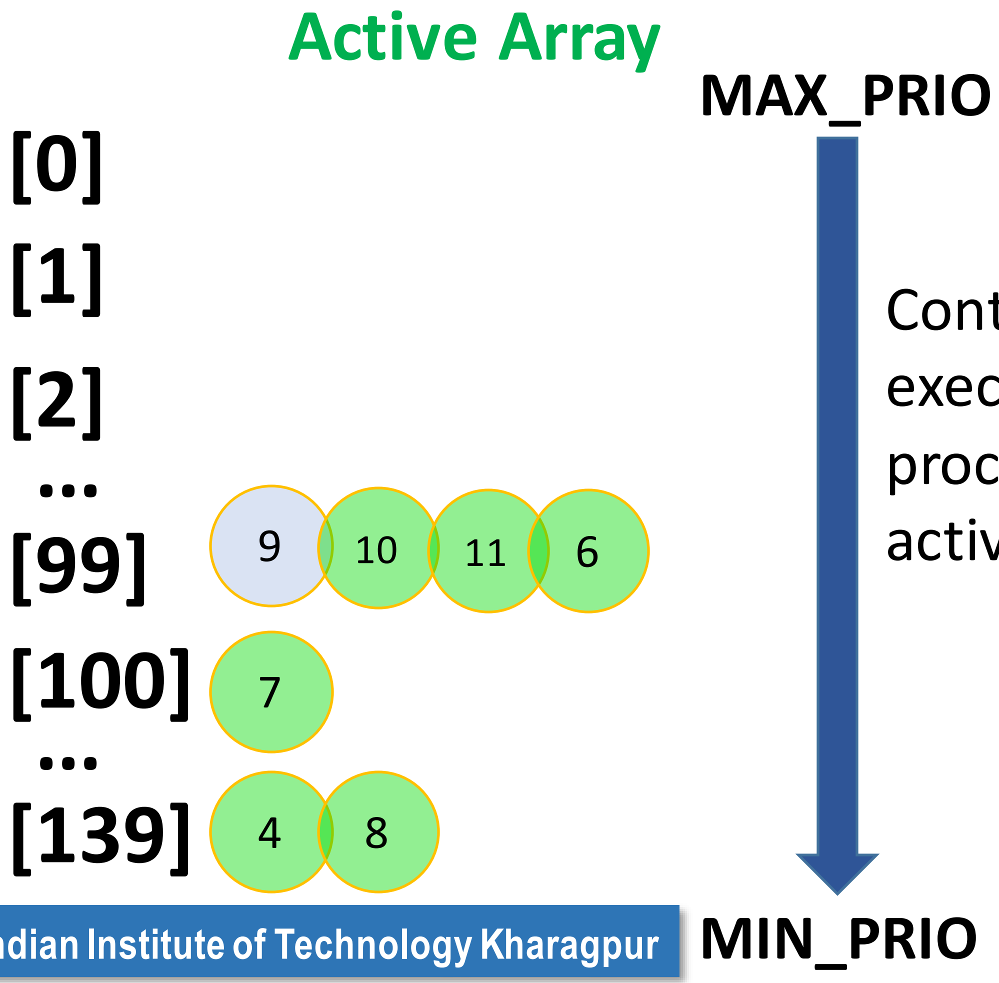
O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure

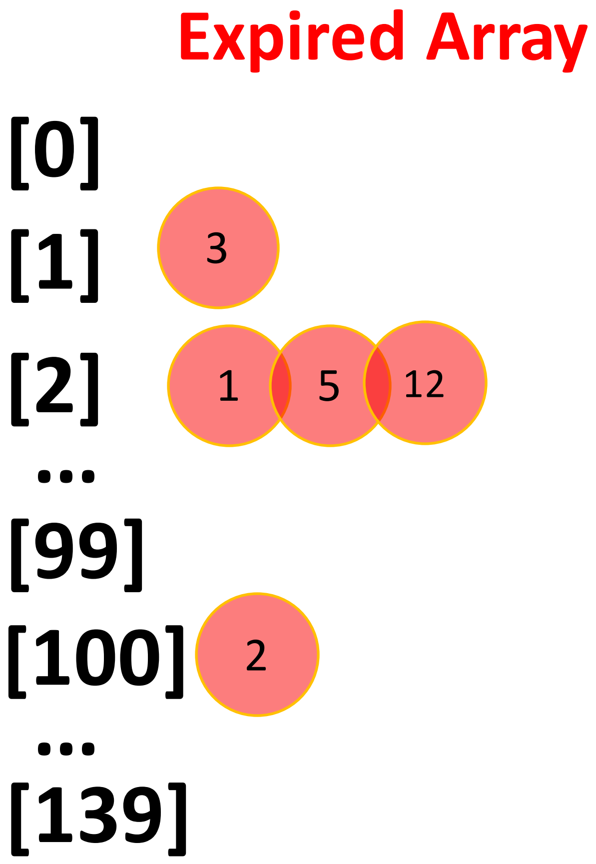


O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure



Continue the execution of all the processes from the active array



O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure

Active Array

MAX_PRIO

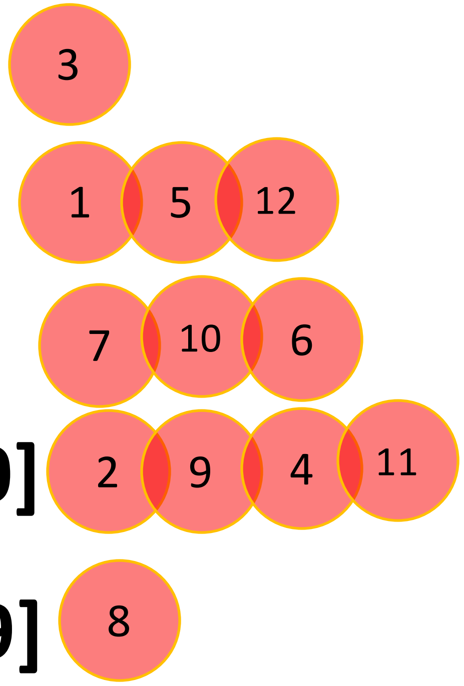
[0]
[1]
[2]
...
[99]
[100]
...
[139]

Continue the execution of all the processes from the active array

MIN_PRIO

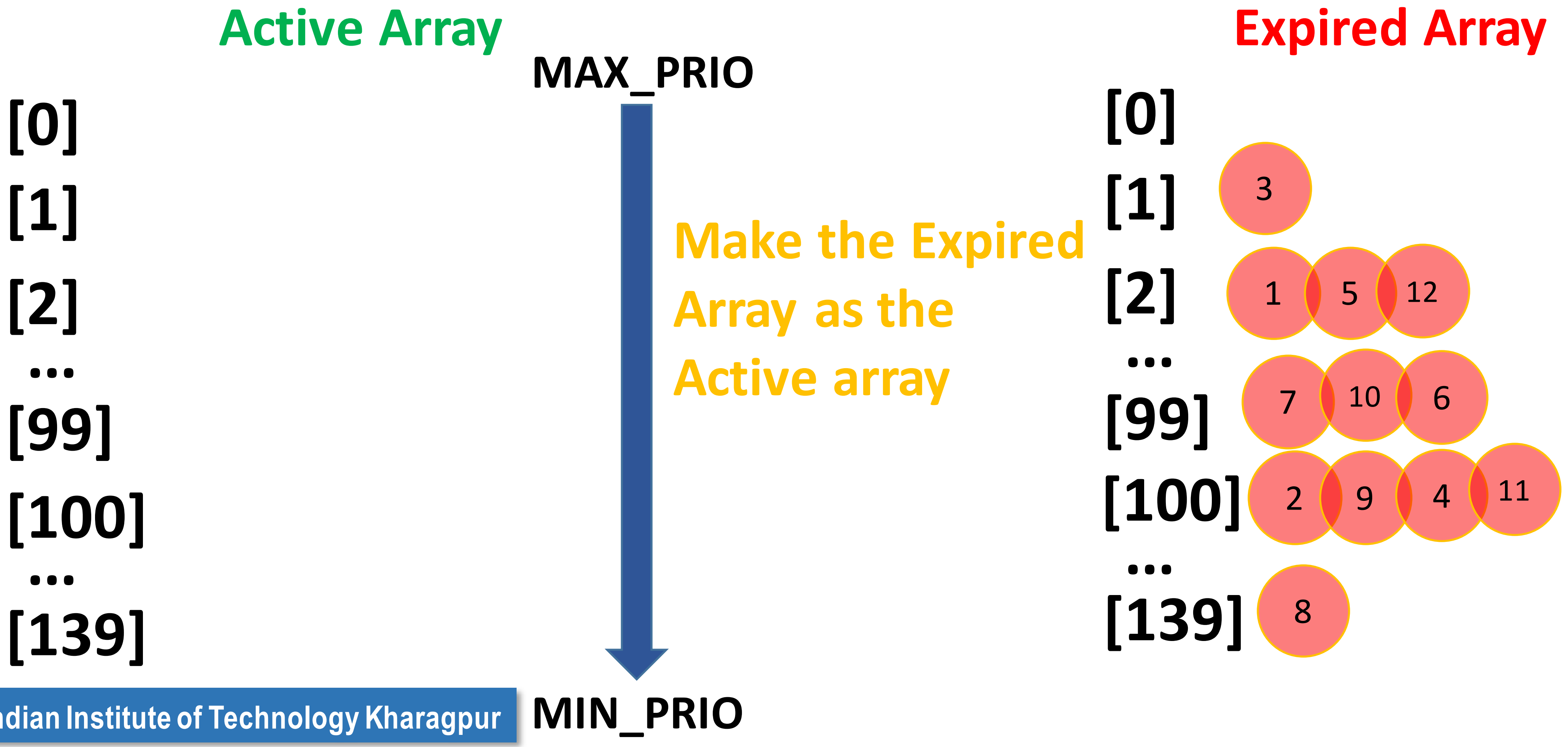
Expired Array

[0]
[1]
[2]
...
[99]
[100]
...
[139]



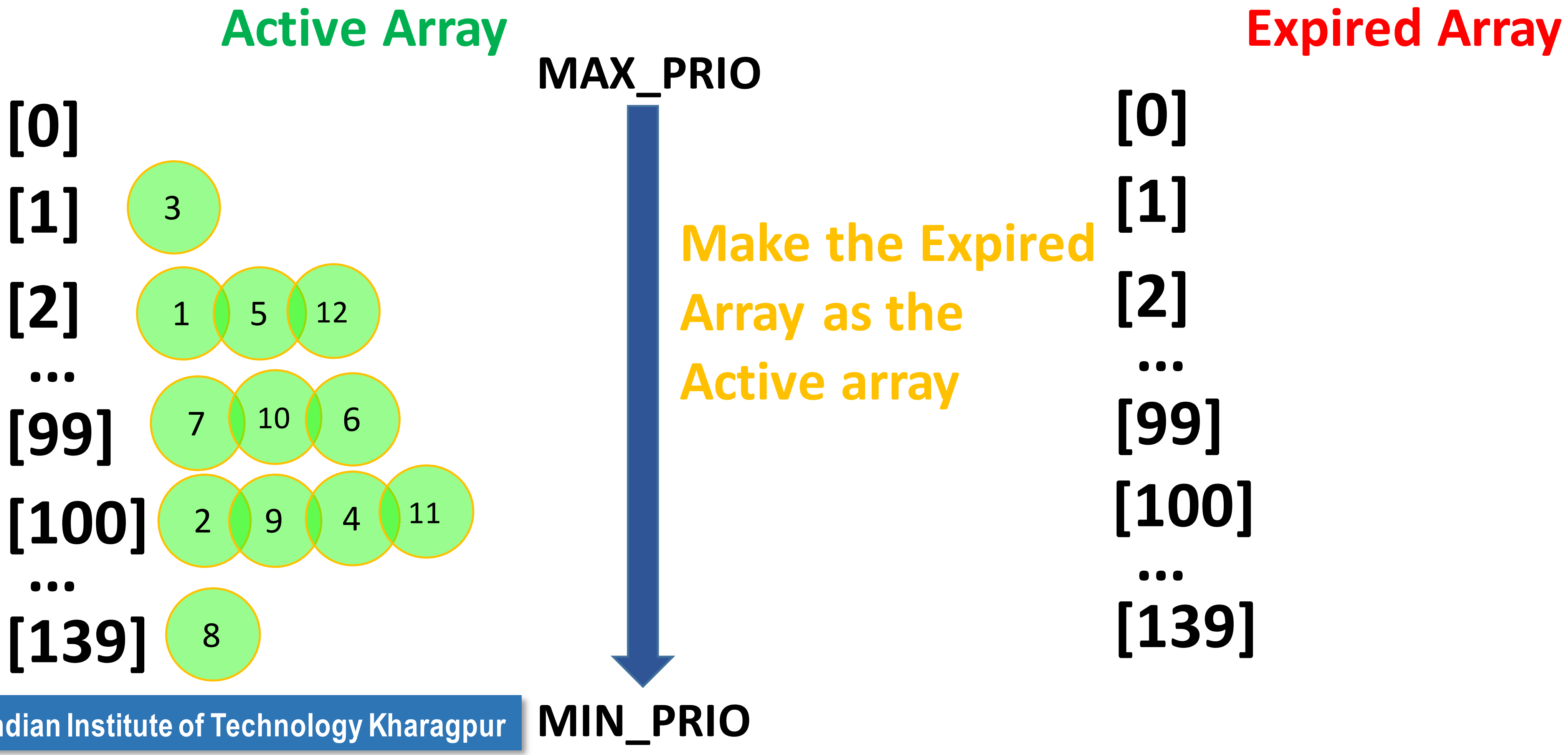
O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure



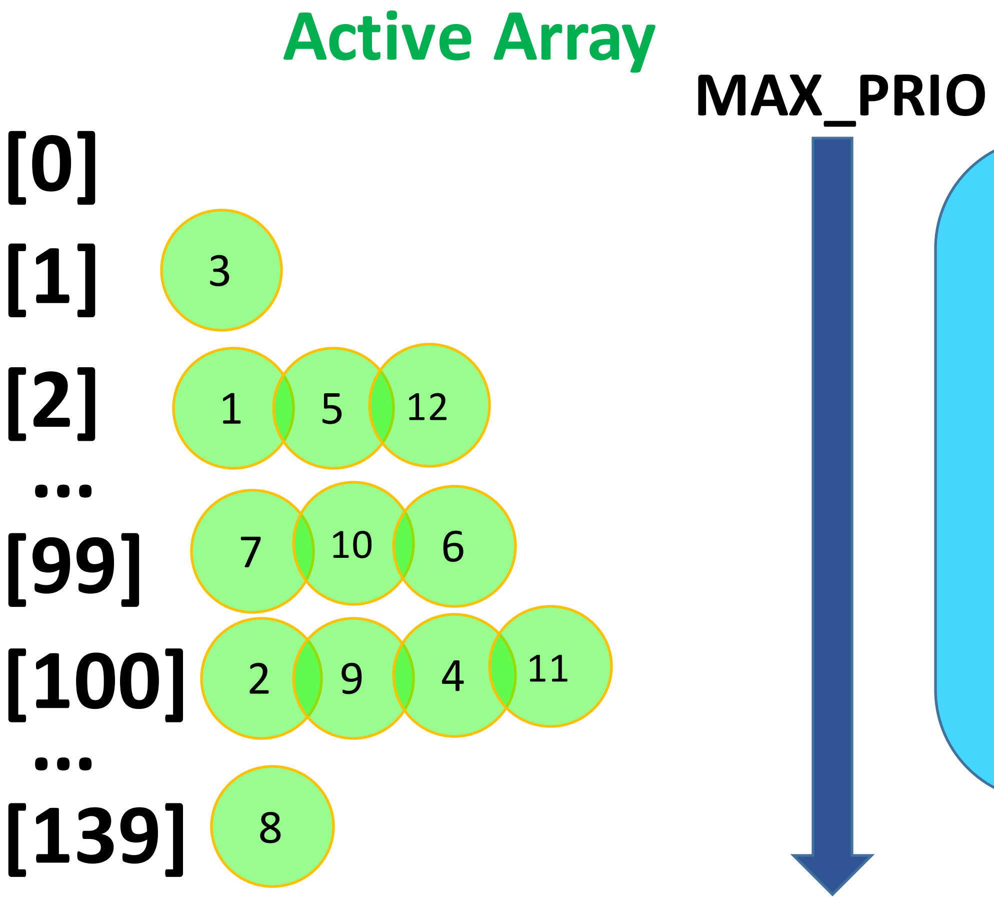
O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure



O(1) Scheduler – Extract MAX Priority Task in O(1)

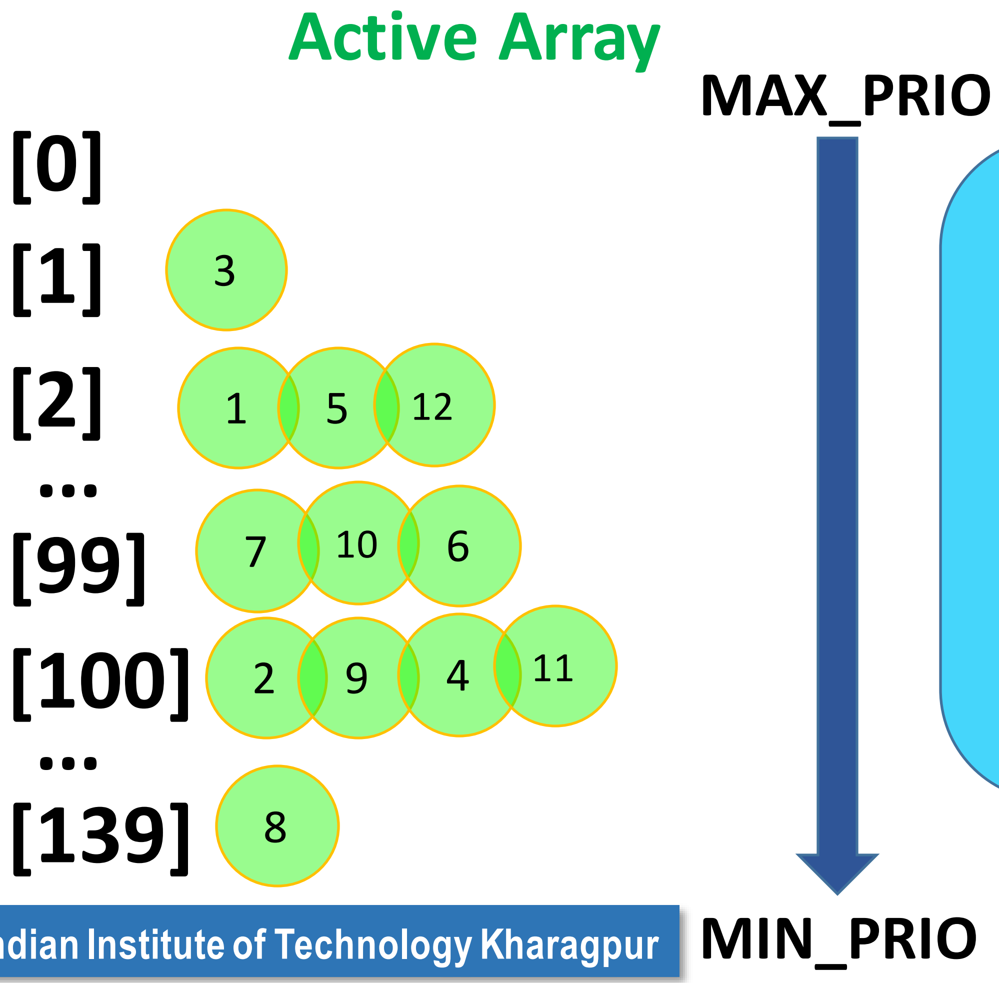
- Reorganize the runqueue data structure



You do not need to iterate over all the processes to update their priority, at the end of the current epoch

O(1) Scheduler – Extract MAX Priority Task in O(1)

- Reorganize the runqueue data structure



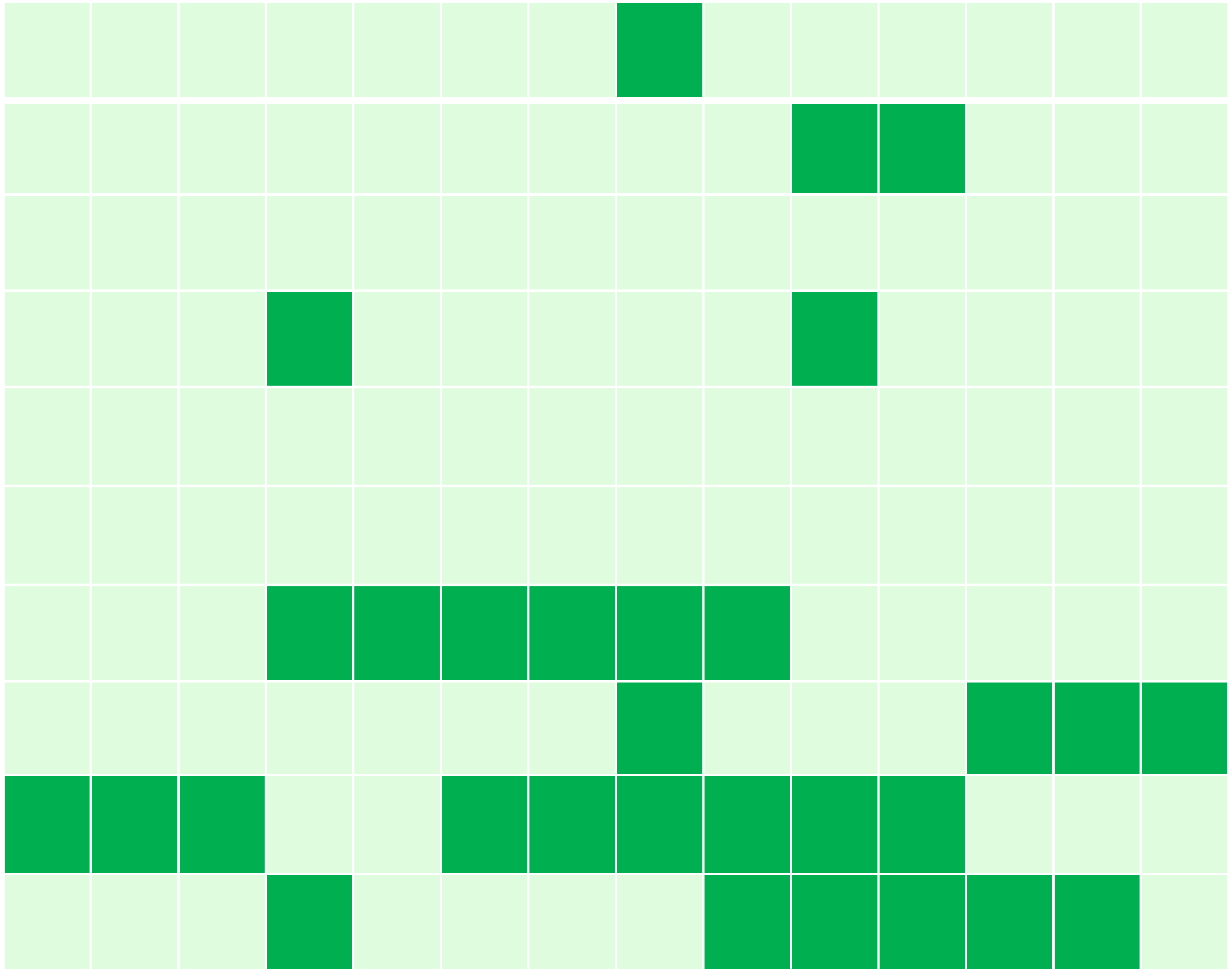
Only thing that remains:

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

Priority 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];

};
```

Priority 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];

};
```

Update Priority based on Interactivity

- Dynamically increase the priority level of the interactive processes (processes which does a lot of I/Os) for better user experience

Update Priority based on Interactivity

- Dynamically increase the priority level of the interactive processes (processes which does a lot of I/Os) for better user experience

- **Sleep Ratio:**
  - Number of clock ticks spent while getting executed in CPU / Number of clock ticks while blocked in the wait queue

Update Priority based on Interactivity

- Dynamically increase the priority level of the interactive processes for better user experience

- **Sleep Ratio:**
  - Number of clock ticks spent while getting executed in CPU / Number of clock ticks while blocked in the wait queue
  - **Mostly Sleeping:** I/O Bound (Interactive)
  - **Mostly Running**: CPU Bound (Batch)

O(1) Scheduler

- **Pros**
  - Better scalability
  - Faster performance
  - Reduced context switching time
  - Incorporated interactivity metric

O(1) Scheduler

- **Pros**
  - Better scalability
  - Faster performance
  - Reduced context switching time
  - Incorporated interactivity metric

- **Cons**
  - Complex heuristic to mark a process as interactive or batch – didn't work well in practice
  - The interactivity may change over time – initially CPU bound, then I/O bound
  - 140*2 runqueues for each processor – complex logic is required for runqueue locking
  - Complicated codebase – difficult for debugging purpose

Happy Learning